Kinetic Metallic Glass Evolution Model

by

Thomas James Hardin

Submitted to the Department of Materials Science and Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Materials Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

Signature redacted

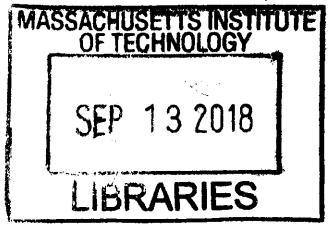
Author .....
.....
Department of Materials Science and Engineering
June 5, 2018

Signature redacted

Certified by.....
Christopher A. Schuh
Head, Department of Materials Science and Engineering
Danae and Vasilis Salapatas Professor of Metallurgy
Thesis Supervisor

Signature redacted

Accepted by ....
.....
Donald R. Sadoway
Chair, Departmental Committee on Graduate Studies



Kinetic Metallic Glass Evolution Model

by

Thomas James Hardin

Submitted to the Department of Materials Science and Engineering
on June 5, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Materials Science and Engineering

Abstract

The structure of metallic glass controls its mechanical properties; this structure can be altered by thermomechanical processing. This manuscript presents a model for this structural evolution of metallic glass under thermal and mechanical stimuli. The foundation of this model is a potential energy landscape; this consists of three pieces: a function for the energy of any given stable state, a density of states function across the landscape, and a model for the energetic barriers between stable states. All three of these pieces are parameterized in terms of the configurational potential energy of the glass, which is split into isochoric and dilatative degrees of freedom. Under a thermal or mechanical stimulus, the glass traverses the potential energy landscape by way of isotropic relaxation or excitation events, and by shear transformations. The rates of these events are calculated using transition state theory. This model is first implemented in homogeneous form, treating the glass nanostructure as a statistical distribution; this implementation, while devoid of spatial detail, is nonetheless able to fit many of the experimental results on homogeneous flow previously in the literature. The second implementation of the model is in a mesoscale discrete shear transformation zone dynamics framework; this couples the model's rate equations to discrete points in a finite element model under realistic thermomechanical loading, and propagates the effects of local events via static elasticity. Emphasis is placed on efficient computer implementation of the new model's physics, improving on the previous state of the art with stiffness matrix factor caching and geometric multigrid methods. These numerical improvements produce a 200x speedup over previous algorithms, enable rapid simulations of glass with evolving elastic properties, and facilitate the first-ever metallic glass simulations of physical nanomechanical experiments with matching length and time scales.

Thesis Supervisor: Christopher A. Schuh
Title: Head, Department of Materials Science and Engineering
Danae and Vasilis Salapatas Professor of Metallurgy

Dedication

To my family. Over four years I created this thesis, met and married my wife Rachel, and we had a son, Henry. In the end I suspect the latter two accomplishments will far exceed the first in importance.

In memory of my mother, the first of her family to attend college, who taught me to read and write and to love learning.

With deep gratitude to Rachel for her unwavering support of my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The defining characteristic of a metal is delocalization of electrons; metallic bonds between atoms are generally not directional in nature. This characteristic produces traits closely associated with metals: crystalline structure, ductility, and thermal and electrical conductivity. The defining characteristic of a glass is atomic disorder. This lack of a crystalline lattice precludes "easy" modes of plastic deformation (for example, dislocations), typically resulting in brittle behavior. Metallic glass is a chimaera: chemically metallic, yet structurally glassy. This section introduces the structure and properties of metallic glass. It also provides background on thermomechanical processing of metallic glass, and on models of such processes.

1.1 Metallic Glass

1.1.1 Heterogeneity, Structure and Parameters

"Amorphous," meaning "lacking long-range atomic order," is a seductively simple label to apply to metallic glass. It seems to suggest isotropic properties and homogeneous structure. Metallic glass, however, possesses richly diverse short-range atomic configurations. These configurations form a heterogeneous glassy structure on the scale of a few nanometers [1–3], with local anisotropy being the rule [4]. This structure dramatically impacts the macroscopic properties of the glass [5–7].

The most obvious indicator of the structure of a glass is its volume, giving rise to "free volume" (roughly defined as the volume of the glass minus the volume of an ideal, usually glassy, reference state) as a scalar structural parameter [8–10]. Free volume has shown its worth in successful models but has proven inadequate to fully describe the structure of a glass [11, 12]. Another useful scalar structural indicator is the configurational potential energy of the glass; this energy is indirectly experimentally measurable by calorimetry [13,14] and is readily obtained by atomistic simulation [12]. Configurational potential energy is a scalar multiple of another scalar parameter, the "fictive temperature" of the glass [15, 16]. These quantities are only useful for comparison of structures within a composition.

Flexibility volume is another recently developed (and promising) scalar measure of glassy structure [17, 18]; this combines the notion of free volume with an atom's "flexibility," that is, its mean vibrational displacement. This parameter seems to be composition-independent, but its evolution under thermomechanical processing has not been well studied.

Other scalar measures of glassy structural state are "granular fluidity" [19] and local solidity/liquidity [20]; these focus on the shear modulus of the glass, noting that the shear modulus of a fluid vanishes at low strain rates.

Moving away from scalar measures of glassy structure, atomistic simulations have enabled categorization of the polyhedral atomic "cages" surrounding each atom in a sample [2, 6, 12, 21–24]. These distributions have revealed that particular atomic environments are more or less energetically favored, with strong correlations to the configurational potential energy and free volume parameters described above.

As a general rule, these scalar parameters purport to place a glass on a spectrum in terms of level of disorder, with crystal and liquid/gas states at the extremes. At the more ordered end one tends to find lower free volumes, lower configurational energies, lower flexibility volumes, and more energetically favored atomic environments.

1.1.2 Thermal Response

One productive avenue for understanding glassy structure has been to track the just-described structural parameters through different heat-treatment pathways. The simplest such experiments cool glass samples from the melt at different rates and then compare resultant properties; faster cooling rates have been shown to produce higher free volume [12], lower shear modulus [22,25], and higher configurational potential energy [12]. Each of these also show that the structure of the unstressed glass becomes essentially fixed at low temperature. A variation on this theme measures properties as a function of annealing times at elevated temperatures, for example showing that higher annealing temperatures and times produced higher viscosity, indicative of structural relaxation [8].

These results suggest a classical kinetic explanation: that the glass locally changes its structure by thermally-activated relaxation events called α-relaxations. These relaxation events have been studied at length both experimentally and computationally [11,26–32]. Activation energies ranging from 1/10 eV to upward of 1eV have been reported for these events; there does seem to be a consensus that in any given glass a range of activation energies will be observed, presumably corresponding to the various atomic environments present. Atomistic studies have shown a strong correlation between the locations of relaxation events and regions of elevated disorder (that is, elevated free volume, configurational potential energy, etc) [2,17,18,33].

1.1.3 Mechanical Response

The mechanical behavior of metallic glass has been the subject of several recent review papers [34–36] and its literature is extensive; this section does not attempt to be comprehensive on the subject.

Elastic Regime

On a macroscopic level, a rule of thumb has been that metallic glass has a shear modulus on the order of 30 percent less than an isocompositional crystalline analog,

while the bulk modulus is reduced by around 5 percent [37]. However, as previously mentioned, elastic properties of a metallic glass depend strongly on its structure [38–40]; metallic glass can therefore be considered as an elastically heterogeneous composite with a structural length scale of a few nanometers [41].

Strains below the elastic limit have also been shown to encourage and bias the α-relaxation events described above; processing at high pressure has been shown to elevate the energy of the glass [42, 43]. Experiments on cyclic loading have also produced elevation or depression of the energetic state of glass [44, 45] with the direction of energy flow depending in nonlinear fashion on the method of loading.

Plastic deformation

The absence of long-range disorder in metallic glass precludes dislocation-mediated plasticity. The shear transformation (also called flow unit) fills a role analogous to dislocations in metallic glass [46–51]. A shear transformation is a localized collective rearrangement of atoms which produces a large local shear strain.

Shear transformations occur over clusters of (on the order of) 20-100 atoms with some variance expected within a given sample [52]. They involve a transitory dilatation producing tension-compression asymmetry in plastic behavior [53, 54]; more intuitively, a cluster of atoms can be thought of as "jammed" or "unjammed" [55], with compressive stress tending to jam the atoms together.

Shear transformations have been observed to localize to regions with lower shear modulus and higher free volume [22], with implications in engineering glassy structures that discourage strain localization [5, 33, 56]. Under certain circumstances shear transformations tend to occur in highly correlated fashion, producing well-defined shear bands [57–63]. These shear bands typically have lower shear modulus and elevated molar volume relative to undeformed glass [64–68] with (in the extreme case) excess volume coalescing into voids [69], thus making shear bands prime crack nucleation sites. Even in the absence of cracks the shear-softening behavior often associated with metallic glass increases the likelihood of catastrophic rather than gradual failure. The structural softening associated with shear transformations is further aggravated

by localized heating in shear bands [70–72].

The effect of plastic strain on the structure of the glass can depend strongly on the loading [73,74]; at temperatures where the glass flows homogeneously the steady-state strain rate correlates strongly with the steady-state stress, reflecting the competing rates at which the structure is excited by shear transformations and subsequently relaxes [75–77]. Similar effects are observed in stress-relaxation experiments [78], while cyclic loading and certain tension tests have actually measured hardening or densification through plastic strain [79–81].]

1.1.4 Rejuvenation

Rejuvenation refers to the process of taking the glass from a (brittle [82]) relaxed state to a less-relaxed state; that is, from a state with high shear modulus and prone to strain localization to a more ductile state with low shear modulus, with obvious benefits with respect to e.g. shape-forming processes [83–85]. This is a very active research area, with successes reported from methods including severe plastic deformation [86], heat treatment [4,87] and ion irradiation [15,88].

1.2 Modeling Metallic Glass

Modeling of metallic glass mechanical deformation and structural evolution spans atomistic length- and time-scales [89] through continuum constitutive models (of which there are many, including [90] and [58]). Early analog models using bubble rafts produced insight into the nature of the shear transformation event [47] which led to expressions for the kinetics of shear transformation activation based on Eshelby's solution for elastic inclusions [46, 49, 91]. These models continue to be relevant and refined in recent years [92–94]. The early models for STZ kinetics also lent themselves to homogenization into constitutive laws by analytical means [46, 50].

These and related kinetic laws have also been incorporated into mesoscale metallic glass models, which consider distributions of shear transformation zones across two-dimensional or three-dimensional samples. These select shear transformations, apply

strain associated with the transformations, and bias future transformations based on the elastic fields associated with preceding transformations (for a generalization of this idea, see [95]). Prototypical of this class (with a focus on the mechanical aspect of the problem) is [96–98], followed by Shear Transformation Zone Dynamics [81,94,99–105] and Discrete STZ Plasticity [106,107]. Additional examples focusing on the evolving structure of the glass are found in [5,56].

Other models step away from the kinetics of shear transformations in favor of a potential energy landscape concept, starting with [108] and [73] and updated more recently in [109–111]. In each of these cases, the details of the kinetics are deemphasized in favor of the structural state of the glass evolving across a more-or-less abstract energetic landscape. These methods lend themselves to studies of homogeneous deformation and thermally activated relaxation, but most make no attempt at spatially resolving the events under consideration (which is important when considering a material failing via shear band formation).

1.3 Thesis Structure

The rest of this thesis is organized as follows. In Chapter 2 the "Kinetics of Metallic Glass Evolution" Model is introduced. This consists of equations of state, a density of states function, and idealized structural transitions with accompanying potential barrier models. In Chapter 3 the KMGEM is implemented in homogenized form, treating the glassy nanostructure as a statistical distribution; this homogenized KMGEM is shown to fit experimental homogeneous flow data from the last forty years of literature on metallic glass. In Chapters 4 and 5 attention turns to mesoscale models of metallic glass, with Chapter 4 discussing a strategy for accelerating the extant Shear Transformation Zone Dynamics model. Chapter 5 presents preliminary work towards a mesoscale implementation of KMGEM. Chapter 6 is a summary and Chapter 7 describes directions for future research.

The appendices to this thesis contain a copy of the KMGEM codebase and a paper on homogenization and continuum percolation theory written by the author during

his doctoral tenure but not germane to the subject of metallic glass modeling.

Chapter 2

Kinetics of Metallic Glass Evolution Model (KMGEM)

This chapter presents a new model for the nanostructural evolution and mechanical deformation of metallic glass under thermomechanical loading.

Glass deforms and its nanostructure evolves by kinetic events on a range of energies and sizes; to simplify matters this model idealizes these into two fundamental kinetic events: isotropic relaxations and shear transformations. Isotropic relaxation events locally modify the structure of clusters of a few atoms, storing or dissipating energy and optionally increasing or decreasing volume. Shear transformations operate on larger clusters of atoms and primarily produce plastic shear strain, with local modifications to structure, energy, and volume as a side effect. These two fundamental kinetic events can be thought of as ways of "hopping" across a potential energy landscape.

This model is unfolded as follows. First, thermodynamics informs the selection of convenient variables to represent the local state of the metallic glass. Equations of state and empirically-based structure-property relations are postulated in connection with the selected state variables. Then, a plausible density of states function is introduced in terms of the state variables. Next, models for the potential barriers between states are presented for isotropic relaxations and shear transformations; the state space, density of states, and potential barrier models together comprise a po-

tential energy landscape. All that then remains is to use transition state theory to compute the rate at which the glass traverses the potential energy landscape.

Table 2.1: Operators appearing in this chapter

Symbol	Meaning
$\underset{1\to 2}{\Delta}$	denotes a change in a state variable between states 1 and 2
:	tensor operator representing elementwise multiplication, followed by summation

Table 2.2: Superscripts appearing in this chapter

Superscript	Meaning
$\circ$	unloaded reference state
$\star$	unloaded reference state at zero Kelvin
Ref	arbitrary reference state
$'$	extensive property for a small subset of atoms in the sample
$''$	extensive property for entire sample body

Table 2.3: Variables appearing in this chapter

Variable	Meaning
B	bulk modulus
$\boldsymbol{C}$	Hooke's stiffness tensor
$\boldsymbol{\epsilon}$	Elastic cauchy strain
G	thermodynamic potential minimized at equilibrium
k_{B}	Boltzmann constant
S	entropy
$\boldsymbol{\sigma}$	Cauchy stress
T	temperature
μ	shear modulus
U	internal energy
$U^\star$	configurational potential energy
$U^\star_{\mathrm{G}}$	ground configurational potential energy
$U^\star_{\mathrm{I}}$	isochoric configurational potential energy
$U^\star_{\mathrm{D}}$	volumetric configurational potential energy
V	volume
$\mathcal{V}$	potential energy of boundary and body force loading
W	total mechanical potential energy
Ω	the three-dimensional region occupied by the body
ω	the three-dimensional region occupied by a small cluster of atoms

2.1 Thermodynamics & State Variables

Assuming that a given sample is temperature-controlled, the following generalization of the Gibbs free energy is minimized at equilibrium:

$$G'' = U'' - TS'' + \mathcal{V}'' \tag{2.1}$$

where $\mathcal{V}$ (in caligraphic font) is the potential energy of applied boundary and body force loads. The double-prime superscript indicates an extensive property over the entire sample; that is,

$$U'' = \int_{\Omega^\circ} U/V^\circ d\Omega \tag{2.2}$$

where energies and generalized thermodynamic displacements (e.g. S, V, or U) with no prime superscript are intensive molar quantities. The variable V° is the local stress-free reference molar volume of the material, and Ω° is the three-dimensional region occupied by the sample in an unloaded state. In general the circle superscript denotes a stress-free (unloaded) reference state.

The internal energy U can be decomposed relative to a stress-free reference state at 0 Kelvin (schematically illustrated in Fig. 2-1); the molar internal energy of this reference state is denoted $U^\star$ and is termed "Configurational Potential Energy." The decomposition is:

$$U = U^\star_{\mathrm{G}} + U^\star + \underset{0\to T}{\Delta} U + \underset{\mathbf{0}\to\boldsymbol{\sigma}}{\Delta} U \tag{2.3}$$

where $U^\star_{\mathrm{G}}$ is the glassy ground state energy (related to the notion of an "ideal glass," as in [34, 108]), and

$$\underset{0\to T}{\Delta} U = \int_0^T c_P dT' \tag{2.4}$$

accounts for finite temperature. Assuming pseudo-static mechanical equilibrium the strain energy term can be written:

$$\underset{\mathbf{0}\to\boldsymbol{\sigma}}{\Delta} U = \frac{1}{2} V^\circ \boldsymbol{\sigma} : \boldsymbol{\epsilon} \tag{2.5}$$

where the colon operator denotes elementwise multiplication followed by summation.

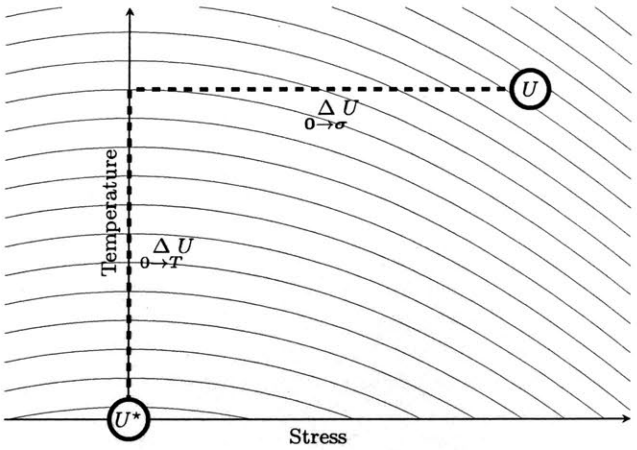


Figure 2-1: Schematic of partition of internal energy between configurational part, thermal part, and strain-energy part.

Substituting Eqn. (2.3) into Eqn. (2.1) one obtains:

$$G'' = U''^{\star}_{\mathrm{G}} + U''^{\star} + \underset{0\to T}{\Delta} U'' - TS'' + \underset{\mathbf{0}\to\boldsymbol{\sigma}}{\Delta} U'' + \mathcal{V}'' \tag{2.6}$$

The last two terms (strain energy and loading potential energy) are combined into the "total mechanical potential energy" W:

$$G'' = U''^{\star}_{\mathrm{G}} + U''^{\star} + \underset{0\to T}{\Delta} U'' - TS'' + W'' \tag{2.7}$$

Now, suppose that the sample (initially in state i) undergoes a single kinetic transition of the sort described above; that is, a small, localized portion of the sample shuffles its atoms. After the kinetic event the sample is in state f. The change in total free energy associated with the event is written:

$$\underset{i\to f}{\Delta} G'' = \underset{i\to f}{\Delta} U''^{\star} + \underset{i\to f}{\Delta}\underset{0\to T}{\Delta} U'' - T\underset{i\to f}{\Delta} S'' + \underset{i\to f}{\Delta} W'' \tag{2.8}$$

which, noting that the transition is localized, reduces to:

$$\underset{i\to f}{\Delta} G'' = \underset{i\to f}{\Delta} U'^{\star} + \underset{i\to f}{\Delta}\underset{0\to T}{\Delta} U' - T\underset{i\to f}{\Delta} S' + \underset{i\to f}{\Delta} W'' \tag{2.9}$$

where a single-prime superscript denotes an extensive property corresponding only

to the localized volume that underwent structural rearrangement in the transition. Note that since the kinetic event may entail a shape change (as in the case of a shear transformation) with associated elastic fields throughout the sample, the mechanical potential term W above is global, not local.

In general, the difference in entropy between two states can be written:

$$\underset{i\to f}{\Delta} S' = \underset{i\to f}{\Delta} S'^{\star} + \int_0^T n \underset{i\to f}{\Delta} c_P/T'dT' \tag{2.10}$$

dividing entropy into a configurational and thermal portion. Recent research has suggested that the difference between the entropy of a glass and the entropy of a similarly-composed crystal is nearly entirely configurational [112] (that is, that the thermal contribution is negligible). While there is a paucity of experimental data correlating entropy to glassy structure, if the difference in thermal entropy between a crystal and a glass is negligible, then it is reasonable to assume that the difference in thermal entropy between two similarly-composed glasses is negligible:

$$\int_0^T n \underset{i\to f}{\Delta} c_P/T'dT' \approx 0 \tag{2.11}$$

This is suggestive of a convenient simplifying assumption: that the local heat capacity of the transformed region is not altered by the transformation.

Under this heat capacity assumption and referencing Eqn. (2.4) the thermal term of Eqn. (2.9) vanishes:

$$\underset{i\to f}{\Delta}\underset{0\to T}{\Delta} U' = \int_0^T n \underset{i\to f}{\Delta} c_P dT' \approx 0 \tag{2.12}$$

Similarly the entropic term of Eqn. (2.9) is reduced:

$$\underset{i\to f}{\Delta} S' \approx \underset{i\to f}{\Delta} S'^{\star} \tag{2.13}$$

where $\underset{i\to f}{\Delta} S'^{\star}$ is the zero-Kelvin configurational entropy. This produces:

$$\underset{i\to f}{\Delta} G'' \approx \underset{i\to f}{\Delta} U'^{\star} - T \underset{i\to f}{\Delta} S'^{\star} + \underset{i\to f}{\Delta} W'' \tag{2.14}$$

In the absence of experimental data to support a model for the relative entropies of various glassy states, this model will subsume the configurational entropy term into degeneracy information contained in the density of states function introduced later in this section. To be clear, this approach is no less arbitrary than introducing an unsupported entropy model, but it has proved more computationally convenient. The resulting thermodynamic potential of interest is:

$$\underset{i\to f}{\Delta G''} \approx \underset{i\to f}{\Delta U'^{\star}} + \underset{i\to f}{\Delta W''} \tag{2.15}$$

2.1.1 Structural State Variables

Experiments on the nanostructure of glass (enumerated in the introduction) have produced large volumes of data on the energy and volume changes associated with glassy kinetic transitions. To capture these, the configurational potential energy is partitioned into an isochoric degree of freedom, and a dilatative degree of freedom:

$$U^{\star} = U^{\star}_{\mathrm{I}} + U^{\star}_{\mathrm{D}} \tag{2.16}$$

The isochoric configurational potential energy ($U^{\star}_{\mathrm{I}}$) does not correlate with the molar volume of the glass; it captures the energetic effects of atomic rearrangement with no associated shape change. The dilatative configurational potential energy ($U^{\star}_{\mathrm{D}}$) is that portion of the configurational potential energy which can be explained entirely by the presence of free volume. These two degrees of freedom will serve as the local structural state variables for metallic glass in this model, capturing both the internal arrangement of the atoms, and the dilatation of the glass.

2.2 Structure–Property Relationships

Experimental studies (both physical and computational) have indicated a strong linear correlation between the configurational potential energy of a glass, and its shear modulus. Accordingly, the following structure-property relationship is proposed for

shear modulus:

$$\mu(T, U^\star) = \mu^{\text{Ref}} \exp\left[\left.\frac{d\mu}{dT}\right|_{U^\star} \frac{(T - T^{\text{Ref}})}{\mu^{\text{Ref}}} + \left.\frac{d\mu}{dU^\star}\right|_{T} \frac{(U^\star - U^{\star\text{Ref}})}{\mu^{\text{Ref}}}\right] \tag{2.17}$$

This function is a first-order expansion around a reference temperature, configurational potential energy, and shear modulus, and is approximately linear around the reference datapoint. The exponential wrapper function is for computational convenience, producing finite positive shear values of the shear modulus for even improbably high values of the configurational potential energy. The derivative with respect to temperature $\left.\frac{d\mu}{dT}\right|_{U^\star}$ is the familiar Debye-Grüneisen slope, which captures the effect of thermal expansion on shear modulus in the absence of configurational structural relaxation.

A parallel law is proposed for the bulk modulus of the glass:

$$B(T, U^\star) = B^{\text{Ref}} \exp\left[\left.\frac{dB}{dT}\right|_{U^\star} \frac{(T - T^{\text{Ref}})}{B^{\text{Ref}}} + \left.\frac{dB}{dU^\star}\right|_{T} \frac{(U^\star - U^{\star\text{Ref}})}{B^{\text{Ref}}}\right] \tag{2.18}$$

Since research has shown the bulk modulus to be relatively configuration-insensitive, the configuration-dependent derivative will subsequently be approximated as zero.

Finally, a law is proposed relating dilatative configurational potential energy to the stress-free molar volume of the glass:

$$V^\circ(T, U^\star_{\text{D}}) = V^{\circ\text{Ref}} \exp\left[\alpha_V \left(T - T^{\text{Ref}}\right) + \left.\frac{dV^\circ}{dU^\star_{\text{D}}}\right|_{T} \frac{(U^\star_{\text{D}} - U^{\star\text{Ref}}_{\text{D}})}{V^{\circ\text{Ref}}}\right] \tag{2.19}$$

where α_V is the usual coefficient of thermal expansion (which does not account for configurational structural relaxation).

The relationships between state variables and properties are schematically illustrated in Fig. 2-2.

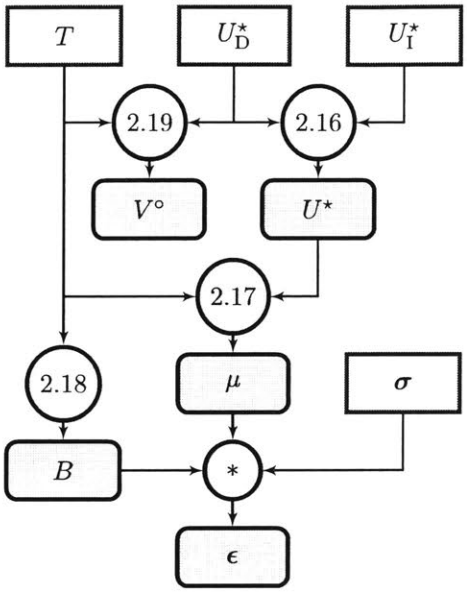


Figure 2-2: Schematic relationship between state variables and properties. The asterisk denotes Hooke's law for isotropic elasticity.

2.3 Density of States

With a thermodynamic potential in place, parameterized by convenient state variables, the next piece of the model is a density of states function. As previously mentioned, there is a dearth of information in the literature regarding the entropy (i.e. degeneracy) of various states of similarly-composed metallic glass, and so this model opts for the simplest possible density of states function with a few desirable properties. Specifically, we postulate:

$$D(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) \propto (U_{\mathrm{I}}^{\star})^{a} \, (U_{\mathrm{D}}^{\star})^{b} \tag{2.20}$$

To fix the constant of proportionality, the total number of states with configurational potential energy less than U^{Ref} is set to n^{Ref} (see Fig. 2-3 to illustrate region of integration):

$$n^{\mathrm{Ref}} = \int_{0}^{U^{\mathrm{Ref}}} \int_{0}^{U^{\mathrm{Ref}} - U_{\mathrm{I}}^{\star}} D(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) dU_{\mathrm{D}}^{\star} dU_{\mathrm{I}}^{\star} \tag{2.21}$$

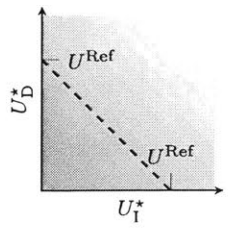


Figure 2-3: Region of integration over density of states for establishing proportionality constant.

resulting in the following explicit representation of the density of states:

$$D(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) = \frac{n^{\mathrm{Ref}}}{(U^{\mathrm{Ref}})^2} \frac{\Gamma(3+a+b)}{\Gamma(1+a)\Gamma(1+b)} \left(\frac{U_{\mathrm{I}}^{\star}}{U^{\mathrm{Ref}}}\right)^{a} \left(\frac{U_{\mathrm{D}}^{\star}}{U^{\mathrm{Ref}}}\right)^{b} \tag{2.22}$$

The desirable properties of this density of states function become apparent when Boltzmann statistics are applied.

As a final note, the integral of this density of states across a rectangular region of state space is:

$$\begin{aligned}\int_{U_{\mathrm{I0}}^{\star}}^{U_{\mathrm{I1}}^{\star}} \int_{U_{\mathrm{D0}}^{\star}}^{U_{\mathrm{D1}}^{\star}} D(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) dU_{\mathrm{D}}^{\star} dU_{\mathrm{I}}^{\star} &= \frac{n^{\mathrm{Ref}}\Gamma(3+a+b)}{\Gamma(2+a)\Gamma(2+b)} \\ &* \left(\left(\frac{U_{\mathrm{I1}}^{\star}}{U^{\mathrm{Ref}}}\right)^{1+a} - \left(\frac{U_{\mathrm{I0}}^{\star}}{U^{\mathrm{Ref}}}\right)^{1+a}\right)\left(\left(\frac{U_{\mathrm{D1}}^{\star}}{U^{\mathrm{Ref}}}\right)^{1+b} - \left(\frac{U_{\mathrm{D0}}^{\star}}{U^{\mathrm{Ref}}}\right)^{1+b}\right)\end{aligned} \tag{2.23}$$

2.3.1 Boltzmann Statistics

Assuming that the local states inside of a metallic glass sample are not correlated, one can apply Boltzmann statistics to determine the equilibrium distribution $\mathcal{P}$ of those states at given temperature. In particular:

$$\mathcal{P}(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) \propto D(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) \exp\left[-G'(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star})/k_{\mathrm{B}}T\right] \tag{2.24}$$

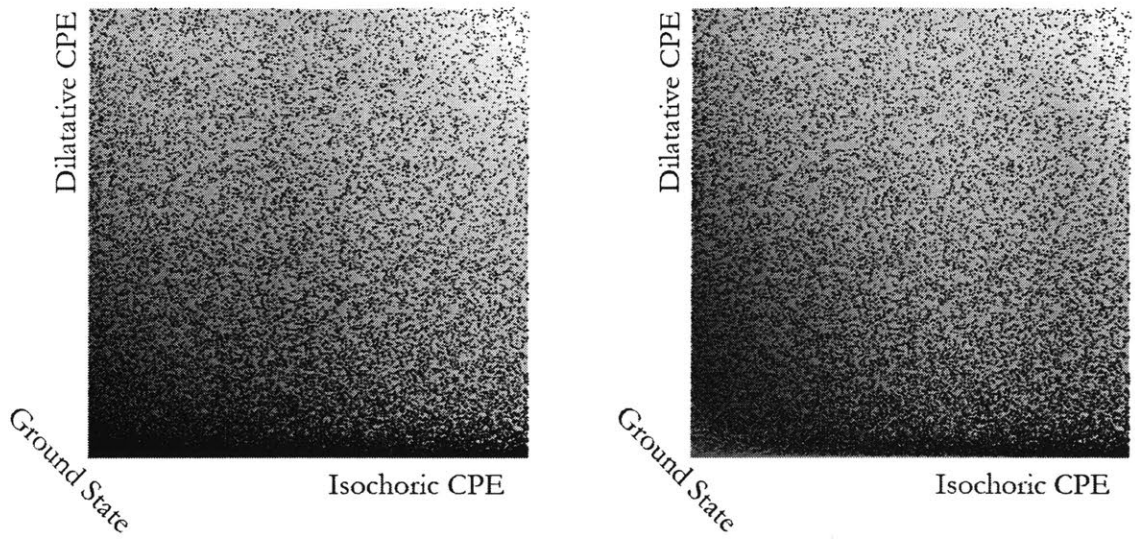


Figure 2-4: Schematic illustrating the distribution of states (denoted by black dots) in CPE state space. On the right, red dots indicate states occupied according to the Boltzmann statistics.

In the zero-stress limit with small dilatations (as would be typical in thermal processing), the normalized probability is written (drawing from Eqn. (2.15) and Eqn. (2.16)):

$$\mathcal{P}(U_{\mathrm{I}}^{\star}, U_{\mathrm{D}}^{\star}) = \left(\frac{U_{\mathrm{I}}^{\star}}{k_{\mathrm{B}}T}\right)^{a} \left(\frac{U_{\mathrm{D}}^{\star}}{k_{\mathrm{B}}T}\right)^{b} \exp\left(\frac{-U_{\mathrm{I}}^{\star} - U_{\mathrm{D}}^{\star}}{k_{\mathrm{B}}T}\right) / \left((k_{\mathrm{B}}T)^{2} ab\Gamma(a)\Gamma(b)\right) \tag{2.25}$$

Only now do some desirable properties of the chosen density of states function become apparent:

$$\langle U_{\mathrm{I}}^{\star} \rangle = (1 + a)k_{\mathrm{B}}T \tag{2.26}$$

$$\langle U_{\mathrm{D}}^{\star} \rangle = (1 + b)k_{\mathrm{B}}T \tag{2.27}$$

$$\langle U^{\star} \rangle = (2 + a + b)k_{\mathrm{B}}T \tag{2.28}$$

In particular, it is possible to choose exponents a and b such that the equilibrium configurational potential energy is $k_B T$ with an arbitrary equilibrium ratio of isochoric to dilatative configurational potential energy:

$$a = \phi - 1 \tag{2.29}$$

$$b = -\phi \tag{2.30}$$

where

$$\phi = \langle U_{\mathrm{I}}^{\star} \rangle \,/\, \langle U^{\star} \rangle \tag{2.31}$$

Having established the density of states function, all that remains are models for the potential barriers associated with transitions between states by way of isotropic relaxations and shear transformations.

2.4 Isotropic Relaxations

The simplest possible barrier model satisfying detailed balance is:

$$\underset{\mathrm{i}\to\mathrm{t}}{\Delta G''} = \max\left\{0, \underset{\mathrm{i}\to\mathrm{f}}{\Delta G''}\right\} + \underset{\mathrm{int}}{\Delta G'} \tag{2.32}$$

where the subscript t denotes the transition state, and $\underset{\mathrm{int}}{\Delta G'}$ is a positive intrinsic barrier height (illustrated in Fig. 2-5).

$$\underset{\mathrm{int}}{\Delta G'}(\mu) = \underset{\mathrm{int}}{\Delta G'}(0) + \frac{\mu}{\mu^{\mathrm{Ref}}}\left(\underset{\mathrm{int}}{\Delta G'}(\mu^{\mathrm{Ref}}) - \underset{\mathrm{int}}{\Delta G'}(0)\right) \tag{2.33}$$

The quantity $\underset{\mathrm{i}\to\mathrm{f}}{\Delta G''}$ is evaluated using Eqn. (2.15). The mechanical potential energy term is analytically tractable as described at length in [113] chapters 2 and 4; in the limit of a small dilatation associated with the transformation (as would be expected in a metallic glass, where an extreme dilatation is less than 1 percent) the mechanical term approximately vanishes (though in any given concrete implementation, this simplification can be viewed as optional). With a barrier model in place, transition state theory [114] predicts the rate of transition between states:

$$\dot{s}_{\mathrm{i}\to\mathrm{f}} = \nu^{\circ} \exp\left(\frac{-\underset{\mathrm{i}\to\mathrm{t}}{\Delta G''}}{k_{\mathrm{B}}T}\right) \tag{2.34}$$

where ν° is an attempt rate.

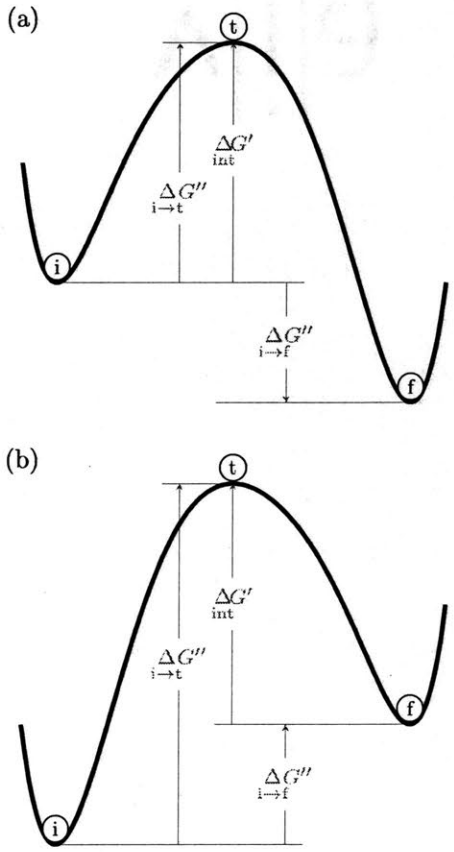


Figure 2-5: Schematic of transition energy calculation for (a) relaxation and (b) excitation events. The initial, transition, and final states are marked.

2.5 Shear Transformations

The model for a transition potential barrier for shear transformation draws from Homer et al:

$$\underset{\text{i}\to\text{t}}{\Delta G''_{\boldsymbol{Q}}} = \underset{\text{i}\to\text{t}}{\Delta F'} - \Omega\boldsymbol{\sigma} : \boldsymbol{\epsilon}^{\text{t}}_{\boldsymbol{Q}} \tag{2.35}$$

where

$$\boldsymbol{\epsilon}^{\text{t}}_{\boldsymbol{Q}} = \boldsymbol{Q}^{\text{T}} \begin{bmatrix} \varepsilon_V/3 & \gamma/4 & 0 \\ \gamma/4 & \varepsilon_V/3 & 0 \\ 0 & 0 & \varepsilon_V/3 \end{bmatrix} \boldsymbol{Q} \tag{2.36}$$

$$\underset{\text{i}\to\text{t}}{\Delta F'}(\mu) = \underset{\text{int}}{\Delta F'}(0) + \frac{\mu}{\mu^{\text{Ref}}} \left(\underset{\text{int}}{\Delta F'}(\mu^{\text{Ref}}) - \underset{\text{int}}{\Delta F'}(0) \right) \tag{2.37}$$

where $\boldsymbol{\epsilon}$ is the simple shear eigenstrain associated with the final state of the shear transformation. The barrier height, denoted $\underset{\text{i}\to\text{t}}{\Delta F''}$, is considered to be a linear function of shear modulus, with slope and intercept set as model parameters. The activation rate of a single shear transition is predicted using transition state theory as before:

$$\dot{s}_{\boldsymbol{Q}} = \nu^{\circ} \exp\left(\frac{-\underset{\text{i}\to\text{t}}{\Delta G''_{\boldsymbol{Q}}}}{k_{\text{B}}T} \right) \tag{2.38}$$

$$\dot{s}^{\text{tot}} = \int \dot{s}_{\boldsymbol{Q}}\, d\boldsymbol{Q} \tag{2.39}$$

The total rate of activation of shear transformation events is accessed by integrating Eqn. (2.38) over all possible simple shears. This is discussed at length in [102], to which the reader is referred.

One issue neglected in [102], however, is stress-activated (athermal) shear transformations; the simple model described above predicts exponentially increasing activation rate with increasing stress magnitudes, with transitions occurring orders of magnitude more frequently than the attempt frequency just above the yield stress of the material. In order to avoid this issue, this model first finds the smallest possible transition potential barrier from among all the possible shear strain orientations. If that potential barrier is negative then a stress-activated shear transformation is assumed to preempt any thermally-activated transformation. A stress-activated shear

transformation occurs with a frequency equal to the attempt frequency, and the only permissible associated shear tensor is the one that maximizes dissipated energy.

Analytical description of the effect of shear transformations on glassy structure is deferred to the next chapter, where it is described in terms of a finite set of states; here it suffices to say that a shear transformation resets its constituent relaxation units to a distribution which would be associated with elevated temperature and a large negative pressure. The effect of these conditions is that shear transformations tend to inject volume and configurational potential energy into the glass, consistent with experiment.

2.6 Concluding Thoughts

This chapter outlined a model for the evolution of metallic glass under thermal and mechanical loading. It introduced two state variables (isochoric and dilatative configurational potential energy) and linked them to the physical properties of the glass. It then postulated a density of states function and equations for calculating the rates at which the glass transitions between the various states by way of two idealized transitions: isotropic relaxations and shear transformations.

The next few chapters implement these ideas at varying levels of detail. The next chapter describes the thermomechanical evolution of glass with its nanostructure described only as a statistical distribution (with no spatial discrimination). Later the model will be implemented in a discrete mesoscale framework, with complete spatial information. While both approaches leverage the same physics described in this chapter, the contrast between the two sets of results will highlight the value of each.

Chapter 3

Homogenous KMGEM

This chapter describes the implementation and results of the Kinetic Metallic Glass Evolution Model (KMGEM) in setting of statistical homogeneity; that is, this chapter eliminates spatial resolution of events in favor of a statistical description of the sample's evolving state. The first part of this chapter covers homogenization of the KMGEM equations described in the previous chapter; the second part of this chapter presents some results in comparison to preexisting literature to demonstrate the power of KMGEM to fit current understanding of metallic glass evolution.

3.1 Discretizing State Space

The density of states function (Eqn. (2.22)) is continuous in state space; it is convenient here to partition state space into a finite number of discrete levels, each with degeneracy arising from the density of states.

The levels can be distributed any number of ways, but the approach taken here is to distribute them in $U_{\mathrm{I}}^{\star}$ and $U_{\mathrm{D}}^{\star}$ space as follows:

$$U_{?}^{\star}(i) = A\sinh(Bi) \tag{3.1}$$

where A and B are calibrated to an appropriate initial step size and final energy value. This function increases the resolution of the levels near the ground state,

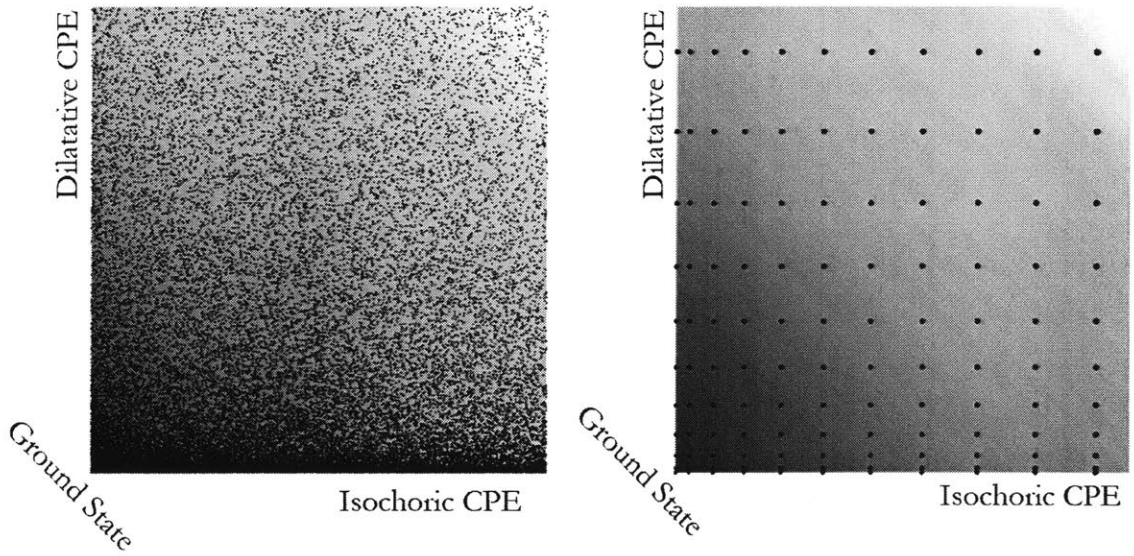


Figure 3-1: Schematic of continuous density of states and corresponding discrete levels.

which is desirable since only a small fraction of the sample's relaxation zones will be occupying high-energy states at any given time. Each level is in the center of a rectangular region of state space; the degeneracy of each level is found by integrating the density of states using Eqn. (2.23). The maximum possible energy level is selected using a cutoff value near zero, where the Boltzmann distribution predicts that the fraction of occupied states in continuous state space above the highest energy level is less than the cutoff.

With a discrete set of levels and degeneracies in hand, the instantaneous structural state of the homogeneous sample can be conceptualized as a bar graph, where each bar corresponds to a level, and the height of the bar is the fraction of relaxation zones occupying that level.

The rule of mixtures (i.e. a weighted average) is used to compute the homogenized properties of the glass.

3.2 Evolution via Relaxation

In this and subsequent sections, levels are indexed by subscripts and the population occupying level i is denoted $\mathcal{P}_i$. The gross rate of relaxation zones relaxing from level

i to level j can be expressed:

$$\dot{\mathcal{P}}_{i\to j} = \mathcal{P}_i \dot{s}_{i\to j} D_j \tag{3.2}$$

where $\dot{s}^{\text{rlx}}_{i\to j}$ is the relaxation rate from the KMGEM physics described in Chapter 2. Considering all possible relaxations one obtains:

$$\dot{\mathcal{P}}_i = \sum_{j=1}^{n} \mathcal{P}_j \dot{s}_{j\to i} D_i - \sum_{j=1}^{n} \mathcal{P}_i \dot{s}_{i\to j} D_j \tag{3.3}$$

representing the net rate of change of the occupation of each level in the sample. This is readily converted to a matrix equation:

$$\dot{\boldsymbol{\mathcal{P}}} = \boldsymbol{A}\boldsymbol{\mathcal{P}} \tag{3.4}$$

where $\boldsymbol{A}$ depends on the temperature of the sample. This form lends itself to solution by various well-established numerical ordinary differential equation methods.

3.3 Evolution via Shear Transformation

A similar approach applies to evolution via shear transformation. Implementing the evolution rule as described in Chapter 2 in a discrete-levels framework, one writes:

$$\dot{s}^{\text{st}}_{i\to j} = \dot{s}^{\text{st:tot}}_i q_j \tag{3.5}$$

where, in order to satisfy convergence to a temperature- and pressure-biased steady-state, the destination levels obey:

$$q_j = \frac{\dot{s}^{\text{tot}}_j D_j \exp\left(-\frac{G''_j + \Delta P V'_j}{k_{\text{B}}(T+\Delta T)}\right)}{\sum_{i=1}^{n} \dot{s}^{\text{tot}}_i D_i \exp\left(-\frac{G''_i + \Delta P V'_i}{k_{\text{B}}(T+\Delta T)}\right)} \tag{3.6}$$

with ΔP being a pressure bias (generally negative) to encourage the accumulation of free volume through shear transformation processes, and ΔT being a (generally positive) temperature bias to encourage accumulation of configurational potential

energy and accompanying strain softening.

Since density of states is already accounted-for in Eqn. (3.6), the net occupation fraction rate of change differs slightly from Eqn. (3.3):

$$\dot{\mathcal{P}}_i = \sum_{j=1}^{n} \mathcal{P}_j \dot{s}_{j\to i} - \sum_{j=1}^{n} \mathcal{P}_i \dot{s}_{i\to j} \tag{3.7}$$

but as before it reduces to a matrix equation:

$$\dot{\boldsymbol{\mathcal{P}}} = \boldsymbol{A}^{\mathrm{st}} \boldsymbol{\mathcal{P}} \tag{3.8}$$

where the matrix $\boldsymbol{A}^{\mathrm{st}}$ depends on temperature and stress of the sample.

The strain rate due to shear transformation is also homogenized by integrating the product of the shear transformation rate with the quantum of shear strain over all possible shear orientations:

$$\dot{\boldsymbol{\epsilon}} = \int \boldsymbol{\epsilon}^{\mathrm{f}}_{\boldsymbol{Q}} \dot{s}_{\boldsymbol{Q}} \, d\boldsymbol{Q} \tag{3.9}$$

where the relevant quantum of shear strain is:

$$\boldsymbol{\epsilon}^{\mathrm{f}}_{\boldsymbol{Q}} = \boldsymbol{Q}^{\mathrm{T}} \begin{bmatrix} 0 & \gamma/2 & 0 \\ \gamma/2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{Q} \tag{3.10}$$

3.3.1 Results: Tension-Compression Asymmetry

Plotting $|\dot{\boldsymbol{\epsilon}}|_2$ in the $\sigma_z = 0$ plane of principal stress space reveals that the strain rate is not pressure-invariant; this is directly a consequence of the ε_V term in Eqn. (2.36), which represents momentary dilatation in the transition state of the shear transformation. Comparison with molecular dynamics data from [51] reveals a close fit between the shapes of the measured yield surface of a model metallic glass and that of a strain-rate isosurface (see Figs. 3-2 and 3-3). The data shown corresponds to a transition state dilatation of 0.0089; the observed tension-compression asymmetry is

sensitive to this parameter. The quality of the fit appears superior to that obtained in [51] using the Mohr-Coulomb yield criterion, particularly in the biaxial tension and biaxial compression regimes.

3.4 Solution of Evolution Equations

The total level occupation rate of change for the sample is the sum of the effects of relaxation and shear transformation:

$$\boldsymbol{A}^{\text{tot}}(T, \boldsymbol{\sigma}) = \boldsymbol{A}^{\text{rlx}}(T) + \boldsymbol{A}^{\text{st}}(T, \boldsymbol{\sigma}) \tag{3.11}$$

$$\dot{\boldsymbol{\mathcal{P}}} = \boldsymbol{A}^{\text{tot}}\boldsymbol{\mathcal{P}} \tag{3.12}$$

Because this ordinary differential equation may be stiff, it is desirable to use an implicit integration scheme; the backwards Euler method is quickly computed for small (less than a few hundred) numbers of discrete levels:

$$\boldsymbol{\mathcal{P}}^{k+1} = (\boldsymbol{I} - \Delta t \boldsymbol{A}^{\text{tot}})^{-1}\boldsymbol{\mathcal{P}}^{k} \tag{3.13}$$

where $\boldsymbol{A}^{\text{tot}}$ is updated with each step.

3.5 Results: cooling rate experiment

To examine the effectiveness of this approach we consider a classic experiment: to cool a metallic glass at varying rates, and then compare the resultant properties at room temperature. Cooling rates from 1 to 10^{12} K/s are modeled in this study; the slower rates are unrealistic for glass formation but are included for the sake of completeness. For comparison the results of atomistic cooling rate experiments from [12] and [22] are shown here; these experiments only cover the faster cooling rates from this study for computational reasons.

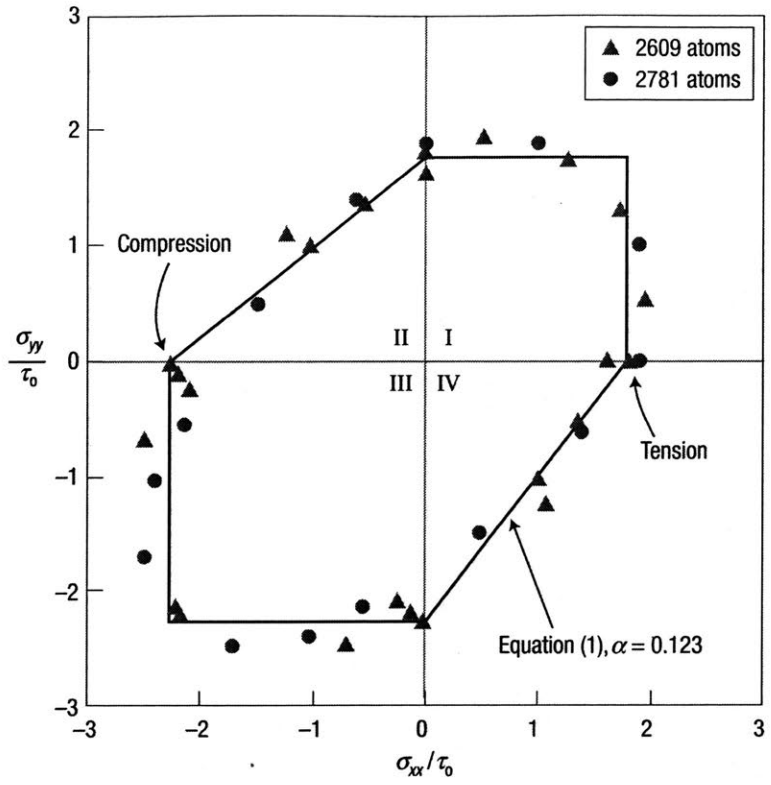


Figure 3-2: Measured yield surface for a model metallic glass from [51] with a fitted Mohr-Coulomb yield surface plotted in black

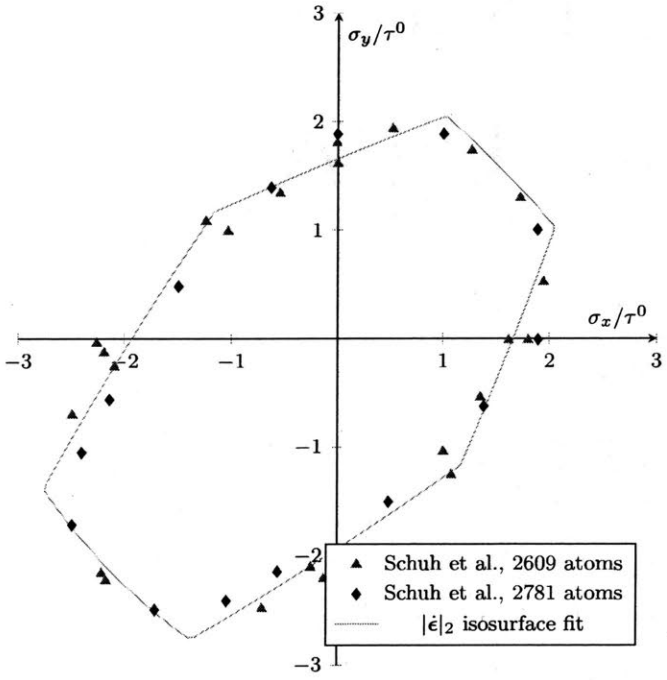


Figure 3-3: Strain rate isosurface plotted in green over data from [51]

3.5.1 Cooling rate and $U^{\star}$

Plots of configurational potential energy vs temperature for the studied cooling rates are shown in Fig. 3-5; for comparison, atomistic results from [12] are shown in Fig. 3-4. Noting that only relative energy values are meaningful here, the four fastest-cooling curves from this study qualitatively track with the data from the literature. As expected, the configurational potential energy becomes "frozen in" at low temperatures, with the onset of this transition occuring at higher temperatures for faster cooling rates.

The room temperature configurational potential energy is plotted for the various cooling rates in Fig. 3-7, with atomistic data for comparison from [22] in Fig. 3-6. Comparison of the fast cooling rate regime shows the same approximately logarithmic dependence in both cases.

3.5.2 Cooling rate and shear modulus

Plots of shear modulus vs temperature for the studied cooling rates are shown in Fig. 3-8 with the Debye-Grüneisen effect producing a nonzero slope in the regime where the structure is frozen in. The room temperature shear modulus is plotted for the various cooling rates in Fig. 3-10, with atomistic data for comparison from [22] in Fig. 3-9. As would be expected given the just-described behavior of the configurational potential energy, comparison of the fast cooling rate regime shows the same approximately logarithmic dependence in both cases.

3.5.3 Cooling rate and molar volume

Glass volume was one of the first discovered indicators of metallic glass structure, so it is included in this study. The molar volume of the glass is plotted with respect to temperature in Fig. 3-12 and atomistic data from [12] is plotted for comparison in Fig. 3-11. The room temperature plot of molar volume as a function of cooling rate is shown in Fig. 3-13 and features the same approximately logarithmic dependence seen previously. Two unexplained features are noted here as subjects for future ex-

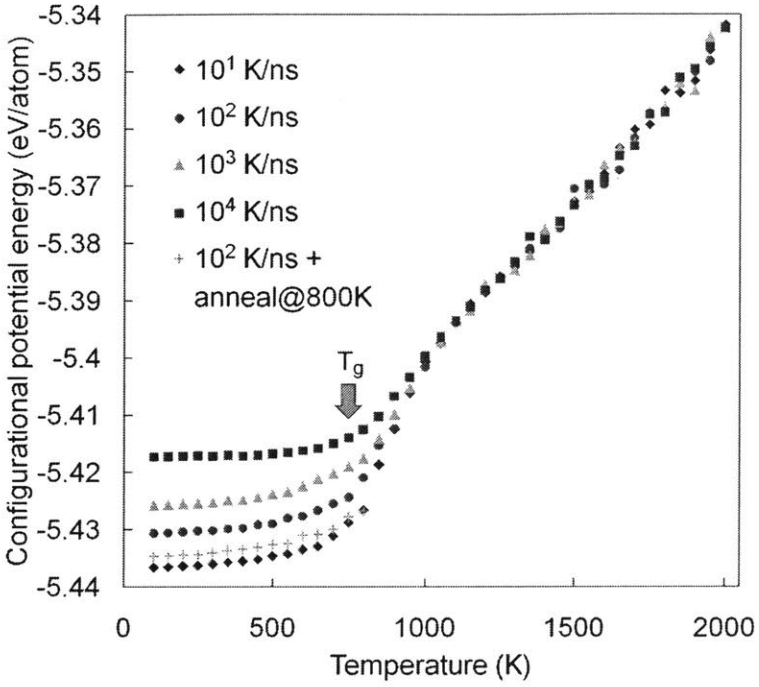


Figure 3-4: Experimental temperature vs configurational potential energy from atomistic experiments in [12].

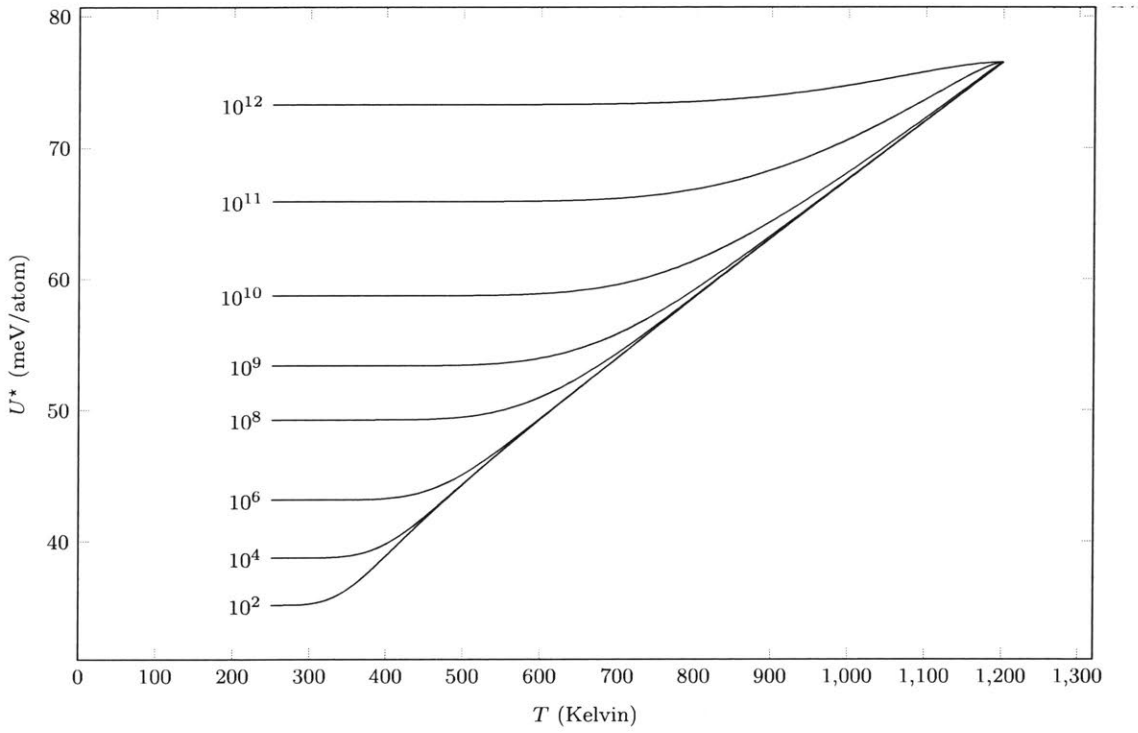


Figure 3-5: Temperature vs configurational potential energy for various cooling rates, calculated using KMGEM.

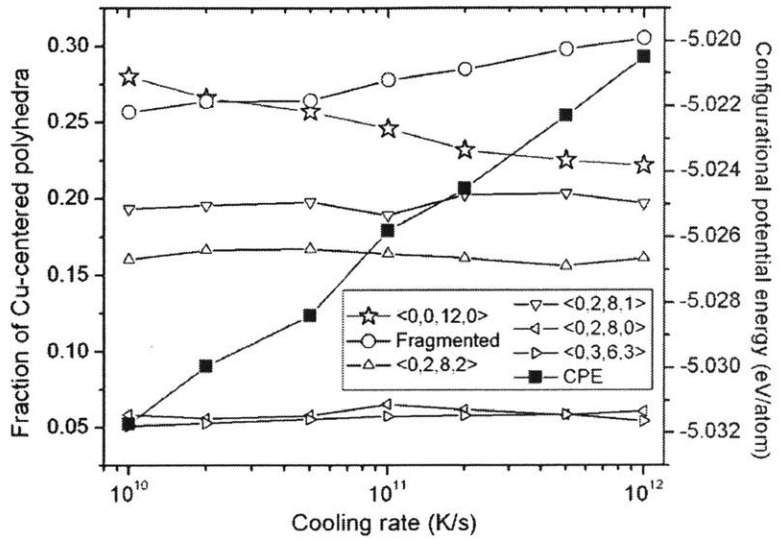


Figure 3-6: Experimental cooling rate vs molar volume at room temperature from atomistic data in [22].

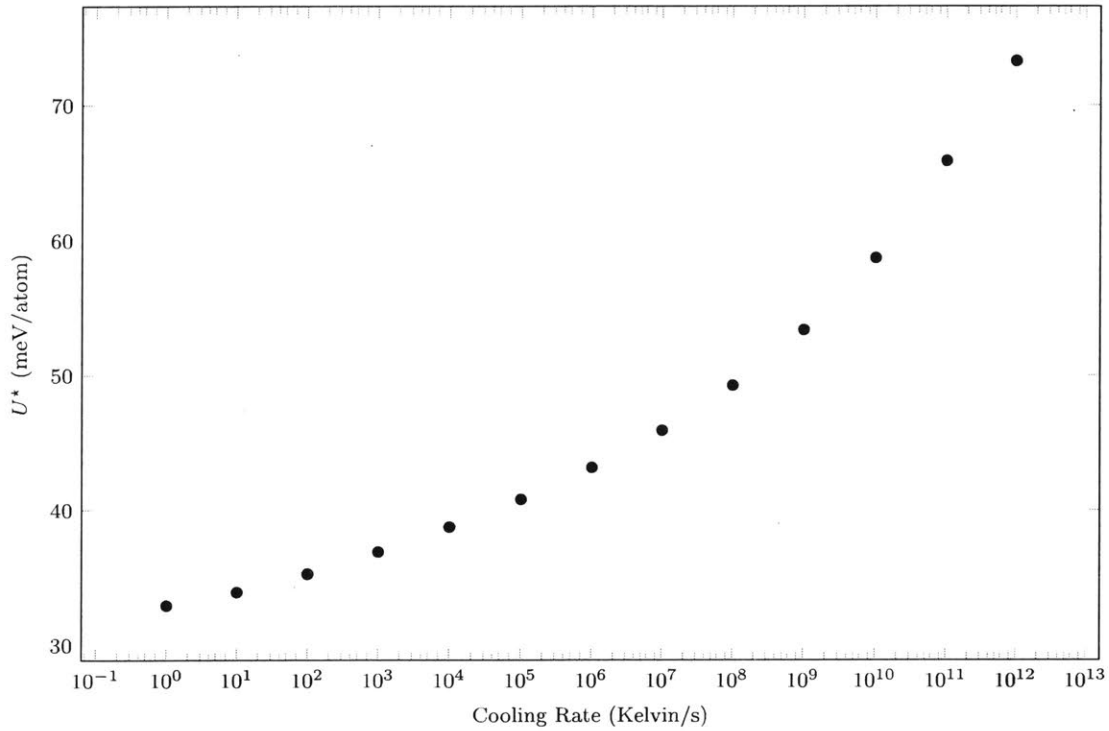


Figure 3-7: Cooling rate vs configurational potential energy at room temperature, calculated using KMGEM.

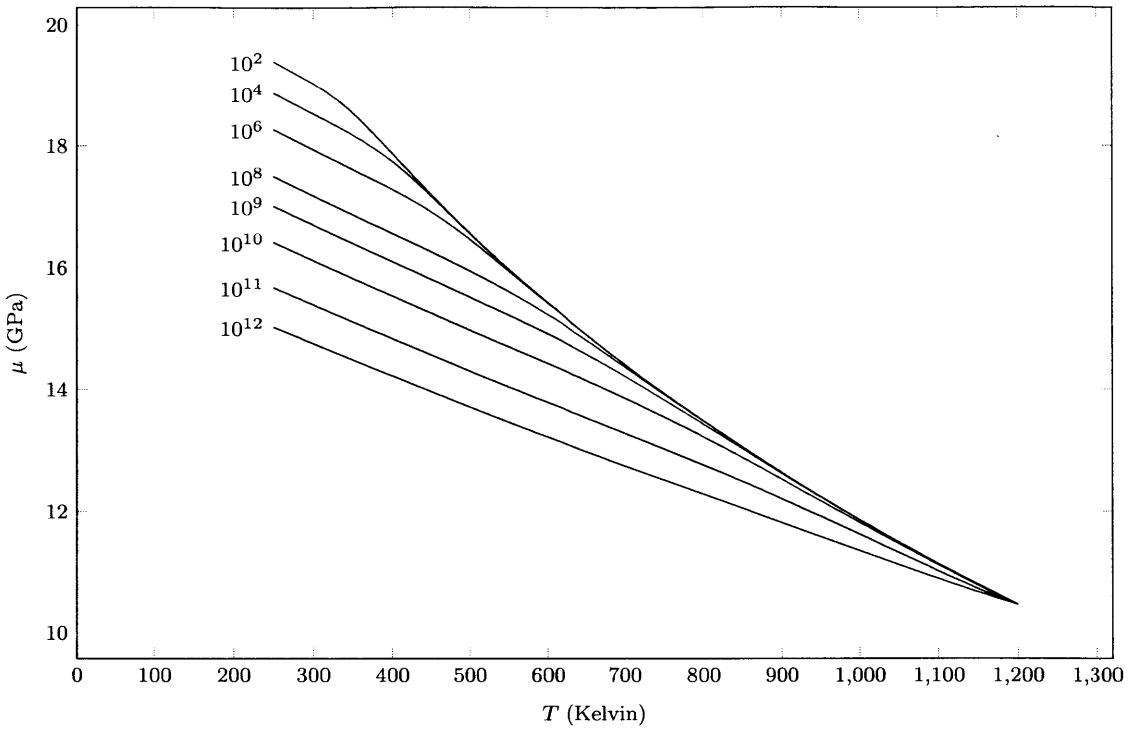


Figure 3-8: Temperature vs shear modulus for various cooling rates, computed using KMGEM.

amination: first, the molar volume undergoes an inflection around 500 Kelvin for the slower cooling rates, and second, the room temperature molar volume for the glass cooled at 10^{12} Kelvin per second seems to exceed the trend from lower cooling rates.

3.6 Conclusions

The strategy described in this section provides an efficient way to model the structural evolution of metallic glass under thermal loading. This approach is able to qualitatively fit atomistic data previously in literature, modeling the effect of cooling rate on configurational potential energy, shear modulus, and molar volume of the glass. The approach in this chapter is limited by the absence of information about the spatial distribution of kinetic events. This shortcoming will be addressed in the next two chapters.

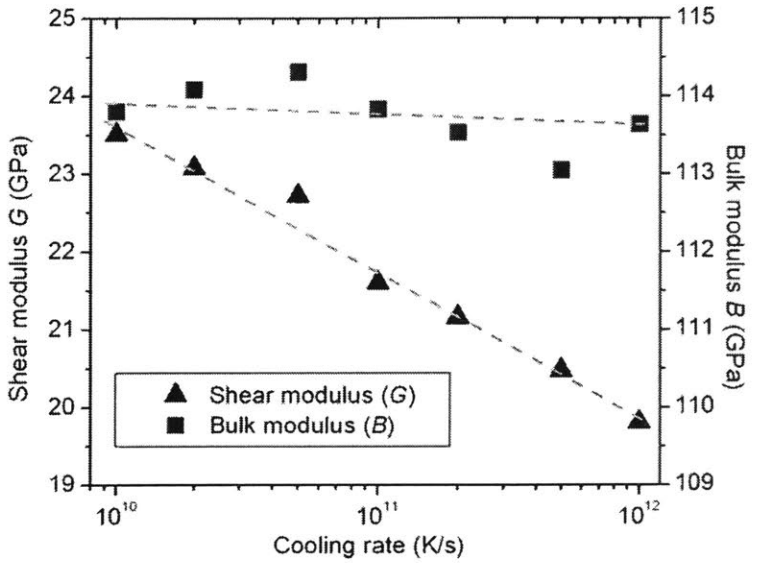


Figure 3-9: Experimental cooling rate vs shear modulus at room temperature from atomistic data in [22].

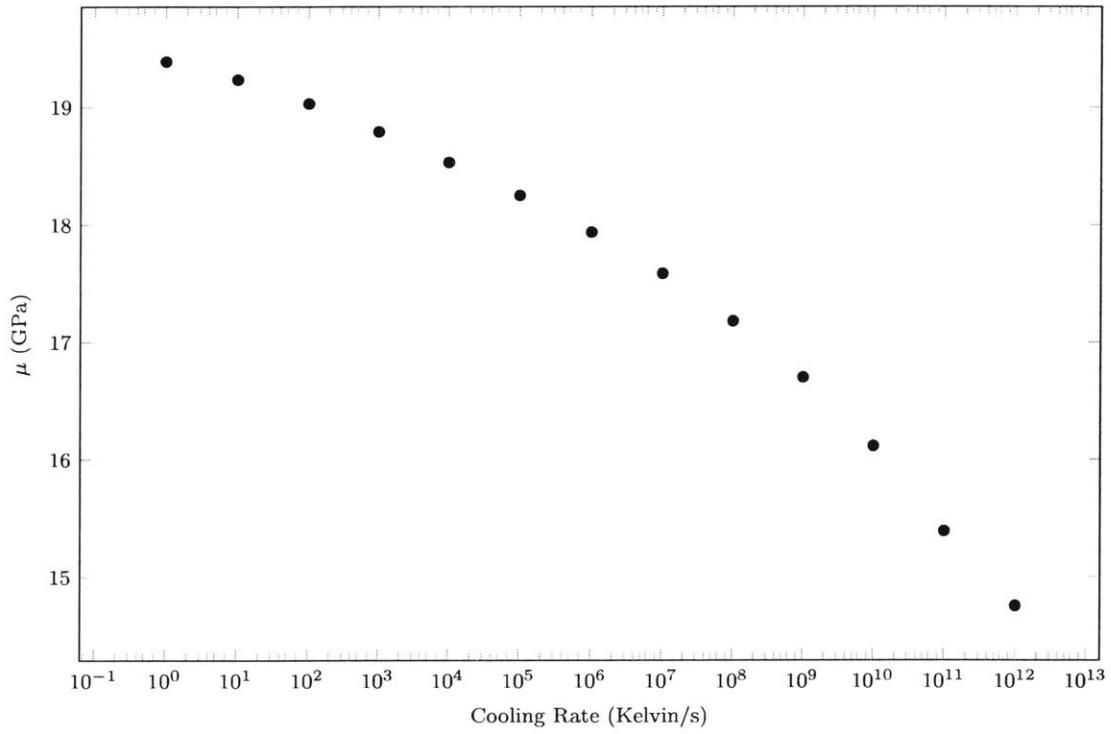


Figure 3-10: Cooling rate vs shear modulus at room temperature, computed using KMGEM.

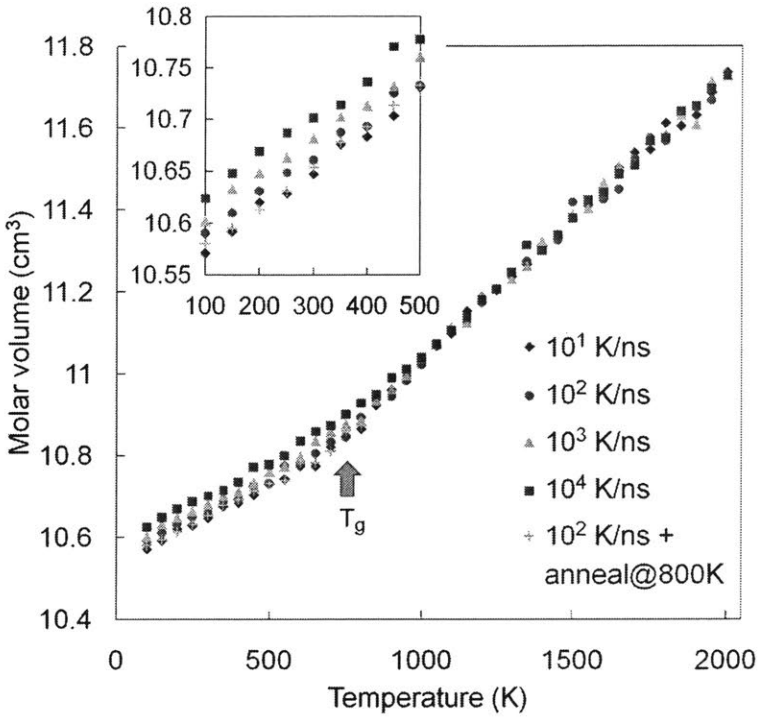


Figure 3-11: Experimental temperature vs molar volume from atomistic data in [12].

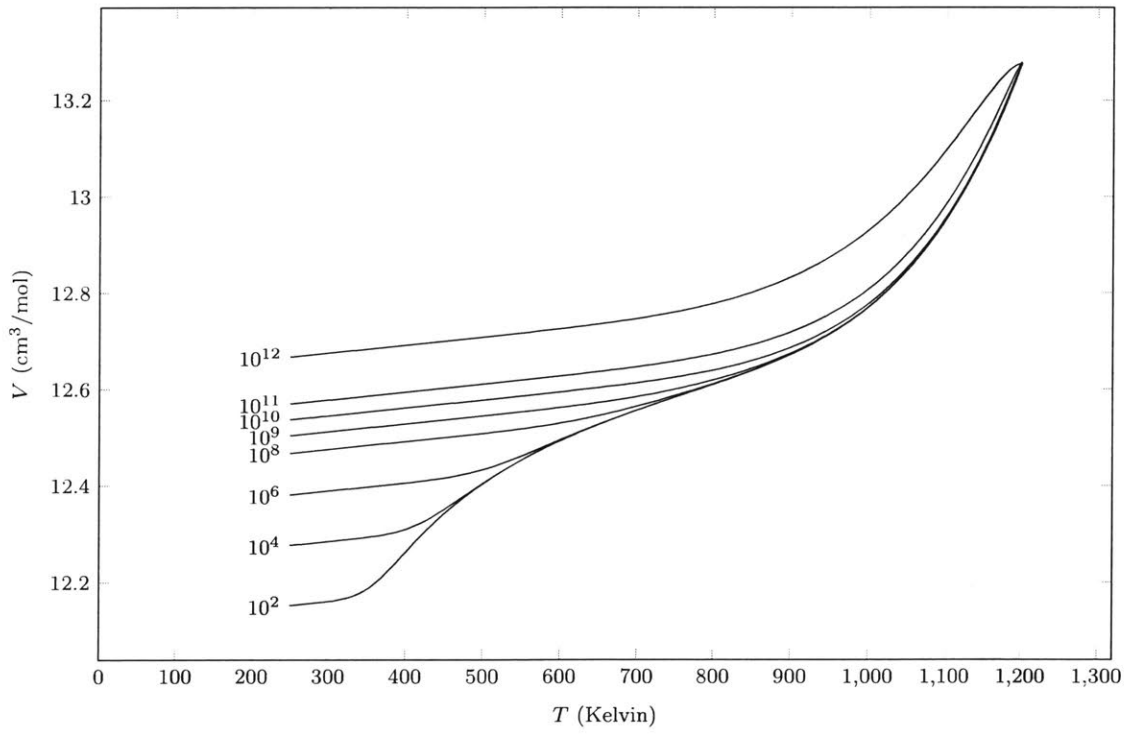


Figure 3-12: Temperature vs molar volume for various cooling rates, calculated using KMGEM.

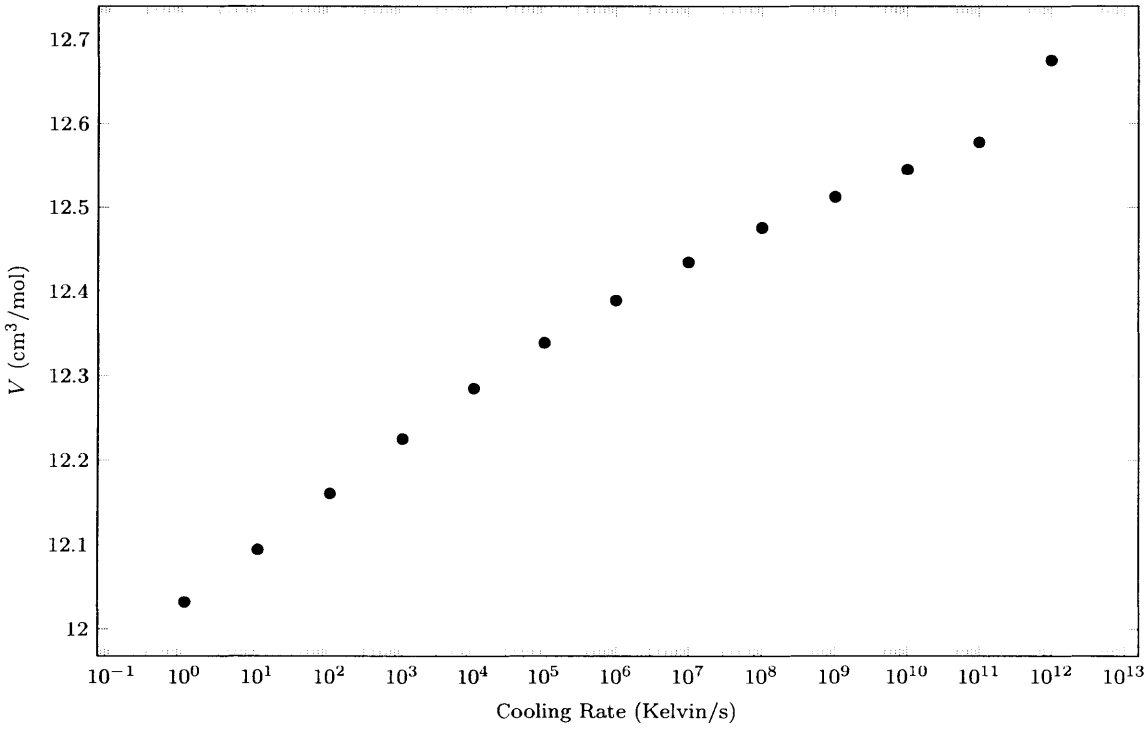


Figure 3-13: Cooling rate vs molar volume at room temperature, computed using KMGEM.

Chapter 4

Mesoscale Modeling: Stiffness Matrix Factor Caching

This chapter was previously published in Computational Mechanics, 2018, under the name "Accelerating coupled finite element-kinetic Monte Carlo models: 200x speedup of shear transformation zone dynamics simulations".

4.1 Introduction

Specialized techniques are available for modeling physical phenomena at the extremes of the length and time scales. At the small/fast end of the spectrum, phenomena involving a few atoms and very fast time scales can often be reproduced using first principles techniques [115, 116]. Ensembles of many more atoms can routinely be simulated on time scales below a millisecond using molecular dynamics [117–119]. At the opposite end of the spectrum, continuum models treat material as a continuous homogenized medium rather than as a granular assembly of atoms [120]; this assumption creates a lower limit on continuum theories' applicable length and time scales, though that limit shifts relative to acceptable error levels. Mesoscale, multiscale, and coupled multiphysics models have proliferated for studying phenomena between or spanning these length and timescale extremes. Mesoscale examples include dislocation dynamics [121–124], phase field models [125–127], and some kinetic Monte Carlo

models [128–131].

This chapter concerns a particular class of mesoscale model that uses kinetic Monte Carlo (kMC) to govern discrete, small-scale, relatively fast deformation events, and the Finite Element Method (FEM) to calculate the interactions between the discrete events and their continuum-level cumulative effect (i.e. a sample's macroscopic shape change). These models are cyclical: the FEM computes the sample's stress field and passes it to kMC; kMC uses that stress field to select a localized "transformation" (e.g. a shear event or phase transformation) which is passed back to the FEM; the FEM then applies that transformation as an eigenstrain [113] and calculates an updated stress field. Because they appeal to the raw deformation mechanism kinetics, these methods are able to capture much more granular detail than would a continuum constitutive law, while avoiding the many atomic vibrations that molecular dynamics so exhaustively simulates. Consequently, these methods have in common an exceptional compromise between simulation fidelity and size (spatially and especially temporally). Prototypical of this class of models is Homer's Shear Transformation Zone Dynamics (STZD) model [94] for deformation of bulk metallic glasses (which will be outlined in the next subsection). Other closely related models (cyclically coupling kinetics and the FEM) include a quantized crystal plasticity model for nanocrystalline materials [132–138] and a kMC model for martensitic phase transformations in shape memory alloys [139]. The Discrete Shear-Transformation-Zone Plasticity model [106, 107] also models metallic glass deformation by cycling between kinetics and elasticity, but uses a hybrid of analytical and FEM calculations in its elastic portion.

The computational scaling of coupled kMC-FEM models is generally dominated by the continuum FEM calculation. The memory consumption and computational time required to evaluate an up-to-date stress field in each step has limited most instantiations of this class of models to two-dimensional approximations, and the few three-dimensional examples in literature (for example, [101, 102]) invariably model very small samples (at most 60 nm in any direction).

To address this shortcoming of the above-described class of models, this manuscript borrows the well-established concept of stiffness matrix factor caching from closely re-

lated modeling techniques. Two particularly relevant examples of reuse of the stiffness matrix decomposition in mesoscale modeling are in discrete dislocation dynamics [140] and coupled atomistic/continuum multiscale models [141]. Despite the historical success of stiffness matrix factor caching in other mesoscale models, this strategy has never before been applied to STZD or its sibling models cited above.

This chapter's Methods section describes stiffness matrix factor caching and shows how it accelerates these models. While these methods apply to the entire class of models described above, the data presented in "Results" focus on STZD as a case study. In anticipation of this, the following subsection provides a brief review of the STZD model; the reader is referred to [105] for a more in-depth presentation. This chapter concludes with a presentation of the largest-ever three-dimensional STZD simulation, which was executed using stiffness matrix factor caching, and which showed an acceleration of nearly 200x over the original approach.

4.1.1 Introduction to the STZD Model

The STZD model is based on Argon's theory of metallic glass deformation [48], which postulates shear transformation zones (STZ), groups of atoms collectively shearing, as the fundamental plastic event. STZD models a sample with a finite element mesh, where the mesh elements coarse-grain the sample's atoms. Clusters of elements (often sharing a common node) constitute potential STZs. The physical size of an STZ therefore bounds the maximum element size of the STZD method's FEM mesh; so the physical sample size that can be simulated by STZD is closely connected to the FEM mesh size that can be handled. That is, simply scaling the FEM mesh size is not an option for reaching longer length scales with STZD.

Each step of the STZD model begins with the sample's stress state, which is calculated by FEM, taking into account the sample's loading and preexisting eigenstrain. The activation rate for each STZ is then estimated using transition state theory [114],

which predicts an Arrhenius-like relation [142]:

$$\dot{s} = \nu_0 \exp\left(-\frac{\Delta F}{k_B T}\right) \int_{g \in G} \exp\left(\frac{\tau(\sigma, g)\gamma_0 \Omega_0}{2 k_B T}\right) \, dg \tag{4.1}$$

where ν_0 is the transition attempt frequency (on the order of the material's Debye frequency), ΔF is a fixed activation energy barrier, G is the set of combinations of shear plane and direction, τ is the shear stress resolved on $g \in G$, γ_0 is the characteristic STZ shear strain, and Ω_0 is STZ volume. The kMC algorithm then stochastically selects a single STZ shear event as the next transition and computes a time step (a "residence time" before the transition). The probability of choosing any STZ shear event is weighted proportionally to its particular rate. To close the cycle, the FEM applies the appropriate eigenstrain (also called "thermal strain" or "initial strain" in FEM literature) to the FEM mesh elements comprising the selected STZ, increments the sample's loading conditions, and computes the updated sample stress field. The STZD cycle then repeats. Gradually the individual STZ activation events cause eigenstrain to accumulate in the FE mesh, resulting in macroscopic plastic deformation of the sample.

The STZD model was originally implemented in two dimensions [94] and then extended to three dimensions [100]. It has been successfully used to simulate shear samples [94], tensile samples [100,103], and single and cyclic nanoindentation [81,100]. It has also been extended to include free volume as an evolving state variable [99,104] and to study metallic glass matrix composite materials [103]. The key papers reporting results from STZD simulations are shown in Table 4.1, along with the dimensionality and the length scales of those simulations. These papers have produced valuable insights into shear band nucleation and structure [101] and metallic glass deformation modes [102], among other phenomena, while relying mostly on two-dimensional approximations. The few three-dimensional samples in the literature never exceeded 60 nm in any direction and typically took weeks on multicore architectures to compute. Comparison to selected micromechanical experiments (cited in Table 4.2; see also [143]) shows a gap between experimentally-relevant length scales and simula-

Table 4.1: Sizes of STZD simulations in literature, with lenth scales and brief descriptions of the simulated loading. The largest simulations reported in this chapter are denoted by an asterisk.

Two-Dimensional STZ Dynamics Simulations			
Ref.	Description	Dimensions (nm)	Volume (nm^3)
[94]	shear	27.6×45.8	
[100]	shear	34.8×57.7	
[81]	nanoindentation	100×35	
[99]	shear	60×120	
[104]	tensile	(not reported)	
[103]	tensile	100×300	
[149]	tensile	50×250	
Three-Dimensional STZ Dynamics Simulations			
Ref.	Description	Dimensions (nm)	Volume (nm^3)
[102]	tensile creep	$10 \times 10 \times 20$	1.57×10^3
	nanoindentation	$30 \times 30 \times 11$	7.78×10^3
[101]	tensile	$20 \times 20 \times 60$	1.88×10^4
*	uniaxial	$60 \times 60 \times 170$	3.46×10^5

Table 4.2: Selected micromechanical experiments on metallic glass from literature, with length scales and brief descriptions of loading.

Metallic Glass Micromechanical Experiments			
Ref.	Description	Dimensions (nm)	Volume (nm^3)
[145]	pillar compression	$70 \times 70 \times 210$	8.08×10^5
[144]	pillar compression	$90 \times 90 \times 360$	2.29×10^6
[146]	pillar bending	$93 \times 93 \times 744$	5.05×10^6
[147]	tensile	$100 \times 100 \times 650$	5.11×10^6
[148]	tensile	$70 \times 70 \times 350$	1.35×10^6
[150]	nanoindentation	depth 50-100	

tion capabilities which prevents side-by-side comparison for calibration, validation, and forward-modeling purposes. This gap is of particular interest in view of the experimentally-observed transition in metallic glass plasticity between 80 nm- and 500 nm-diameter uniaxially loaded samples [143–148]; the technique in this chapter brings STZD much closer to being able to study this transition *in silico*.

4.2 Method

The method to follow uses the FEM in its constituent pieces rather than as a "black box." The reader is referred to the first two chapters of [151] for an in-depth introduction to the FEM, but a high-level overview is provided here for context. The FEM

takes a discretized mesh of a sample and constructs interpolation functions ("shape functions") on the mesh elements. Then, under the postulate that (in the case of elasticity) the displacement field satisfying stress equilibrium can be approximated by a weighted sum of the shape functions, the FEM constructs a linear system:

$$\mathbf{Kd} = \mathbf{F} \tag{4.2}$$

where the unknown vector $\mathbf{d}$ consists of the shape function weights best satisfying the underlying differential equation. The symmetric positive definite matrix $\mathbf{K}$ is termed the "stiffness matrix," and is constructed from the elastic constants of the sample and the mesh shape functions. The vector $\mathbf{F}$ is termed the "force vector," and contains (in addition to the stiffness matrix's ingredients) information on both Dirichlet and Neumann boundary values, body forces, eigenstrains, and eigenstresses. Eqn. (4.2) is often solved by Cholesky decomposition [152–154] of the stiffness matrix $\mathbf{K} = \mathbf{L}\mathbf{L}^T$, followed by solution of $\mathbf{L}\mathbf{L}^T\mathbf{d} = \mathbf{F}$ by forward- and back-substitution.

The STZD algorithm can be framed as a cycle with six steps (as shown in Fig. 4-1(a)). After a brief setup phase, the stiffness matrix $\mathbf{K}$ is constructed using the sample mesh and elastic stiffness tensors, at a computational cost of $O(n)$ where n is the number of mesh nodes. Second, a sparse Cholesky algorithm factors $\mathbf{K} = \mathbf{L}\mathbf{L}^T$; this step is the most computationally expensive, with theoretical $O(n^3)$ complexity, empirically closer to $O(n^2)$ when sparse linear algebra is leveraged. Third, the force vector $\mathbf{F}$ is constructed from the sample loading, body force, and preexisting eigenstrain fields, in $O(n)$ time. Fourth, the system $\mathbf{Kd} = \mathbf{F}$ is solved by forward- and back-substituion, with theoretical $O(n^2)$ complexity, empirically closer to $O(n)$ with sparse computations. Fifth, in $O(n)$ time the displacement field is postprocessed into stress and strain fields for the sample. Sixth and finally, kMC selects the next transition, also in $O(n)$ time. The asymptotic complexities of these steps are listed in Table 4.3.

The transition selected by kMC takes the form of an eigenstrain which is applied to a cluster of elements in the FEM mesh. Under the original algorithm the cycle

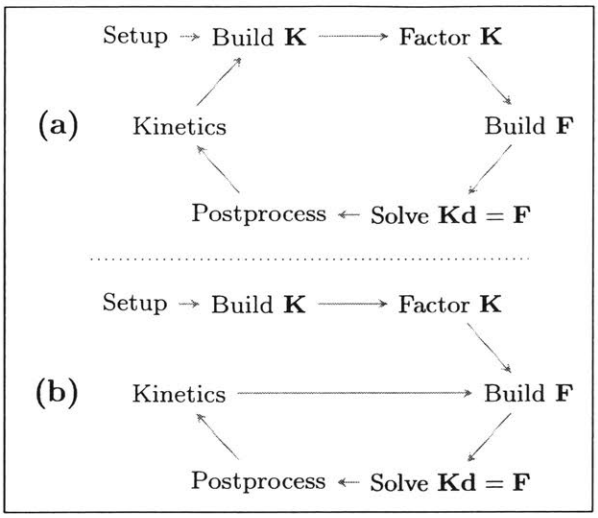


Figure 4-1: (a) kinetic-FEM cycle; (b) kinetic-FEM cycle with shortcut shown

then repeats itself, starting with construction of a new stiffness matrix. However, note that the new stiffness matrix will be identical to the previous one; modifying the boundary values and adding eigenstrain to the model changes neither the sample's elastic constants nor the shape functions. Therefore, construction and factorizaton of $\mathbf{K}$ is completely redundant after the first step.

This suggests a simple innovation: to calculate the FEM stiffness matrix and its factors once as a setup step, and then to cache those stiffness matrix factors in memory. This eliminates the necessity of calculating and factoring $\mathbf{K}$ in each simulation cycle; each cycle simply calculates the new force vector, solves the cached stiffness matrix factors against the new force vector, and then postprocesses the newly calculated displacement field to obtain strain and stress data (see Fig. 4-1(b)). This strategy is termed "stiffness matrix factor caching." It produces precisely the same results as the original approach; the physics are not altered, nor is the numerical approximation. This is simply an adjustment to the code's logical flow to eliminate redundant calculations.

The potential value of this optimization is apparent from the complexities of each part of the STZD algorithm in Table 4.3; by eliminating the need to factor a stiffness matrix with each step, the overall asymptotic complexity of each step is reduced from $O(n^3)$ to $O(n^2)$, assuming dense numerical linear algebra, and by a similar mar-

Table 4.3: Complexity of pieces of the STZD method, assuming dense numerical linear algebra, where n is the number of nodes in the mesh.

Step	Dense Theoretical	Sparse Empirical
Setup	$O(n)$	$O(n)$
Build $\mathbf{K}$	$O(n^2)$	$O(n)$
Factor $\mathbf{K} \rightarrow \mathbf{L}\mathbf{L}^T$	$O(n^3)$	$O(n^2)$
Build $\mathbf{F}$	$O(n)$	$O(n)$
Solve $\mathbf{L}\mathbf{L}^T\mathbf{d} = \mathbf{F}$	$O(n^2)$	$O(n)$
Postprocess	$O(n)$	$O(n)$
Kinetics	$O(n)$	n/a

gin using sparse linear algebra. Of course this approach, while novel in the context of STZD modeling (and of the other closely related models mentioned in the introduction), is a straightforward application of well-established ideas within mesoscale modeling [140, 141]; also, commercial FEM packages routinely reuse stiffness matrix factors in time-series calculations. More broadly, reusing matrix factors or inverses is a standard practice in algorithms for fields as diverse as optimization and image processing.

4.2.1 Implementation Details

For this study two STZD codes were constructed, one of which follows the conventional algorithm in Fig. 4-1(a) and one of which leverages stiffness matrix factor caching as in Fig. 4-1(b), but both of which are otherwise as similar as possible. Both codes are composed in C++11, and in lieu of a commercial FEM solver both codes use a simple in-house FEM library which takes advantage of the Eigen3 matrix library [155] and the Cholmod sparse linear system solver [156]. Both codes were compiled using the Intel compiler, linked against a single-threaded version of Intel's MKL library, and were run in serial on an Intel Xeon processor clocked at 2.6 GHz in a workstation with 128 Gb of memory. Both codes are instrumented to report the timing breakdown between parts of the STZD cycle to enable more granular comparison between the algorithms. The simulation input is in the form of an .ini file, and the output uses the HDF5 file format [157]. The code that does not use stiffness matrix caching performs similarly to commercial linear FEM implementations in serial execution mode.

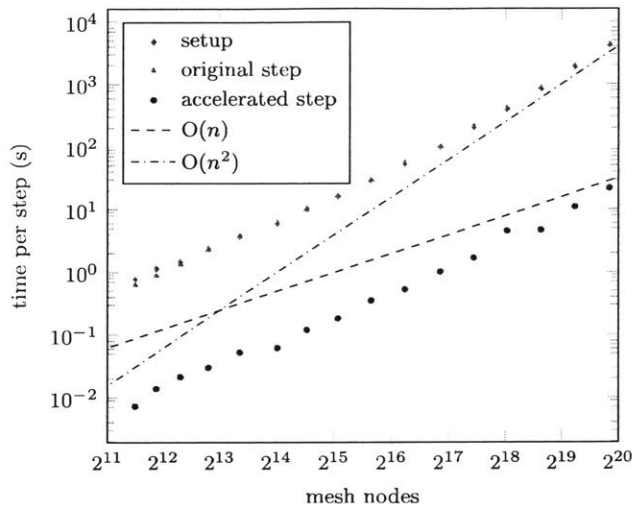


Figure 4-2: Time required to execute STZD code described in the text, as a function of FEM mesh size.

4.3 Results

To examine the scaling of the STZD algorithm with respect to mesh size, wall-clock times were averaged over 10 STZD steps for meshes with between 1606 and 938407 nodes; the timings are plotted on a log-log axis in Fig. 4-2. It is evident that the STZD code using stiffness matrix factor caching is empirically faster than the original approach, with a speedup of 196x for the largest meshes studied for this chapter. The observed deviation from dense matrix asymptotic behavior is due to extensive use of sparse numerical linear algebra.

The effectiveness of caching stiffness matrix factors is further illustrated by fractionally breaking the execution time of an original STZD step into pieces in Fig. 4-3. The optimization described in this chapter eliminates the striped regions of that plot (corresponding to building and factoring the FEM stiffness matrix), cutting 98-99.5% of the computation per STZD step.

To concretely illustrate the utility of this technique the next subsection contains a series of simulated uniaxial tensile and compression tests on the largest-ever STZD samples.

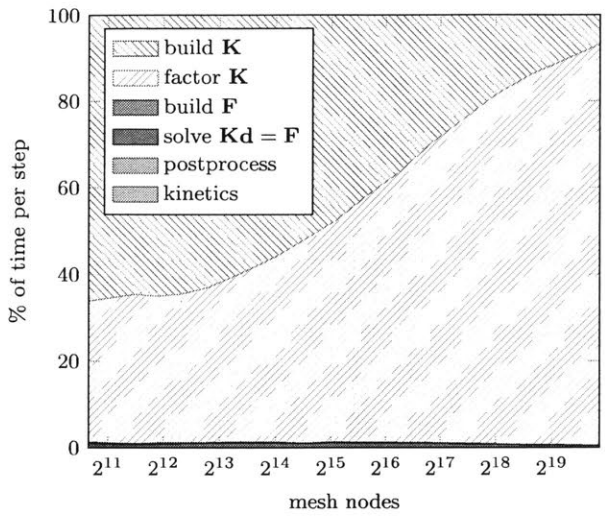


Figure 4-3: Fraction of time spent on each part of a step of the STZD algorithm described in the text, as a function of FEM mesh size.

4.3.1 Uniaxial Tensile & Compression Tests

This section describes uniaxial tests on nanoscale cylindrical samples with gauge diameters from 10 nm to 50 nm and gauge lengths from 30 nm to 150 nm; the geometry of these samples is drawn in Fig. 4-4(a). The relative sizes of this chapter's samples in comparison to three-dimensional STZD samples in literature are shown in Fig. 4-5. Each simulation was run for a number of steps proportional to the volume of the sample, to ensure roughly equal amounts of plastic deformation between the simulations. These simulations were run under the conditions described in "Implementation Details" above; in particular, they were run in serial fashion. A selection of the simulations are plotted in Figures 4-6 through 4-9, and the remainder are included in supplementary material to this article; in these plots, STZs are plotted as small dots, with the size and color of the dot corresponding to the norm of the cumulative STZ strain. The colorbar for STZs is shown in Fig. 4-4(b). The STZD parameters for all the simulations are given in Table 4.4.

The most notable feature of the $\varnothing$=10 nm compression sample in Fig. 4-6 is its runtime of less than five minutes. This is a dramatic improvement on the original approach, which would have taken at least a day to run a comparable simulation on multiple cores. This suggests that stiffness matrix caching will enable simulation of

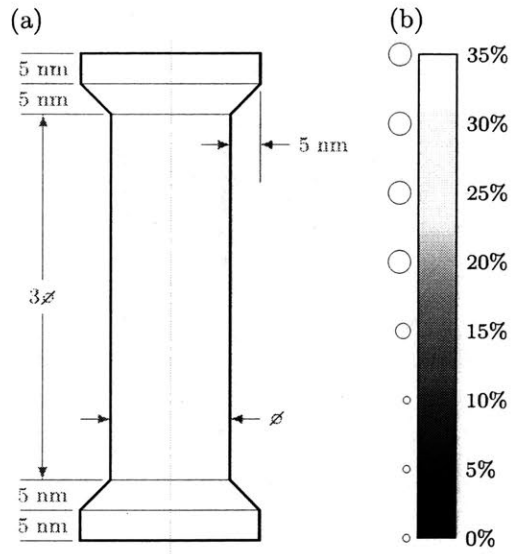


Figure 4-4: Legend to the STZD simulation Figures 4-6 through 4-9. Part (a) shows the dimensions of the uniaxial samples in terms of the parameter $\varnothing$, the diameter of the gauge portion of the sample. Part (b) maps the norm of STZ strain to color and dot size. The dot sizes shown here are scaled much larger than those in the figures to follow, but are proportionally correct relative to each other.

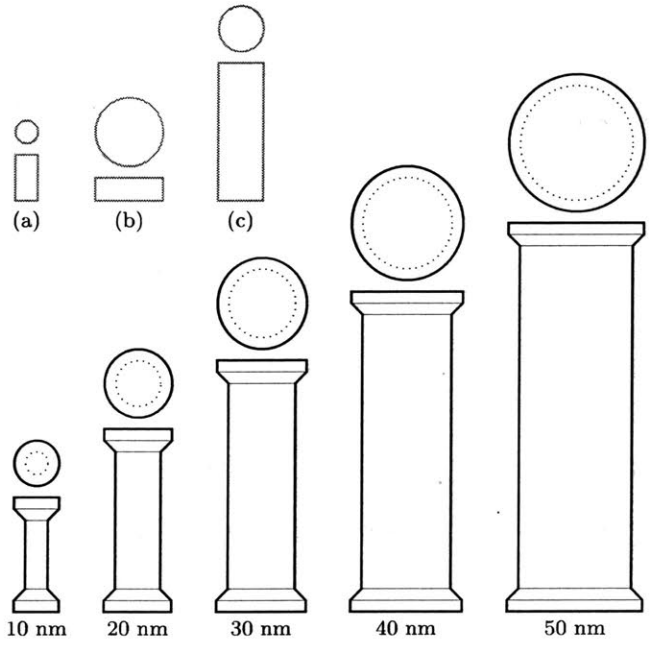


Figure 4-5: Relative sizes of three-dimensional STZD simulations in literature and this chapter. In the top-left corner are the three largest three-dimensional STZD simulations from literature, with (a) and (b) from [102], and (c) from [101]. Along the bottom are the various samples reported in this chapter with their respective gauge section diameters ($\varnothing$).

Table 4.4: Parameters to all of the STZD simulations appearing in this chapter.

Parameter	Value	Units
ν_0	6.814×10^{12}	s^{-1}
ΔF	2.5×10^{-19}	J
Γ_0	0.1	m/m
T	300	K
G	37	GPa
ν	0.352	
$\dot{\epsilon}_{zz}$	± 1.0	m/(m·s)
$\delta t_{\max}$	0.001	s

large ensembles of small samples for statistical analysis; this is of particular value because these simulations are stochastic in nature, so analysis of any one simulation might not be representative of the ensemble.

Of course, stiffness matrix factor caching could also enable use of a finer FEM mesh on these small samples. This has been shown to not be a particular issue in STZ Dynamics (assuming that the mesh size is an appropriate fraction of the material's characteristic STZ volume, as is the case in these simulations), but may be useful as kMC-FEM models are extended to new materials systems in the future.

Moving up to the $\varnothing$=30 nm tensile sample in Fig. 4-7, which is already larger than any previously published STZD sample, nucleation of orthogonal competing shear bands is observed. The interaction between the shear bands apparently obstructs both of them from crossing the full diameter of the gauge section. This behavior has implications for understanding shear band nucleation and growth, and can only be observed in samples large enough to sustain multiple instances of shear localization. This issue will be thoroughly explored in future articles.

Looking closely at the $\varnothing$=40 nm tensile sample in Fig. 4-8, one can observe periodic "waves" in the STZ strain field perpendicular to and along the length of the main shear band. Interestingly, these appear very early in the shear band nucleation process (they are visible as early as step 23930 of the simulation). The wavelength of these oscillations (between 10 and 15 nm) is such that they would be impossible to observe in the smaller STZD samples published to date.

The $\varnothing$=50 nm compression sample in Fig. 4-9 shows nucleation of four shear bands along orthogonal planes, but one of the shear bands ultimately dominates the

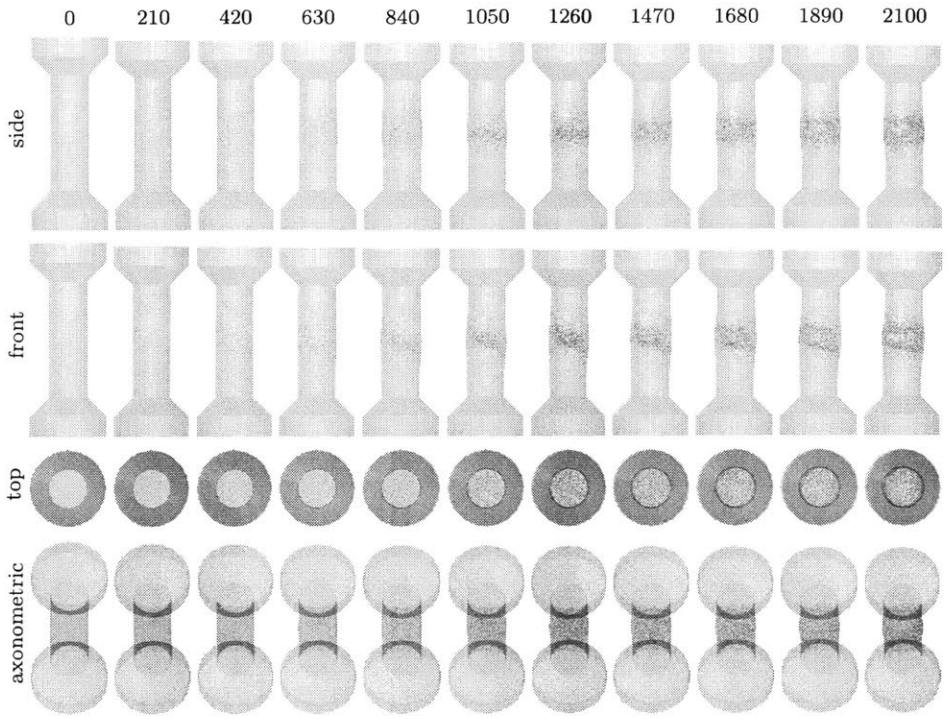


Figure 4-6: STZD compression test of $\varnothing$=10 nm sample. Numbers along the top are KMC steps. This simulation took less than five minutes to run on the machine described in the "Implementation Details" section of the text.

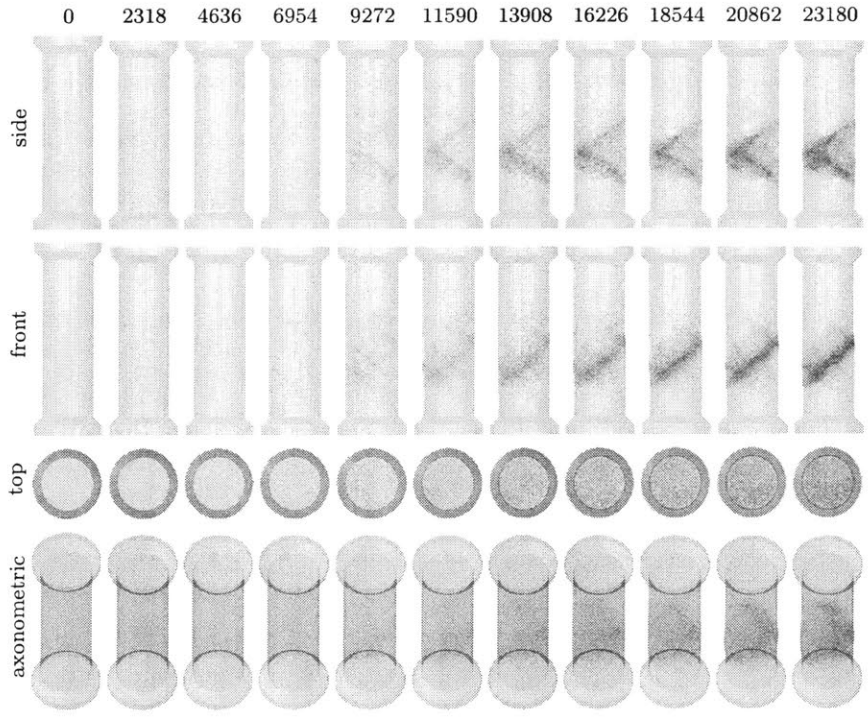


Figure 4-7: STZD compression test of $\varnothing$=30 nm sample. Numbers along the top are KMC steps. Execution time: 13 hours.

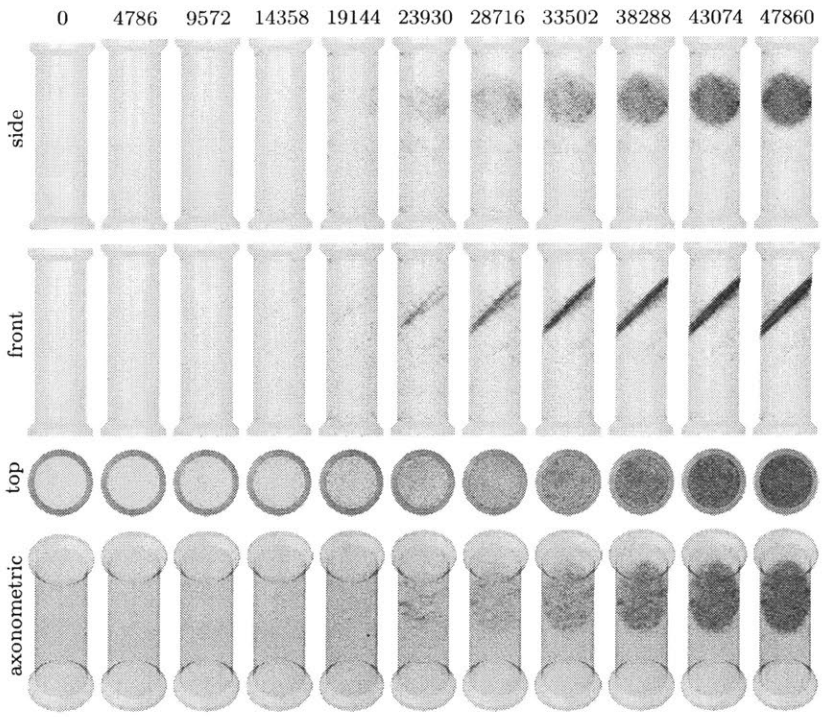


Figure 4-8: STZD tension test of $\varnothing$=40 nm sample. Numbers along the top are KMC steps. Execution time: 3 days.

others and propagates across the diameter of the sample (unlike Fig. 4-7). It is, however, apparent from the axonometric view that the dominant shear band is impeded by its orthogonal competitors. Again, a smaller STZD sample would be unable to sustain these multiple shear localization instances. This sample and the $\varnothing$=50 nm tension sample in the supplementary material are the largest STZD simulations run to date, exceeding the volume of the previous maximum by a factor of eighteen; these simulations' dimensions approach the scale of physical nanomechanical experiments currently in the literature (as in Table 4.2).

It is worth noting here that stiffness matrix factor caching does not negate the necessity of remeshing when plastic deformation to the sample invalidates the linear expansion underpinning the FEM. The FEM stiffness matrix will need to be reconstructed and factored after each remeshing. However, in the context of STZD, remeshing events should be spaced many STZD steps apart, so the speed gains described above remain representative of expected performance even with remeshing.

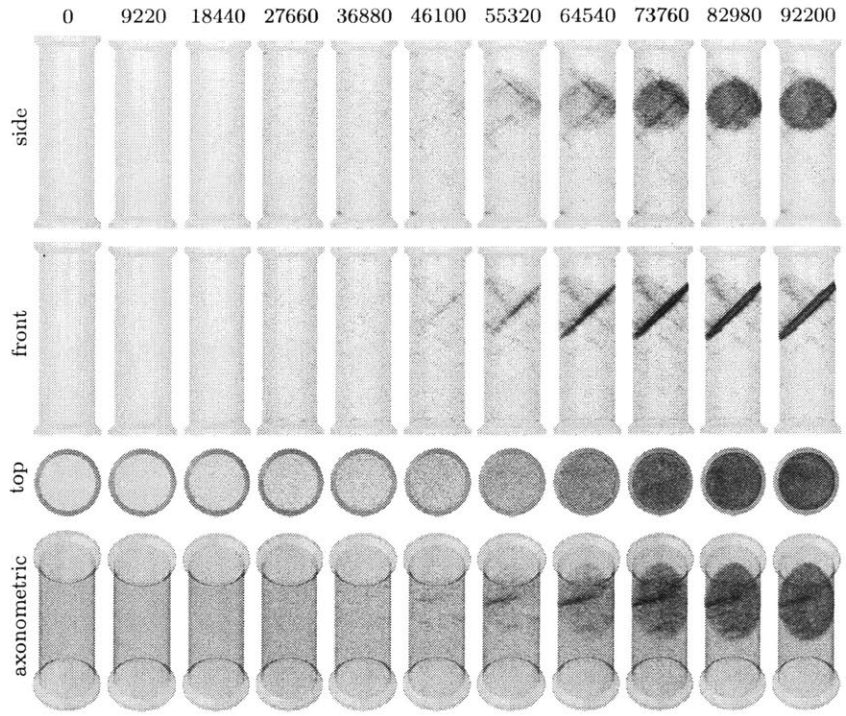


Figure 4-9: STZD compression test of $\varnothing$=50 nm sample. Numbers along the top are KMC steps. Execution time: 19 days.

4.4 Conclusions

Simulation methods that iteratively link kinetics with localized updates in the FEM, typified by the STZ Dynamics method, have suffered from long run times due to superlinear scaling of the FEM with mesh size. However, if these methods do not require modifications to the mesh or the elastic properties of the sample from step to step, as is the case in STZD, then the FEM stiffness matrix is also unchanged from step to step. This enables an acceleration strategy: to calculate the FEM stiffness matrix, factor it, and cache the factorization in an initialization step, and then to use and reuse the factorization in each step. This is termed "stiffness matrix factor caching." While reuse of stiffness matrix factors is a common practice in mesoscale modeling, it has never before been applied to STZD and closely-related methods. Stiffness matrix caching constitutes an asymptotic improvement and empirically has produced a speedup of nearly 200x over the original method. This speedup is useful in two respects: it enables simulation of large numbers of small samples to form

an ensemble, and it enables simulation of samples on experimentally-relevant length scales in three dimensions. These simulations of larger samples exhibit multiple (often competing) shear bands, sometimes in apparently periodic arrangements. Future work looks to directly compare real nanomechanical experiments to these large STZD simulations for validation purposes, or to illuminate avenues for improvement of the STZD model's physics. The results in this chapter are readily extensible to similarly-designed methods in both two and three dimensions; it is hoped that stiffness matrix caching will make three dimensional simulation the norm rather than the exception for STZD and its sibling methods, and that studies comparing these simulations to physical nanomechanical experiments will be forthcoming.

Chapter 5

Towards Mesoscale KMGEM

KMGEM is designed to be implemented in three contexts: a homogeneous statistical-mechanical model, a continuous finite-element model, and a mesoscale discrete shear transformation model. This chapter discusses preliminary and continuing work towards constructing the mesoscale model.

KMGEM Mesoscale must overcome three major hurdles. The first is that an evolving shear modulus precludes use of the stiffness matrix caching approach described in Chapter 4. This could be partially overcome by use of sparse Cholesky update algorithms, but this chapter will suggest a better way forward. The second issue is that of memory-boundedness for very large samples; the samples in the previous chapter maxed out the author's available computing resources, and while turning to ever more expensive computing platforms is an option, an algorithmic improvement seems desirable. The third issue has more to do with the physics of KMGEM itself: the structural relaxation events occur much more frequently than shear transformations in the course of a simulation, so a naive kMC implementation would see the vast majority of steps produce no plastic strain. Put another way, the ordinary differential equation describing plastic deformation and structural evolution of metallic glass is inherently stiff.

5.1 Algorithmic Improvements: Geometric Multigrid

Geometric multigrid is a method for solving partial differential equations; a description of this method falls outside of the scope of this thesis, but excellent introductions to the subject are available [158, 159]. It suffices here to say that by iteratively solving the PDE problem using multiple discretizations, each with a different level of coarseness, one may obtain a solution in linear time with minimal memory usage compared to a Cholesky decomposition. The memory savings occur because multigrid operates directly on the finite element stiffness matrices without ever factoring them; conversely, even a skillfully permuted Cholesky decomposition of a finite element stiffness matrix will have significant fill-in, resulting in a memory footprint several times larger than that of the original matrix.

Geometric multigrid also extends readily to parallel architectures [160]. The core operation in geometric multigrid is matrix-vector multiplication; each row of the resulting vector can be computed independently of the other rows, making this operation relatively easy to parallelize with a minimum of inter-node communication. This is compared to direct matrix factorization, in which each row depends on the previously processed rows; so parallel implementations of direct matrix factorization generally are not able to avoid a large amount of communication between nodes.

In addition to linear complexity, small memory footprint, and ready parallelization, geometric multigrid is exceptionally easy to update when a local region of a mesh undergoes a change in elastic properties. Here we assume a basic grasp of geometric multigrid—see the treatments of the subject cited above; there are also outstanding notes on geometric multigrid freely available on the internet.

The finite element stiffness matrix $\boldsymbol{K}^h$ (where h denotes the coarseness of the mesh) is restricted to the coarse meshes by interpolation/restriction matrices $\boldsymbol{I}$:

$$\boldsymbol{K}^{2h} = \left(\boldsymbol{I}_h^{2h}\right)^{\mathrm{T}} \boldsymbol{K}^h \boldsymbol{I}_h^{2h} \tag{5.1}$$

The compact support property of the finite element method ensures that $\boldsymbol{K}^h$ is sparse; similarly, the compact support of elements of the coarser grids ensures that $\boldsymbol{I}$ is

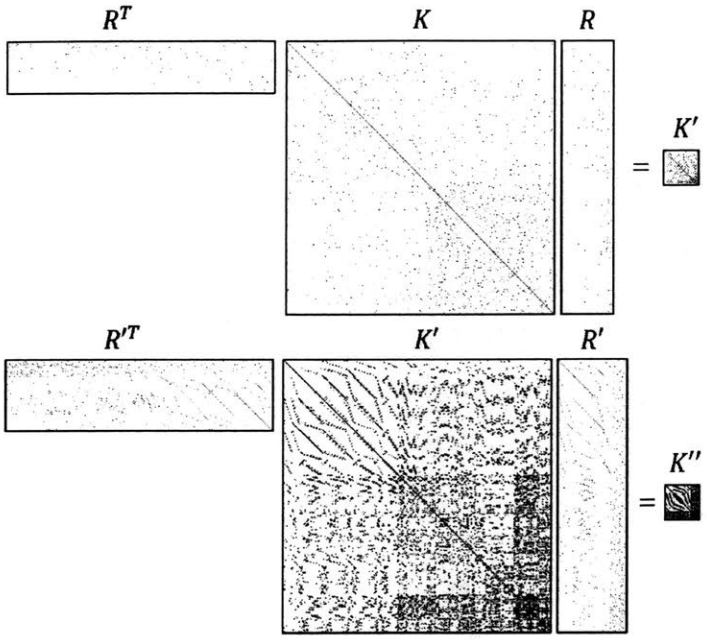


Figure 5-1: Schematic illustration of Eqn. (5.1), showing sparsity at multiple levels of geometric multigrid. Note that here $\boldsymbol{R}$ is acting as the interpolation matrix; this differs in notation from the text.

also sparse. Consequently, the coarsened stiffness matrices are also sparse. This is schematically illustrated in Figure 5-1.

Now, suppose that one element of the fine mesh experiences a change in its elastic properties (e.g. it undergoes a structural relaxation event). Compact support ensures that only on the order of tens to a hundred entries of the finest stiffness matrix will be correspondingly altered (denote the update as $\Delta \boldsymbol{K}$). Construction of updated coarsened stiffness matrices is straightforward:

$$\boldsymbol{K}^{2h} + \Delta \boldsymbol{K}^{2h} = \left(\boldsymbol{I}_h^{2h}\right)^{\mathrm{T}} \left(\boldsymbol{K}^h + \Delta \boldsymbol{K}\right) \boldsymbol{I}_h^{2h} \tag{5.2}$$

and by a similar argument to above, the update to the coarsened stiffness matrix remains sparse. This is schematically illustrated in Figure 5-2.

The upshot to all of this is that a local update in the finite element model corresponds to a similarly local update to the machinery of the geometric multigrid method; this update process has constant complexity with respect to system size. Returning to the three issues described at the outset of this chapter, the geometric

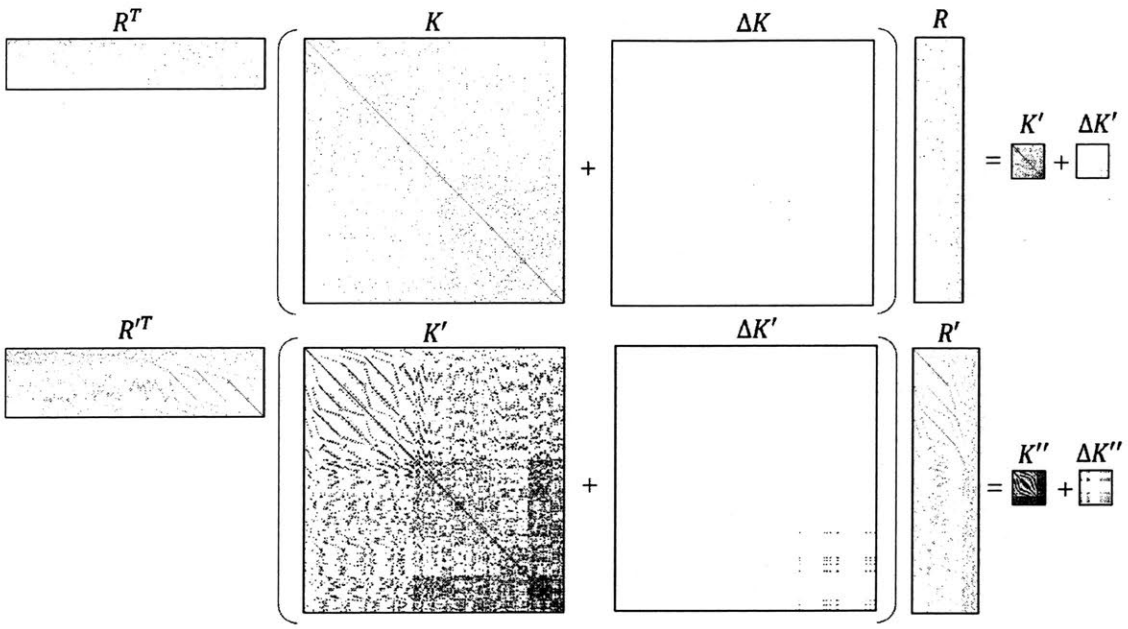


Figure 5-2: Schematic illustration of Eqn. (5.2), showing sparsity at multiple levels of geometric multigrid preserved through a local update. Again, the notation here differs from the text.

multigrid method resolves both the memory-boundedness issue and the update time issues.

5.1.1 Implementation Notes

A rudimentary implementation of the geometric multigrid method is included in the code appended to this thesis. The strategy pursued therein constructs the heirarchy of grids over an unstructured tetrahedral mesh. Each coarse mesh is made up of cuboidal elements, making them far easier to interpolate than an unstructured mesh.

The implementation included with this thesis uses Gauss-Seidel as its relaxation method; in practice, in a distributed-memory computing environment, block Gauss-Seidel or even Jacobi iteration would be preferable.

Since the implementation here is far from optimal (it is very much a work-in-progress), I will defer reporting of its time and memory performance to a later publication. I can, however, report that it does successfully solve the elasticity equations underpinning KMGEM Mesoscale.

5.2 Hybrid Kinetic Monte Carlo

Since the energy barriers associated with structural relaxation are lower than those associated with shear transformations, relaxation events would be expected to occur much more frequently than shear transformations. If relaxation events were naively included in the kinetic Monte Carlo rate catalog, the vast majority of kMC steps would be random thermal fluctuations. This would make the simulation of mechanical processes unacceptably slow.

To deal with this, each element of the finite element model is given a structural distribution (as in the homogeneous KMGEM described in Chapter 3). The structural distribution determines the instantaneous elastic properties of the element. Kinetic Monte Carlo is then used to select shear transformation events and compute residence times, with rates computed using KMGEM. Once a residence time is computed, the corresponding backwards Euler state relaxation matrix is computed and applied to the structural distribution for each element in the sample. Each shear transformation event also locally alters the structural distribution (and, consequently, the elastic properties) in the affected elements. The corresponding update to the elastic solution is conducted through the mechanisms of the Geometric Multigrid method described above. With this strategy, kMC steps only correspond to mechanically important events (shear transformations), but the simulation continues to model structural evolution.

5.3 Looking Forward

By hybridizing the kinetic Monte Carlo portion of KMGEM Mesoscale with a classical ordinary differential equation integrator, we are able to deal with the inherent numerical stiffness of the metallic glass structural evolution process. This enables us to only take kMC steps for mechanically significant events—shear transformations—while still retaining much information about the underlying structural evolution of the glass. Coupling this to Geometric Multigrid will resolve many of the remaining

memory- and time-related issues related to mesoscale simulation of metallic glass, and will open doors to parallel computing on distributed architectures. It may also become desirable to re-implement STZ Dynamics with Geometric Multigrid in lieu of direct matrix factorization.

Chapter 6

Summary

Metallic glass is isotropic and homogeneous on the macroscale, but on the scale of tens of angstroms it is locally anisotropic and heterogeneous. The mechanical response of the glass (elastic and plastic) depends strongly on this structure, with the kinetic events mediating plasticity localizing to the more disordered/softer regions. Structural rearrangements of just a few atoms are thermally activated, enabling manipulation of the glassy structure via heat treatment. Shear transformations occur on larger scales and accommodate strain; they are stress-biased and optionally thermally activated. These events also alter the structure of the glass, tending to inject excess volume and energy into the glass.

This thesis presents a model addressing both the structural evolution and mechanical deformation processes in metallic glass, called the Kinetic Metallic Glass Evolution Model (KMGEM). This model consists of a potential energy landscape which is traversed by way of idealized relaxation and shear transformation events, whose rates are computed using transition state theory. The potential energy landscape is expressed in terms of two structural state variables which correspond to the dilatation and the rigidity of the glass. By explicitly including dilatation in the shear transformation rate equations this model incorporates tunable tension-compression asymmetry, which is shown to fit atomistic data previously in literature.

The KMGEM is first implemented in a homogeneous statistical sense, where the glassy structure is represented as a distribution over discrete levels in structural state

space. The KMGEM rate equations provide a numerically convenient way of computing the time-domain evolution of the glass under dynamic temperature and stress conditions. The model is able to roughly replicate experiments in the literature varying the cooling rate of the glass, predicting roughly logarithmic variation of shear modulus, volume, and configurational potential energy with cooling rate.

In anticipation of implementing the KMGEM in a mesoscale format, we then shift our attention to numerical considerations in discrete shear transformation zone dynamics (STZD) models. In models where the elastic properties of the sample are not altered by shear transformation, the finite element stiffness matrices are identical from step to step. By eliminating this redundancy from the calculations these models accelerate by a factor of 200x. This enables STZD simulations on a record-breaking length scale.

Finally we consider implementation of mesoscale KMGEM using a hybrid kinetic Monte Carlo method underpinned by the Geometric Multigrid method. A preliminary codebase for this is provided, along with a discussion of implementation details and concerns surrounding the method. It is expected that this approach will ultimately enable full spatiotemporal parity with nanomechanical experiments.

Chapter 7

Directions for Future Work

This section breaks future work into three categories: uses of KMGEM, improvements for KMGEM, and broader suggestions to move the field's fundamental understanding of metallic glass forward.

7.1 Using KMGEM

The homogeneous implementation of KMGEM should readily extend to mechanical experiments in the homogeneous flow regime. This will enable studying the effect of strain rate on steady-state stress, stress relaxation, and viscosity at temperature. This document already describes the machinery necessary to simulate these experiments, and they are excluded only due to time constraints.

The mesoscale implementation of KMGEM is designed to facilitate large-scale simulations of heterogeneous deformation in metallic glass nanosamples. Examples on the list include nanoindentation, nanotensile tests, nanopillar compression and bending tests, and cyclic loading experiments. It is hoped that KMGEM on distributed-memory computers will be capable of achieving spatiotemporal parity with these actual physical nanomechanical experiments, which should highlight the model's deficiencies and also provide insight into the deformation mechanisms in the experiments.

KMGEM mesoscale also lends itself to the currently-popular practice of correlating shear localization to the preexisting heterogeneous structure; also, studies examining

how the "tail" of the structural distribution (that is, the regions that are most damaged or "fluid-like") evolves in heterogeneously deformed glass may speak to crack nucleation and failure in metallic glass samples.

Also, the principles described previously may be applicable to other amorphous materials; this document focused on metallic glass only to restrict its scope (particularly with respect to the literature review).

7.2 Improving on KMGEM

As was just mentioned, KMGEM mesoscale is designed to scale well in a distributed-memory parallel computing environment, so it should be a relatively simple matter to extend its implementation in that direction. Initial efforts have found the Trilinos package [161] to be useful for this purpose.

Looking forward, two possible improvements to KMGEM would be to incorporate thermal dissipation (that is, that shear transformations produce a local temperature spike which then dissipates out) and inertial effects (switching from static to dynamic elasticity). Related to relaxing the static elasticity assumption would be to make shear transformations occur over a finite period of time (as in [106, 107]).

7.3 Broader Suggestions

Composing this thesis revealed substantial holes in the literature surrounding the structure of metallic glasses. For example, while Ma and co-workers have done inspiring work correlating atomic structure to properties and kinetics [6], there remains much to be done enumerating the structural units of metallic glass and correlating them to configurational entropies and energies. Corresponding constraints on valid distributions of those structural units are also needed. Given this information, one could construct relaxation evolution equations similar to those appearing in this document.

A catalog of structural units would also enable systematic study of the evolution

of metallic glass structure due to shear transformation events.

Finally, the tensorial stress dependence of shear transformation zone barrier height is badly understood, but could be easily explored using molecular dynamics and nudged elastic band methods. This would produce a transition barrier model to be plugged into shear transformation zone dynamics and KMGEM mesoscale models.

Appendix A

Poincare-Steklov Method

This chapter was previously published in the Journal of Computational Physics, 2017, under the name "Fast finite element calculation of effective conductivity of random continuum microstructures: The recursive Poincaré-Steklov operator method".

A.1 Introduction

Homogenization is extrapolation of macro-scale properties of a composite from its microstructure—both the character and the spatial arrangement of its constituent phases. This paper deals with homogenization of random composites, focusing on properties related to transport of charge, diffusion of species, or conduction of heat. These phenomena are governed by flux laws of the type [162]:

$$\vec{J} = -\gamma_{\text{local}} \nabla \phi \tag{A.1}$$

where γ is a tensor which stands in for local electrical conductivity, diffusivity, or heat conductivity, and ϕ stands in for electrical potential, solute concentration, or temperature, depending on the context. This paper will, without loss of generality, use the language of electrostatics in referring to these quantities. At steady-state, the

divergence of the flux field vanishes:

$$0 = \nabla \cdot (\gamma_{\text{local}} \nabla \phi) \tag{A.2}$$

for ϕ subject to prescribed boundary conditions [163,164]. In a composite where there is contrast between the respective γ values of constituent phases, it is desirable to determine an *effective conductivity tensor* γ_{eff} for the homogenized system, which (at steady-state) satisfies:

$$\langle \vec{J} \rangle = -\gamma_{\text{eff}} \langle \nabla \phi \rangle \tag{A.3}$$

where $\langle \cdot \rangle$ is a volume average [165]. In the limit of averaging over sufficiently large volumes of the composite microstructure, γ_{eff} converges to a single tensor representative of the composite as a whole, accounting for both the constituents' respective γ_{local} tensors and their spatial arrangement.

There are mathematically elegant ways to access the effective properties of simple microstructures [165–171]; however, for more arbitrary composite microstructures a straightforward approach to determining γ_{eff} is to:

1. generate explicit finite realizations of the microstructure
2. use numerical partial differential equation techniques to solve Eqn. (A.2) under a variety of boundary conditions
3. calculate $\langle \vec{J} \rangle$ and $\langle \nabla \phi \rangle$ for each instance
4. solve Eqn. (A.3) for γ_{eff} in a least-squares sense.

There are two broad classes of numerical methods for solving Eqn. (A.2) over composite microstructures with sharp boundaries between constituent phases: integral methods and finite element methods (FEMs).

Integral methods can be elegant, accurate, and fast, can extend to arbitrarily many dimensions, and can be specialized to periodic microstructures [169–171]. These methods use Green's identities to transform Eqn. (A.2) from a volumetric partial differential equation to a surface integral equation [172,173] where the relevant integral

is over all the boundaries between phases in the composite. Unfortunately, evaluating this integral is a challenging proposition for microstructures not having obvious discretizations.

The finite element method is an alternative to integral methods for solving Eqn. (A.2). This method is extremely flexible and well-studied [151,174–179], with wide application in solving partial differential equations of many kinds. Using the finite element method to solve Eqn. (A.2) is straightforward. The technique is also tolerant of arbitrary random composite microstructures; this in addition to its flexibility make the finite element method the technique of interest in this paper.

Many approaches to solve the finite element method in nearly linear time have been developed. The main idea in most of these methods is to divide the finite element mesh into submeshes with continuity equations between them; the individual submeshes can be very efficiently analyzed independently of each other, so the speed of these methods hinges on rapidly solving the continuity equations [180–182]. Such approaches are categorized as multigrid, domain decomposition, or reduction to the interface methods [183–186].

This paper describes a numerical technique for computing the effective conductivity of a block of composite. The new method is based on the same equations used in the finite element method, and produces the same results, but with much less computational time and memory consumption. This is not only efficient, but also provides more accurate homogenized calculations by increasing the size of the finite element meshes that can be analyzed for given computer hardware.

This effort is motivated by an attempt to numerically compute the effective conductivity of continuum composite microstructures with phase fractions near the percolation threshold (a structural phase transition characterized by global connectivity of the reinforcing phase at phase fractions above the percolation threshold and only short-range connectivity below). When Eqn. A.2 is solved over a high-contrast random composite near its percolation threshold, the resulting solution often has a somewhat fractal nature [187,188] with features on both long and short length-scales (see Figure A-1). Consequently, when using the finite element method for such cases

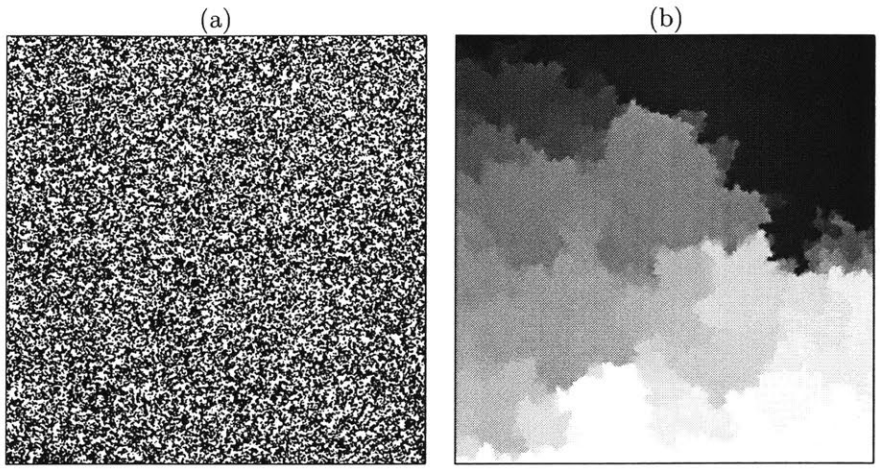


Figure A-1: (a): a two-dimensional composite microstructure at its percolation threshold. (b): solution to Eqn. (A.2) over the microstructure in (a) with a boundary condition of $\phi = 0$ at the top and $\phi = 1$ at the bottom, and isotropic phase γ values of 1 and 10^6 for the dark and light phases, respectively. Note fractal-like features in the solution on a wide range of length scales.

both the composite block size and the sampling density must be higher than would be necessary for composites away from the percolation threshold; the resulting finite element mesh is very large. Ultimately, the traditional finite element method is prohibitively slow and memory-intensive for some microstructures near the percolation threshold. The present development of a faster, less memory-intensive method for computing the effective conductivity of a composite speaks to this need.

A.2 Method

This section is built on a very simple first-order linear finite element framework (as found in the first three chapters of [151]) but it is readily extensible to higher-order finite element spaces. The figures in this section show a regular two-dimensional grid of square finite elements; however, the equations given here are general with respect to dimensionality and are immediately applicable to arbitrary finite element meshes.

This section is broken into four subsections. In the first subsection, a mathematical object called the Poincaré-Steklov Operator (PSO) is reviewed and a method for representing and calculating the PSO for a finite element mesh is presented. In the

second subsection, a case is considered where two adjacent blocks of finite element mesh respectively have known PSOs, and a method is presented for combining the two respective PSOs into a single PSO for the merged mesh. In the third subsection, an algorithm is presented to calculate the PSO for a large finite element mesh. This algorithm exploits the methods in the first two subsections recursively to achieve both memory and time efficiency. Finally, the fourth subsection deals with the boundary conditions to extract γ_{eff} from the PSO. Put together, these pieces comprise what we shall term the "Recursive Poincaré-Steklov Operator Method" (RPSOM).

A.2.1 PSO Representation & Calculation

One of the properties of Eqn. A.2 is the existence of a function that maps every possible Dirichlet boundary condition to a corresponding Neumann boundary condition. That is, given a domain and a γ_{local} distribution across it, it is possible to construct an operator that takes the boundary ϕ values and returns the corresponding boundary fluxes that would be observed upon solving Eqn. A.2 with those boundaries. This is the (Dirichlet-to-Neumann) PSO [189, 190].

The PSO can be approximated in a discrete setting using the Finite Element Method. Consider a finite element mesh with only Dirichlet boundaries (see Figure A-2). Each element has a local γ_{local} tensor assigned to it. The elements containing boundary nodes are called *outer elements*. Let the values of boundary nodes be designated $\vec{y}$. Let the values of the non-boundary nodes in outer elements be designated $\vec{x}$. Let the values of all other nodes in the mesh—nodes not belonging to an outer element—be designated $\vec{z}$.

Now suppose that one wants to calculate the flux crossing the boundary between nodes c and d in the detail of Figure A-2. The gradient of the potential field in element *abcd* can in this case be approximated using finite differences:

$$\nabla\phi \approx \left(\frac{(c+d)-(a+b)}{2\lambda}, \frac{(a+c)-(b+d)}{2\lambda} \right) \tag{A.4}$$

where λ is the element side length. The flux passing through the element can then

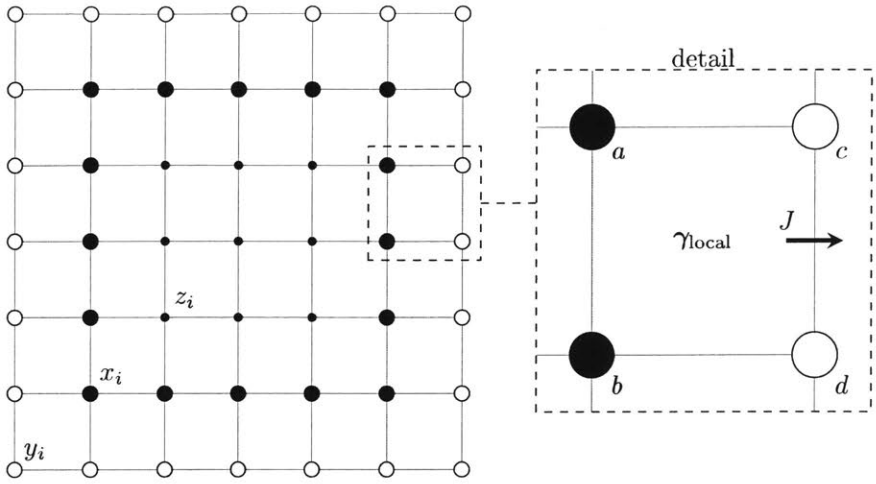


Figure A-2: The elements in this mesh are partitioned into outer elements (lightly shaded) and interior elements. The nodes are partitioned into three sets: the boundary nodes have values $\vec{y}$, the set of interior nodes in outer elements have values $\vec{x}$, and the remaining interior nodes have values $\vec{z}$.

be approximated:

$$\vec{J} \approx -\gamma_{\text{local}} \nabla \phi \tag{A.5}$$

from which the flux component crossing the boundary is readily obtained.

While the details of computing the flux across the boundary may vary depending on the finite element mesh in question (the square grid in Figure A-2 is particularly simple), there is a key characteristic that will remain general: the computation required only γ_{local} and nodal potential values from outer elements. That is, to compute the boundary flux for the entire mesh, all that need be known are $\vec{x}$, $\vec{y}$, and γ_{local} from outer elements.

Consequently, if γ_{local} is known for the outer elements of a mesh, the PSO can be represented as simply a function mapping $\vec{y}$ (which, recall, represents the Dirichlet boundary condition) to $\vec{x}$. This function can be represented as a matrix $\boldsymbol{P}$ such that $\boldsymbol{P}\vec{y} = \vec{x}$. Each of the $n+p$ interior nodes corresponds (by the finite element method) to a linear equation linking the $n+p$ values of the interior nodes to the m boundary

node values; these equations can be compiled into a block matrix equation [151]:

$$\begin{bmatrix} \boldsymbol{K}_{xx} & \boldsymbol{K}_{xz} \\ \boldsymbol{K}_{zx} & \boldsymbol{K}_{zz} \end{bmatrix} \cdot \begin{bmatrix} \vec{x} \\ \vec{z} \end{bmatrix} = \begin{bmatrix} \boldsymbol{F}_{xy} \\ \boldsymbol{F}_{zy} \end{bmatrix} \cdot \vec{y} \tag{A.6}$$

where $\boldsymbol{K}$ and $(\boldsymbol{F}\vec{y})$ are respectively the familiar stiffness matrix and force vector from the finite element method.

At this point, one could potentially simply factor $\boldsymbol{K}$ and solve for $\boldsymbol{K}^{-1}\boldsymbol{F}$; however, since the PSO requires only the solution to $\vec{x}$ as a function of $\vec{y}$, there is no need to solve against rows of $\boldsymbol{F}$ corresponding to $\vec{z}$. The Schur complement [191–193] provides a way to avoid extraneous computations. Multiplying the bottom row of (A.6) by $\boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}$ and subtracting from the top row yields:

$$\begin{bmatrix} \boldsymbol{K}_{xx} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{K}_{zx} & \boldsymbol{K}_{xz} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{K}_{zz} \end{bmatrix} \cdot \begin{bmatrix} \vec{x} \\ \vec{z} \end{bmatrix} = \begin{bmatrix} \boldsymbol{F}_{xy} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{F}_{zy} \end{bmatrix} \cdot \vec{y} \tag{A.7}$$

which reduces to

$$\left(\boldsymbol{K}_{xx} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{K}_{zx}\right)\vec{x} = \left(\boldsymbol{F}_{xy} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{F}_{zy}\right)\vec{y} \tag{A.8}$$

The quantity $\boldsymbol{S} = \boldsymbol{K}_{xx} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{K}_{zx}$ is the Schur complement of block $\boldsymbol{K}_{zz}$; since $\boldsymbol{K}$ is singular positive definite, it can be shown (see [194] page 834) that $\boldsymbol{S}$ is also singular positive definite. Eq. (A.8) can be numerically solved to obtain:

$$\vec{x} = \left[\boldsymbol{S}^{-1}\left(\boldsymbol{F}_{xy} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{F}_{zy}\right)\right]\vec{y} \tag{A.9}$$

So the matrix $\boldsymbol{P} = \boldsymbol{S}^{-1}\left(\boldsymbol{F}_{xy} - \boldsymbol{K}_{xz}\boldsymbol{K}_{zz}^{-1}\boldsymbol{F}_{zy}\right)$, in addition to γ_{local} for the outer elements, represents the PSO for the mesh in Figure A-2.

A.2.2 Merging two PSOs

Consider now a finite element mesh whose elements have been partitioned into two submeshes—suppose, into a left and a right half. There will be nodes shared between

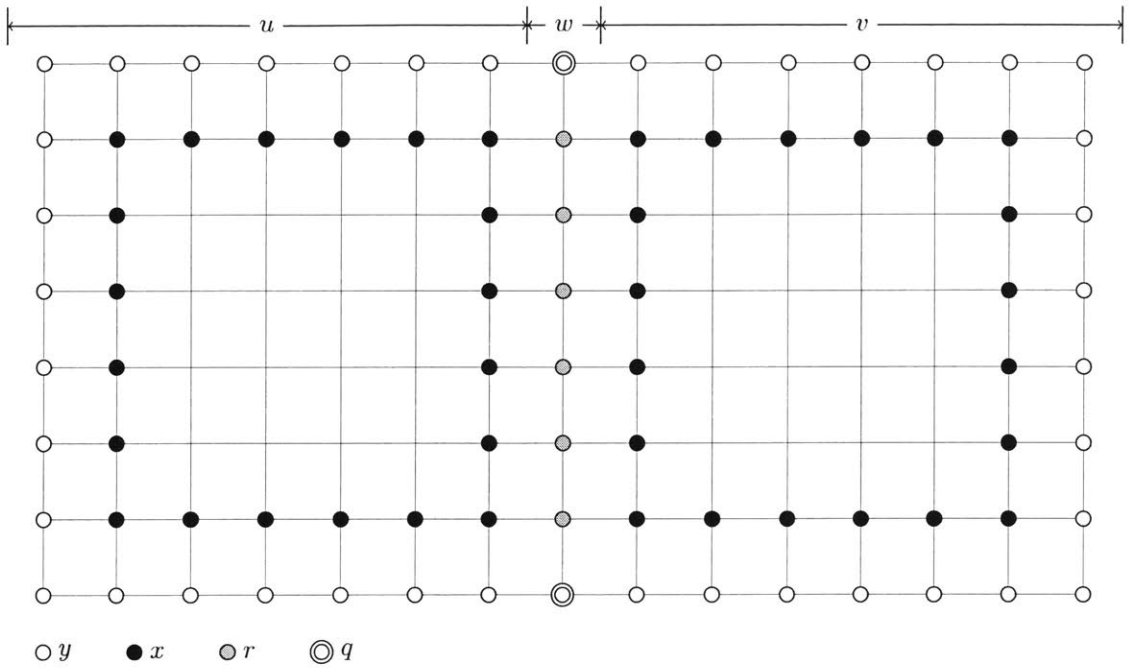


Figure A-3: Scheme for partitioning nodes when combining two PSOs (sharing an edge) into one.

the two submeshes (on the interface, see Figure A-3); put the values of those nodes into $\vec{w}$. The remaining nodes on the left will have values in a vector $\vec{u}$ and the remaining nodes on the right will have values in a vector $\vec{v}$.

As shown in Figure A-3, we further subdivide the nodes as follows. Let $\vec{w}_q$ be the subset of $\vec{w}$ representing nodes on the boundary of the combined supermesh. Let $\vec{w}_q$ be the part of $\vec{w}$ not in $\vec{w}_q$. Let $\vec{u}_y$ be the subset of $\vec{u}$ representing nodes on the boundary of the combined supermesh. Let $\vec{u}_x$ be the part of $\vec{u}$ representing interior nodes belonging to outer elements of the left submesh. Let $\vec{v}_y$ and $\vec{v}_x$ be defined similarly.

Now suppose that a PSO is known for each of the two halves individually; that is, γ_{local} is known for all the outer elements of the two submeshes and the two parent PSOs are $\boldsymbol{P}^u$ and $\boldsymbol{P}^v$, respectively:

$$\vec{u}_x = \boldsymbol{P}^u_{xy}\vec{u}_y + \boldsymbol{P}^u_{xq}\vec{w}_q + \boldsymbol{P}^u_{xr}\vec{w}_r \tag{A.10}$$

$$\vec{v}_x = \boldsymbol{P}^v_{xy}\vec{v}_y + \boldsymbol{P}^v_{xq}\vec{w}_q + \boldsymbol{P}^v_{xr}\vec{w}_r \tag{A.11}$$

The finite element equations corresponding to r nodes may be expressed similarly to (A.6):

$$\boldsymbol{K}_{rr}\vec{w}_r = \boldsymbol{F}^w_{rq}\vec{w}_q + \boldsymbol{F}^u_{ry}\vec{u}_y + \boldsymbol{F}^u_{rx}\vec{u}_x + \boldsymbol{F}^v_{ry}\vec{v}_y + \boldsymbol{F}^v_{rx}\vec{v}_x \tag{A.12}$$

Note that $\boldsymbol{F}_{ry}$ and $\boldsymbol{F}_{rx}$ will be very sparse and can be manipulated into a block structure by judicious node numbering. Substitute Eqs. (A.10) and (A.11) into (A.12):

$$\begin{aligned}\boldsymbol{K}_{rr}\vec{w}_r = \boldsymbol{F}^w_{rq}\vec{w}_q + \boldsymbol{F}^u_{ry}\vec{u}_y + \boldsymbol{F}^u_{rx}\left(\boldsymbol{P}^u_{xy}\vec{u}_y + \boldsymbol{P}^u_{xq}\vec{w}_q + \boldsymbol{P}^u_{xr}\vec{w}_r\right) \\ + \boldsymbol{F}^v_{ry}\vec{v}_y + \boldsymbol{F}^v_{rx}\left(\boldsymbol{P}^v_{xy}\vec{v}_y + \boldsymbol{P}^v_{xq}\vec{w}_q + \boldsymbol{P}^v_{xr}\vec{w}_r\right)\end{aligned} \tag{A.13}$$

which simplifies to:

$$\begin{aligned}\left(\boldsymbol{K}_{rr} - \boldsymbol{F}^u_{rx}\boldsymbol{P}^u_{xr} - \boldsymbol{F}^v_{rx}\boldsymbol{P}^v_{xr}\right)\vec{w}_r = \left(\boldsymbol{F}^w_{rq} + \boldsymbol{F}^u_{rx}\boldsymbol{P}^u_{xq} + \boldsymbol{F}^v_{rx}\boldsymbol{P}^v_{xq}\right)\vec{w}_q + \\ \left(\boldsymbol{F}^u_{ry} + \boldsymbol{F}^u_{rx}\boldsymbol{P}^u_{xy}\right)\vec{u}_y + \left(\boldsymbol{F}^v_{ry} + \boldsymbol{F}^v_{rx}\boldsymbol{P}^v_{xy}\right)\vec{v}_y\end{aligned} \tag{A.14}$$

Define the following:

$$\boldsymbol{\Sigma} = \boldsymbol{K}_{rr} - \boldsymbol{F}^u_{rx}\boldsymbol{P}^u_{xr} - \boldsymbol{F}^v_{rx}\boldsymbol{P}^v_{xr} \tag{A.15}$$

$$\boldsymbol{\Pi}^w_{rq} = \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{F}^w_{rq} + \boldsymbol{F}^u_{rx}\boldsymbol{P}^u_{xq} + \boldsymbol{F}^v_{rx}\boldsymbol{P}^v_{xq}\right) \tag{A.16}$$

$$\boldsymbol{\Pi}^u_{ry} = \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{F}^u_{ry} + \boldsymbol{F}^u_{rx}\boldsymbol{P}^u_{xy}\right) \tag{A.17}$$

$$\boldsymbol{\Pi}^v_{ry} = \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{F}^v_{ry} + \boldsymbol{F}^v_{rx}\boldsymbol{P}^v_{xy}\right) \tag{A.18}$$

So, substituting into (A.14) one obtains:

$$\vec{w}_r = \boldsymbol{\Pi}^w_{rq}\vec{w}_q + \boldsymbol{\Pi}^u_{ry}\vec{u}_y + \boldsymbol{\Pi}^v_{ry}\vec{v}_y \tag{A.19}$$

This can in turn be introduced into (A.10):

$$\begin{aligned}\vec{u}_x &= \boldsymbol{P}^u_{xy}\vec{u}_y + \boldsymbol{P}^u_{xq}\vec{w}_q + \boldsymbol{P}^u_{xr}\left(\boldsymbol{\Pi}^w_{rq}\vec{w}_q + \boldsymbol{\Pi}^u_{ry}\vec{u}_y + \boldsymbol{\Pi}^v_{ry}\vec{v}_y\right) \\ &= \left(\boldsymbol{P}^u_{xy} + \boldsymbol{P}^u_{xr}\boldsymbol{\Pi}^u_{ry}\right)\vec{u}_y + \left(\boldsymbol{P}^u_{xq} + \boldsymbol{P}^u_{xr}\boldsymbol{\Pi}^w_{rq}\right)\vec{w}_q + \boldsymbol{P}^u_{xr}\boldsymbol{\Pi}^v_{ry}\vec{v}_y\end{aligned} \tag{A.20}$$

Algorithm 1: Calculate PSO Recursively

```
1: function psoCalcRecursive(Mesh M)                    ▷ Returns the PSO of M
2:     if the size of M is less than a threshold then
3:         return psoCalcDirect(M)
4:     else
5:         Mesh L,R ← partition(M)
6:         PSO psoL ← psoCalcRecursive(L)
7:         PSO psoR ← psoCalcRecursive(R)
8:         return psoMerge(psoL,psoR)
9:     end if
10: end function

11: function psoCalcDirect(Mesh M)          ▷ Returns the PSO of M using Sec. 2.1

12: function partition(Mesh M)              ▷ Returns two half-meshes partitioned from M

13: function psoMerge(PSO psoL, PSO psoR)   ▷ Returns a single PSO merged from
                                              psoL and psoR using Sec. 2.2
```

$$\begin{aligned}\vec{v}_x &= \boldsymbol{P}^v_{xy}\vec{v}_y + \boldsymbol{P}^v_{xq}\vec{w}_q + \boldsymbol{P}^v_{xr}\left(\boldsymbol{\Pi}^w_{rq}\vec{w}_q + \boldsymbol{\Pi}^u_{ry}\vec{u}_y + \boldsymbol{\Pi}^v_{ry}\vec{v}_y\right) \\ &= \left(\boldsymbol{P}^v_{xy} + \boldsymbol{P}^v_{xr}\boldsymbol{\Pi}^v_{ry}\right)\vec{v}_y + \left(\boldsymbol{P}^v_{xq} + \boldsymbol{P}^v_{xr}\boldsymbol{\Pi}^w_{rq}\right)\vec{w}_q + \boldsymbol{P}^v_{xr}\boldsymbol{\Pi}^u_{ry}\vec{u}_y \qquad \text{(A.21)}\end{aligned}$$

By selecting the relevant rows of (A.19), (A.20), and (A.21), one may finally construct a new PSO for the combined mesh block. One may discard the rows of (A.19), (A.20), and (A.21) corresponding to nodes that do not belong to edge elements of the merged block; these rows (equations) are totally extraneous to the PSO of the merged block of mesh. Similarly, one may discard the γ_{local} values for mesh elements that are not outer elements of the merged block.

A.2.3 A Recursive Algorithm for PSO Calculation

Conceptually, the approach in this section is to break a large finite element mesh into small blocks, calculate the PSO for each block, pairwise merge the PSOs of the small blocks into larger blocks, then pairwise combine those into even larger blocks, and so forth, until the PSO is known for the full mesh. After each merge step, information relating to γ_{local} and nodal values not belonging to outer elements of the mesh block are discarded. The new method is outlined in pseudocode in Algorithm 1.

This algorithm bears some obvious similarity to the multigrid, domain decomposi-

tion, and reduction to the interface methods mentioned in the introduction. However, it achieves improved performance and memory efficiency by discarding information on unnecessary degrees of freedom.

A.2.4 Boundary Conditions and Extracting the Effective Conductivity

Having constructed the PSO for a finite element mesh, all that remains is configuring the boundary conditions in a way that is conducive to extracting γ_{eff}. So, consider again the mesh in Figure A-2. Suppose that γ_{local} is known for the outer elements of the mesh, and that a matrix $\boldsymbol{P}$ is known such that $\vec{x} = \boldsymbol{P}\vec{y}$.

Now suppose that one wants to switch one of the Dirichlet boundary edges to an insulated boundary — let the values of nodes that are to be transformed be contained in $\vec{a}$ and let $\vec{b}$ contain the node values in $\vec{y}$ but not in $\vec{a}$. So:

$$\vec{x} = \boldsymbol{P}_{xa}\vec{a} + \boldsymbol{P}_{xb}\vec{b} \tag{A.22}$$

Now, going through each of the nodes that are to be transformed, the finite element method can construct a system of linear equations linking the ex-Dirichlet node values to the rest of the nodes in $\vec{y}$ and $\vec{x}$. That is, these equations can be expressed similarly to Eqn. (A.6):

$$\boldsymbol{K}_{aa}\vec{a} = \boldsymbol{F}_{ab}\vec{b} + \boldsymbol{F}_{ax}\vec{x} \tag{A.23}$$

Substituting Eqn. (A.22) into Eqn. (A.23) and solving for $\vec{a}$, one finds:

$$\vec{a} = \left(\boldsymbol{K}_{aa} - \boldsymbol{F}_{ax}\boldsymbol{P}_{xa}\right)^{-1}\left(\boldsymbol{F}_{ab} + \boldsymbol{F}_{ax}\boldsymbol{P}_{xb}\right)\vec{b} \tag{A.24}$$

which can be substituted back into Eqn. (A.22) to obtain a new PSO for the new boundary configuration. Similarly, to construct a periodic boundary condition, one need simply construct the necessary finite element equations, substitute Eqn. (A.22) into them, and solve as above.

To extract the effective conductivity from a composite block for which the PSO is

known, one strategy is to establish a "global" potential gradient by constraining one side to $\phi = 0$, the opposing side to $\phi = \Phi$, and insulating all other sides. This fixes $\langle \nabla \phi \rangle \approx (\Phi / L)\hat{n}$ (where $\hat{n}$ is a unit vector in the direction of the imposed potential gradient). This configuration also ensures that $\langle \vec{J} \rangle \cdot \hat{n} = \langle \vec{J} \cdot \hat{n} \rangle_\Gamma$ where Γ is either of the two Dirichlet boundary edges. Substitution of these expressions into Eqn. (A.3) yields a term of γ_{eff}. Unfortunately, when the block is insulated on its sides, it is not possible to recover a meaningful mean flux measurement in directions other than that of the global potential gradient. This reduces the amount of information that can be extracted from each simulation; nonetheless, by rotating the composite domain relative to the global potential gradient, it is possible to construct the $\boldsymbol{\gamma}_{\text{eff}}$ tensor.

A.3 Results

A.3.1 Performance

The RPSOM was benchmarked against a simple traditional finite element method for two-dimensional square-grid meshes with a variety of side lengths. In all cases where a traditional finite element solution was computed, it matched the boundary solution produced by the RPSOM to within floating-point error. Both methods were coded in C++11 and compiled using the Intel compiler with full optimization enabled, and were linked against the Intel MKL library's single-thread linear algebra routines. Both methods were tested on a Linux workstation with an Intel Xeon E5-2640 V3 processor, clocked at 2.6GHz. The code was written for strictly serial execution, though the RPSOM algorithm does suggest the possibility of at least partial parallelization. The traditional finite element method used an optimized-bandwidth sparse stiffness matrix, factored using a Simplicial Cholesky LDLt algorithm from the Eigen 3 package [155]. The RPSOM implementation utilized both Eigen 3's matrix manipulation utilities and MKL's single-threaded BLAS and LAPACK algorithms. The benchmarks shown here were generated using a Python wrapper running the algorithms from the shell; the wrapper program measured memory consumption (taking

100 samples per second) and reported the peak memory consumed by the program along with the total execution time.

As shown in Figure A-4a, the RPSOM realized computation times an order of magnitude faster than the traditional finite element method. The performance gains observed from this method stem from subdividing the mesh and discarding extraneous degrees of freedom from the calculation. Consider a mesh of $N \times N$ nodes. Using a banded Cholesky solver, factoring the stiffness matrix of the mesh would cost $O(N^5)$ operations [195]. Conversely, suppose that the mesh is subdivided into $\frac{N \times N}{n \times n}$ submeshes of dimension $n \times n$. Computing the PSO for a single submesh using the Schur complement method outlined in Section A.2.1 costs $O(n^6)$, assuming fully dense matrix arithmetic. Therefore, to compute all the submeshes' PSO costs $O(n^6 \times \frac{N \times N}{n \times n})$ $= O(N^2 n^4)$, which (for fixed n) is a dramatic improvement on factoring the full stiffness matrix. Of course, the downside is the necessity of then stitching together the submeshes' PSOs. The operation count for stitching two PSOs together that share an edge of p nodes is $O(p^3)$. Since at least the last merge operation occurs with a merge edge length of N, the asymptotic performance of the algorithm is $O(N^3)$. Nonetheless, the dramatic speedup shown in Figure A-4a speaks to the utility of the method.

As shown in Figure A-4b, the memory consumed by the new method is less than a quarter of that consumed by the standard finite element method (which, as previously mentioned, leveraged a fully sparse matrix framework for its calculations).

The reduction in memory achieved by this method stems entirely from discarding extraneous degrees of freedom. Returning to the $N \times N$ square-element mesh, for the standard finite element method to store its Cholesky factorization costs $O(N^3)$. Conversely, since the new method only ever stores information regarding mesh and submesh boundaries, the storage requirement for the new method is theoretically closer to $O(N^2)$.

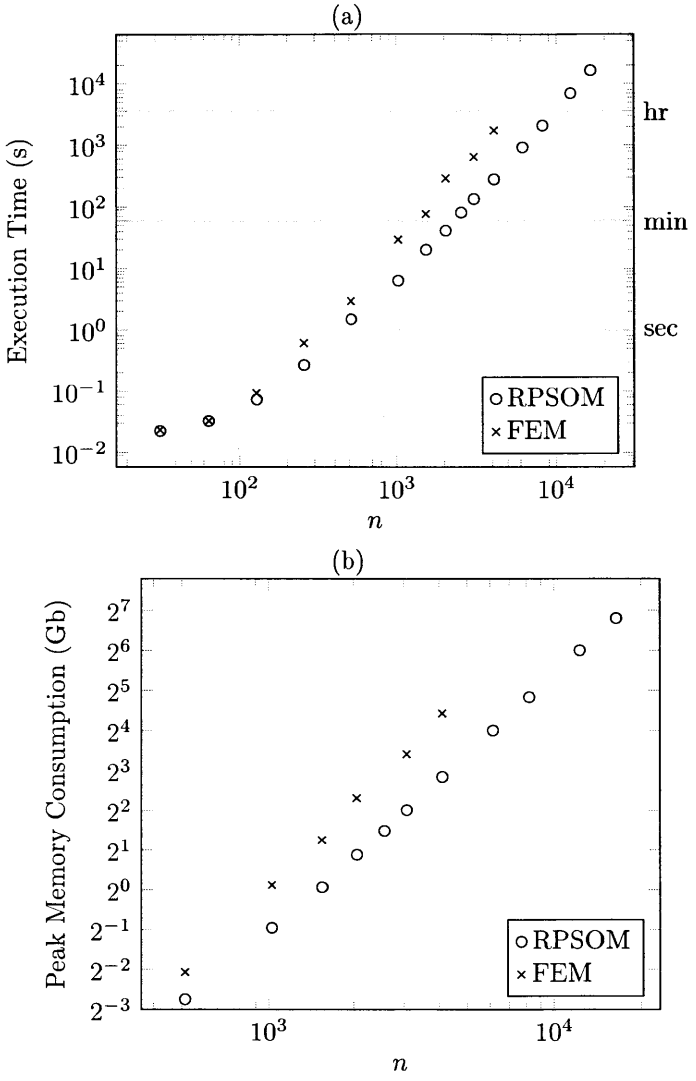


Figure A-4: Time and peak memory consumption to compute the effective conductivity for a square grid of $n \times n$ elements, using the benchmark described in the body of this paper.

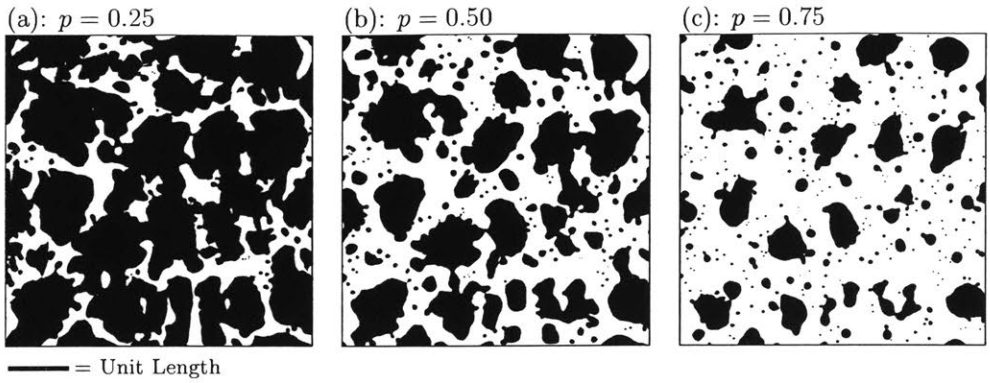


Figure A-5: Test microstructure at three phase fractions; each sample edge has a length of five units.

A.3.2 Calculations on a Random Composite Microstructure

As described in Section A.1, this study is motivated by a desire to measure γ_{eff} near the percolation threshold of random composites. This section shows some preliminary results along those lines for a random two-dimensional composite microstructure.

The test microstructure selected is shown at three phase fraction values in Figure A-5 and its exact specifications are provided in the Appendix. The microstructure is constructed around "seed points" randomly distributed in the plane, and has irregularly-shaped features at three distinct length scales (similar to a fractal). The irregularity, randomness, and fractal-like qualities of this microstructure make it a challenging candidate for homogenization by many existing methods. This section shows how the RPSOM nonetheless enables extraction of the test microstructure's effective conductivity.

The fundamental unit of this computational experiment is an "instantiation" of the microstructure. Each instantiation is a square block with a characteristic edge length (L) and a sampling resolution (R) equal to the number of elements per unit length. Each instantiation also has a unique random spatial phase configuration and can assume any phase fraction in $[0, 1]$. For this study, the conductivity of the dark phase was set to 1 and that of the light phase to 10^6.

For each instantiation, the RPSOM was used to calculate effective conductivity at a variety of phase fractions (p); the phase fractions were sampled between 0.05-

	R					
L	29	32	37	43	51	64
29	2051	2052	2052	2052	2052	2052
32	1820	1820	1820	1820	1820	1820
37	1362	1362	1362	1362	1362	1362
43	1135	1135	1135	1135	1135	1135
51	681	681	681	681	681	681
64	454	454	454	454	454	454
85	227	227	227	227	227	227

Table A.1: Number of replicates for each combination of L and R levels.

0.95 at intervals of 0.05, and between 0.27-0.33 at intervals of 0.005. The resulting $p \mapsto \gamma_{\text{eff}}$ curve was subsequently least-squares fitted to the Generalized Effective Medium (GEM) equation [196, 197]. The GEM equation has three parameters: p_c, s, and t. The parameter p_c corresponds with the percolation threshold, and s and t are critical exponents that respectively describe the curvature of the $p \mapsto \gamma_{\text{eff}}$ function immediately below and above the percolation threshold. The least-squares fitting procedure varied these three parameters to minimize p-direction error:

$$\min_{p_c,s,t} \sum_i \left(p_{p_c,s,t}(\gamma_{\text{eff}}^i) - p^i \right)^2 \tag{A.25}$$

where $(p^i, \gamma_{\text{eff}}^i)$ are points sampled in the $p \mapsto \gamma_{\text{eff}}$ curve, and $p_{p_c,s,t}$ is the GEM-fit function. This approach varies from traditional least-squares, but avoids difficulties resultant from the fact that the range of γ_{eff} covers several orders of magnitude. A histogram of the fit error is shown in Figure A-6 and the 50th and 99th percentile (nearly the worst) fits are shown in Figure A-7. These indicate that the GEM is a satisfactory fit for each instantiation's $p \mapsto \gamma_{\text{eff}}$ curve.

The full experiment produced microstructure instantiations across a grid in (L, R) space, with the number of replicates for each (L, R) combination shown in Table A.1. Each instantiation produced a value of p_c, s, and t; for each (L, R) combination, the p_c, s, and t values were averaged, and the standard error and 95% confidence intervals were calculated. These are shown in Figure A-8; note that in the plots in that figure, the abscissa axes are inverted. For each of p_c, s, and t, the following bilinear model

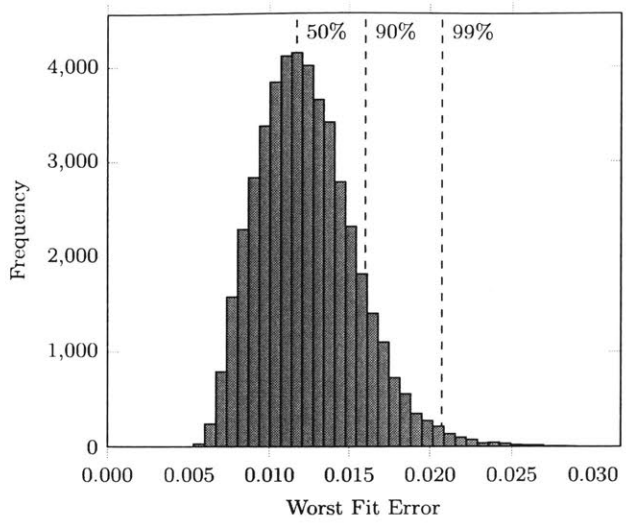


Figure A-6: For each of the 46385 microstructure instantiations, (p, γ_{eff}) points were fitted by the GEM equation, and then the term of the residual with the largest magnitude was found (ϵ). This figure shows a histogram of the ϵ statistics; in about 99% of cases, the worst error in the fit is less than 0.02.

was fit to the means using weighted least-squares:

$$\{p_c|s|t\} \approx a_{11} + a_{L1}\frac{1}{L} + a_{1R}\frac{1}{R} + a_{LR}\frac{1}{LR} \tag{A.26}$$

which enabled extrapolation to $L \to \infty$ and $R \to \infty$ to eliminate edge and finite sampling effects from the experiment. The resulting measurements for the percolation threshold and critical exponents for the test microstructure are: $p_c = 0.3080 \pm 0.0013$, $s = 1.423 \pm 0.013$, and $t = 1.3707 \pm 0.0045$.

To validate these measurements, a "traditional" p_c estimate was calculated using methods roughly analogous to [198]; details of this estimate are presented in the Appendix. The "traditional" percolation threshold measurement was 0.30998 ± 0.00087. This estimate is statistically distinct from the RPSOM+GEM-measured percolation threshold (z-test $p = 0.011$), but it is notable that the RPSOM+GEM-estimated percolation threshold varies from the traditionally-estimated threshold by a mere 0.002, which inspires some confidence in the RPSOM+GEM-measured s and t parameters.

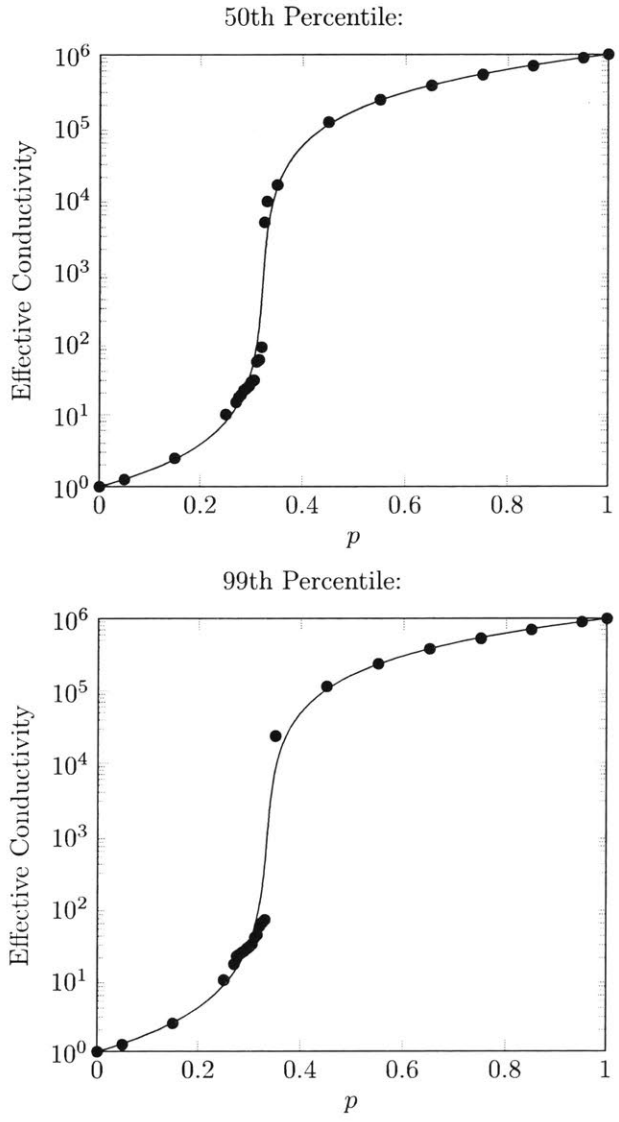


Figure A-7: The 50th and 99th percentile (so, the median and nearly the worst) fits of the GEM equation to effective conductivity measurements of an instantiation.

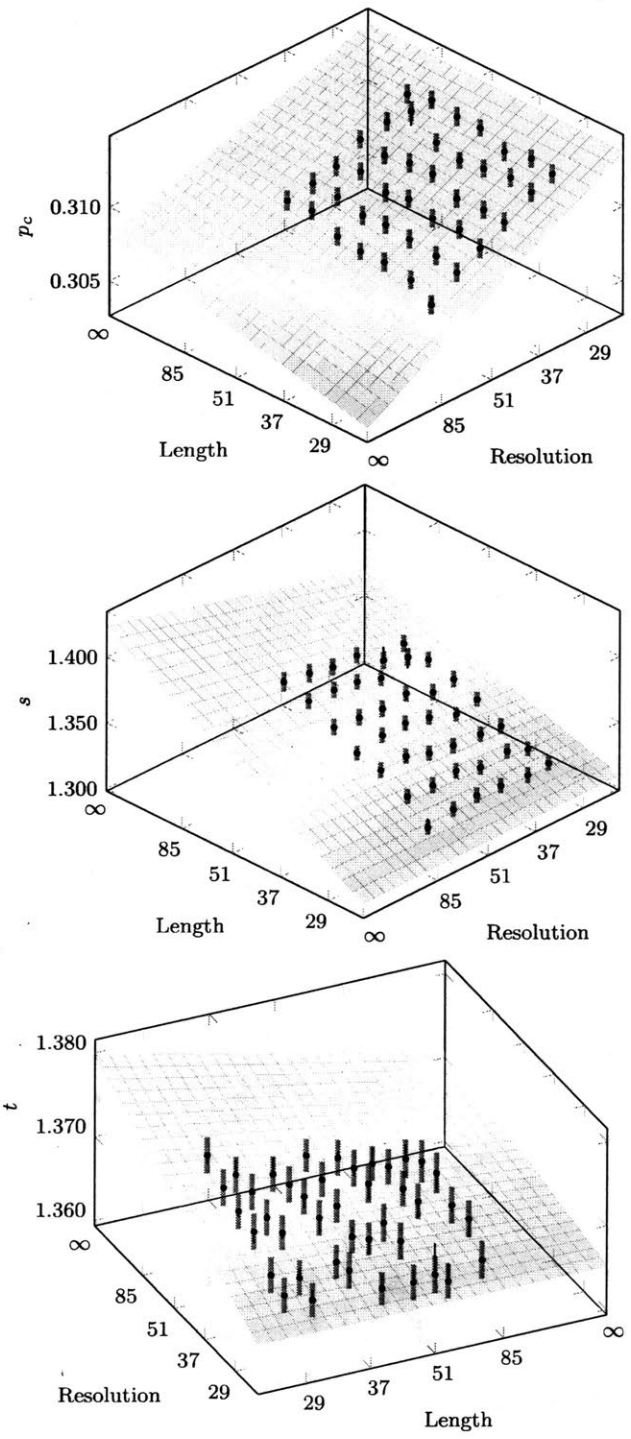


Figure A-8: GEM-fitted parameter data: (from top to bottom) p_c, s, and t. Sample means are marked with •, along with 95% confidence intervals in solid gray. The colored surface fits the sample means; the vertical black lines descend from the sample means to the fit surface. The abscissa axes are inverted in these plots.

A.4 Conclusion

Whereas a simple finite element method provides a way of calculating the effective electrical conductivity, diffusivity, or heat conductivity of a block of composite (these properties are generalized by the symbol γ), the method in this paper does so more efficiently by discarding all unnecessary internal computation and focusing on the domain boundary. The Poincaré-Steklov Operator (PSO) (which maps Dirichlet to corresponding Neumann boundary conditions) captures the information necessary to calculate γ_{eff} for a finite block of composite. Here we show that it can be represented for a finite element mesh using only nodal and γ_{local} values from elements containing boundary nodes; nodal and γ_{local} information from interior elements is extraneous to the PSO. By recursively subdividing a large finite element mesh, calculating individual PSOs for the smallest subdivisions, and then recursively merging the PSOs to construct the full mesh's PSO, the "Recursive Poincaré-Steklov Operator Method" (RPSOM) discards the internal degrees of freedom that are not necessary to represent the PSO at each step and consumes less than a quarter the memory of a traditional finite element algorithm. Similarly, by disregarding extraneous internal degrees of freedom, the RPSOM achieves a speedup of an order of magnitude over the traditional finite element method.

The RPSOM is demonstrated here by calculating the effective conductivity of a random composite microstructure at a variety of phase fractions. The p-γ curves thus generated are well fitted by the Generalized Effective Medium equation and can be extrapolated to eliminate finite size and finite sampling effects. Although the RPSOM is illustrated here for two-dimensional square-grid meshes, it straightforwardly is extensible to three dimensions and general finite element meshes. While this paper focused on phenomena governed by Ohm's Law-like equations, other elliptic partial differential equations feature PSOs, including elasticity. The concepts in this paper may generalize to, for example, fast calculation of forces at mesh boundaries in response to boundary displacement.

A.5 Subappendix A: Test Microstructure Specification

A block D of the test microstructure is constructed using a multi-step numerical process as follows:

1. The region D, scaled by a factor of four, (call it $4D$) and the region immediately surrounding $4D$ are randomly seeded with points P using a Poisson process with parameter $\Lambda = 1$.

2. Define two functions from $4D \mapsto [0, \infty)$ as the distances to the first and second nearest seed points; call those distances respectively $r_1(x)$ and $r_2(x)$ (where $x \in 4D$; note that $r_1(x) \leq r_2(x)$ in general).

3. A scalar field $z : 4D \mapsto [1, \infty)$ is defined: $z(x) = r_2(x)/r_1(x)$ (see Figure A-9a).

4. Define L as the coordinates of the corner of D opposite the origin. Define a new scalar field $Z : D \mapsto [1, \infty)$: $Z(x) = z(x) + z(2L - 2x)/2 + z(4x)/4$ (see Figure A-9b).

5. Let $\lambda(p)$ be a function $\lambda \mapsto [0, \infty)$ be a function such that for random $x \in D$, the probability that $Z(x) < \lambda(p)$ is p.

6. Define the microstructure as follows: for point $x \in D$ and $p \in [0, 1]$, if $Z(x) < \lambda(p)$, then x is phase 0. Otherwise, x is phase 1. The resulting microstructure has phase fraction p of phase 0.

The resulting microstructure is pictured in Figure A-5 at three phase fraction levels.

A.6 Subappendix B: Test Microstructure Percolation Threshold Measurement

For validation purposes, the percolation threshold is calculated using methods similar to those in [198]. The test microstructure is generated for blocks of side length L,

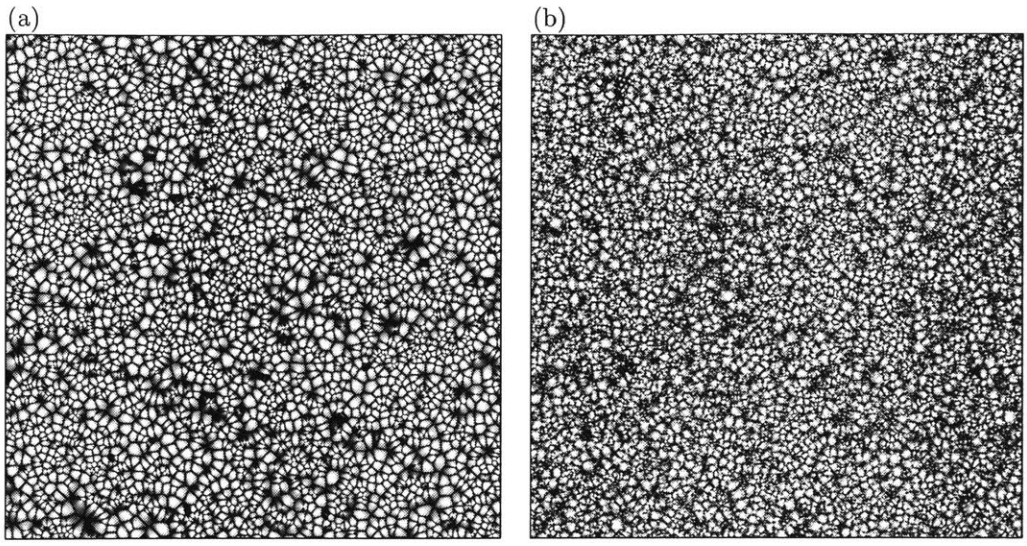


Figure A-9: (a): A sample of field z, and (b) a sample of field Z, as described in Appendix A. Both fields as shown in this figure have been scaled to $[0, 1]$ such that by thresholding at p will produce a structure with phase fraction p.

sampled in a square grid of side length n with a sampling density of $R = n/L$. The number of replicates for each combination of L and R levels is found in Table A.2. For each microstructure block, the percolation threshold was calculated to within 0.1% using a graph-searching algorithm, assuming both edge-only connectivity and edges plus corners connectivity. For each (L,R) treatment, the mean percolation threshold and standard error is calculated. These mean percolation thresholds were fitted using weighted least-squares on the following model:

$$p_c \approx a_{11} + a_{L1}\frac{1}{L} + a_{1R}\frac{1}{R} + a_{LR}\frac{1}{LR} + a_{1R^2}\frac{1}{R^2} + a_{LR^2}\frac{1}{LR^2} \tag{A.27}$$

The fits of the percolation threshold were satisfactory. The model is then extrapolated out to $L = \infty$ and $R = \infty$; the model fits are shown plotted against R in the $L = \infty$ plane in Figure A-10. Both the edges-only and the edges+corners case produce essentially the same extrapolated solution: 0.31003 ± 0.00086 and 0.30998 ± 0.00087, respectively. As mentioned in this paper's body, these measurements of p_c show satisfactory agreement with the GEM-calculated percolation threshold.

	R							
L	29	32	37	43	51	64	85	128
26	15100	16129	15100	16129	15100	16129	15100	16129
32	10784	11652	10784	11652	10784	11652	10784	11652
43	5850	6412	5850	6412	5850	6412	5851	6413
64	2509	2638	2509	2638	2509	2638	2509	2638
128	607	561	607	561	607	561	607	561

Table A.2: Number of replicates for each combination of L and R levels.

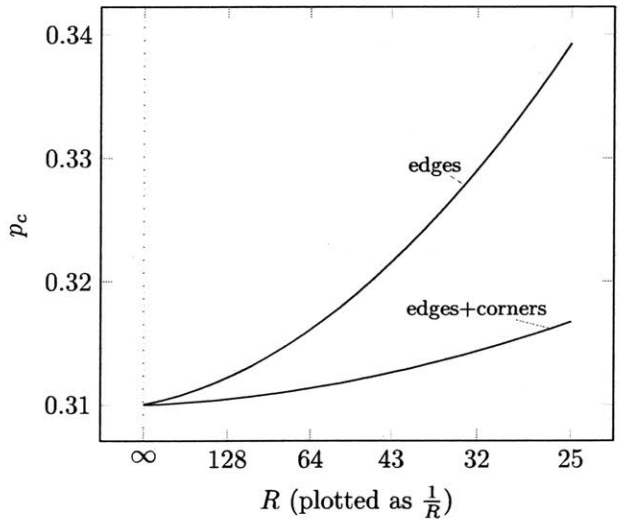


Figure A-10: Cross-sections of the model fit for both the edges-only and the edges+corners case, sectioned at the $L = \infty$ plane and plotted with respect to $\frac{1}{R}$. The point where the curves cross $R = \infty$ is the final estimate for the percolation threshold of the microstructure.

Appendix B

Codebase

This appendix contains a snapshot of the KMGEM codebase used in this thesis. The codebase is written in C++11.

This and several auxiliary libraries (for example, *HardinUtil2* and *HardinFE3*) are available on Github at: `https://github.com/thomasjhardin/`

Listing B.1: KMGEM/Constants.h

```
#ifndef KMGEM_CONSTANTS_H_
#define KMGEM_CONSTANTS_H_

namespace KMGEM {

//Mathematical and physical constants
class Constants {
public:
        static constexpr double kBoltzmann = 1.3806485279e-23; //J/Kelvin
        static constexpr double joulePerEv = 1.602176620898e-19; //unitless
        static constexpr double eVPerJoule = 1 / joulePerEv; //unitless
        static constexpr double atomPerMol = 6.02214085774e23; //per-mol
        static constexpr double molPerAtom = 1 / atomPerMol; //mol
        static constexpr double PI = 3.14159265358979323846264338327950; //↩
            unitless
        static constexpr double TWOPI = 2 * PI; //unitless
        static constexpr double PIHALVES = PI / 2; //unitless
        static constexpr double joulePerMolPerEvPerAtom = joulePerEv * atomPerMol↩
            ; //unitless
};

} //namespace KMGEM

#endif /* KMGEM_CONSTANTS_H_ */
```

Listing B.2: KMGEM/Core.h

```
#ifndef KMGEM_CORE_H_
#define KMGEM_CORE_H_

#define KMGEM_ENABLE_HDF

#include <algorithm>
#include <array>
#include <cstddef>
#include <future>
#include <memory>
#include <numeric>
#include <thread>
#include <vector>

#ifdef KMGEM_ENABLE_HDF
#include "H5Cpp.h"
#include "HardinUtil2/HDF.h"
#endif //KMGEM_ENABLE_HDF

#include "Eigen/Dense"

#include "HardinUtil2/CleanupDirectoryPath.h"
#include "HardinUtil2/FileExists.h"
#include "HardinUtil2/GammaIncomplete.h"
#include "HardinUtil2/HashCombine.h"
#include "HardinUtil2/IniReader.h"
#include "HardinUtil2/IntToHexPadded.h"
#include "HardinUtil2/Elasticity/PrincipalStresses.h"
#include "HardinUtil2/Elasticity/RotateVoigt.h"

#include "KMGEM/Constants.h"

#endif /* KMGEM_CORE_H_ */
```

Listing B.3: KMGEM/Homogeneous.h

```
1  #ifndef KMGEM_HOMOGENEOUS_H_
2  #define KMGEM_HOMOGENEOUS_H_
3
4  #include "KMGEM/Core.h"
5  #include "KMGEM/Kinetics.h"
6
7  #include "KMGEM/Homogeneous/RateMatrixRelaxation.h"
8  #include "KMGEM/Homogeneous/RateMatricesShear.h"
9
10 #include "KMGEM/Homogeneous/EvolveStress.h"
11
12 #endif /* KMGEM_HOMOGENEOUS_H_ */
```

Listing B.4: KMGEM/Kinetics.h

```
#ifndef KMGEM_KINETICS_H_
#define KMGEM_KINETICS_H_

#include "KMGEM/Core.h"

#include "KMGEM/Kinetics/EquationsOfState.h"
#include "KMGEM/Kinetics/Levels.h"

#include "KMGEM/Kinetics/RateRelaxation.h"
#include "KMGEM/Kinetics/RateShear.h"

#include "KMGEM/Kinetics/IntegralOrientation.h"
#include "KMGEM/Kinetics/RateShearEvolution.h"

#endif /* KMGEM_KINETICS_H_ */
```

Listing B.5: KMGEM/Kinetics/RateShearEvolution.h

```
#ifndef KMGEM_KINETICS_RATESHEAREVOLUTION_H_
#define KMGEM_KINETICS_RATESHEAREVOLUTION_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics/Levels.h"
#include "KMGEM/Kinetics/RateShear.h"

namespace KMGEM {
namespace Kinetics {

class RateShearEvolution {
public:
	class Parameters {
	public:
		Levels levels;
		RateShear rateShear;

		double temperatureBias;
		double pressureBias;

		static Parameters vanilla(
			const Levels &levels,
			const RateShear &rateShear
		) {
			Parameters params;
			params.levels = levels;
			params.rateShear = rateShear;

			params.temperatureBias = 600;
			params.pressureBias = -1e9;

			return params;
		}
	};

```

```
36      Parameters params;
37      Levels levels; //Alias
38      RateShear rateShear; //Alias
39      EquationsOfState equationsOfState; //Alias
40
41      void alias() {
42              levels = params.levels;
43              rateShear = params.rateShear;
44              equationsOfState = params.rateShear.equationsOfState;
45      }
46      void setup(const Parameters params_) { params = params_; alias(); }
47      RateShearEvolution() {}
48      RateShearEvolution(const Parameters params_) { setup(params_); }
49      static RateShearEvolution vanilla(
50              const Levels &levels,
51              const RateShear &rateShear
52      ) { return RateShearEvolution(Parameters::vanilla(levels, rateShear)); }
53
54      void getRateShearEvolution(
55              const double temperature,
56              const double* stressPrincipalInitial,
57              double* levelRates, //[nLevels]
58              double* levelStrainRates, //[6, nLevels]{COL-Major}
59              double* levelProbabilities, //[nLevels]
60              const std::int8_t nThreads = 1
61      ) const {
62              double pressure = -(stressPrincipalInitial[0] + ↩
                    stressPrincipalInitial[1] + stressPrincipalInitial[2]) / 3.0;
63
64              Eigen::Map<Eigen::Matrix<double, 6, -1, Eigen::ColMajor>> ↩
                    strainRates(levelStrainRates, 6, levels.nLevels);
65
66              auto threadLambda = [&](const std::int8_t iThread) {
67                      std::array<double, 6> strainRate;
68                      for (std::int64_t iLevel = iThread; iLevel < levels.↩
                            nLevels; iLevel += nThreads) {
```

```
69                          double cpe = levels.cpeLevels(iLevel);
70                          double cpemIsochoricInitial = levels.↩
                                cpeIsochoricLevels(iLevel) / levels.zoneSize;
71                          double cpemDilatativeInitial = levels.↩
                                cpeDilatativeLevels(iLevel) / levels.zoneSize;
72                          rateShear.getRateTotal(temperature, ↩
                                stressPrincipalInitial, cpemIsochoricInitial, ↩
                                cpemDilatativeInitial, levelRates + iLevel, ↩
                                strainRate.data());
73                          for (std::int8_t dim = 0; dim < 6; dim++) { ↩
                                strainRates(dim,iLevel) = strainRate[dim]; }
74
75                          double volume = equationsOfState.getVolumeMolar(↩
                                temperature, cpemDilatativeInitial) * levels.↩
                                zoneSize;
76                          double pressureBiased = pressure + params.↩
                                pressureBias;
77                          double temperatureBiased = temperature + params.↩
                                temperatureBias;
78                          double potentialBiased = cpe + pressureBiased * ↩
                                volume;
79                          double expArg = -potentialBiased / (Constants::↩
                                kBoltzmann * temperatureBiased);
80                          double targetFraction = levels.degeneracies(iLevel)↩
                                 * std::exp(expArg);
81
82                          levelProbabilities[iLevel] = targetFraction * ↩
                                levelRates[iLevel];
83                  }
84          };
85
86          if (nThreads == 1) { threadLambda(0); }
87          else {
88                  std::vector<std::thread> threads;
89                  for (std::int8_t iThread = 0; iThread < nThreads; iThread↩
                        ++) { threads.push_back(std::thread(threadLambda, ↩
```

```
                        iThread)); }
90                      for (std::thread &thread : threads) { thread.join(); }
91              }
92
93              double totalLevelProbabilities = 0;
94              for (std::int64_t iLevel = 0; iLevel < levels.nLevels; iLevel++) {↩
                    totalLevelProbabilities += levelProbabilities[iLevel]; }
95              for (std::int64_t iLevel = 0; iLevel < levels.nLevels; iLevel++) {↩
                    levelProbabilities[iLevel] /= totalLevelProbabilities; }
96      }
97  };
98
99  } //namespace Kinetics
100 } //namespace KMGEM
101
102 #endif /* KMGEM_KINETICS_RATESHEAREVOLUTION_H_ */
```

Listing B.6: KMGEM/Kinetics/RateShear.h

```
#ifndef KMGEM_KINETICS_RATESHEAR_H_
#define KMGEM_KINETICS_RATESHEAR_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics/EquationsOfState.h"
#include "KMGEM/Kinetics/IntegralOrientation.h"

namespace KMGEM {
namespace Kinetics {

class RateShear {
public:
        class Parameters {
        public:
                EquationsOfState equationsOfState;
                TableIntegralOrientation tableIntegralOrientation;

                double attemptFrequency; //per-second; How often a zone attempts ↩
                    to transform
                double zoneSize; //mol; Number of atoms rearranged by a single ↩
                    event
                double dilatationTransition; //Unitless; volumetric strain ↩
                    associated with the transition state
                double potentialBarrierXsZero; //J; excess potential barrier ↩
                    associated with zero pressure and zero shear modulus
                double potentialBarrierXsRef; //J; excess potential barrier ↩
                    associated with zero pressure and reference shear modulus (as ↩
                    in equationsOfState)
                double gammaFinal; //Unitless; quantum of engineering shear strain↩
                     for a complete shear transition

                static Parameters vanilla(
                        const EquationsOfState &equationsOfState,
                        const TableIntegralOrientation &tableIntegralOrientation
                        ) {
```

```
29                  Parameters params;
30                  params.equationsOfState = equationsOfState;
31                  params.tableIntegralOrientation = tableIntegralOrientation↩
                        ;
32
33                  params.attemptFrequency = 1e10; //Per-second
34                  params.zoneSize = 75 * KMGEM::Constants::molPerAtom; //↩
                        Mols
35                  params.dilatationTransition = 0.01; //unitless
36                  params.potentialBarrierXsRef = 1 * Constants::joulePerEv;
37                  params.potentialBarrierXsZero = params.↩
                        potentialBarrierXsRef * 1;
38                  params.gammaFinal = 0.1; //Unitless
39
40                  return params;
41          }
42      };
43
44      Parameters params;
45      EquationsOfState equationsOfState; //Alias
46      TableIntegralOrientation tableIntegralOrientation; //Alias
47      IntegralOrientation integralOrientation; //Alias
48
49      void alias() {
50              equationsOfState = params.equationsOfState;
51              tableIntegralOrientation = params.tableIntegralOrientation;
52              integralOrientation = params.tableIntegralOrientation.↩
                    integralOrientation;
53      }
54      void setup(const Parameters params_) { params = params_; alias(); }
55      RateShear() {}
56      RateShear(const Parameters params_) { setup(params_); }
57      static RateShear vanilla(
58              const EquationsOfState &equationsOfState,
59              const TableIntegralOrientation &tableIntegralOrientation
60      ) { return RateShear(Parameters::vanilla(equationsOfState, ↩
```

```
    tableIntegralOrientation)); }
61
62      void getXi(
63              const double temperature, //Kelvin
64              const double* stressPrincipal, //Pa[3]; Principal stresses
65              const double volumeZone, //m^3
66              double* xi //Unitless[2]
67      ) const {
68              double xiFactor = 0.5 * params.gammaFinal * volumeZone / (↩
                    Constants::kBoltzmann * temperature);
69              double pressure = -(stressPrincipal[0] + stressPrincipal[1] + ↩
                    stressPrincipal[2]) * (1.0 / 3.0);
70              xi[0] = (stressPrincipal[0] + pressure) * xiFactor;
71              xi[1] = (stressPrincipal[1] + pressure) * xiFactor;
72      }
73
74      double getRateFactorOriented(
75              const double temperature,
76              const double* stressPrincipalInitial,
77              const double volumeZone,
78              const double* orientationRodrigues,
79              double* strainOriented
80      ) const {
81              std::array<double, 2> xi;
82              getXi(temperature, stressPrincipalInitial, volumeZone, xi.data());
83              double integrand = integralOrientation.getIntegrandOriented(xi.↩
                    data(), orientationRodrigues, strainOriented);
84              for (std::int8_t i = 0; i < 6; i++) { strainOriented[i] *= params.↩
                    gammaFinal; }
85              return integrand;
86      }
87
88      double getRateFactorIntegrated(
89              const double temperature,
90              const double* stressPrincipalInitial,
91              const double volumeZone,
```

```
92              double* strainRateFactorIntegrated
93          ) const {
94              std::array<double, 2> xi;
95              getXi(temperature, stressPrincipalInitial, volumeZone, xi.data());
96              double rateFactorIntegrated = tableIntegralOrientation.getIntegral↩
                    (xi.data(), strainRateFactorIntegrated);
97              for (std::int8_t i = 0; i < 6; i++) { strainRateFactorIntegrated[i↩
                    ] *= params.gammaFinal; }
98              return rateFactorIntegrated;
99          }
100
101         double getPotentialBarrierInvariant(
102             const double temperature,
103             const double* stressPrincipalInitial,
104             const double cpem,
105             const double volumeZone
106         )const {
107             double shearModulus = equationsOfState.getShearModulus(temperature↩
                    , cpem);
108             double potentialBarrierXs = params.potentialBarrierXsZero + ↩
                    shearModulus * (params.potentialBarrierXsRef - params.↩
                    potentialBarrierXsZero) / equationsOfState.params.↩
                    shearModulusReference;
109
110             double deltaVolumeTransition = params.dilatationTransition * ↩
                    volumeZone;
111             double pressureInitial = -0.33333333333333333333333333333333333333 ↩
                    * (stressPrincipalInitial[0] + stressPrincipalInitial[1] + ↩
                    stressPrincipalInitial[2]);
112
113             double potentialBarrier = potentialBarrierXs + pressureInitial * ↩
                    deltaVolumeTransition;
114             return potentialBarrier;
115         }
116
117         double getPotentialBiasMostFavorable(
```

```
118             const double* stressPrincipalInitial,
119             const double volumeZone,
120             double* strainMostFavorable
121         ) const {
122             for (std::int8_t i = 0; i < 6; i++) { strainMostFavorable[i] = 0; ↩
                    }
123
124             std::array<std::int8_t, 3> orderIndices{ {0,1,2} };
125             std::sort(orderIndices.begin(), orderIndices.end(), [↩
                    stressPrincipalInitial](const std::int8_t i, const std::int8_t↩
                     j) { return stressPrincipalInitial[i] <= ↩
                    stressPrincipalInitial[j]; });
126             assert(stressPrincipalInitial[orderIndices[0]] <= ↩
                    stressPrincipalInitial[orderIndices[1]] && ↩
                    stressPrincipalInitial[orderIndices[1]] <= ↩
                    stressPrincipalInitial[orderIndices[2]]);
127             double sMin = stressPrincipalInitial[orderIndices[0]];
128             double sMid = stressPrincipalInitial[orderIndices[1]];
129             double sMax = stressPrincipalInitial[orderIndices[2]];
130
131             if (sMin == sMid && sMid == sMax) {
132                 strainMostFavorable[0] = -0.5 * params.gammaFinal;
133                 strainMostFavorable[2] = 0.5 * params.gammaFinal;
134             }
135             else if (sMin == sMid) {
136                 strainMostFavorable[orderIndices[2]] = 0.5 * params.↩
                        gammaFinal;
137                 strainMostFavorable[orderIndices[1]] = -0.25 * params.↩
                        gammaFinal;
138                 strainMostFavorable[orderIndices[0]] = -0.25 * params.↩
                        gammaFinal;
139             }
140             else if (sMid == sMax) {
141                 strainMostFavorable[orderIndices[0]] = -0.5 * params.↩
                        gammaFinal;
142                 strainMostFavorable[orderIndices[1]] = 0.25 * params.↩
```

```
                    gammaFinal;
143                 strainMostFavorable[orderIndices[2]] = 0.25 * params.↩
                        gammaFinal;
144             }
145             else {
146                 strainMostFavorable[orderIndices[0]] = -0.5 * params.↩
                        gammaFinal;
147                 strainMostFavorable[orderIndices[2]] = 0.5 * params.↩
                        gammaFinal;
148             }
149
150             double potentialBiasMostFavorable = -0.5 * 0.5 * params.gammaFinal↩
                    * volumeZone * (sMax - sMin);
151             return potentialBiasMostFavorable;
152         }
153
154         char getRateTotal(
155             const double temperature,
156             const double* stressPrincipalInitial,
157             const double cpemIsochoricInitial,
158             const double cpemDilatativeInitial,
159             double* activationRate,
160             double* strainRate
161         ) const {
162             double cpemInitial = cpemIsochoricInitial + cpemDilatativeInitial;
163             double volumeMolar = equationsOfState.getVolumeMolar(temperature, ↩
                    0);
164             double volumeZone = params.zoneSize * volumeMolar;
165
166             double potentialBarrierInvariant = getPotentialBarrierInvariant(
167                 temperature,
168                 stressPrincipalInitial,
169                 cpemInitial,
170                 volumeZone
171             );
172             double potentialBiasMostFavorable = getPotentialBiasMostFavorable(
```

```
173                     stressPrincipalInitial,
174                     volumeZone,
175                     strainRate
176             );
177
178             double potentialBarrierMostFavorable = potentialBarrierInvariant +↩
                    potentialBiasMostFavorable;
179             if (potentialBarrierMostFavorable < 0) { //If stress-activated
180                     (*activationRate) = params.attemptFrequency;
181                     for (std::int8_t i = 0; i < 6; i++) { strainRate[i] *= (*↩
                        activationRate); }
182                     return 's';
183             }
184             else {
185                     double expArgInvariant = -potentialBarrierInvariant / (↩
                        Constants::kBoltzmann * temperature);
186                     double rateFactorInvariant = params.attemptFrequency * std↩
                        ::exp(expArgInvariant);
187                     double rateFactorIntegrated = getRateFactorIntegrated(↩
                        temperature, stressPrincipalInitial, volumeZone, ↩
                        strainRate);
188                     for (std::int8_t i = 0; i < 6; i++) { strainRate[i] *= ↩
                        rateFactorInvariant; }
189                     (*activationRate) = rateFactorInvariant * ↩
                        rateFactorIntegrated;
190                     return 't';
191             }
192     }
193
194 };
195
196 } //namespace Kinetics
197 } //namespace KMGEM
198
199 #endif /* KMGEM_KINETICS_RATESHEAR_H_ */
```

Listing B.7: KMGEM/Kinetics/RateRelaxation.h

```
#ifndef KMGEM_KINETICS_RATERELAXATION_H_
#define KMGEM_KINETICS_RATERELAXATION_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics/EquationsOfState.h"

namespace KMGEM {
namespace Kinetics {

class RateRelaxation {
public:
        class Parameters {
        public:
                EquationsOfState equationsOfState;
                Levels levels;

                double potentialBarrierXsZero; //J; excess potential barrier ↩
                    associated with zero pressure and zero shear modulus
                double potentialBarrierXsRef; //J; excess potential barrier ↩
                    associated with zero pressure and reference shear modulus (as ↩
                    in equationsOfState)
                double attemptFrequency; //per-second; How often a zone attempts ↩
                    to transition

                static Parameters vanilla(const EquationsOfState &equationsOfState↩
                    , const Levels &levels) {
                        Parameters params;
                        params.equationsOfState = equationsOfState;
                        params.levels = levels;

                        params.potentialBarrierXsRef = 0.8 * KMGEM::Constants::↩
                            joulePerEv; //Joule
                        params.potentialBarrierXsZero = 0.5 * KMGEM::Constants::↩
                            joulePerEv; //Joule
                        params.attemptFrequency = 1e10; //Per-second
```

```
29
30                  return params;
31              }
32          };
33
34          Parameters params;
35          EquationsOfState equationsOfState; //Alias
36          Levels levels;
37
38          void alias() { equationsOfState = params.equationsOfState; levels = ↩
                params.levels; }
39          void setup(const Parameters params_) { params = params_; alias(); }
40          RateRelaxation() {}
41          RateRelaxation(const Parameters params_) { setup(params_); }
42          static RateRelaxation vanilla(const EquationsOfState &equationsOfState, ↩
                const Levels &levels) { return RateRelaxation(Parameters::vanilla(↩
                equationsOfState,levels)); }
43
44          double getPotentialBarrierHeight(
45                  const double temperature,
46                  const double cpeIsochoricInitial,
47                  const double cpeDilatativeInitial,
48                  const double cpeIsochoricFinal,
49                  const double cpeDilatativeFinal
50          ) const {
51                  double cpemInitial = (cpeIsochoricInitial + cpeDilatativeInitial) ↩
                        / levels.zoneSize;
52                  double cpemFinal = (cpeIsochoricFinal + cpeDilatativeFinal) / ↩
                        levels.zoneSize;
53                  double cpemMean = 0.5 * (cpemInitial + cpemFinal);
54                  double shearModulusMean = equationsOfState.getShearModulus(↩
                        temperature, cpemMean);
55                  double potentialBarrierHeight = params.potentialBarrierXsZero + (↩
                        shearModulusMean / equationsOfState.params.↩
                        shearModulusReference) * ( params.potentialBarrierXsRef - ↩
                        params.potentialBarrierXsZero );
```

```
56              return potentialBarrierHeight;
57      }
58
59      double getPotentialBarrier(
60              const double temperature,
61              const double cpeIsochoricInitial,
62              const double cpeDilatativeInitial,
63              const double cpeIsochoricFinal,
64              const double cpeDilatativeFinal
65      ) const {
66              double potentialInitial = cpeIsochoricInitial + ↩
                    cpeDilatativeInitial;
67              double potentialFinal = cpeIsochoricFinal + cpeDilatativeFinal;
68              double potentialDifference = potentialFinal - potentialInitial;
69              double potentialBarrierHeight = getPotentialBarrierHeight(
70                      temperature,
71                      cpeIsochoricInitial,
72                      cpeDilatativeInitial,
73                      cpeIsochoricFinal,
74                      cpeDilatativeFinal
75              );
76              double potentialBarrier = std::max<double>(0, potentialDifference)↩
                     + potentialBarrierHeight;
77              return potentialBarrier;
78      }
79
80      double getRate(
81              const double temperature,
82              const double cpeIsochoricInitial,
83              const double cpeDilatativeInitial,
84              const double cpeIsochoricFinal,
85              const double cpeDilatativeFinal
86      ) const {
87              double potentialBarrier = getPotentialBarrier(
88                      temperature,
89                      cpeIsochoricInitial,
```

```
90                      cpeDilatativeInitial,
91                      cpeIsochoricFinal,
92                      cpeDilatativeFinal
93                  );
94
95                  double expArg = -potentialBarrier / (Constants::kBoltzmann * ↩
                        temperature);
96                  double rate = params.attemptFrequency * std::exp(expArg);
97                  return rate;
98          }
99  };
100
101 } //namespace Kinetics
102 } //namespace KMGEM
103
104 #endif /* KMGEM_KINETICS_RATERELAXATIONEXCITATION_H_ */
```

Listing B.8: KMGEM/Kinetics/Levels.h

```
#ifndef KMGEM_KINETICS_LEVELS_H_
#define KMGEM_KINETICS_LEVELS_H_

#include "KMGEM/Core.h"

namespace KMGEM {
namespace Kinetics {

class Levels {
public:
        std::int64_t nLevels;
        double zoneSize; //Mols; fundamental relaxation zone size
protected:
        std::shared_ptr<double> cpeLevelsBuffer;
        std::shared_ptr<double> cpeIsochoricLevelsBuffer;
        std::shared_ptr<double> cpeDilatativeLevelsBuffer;
        std::shared_ptr<double> degeneraciesBuffer;
public:
        virtual std::size_t hash() const { return -1; }
        virtual bool operator==(const Levels &other) const { return 0; }

        double* cpeLevels() { return cpeLevelsBuffer.get(); }
        double* cpeIsochoricLevels() { return cpeIsochoricLevelsBuffer.get(); }
        double* cpeDilatativeLevels() { return cpeDilatativeLevelsBuffer.get(); }
        double* degeneracies() { return degeneraciesBuffer.get(); }

        double* cpeLevels() const { return cpeLevelsBuffer.get(); }
        double* cpeIsochoricLevels() const { return cpeIsochoricLevelsBuffer.get
            (); }
        double* cpeDilatativeLevels() const { return cpeDilatativeLevelsBuffer.
            get(); }
        double* degeneracies() const { return degeneraciesBuffer.get(); }

        double& cpeLevels(const std::int64_t i) { return cpeLevelsBuffer.get()[i
            ]; }
```

```
33          double& cpeIsochoricLevels(const std::int64_t i) { return ↩
                cpeIsochoricLevelsBuffer.get()[i]; }
34          double& cpeDilatativeLevels(const std::int64_t i) { return ↩
                cpeDilatativeLevelsBuffer.get()[i]; }
35          double& degeneracies(const std::int64_t i) { return degeneraciesBuffer.↩
                get()[i]; }
36
37          double& cpeLevels(const std::int64_t i) const { return cpeLevelsBuffer.↩
                get()[i]; }
38          double& cpeIsochoricLevels(const std::int64_t i) const { return ↩
                cpeIsochoricLevelsBuffer.get()[i]; }
39          double& cpeDilatativeLevels(const std::int64_t i) const { return ↩
                cpeDilatativeLevelsBuffer.get()[i]; }
40          double& degeneracies(const std::int64_t i) const { return ↩
                degeneraciesBuffer.get()[i]; }
41
42          void allocate(const std::int64_t nLevels) {
43                  this->nLevels = nLevels;
44                  cpeLevelsBuffer.reset(new double[nLevels], std::default_delete<↩
                        double[]>());
45                  cpeIsochoricLevelsBuffer.reset(new double[nLevels], std::↩
                        default_delete<double[]>());
46                  cpeDilatativeLevelsBuffer.reset(new double[nLevels], std::↩
                        default_delete<double[]>());
47                  degeneraciesBuffer.reset(new double[nLevels], std::default_delete<↩
                        double[]>());
48          }
49
50  //#ifdef KMGEM_ENABLE_HDF
51  //      virtual void save(const H5::Group &target) const {
52  //              HardinUtil2::saveScalar<std::int64_t>(&nLevels, "nLevels", target)↩
        ;
53  //              HardinUtil2::saveArray<double, 1>(cpeLevelsBuffer.get(), { { ↩
        hsize_t(nLevels) } }, "cpeLevelsBuffer", target);
54  //              HardinUtil2::saveArray<double, 1>(cpeIsochoricLevelsBuffer.get(), ↩
        { { hsize_t(nLevels) } }, "cpeIsochoricLevelsBuffer", target);
```

```
55 //             HardinUtil2::saveArray<double, 1>(cpeDilatativeLevelsBuffer.get(),↩
       { { hsize_t(nLevels) } }, "cpeDilatativeLevelsBuffer", target);
56 //             HardinUtil2::saveArray<double, 1>(degeneraciesBuffer.get(), { { ↩
       hsize_t(nLevels) } }, "degeneraciesBuffer", target);
57 //     }
58 //     virtual void load(const H5::Group &target) {
59 //             nLevels = HardinUtil2::loadScalar<std::int64_t>("nLevels", target)↩
       ;
60 //             allocate(nLevels);
61 //             HardinUtil2::loadArray<double, 1>(target, "cpeLevelsBuffer", ↩
       cpeLevelsBuffer.get());
62 //             HardinUtil2::loadArray<double, 1>(target, "↩
       cpeIsochoricLevelsBuffer", cpeIsochoricLevelsBuffer.get());
63 //             HardinUtil2::loadArray<double, 1>(target, "↩
       cpeDilatativeLevelsBuffer", cpeDilatativeLevelsBuffer.get());
64 //             HardinUtil2::loadArray<double, 1>(target, "degeneraciesBuffer", ↩
       degeneraciesBuffer.get());
65 //     }
66 //#endif
67
68         void sort() {
69                 std::vector<double> cpeLevelsVector(nLevels);
70                 std::vector<double> cpeIsochoricLevelsVector(nLevels);
71                 std::vector<double> cpeDilatativeLevelsVector(nLevels);
72                 std::vector<double> degeneraciesVector(nLevels);
73                 for (std::int64_t i = 0; i < nLevels; i++) {
74                         cpeLevelsVector[i] = cpeLevels(i);
75                         cpeIsochoricLevelsVector[i] = cpeIsochoricLevels(i);
76                         cpeDilatativeLevelsVector[i] = cpeDilatativeLevels(i);
77                         degeneraciesVector[i] = degeneracies(i);
78                 }
79
80                 std::vector<std::int64_t> indices(nLevels);
81                 for (std::int64_t i = 0; i < nLevels; i++) { indices[i] = i; }
82                 std::sort(indices.begin(), indices.end(),
83                         [this] (const std::int64_t i, const std::int64_t j) { ↩
```

```
                        return cpeLevels(i) < cpeLevels(j); });
84              for (std::int64_t i = 0; i < nLevels; i++) {
85                      cpeLevels(i) = cpeLevelsVector[indices[i]];
86                      cpeIsochoricLevels(i) = cpeIsochoricLevelsVector[indices[i↩
                            ]];
87                      cpeDilatativeLevels(i) = cpeDilatativeLevelsVector[indices↩
                            [i]];
88                      degeneracies(i) = degeneraciesVector[indices[i]];
89              }
90      }
91  };
92
93  class LevelsPower : public Levels {
94  public:
95      class Parameters {
96      public:
97              double numberOfStatesReference;
98              double energyReference;
99              double cpeIsochoricExponent;
100             double cpeDilatativeExponent;
101
102             std::int64_t nLevelsIsochoric;
103             std::int64_t nLevelsDilatative;
104
105             double zoneSize;
106             double temperatureMax;
107             double occupationFractionCutoff;
108             double energyResolutionBest;
109
110             static Parameters vanilla() {
111                     Parameters params;
112
113                     params.numberOfStatesReference = 1;
114                     params.energyReference = KMGEM::Constants::kBoltzmann * ↩
                            300;
115                     params.cpeIsochoricExponent = 1.0;
```

```
116                     params.cpeDilatativeExponent = 1.0;
117
118                     params.nLevelsIsochoric = 40;
119                     params.nLevelsDilatative = 15;
120
121                     params.zoneSize = 2 * KMGEM::Constants::molPerAtom;
122                     params.temperatureMax = 1200;
123                     params.occupationFractionCutoff = 0.9999;
124                     params.energyResolutionBest = 1e-3 * KMGEM::Constants::↩
                            joulePerEv;
125
126                     return params;
127             }
128     };
129
130     Parameters params;
131
132     void discretizeStates() {
133             nLevels = params.nLevelsDilatative * params.nLevelsIsochoric;
134             allocate(nLevels);
135
136             //Calculate cpe[Isochoric|Dilatative]Max
137             auto getCpeXMax = [this](const double cpeXExponent) {
138                     auto occupationFraction = [this, cpeXExponent](const ↩
                            double cpeX) {
139                             double gammaParameter = cpeXExponent;
140                             double gammaArgument = cpeX / (Constants::↩
                                    kBoltzmann * params.temperatureMax);
141                             double of = HardinUtil2::gammaIncomplete(↩
                                    gammaArgument, gammaParameter);
142                             return of;
143                     };
144                     double cpeLo = 0;
145                     double ofLo = 0;
146                     double cpeHi = 1e-23;
147                     double ofHi = occupationFraction(cpeHi);
```

```
148                 //Find a bracket
149                 while (ofHi <= params.occupationFractionCutoff) {
150                         cpeLo = cpeHi;
151                         ofLo = ofHi;
152                         cpeHi *= 2;
153                         ofHi = occupationFraction(cpeHi);
154                 }
155                 //Narrow the bracket using bisection
156                 while (cpeHi - cpeLo > .01 || ofHi - ofLo > (1 - params.↩
                        occupationFractionCutoff)*1e-3) {
157                         double cpeMid = 0.5*(cpeLo + cpeHi);
158                         double ofMid = occupationFraction(cpeMid);
159                         if (ofMid <= params.occupationFractionCutoff) {
160                                 cpeLo = cpeMid;
161                                 ofLo = ofMid;
162                         }
163                         else {
164                                 cpeHi = cpeMid;
165                                 ofHi = ofMid;
166                         }
167                 }
168                 double cpeXMax = 0.5*(cpeLo + cpeHi);
169                 return cpeXMax;
170         };
171
172         double cpeIsochoricMax = getCpeXMax(params.cpeIsochoricExponent);
173         double cpeDilatativeMax = getCpeXMax(params.cpeDilatativeExponent)↩
                ;
174         //OK now we've got cpe upper bounds
175
176         //Compute the sinhPrefactor
177         double cpeResolutionBest = params.energyResolutionBest;
178         auto getSinhPrefactor = [cpeResolutionBest](const std::int64_t ↩
                nMax, const double cpeMax) {
179                 double sinhPrefactor = 50.0e-23;
180
```

```
181                 //Define the function we're trying to solve
182                 auto f = [cpeResolutionBest, nMax](const double a) {
183                         double arg = cpeResolutionBest * (nMax - 1) / a;
184                         double fVal = a*std::sinh(arg);
185                         double fPrime = -arg*std::cosh(arg) + std::sinh(arg↩
                            );
186                         std::pair<double, double> ret;
187                         ret.first = fVal;
188                         ret.second = fPrime;
189                         return ret;
190                 };
191                 //Use Newton's method to find the correct sinhPrefactor
192                 auto fPairSinhPrefactor = f(sinhPrefactor);
193                 while (std::abs(fPairSinhPrefactor.first - cpeMax) > ↩
                        cpeResolutionBest * 1e-3) {
194                         double fVal = fPairSinhPrefactor.first;
195                         double fPrime = fPairSinhPrefactor.second;
196                         double sinhPrefactorProposed = sinhPrefactor + (↩
                            cpeMax - fVal) / fPrime;
197                         if (sinhPrefactorProposed > 0) { sinhPrefactor = ↩
                            sinhPrefactorProposed; }
198                         else { sinhPrefactor *= 0.5; }
199                         fPairSinhPrefactor = f(sinhPrefactor);
200                 }
201                 return sinhPrefactor;
202         };
203         double sinhPrefactorIsochoric = getSinhPrefactor(params.↩
                nLevelsIsochoric, cpeIsochoricMax);
204         double sinhPrefactorDilatative = getSinhPrefactor(params.↩
                nLevelsDilatative, cpeDilatativeMax);
205
206         //Get the levels and bin edges
207         auto getLevels = [cpeResolutionBest](
208                 const std::int64_t iSubLevel,
209                 const std::int64_t nSubLevels,
210                 const double sinhPrefactor,
```

```
211                     double* cpeLevel,
212                     double* cpeBinLo,
213                     double* cpeBinHi
214                     ) {
215                     (*cpeLevel) = sinhPrefactor * std::sinh(cpeResolutionBest ↩
                            * iSubLevel / sinhPrefactor);
216
217                     double iSubLevelLo = iSubLevel - 0.5;
218                     if (iSubLevelLo < 0) { iSubLevelLo = 0; }
219                     double iSubLevelHi = iSubLevel + 0.5;
220                     if (iSubLevelHi > nSubLevels - 1) { iSubLevelHi = ↩
                            nSubLevels - 1; }
221
222                     (*cpeBinLo) = sinhPrefactor * std::sinh(cpeResolutionBest ↩
                            * iSubLevelLo / sinhPrefactor);
223                     (*cpeBinHi) = sinhPrefactor * std::sinh(cpeResolutionBest ↩
                            * iSubLevelHi / sinhPrefactor);
224             };
225             //Get the degeneracies
226             auto integrateDegeneracy = [this](
227                     const double cpeIsochoricLo,
228                     const double cpeIsochoricHi,
229                     const double cpeDilatativeLo,
230                     const double cpeDilatativeHi
231                     ) {
232
233                     double prefactor = params.numberOfStatesReference
234                             * std::pow(params.energyReference, -params.↩
                                    cpeIsochoricExponent - params.↩
                                    cpeDilatativeExponent)
235                             * std::tgamma(1 + params.cpeIsochoricExponent + ↩
                                    params.cpeDilatativeExponent)
236                             / (params.cpeIsochoricExponent*params.↩
                                    cpeDilatativeExponent* std::tgamma(params.↩
                                    cpeIsochoricExponent) * std::tgamma(params.↩
                                    cpeDilatativeExponent));
```

```
237
238                 double deltaIsochoric = std::pow(cpeIsochoricHi, params.↩
                        cpeIsochoricExponent) - std::pow(cpeIsochoricLo, params↩
                        .cpeIsochoricExponent);
239                 double deltaDilatative = std::pow(cpeDilatativeHi, params.↩
                        cpeDilatativeExponent) - std::pow(cpeDilatativeLo, ↩
                        params.cpeDilatativeExponent);
240
241                 double degeneracy = prefactor * deltaIsochoric * ↩
                        deltaDilatative;
242                 return degeneracy;
243             };
244
245             std::int64_t iLevel = 0;
246             for (std::int64_t iLevelIsochoric = 0; iLevelIsochoric < params.↩
                    nLevelsIsochoric; iLevelIsochoric++) {
247                 double cpeIsochoricLevel, cpeIsochoricLo, cpeIsochoricHi;
248                 getLevels(iLevelIsochoric, params.nLevelsIsochoric, ↩
                        sinhPrefactorIsochoric, &cpeIsochoricLevel, &↩
                        cpeIsochoricLo, &cpeIsochoricHi);
249
250                 for (std::int64_t iLevelDilatative = 0; iLevelDilatative <↩
                         params.nLevelsDilatative; iLevelDilatative++) {
251                     double cpeDilatativeLevel, cpeDilatativeLo, ↩
                            cpeDilatativeHi;
252                     getLevels(iLevelDilatative, params.↩
                            nLevelsDilatative, sinhPrefactorDilatative, &↩
                            cpeDilatativeLevel, &cpeDilatativeLo, &↩
                            cpeDilatativeHi);
253
254                     double cpe = cpeIsochoricLevel + cpeDilatativeLevel↩
                            ;
255
256                     cpeLevels(iLevel) = cpe;
257                     cpeIsochoricLevels(iLevel) = cpeIsochoricLevel;
258                     cpeDilatativeLevels(iLevel) = cpeDilatativeLevel;
```

```
259                             degeneracies(iLevel) = integrateDegeneracy(↩
                                    cpeIsochoricLo, cpeIsochoricHi, cpeDilatativeLo↩
                                    , cpeDilatativeHi);
260
261                             iLevel++;
262                     }
263             }
264
265             sort();
266         }
267
268         void setup(const Parameters params_) {
269                 params = params_;
270                 this->zoneSize = params.zoneSize;
271                 discretizeStates();
272         }
273         LevelsPower() {}
274         LevelsPower(const Parameters params_) { setup(params_); }
275         static LevelsPower vanilla() { return LevelsPower(Parameters::vanilla());↩
                 }
276 };
277
278 } //namespace Kinetics
279 } //namespace KMGEM
280
281 #endif /* KMGEM_KINETICS_LEVELS_H_ */
```

Listing B.9: KMGEM/Kinetics/EquationsOfState.h

```
#ifndef KMGEM_KINETICS_EQUATIONSOFSTATE_H_
#define KMGEM_KINETICS_EQUATIONSOFSTATE_H_

#include "KMGEM/Core.h"

namespace KMGEM {
namespace Kinetics {

class EquationsOfState {
public:
	class Parameters {
	public:
		//Note that "CPEM" stands for "Configurational Potential Energy (Molar)"
		//We define a reference state with a particular temperature, isochoric CPEM, and volumetric CPEM

		double temperatureReference; //K; temperature in reference state
		double cpemReference; //J/mol; molar CPE in reference state
		double cpemDilatativeReference;

		double shearModulusReference; //Pa; Shear modulus at the reference temperature and CPEM; that is, the max shear modulus at the reference temperature
		double shearModulusSlopeDebyeGruneisen; //Pa/K; Derivative of the shear modulus w.r.t. an instantaneous temperature change, sans relaxation, but including thermal expansion
		double shearModulusSlopeCpemIsothermal; //Pa/(J/mol); Derivative of the shear modulus w.r.t. molar CPE at a constant temperature

		double bulkModulusReference; //Pa; Bulk modulus at the reference temperature
		double bulkModulusSlopeDebyeGruneisen; //Pa/K; Derivative of the bulk modulus w.r.t. an instantaneous temperature change, sans
```

```
            relaxation, but including thermal expansion
26
27          double volumeMolarReference; //m^3/mol; Stress-free molar volume ↩
                at the reference temperature and volumetric CPEM
28          double volumetricThermalExpansionCoefficient; //per-Kelvin
29          double volumeMolarSlopeCpemDilatativeIsothermal;
30
31          static Parameters vanilla() {
32                  Parameters params;
33
34                  params.temperatureReference = 300;
35                  params.cpemReference = 0;
36                  params.cpemDilatativeReference = 0;
37
38                  params.shearModulusReference = 25e9; //Pa
39                  params.shearModulusSlopeDebyeGruneisen = -0.009115e9; //Pa↩
                        /K
40                  params.shearModulusSlopeCpemIsothermal = -1e9 / (5e-3 * ↩
                        KMGEM::Constants::joulePerMolPerEvPerAtom); //Pa/(J/mol↩
                        )
41
42                  params.bulkModulusReference = 114e9; //Pa
43                  params.bulkModulusSlopeDebyeGruneisen = -0.006815e9; //Pa/↩
                        Kelvin
44
45                  params.volumeMolarReference = 9.855e-6; //m^3/mol
46                  params.volumetricThermalExpansionCoefficient = 13e-6;
47                  params.volumeMolarSlopeCpemDilatativeIsothermal = 1e-6 / ↩
                        1000;
48
49                  return params;
50          }
51      };
52
53      //Input parameters
54      Parameters params;
```

```
55
56          //Local member functions
57          void setup(const Parameters params_) { params = params_; }
58          EquationsOfState() {}
59          EquationsOfState(const Parameters params_) { setup(params_); }
60          static EquationsOfState vanilla() { return EquationsOfState(Parameters::↩
                vanilla()); }
61
62          //Pa <- (Kelvin, J/mol)
63          double getShearModulus(const double temperature, const double cpem) const↩
                {
64                  double shearModulusReferenceInverse = 1.0 / params.↩
                        shearModulusReference;
65                  double multiplierTemperature = params.↩
                        shearModulusSlopeDebyeGruneisen * shearModulusReferenceInverse↩
                        ;
66                  double multiplierCpem = params.shearModulusSlopeCpemIsothermal * ↩
                        shearModulusReferenceInverse;
67                  double expArg = multiplierCpem * (cpem - params.cpemReference) + ↩
                        multiplierTemperature * (temperature - params.↩
                        temperatureReference);
68                  double shearModulus = params.shearModulusReference * std::exp(↩
                        expArg);
69                  return shearModulus; //Pa
70          }
71
72          //Pa <- (Kelvin, J/mol)
73          double getBulkModulus(const double temperature) const {
74                  double bulkModulusReferenceInverse = 1.0 / params.↩
                        bulkModulusReference;
75                  double multiplierTemperature = params.↩
                        bulkModulusSlopeDebyeGruneisen * bulkModulusReferenceInverse;
76                  double expArg = multiplierTemperature * (temperature - params.↩
                        temperatureReference);
77                  double bulkModulus = params.bulkModulusReference * std::exp(expArg↩
                        );
```

```
78                return bulkModulus; //Pa
79        }
80
81        //m^3/mol <- (Kelvin, J/mol)
82        double getVolumeMolar(const double temperature, const double ↩
              cpemDilatative) const {
83                double volumeMolarReferenceInverse = 1.0 / params.↩
                      volumeMolarReference;
84                double multiplierTemperature = params.↩
                      volumetricThermalExpansionCoefficient;
85                double multiplierCpem = params.↩
                      volumeMolarSlopeCpemDilatativeIsothermal * ↩
                      volumeMolarReferenceInverse;
86                double expArg = multiplierCpem * (cpemDilatative - params.↩
                      cpemDilatativeReference) + multiplierTemperature * (↩
                      temperature - params.temperatureReference);
87                double volumeMolar = params.volumeMolarReference * std::exp(expArg↩
                      );
88                return volumeMolar;
89        }
90 };
91
92 } //namespace Kinetics
93 } //namespace KMGEM
94
95 #endif /* KMGEM_KINETICS_EQUATIONSOFSTATE_H_ */
```

Listing B.10: KMGEM/Kinetics/IntegralOrientation.h

```
#ifndef KMGEM_KINETICS_INTEGRALORIENTATION_H_
#define KMGEM_KINETICS_INTEGRALORIENTATION_H_

#include "KMGEM/Core.h"

namespace KMGEM {
namespace Kinetics {

//Int exp[<xi1,xi2,-xi1-xi2>.<exx,eyy,ezz>(r)] dr
//  e = Q(r)^-1.E.Q(r)
//  E_xy = 1, else 0
class IntegralOrientation {
public:
        class Parameters {
        public:
                std::int64_t nSamples;

                static Parameters vanilla() {
                        Parameters params;
                        params.nSamples = std::int64_t(1.2e5);
                        return params;
                }

                static Parameters fromIni(const HardinUtil2::Ini &ini) {
                        Parameters params = Parameters::vanilla();
                        ini.getVar<std::int64_t>("IntegralOrientation/nSamples", ↩
                            &(params.nSamples));
                        return params;
                }

                std::size_t hash() const {
                        std::size_t ret = 0;
                        HardinUtil2::hashCombine(&ret,
                                nSamples
                        );
```

```
35                      return ret;
36              }
37
38              bool operator==(const Parameters &other) const {
39                      bool ret = true;
40                      ret = ret && (nSamples == other.nSamples);
41                      return ret;
42              }
43
44  #ifdef KMGEM_ENABLE_HDF
45              void save(const H5::Group &target) const {
46                      HardinUtil2::saveScalar<std::int64_t>(&nSamples, "nSamples↩
                            ", target);
47              }
48
49              void load(const H5::Group &target) {
50                      nSamples = HardinUtil2::loadScalar<std::int64_t>("nSamples↩
                            ", target);
51              }
52  #endif
53      };
54
55      Parameters params;
56      //Discretization of Rodrigues space
57      std::int64_t nRodriguesSamples;
58      std::shared_ptr<std::array<double, 3>> orientationRodriguesSamples;
59      double orientationRodriguesSampleVolume;
60
61      //Evaluate the Rodrigues integration factor
62      static double getRodriguesFactor(const double* orientationRodrigues) {
63              auto rSqr = [orientationRodrigues](std::int8_t i) { return ↩
                    orientationRodrigues[i] * orientationRodrigues[i]; };
64              double sqrtRodriguesFactor = 0.6366197723675813430755350534900 57 /↩
                     (1 + rSqr(0) + rSqr(1) + rSqr(2));
65              double rodriguesFactor = sqrtRodriguesFactor * sqrtRodriguesFactor↩
                    ;
```

```
66              return rodriguesFactor;
67      }
68
69      //Break the fundamental zone in Rodrigues space into voxels
70      void partitionRSpace() {
71              std::int64_t nQuadratureSide = std::int64_t(std::ceil(std::cbrt(↩
                    double(params.nSamples))));
72              nRodriguesSamples = nQuadratureSide*nQuadratureSide*↩
                    nQuadratureSide;
73              orientationRodriguesSamples.reset(new std::array<double, 3>[↩
                    nRodriguesSamples], std::default_delete<std::array<double, ↩
                    3>[]>());
74
75              double du = 2.0 / nQuadratureSide;
76              double baseVal = -1.0 + 0.5*du;
77              const double sqrt2Over2 = std::sqrt(2) / 2.0;
78
79              std::int64_t I = 0;
80              for (std::int64_t i1 = 0; i1 < nQuadratureSide; i1++) {
81                      double u = baseVal + du*i1;
82                      for (std::int64_t i2 = 0; i2 < nQuadratureSide; i2++) {
83                              double v = baseVal + du*i2;
84                              double r1 = sqrt2Over2*u - sqrt2Over2*v;
85                              double r2 = sqrt2Over2*u + sqrt2Over2*v;
86                              for (std::int64_t i3 = 0; i3 < nQuadratureSide; i3↩
                                    ++) {
87                                      double w = baseVal + du*i3;
88                                      double r3 = w;
89                                      orientationRodriguesSamples.get()[I++] = { {↩
                                            r1,r2,r3 } };
90                              }
91                      }
92              }
93              orientationRodriguesSampleVolume = du*du*du;
94      }
95
```

```
96      //returns Q^-1 . e^hat . Q
97      //   e^hat_xy = e^hat_yx = 1/2; else 0
98      static void getStrainOriented(
99              const double* orientationRodrigues,
100             double* strainOriented
101     ) {
102             auto r = [orientationRodrigues](std::int8_t i) { return ↩
                    orientationRodrigues[i]; };
103             auto rSqr = [orientationRodrigues](std::int8_t i) { return ↩
                    orientationRodrigues[i] * orientationRodrigues[i]; };
104             auto rCub = [orientationRodrigues](std::int8_t i) { return ↩
                    orientationRodrigues[i] * orientationRodrigues[i] * ↩
                    orientationRodrigues[i]; };
105
106             double rotMatDenom = 1 + rSqr(0) + rSqr(1) + rSqr(2);
107             double rotMatDenomInv = 1.0 / rotMatDenom;
108             double rotMatDenomSqr = rotMatDenom * rotMatDenom;
109             double rotMatDenomSqrInv = rotMatDenomInv * rotMatDenomInv;
110
111             strainOriented[0] = -2 * rotMatDenomSqrInv*(r(0)*r(1) + r(2))*(↩
                    rotMatDenom - 2 * (1 + rSqr(0)));
112             strainOriented[1] = -2 * rotMatDenomSqrInv*(r(0)*r(1) - r(2))*(↩
                    rotMatDenom - 2 * (1 + rSqr(1)));
113             strainOriented[2] = 4 * rotMatDenomSqrInv*(r(1) + r(0)*r(2))*(-r↩
                    (0) + r(1)*r(2));
114             strainOriented[3] = 2 * rotMatDenomSqrInv*(2 * rCub(1) + r(0)*r(2)↩
                    *(4 - rotMatDenom + 4 * rSqr(1)) - r(1)*(rotMatDenom + 2 * (-1↩
                     + rSqr(0) + rSqr(2))));
115             strainOriented[4] = 2 * rotMatDenomSqrInv*(-2 * rCub(0) + r(1)*r↩
                    (2)*(4 - rotMatDenom + 4 * rSqr(0)) + r(0)*(rotMatDenom + 2 * ↩
                    (-1 + rSqr(1) + rSqr(2))));
116             strainOriented[5] = rotMatDenomSqrInv*(rotMatDenomSqr - 2 * ↩
                    rotMatDenom*(2 + rSqr(0) + rSqr(1)) + 4 * (1 + rSqr(0) + rSqr↩
                    (1) + 2 * rSqr(0)*rSqr(1) - rSqr(2)));
117     }
118
```

```
119         double getIntegrandOriented(
120                 const double* xi,
121                 const double* orientationRodrigues,
122                 double* strainOriented
123         ) const {
124                 getStrainOriented(orientationRodrigues, strainOriented);
125                 double exponentialArgument = xi[0] * strainOriented[0]
126                         + xi[1] * strainOriented[1]
127                         - (xi[0] + xi[1]) * strainOriented[2];
128                 double integrand = std::exp(exponentialArgument);
129                 return integrand;
130         }
131
132         double getSummandOriented(
133                 const double* xi,
134                 const double* orientationRodrigues,
135                 double* summandOrientedTensor
136         ) const {
137                 double integrand = getIntegrandOriented(xi, orientationRodrigues, ↩
                        summandOrientedTensor);
138                 double rodriguesFactor = getRodriguesFactor(orientationRodrigues);
139                 double summand = integrand * rodriguesFactor * ↩
                        orientationRodriguesSampleVolume;
140                 for (std::int8_t i = 0; i < 6; i++) { summandOrientedTensor[i] *= ↩
                        summand; }
141                 return summand;
142         }
143
144         double integrate(
145                 const double* xi,
146                 double* integralTensor,
147                 const std::int8_t nThreads = 1,
148                 const bool verbose = false
149         ) const {
150                 auto futureLambda = [&](const std::int8_t iThread) {
151                         std::pair<double, std::array<double, 6>> integralThread;
```

```
152                     integralThread.first = 0;
153                     integralThread.second = {};
154
155                     double summandOriented = 0;
156                     std::array<double, 6> summandOrientedTensor;
157                     for (std::int64_t iSample = iThread; iSample < ↩
                            nRodriguesSamples; iSample += nThreads) {
158                         summandOriented = getSummandOriented(xi, ↩
                                orientationRodriguesSamples.get()[iSample].data↩
                                (), summandOrientedTensor.data());
159                         integralThread.first += summandOriented;
160                         for (std::int8_t i = 0; i < 6; i++) { ↩
                                integralThread.second[i] += ↩
                                summandOrientedTensor[i]; }
161                         if (verbose && iThread == 0 && iSample % (nThreads ↩
                                * 100000) == 0) { std::cout << "↩
                                IntegralOrientation:␣" << (iSample + 1) << "/" ↩
                                << nRodriguesSamples << std::endl; }
162                     }
163
164                     return integralThread;
165             };
166
167             double integral = 0;
168             if (nThreads == 1) {
169                     auto integralThread = futureLambda(0);
170                     integral = integralThread.first;
171                     for (std::int8_t i = 0; i < 6; i++) { integralTensor[i] = ↩
                            integralThread.second[i]; }
172             }
173             else {
174                     for (std::int8_t i = 0; i < 6; i++) { integralTensor[i] = ↩
                            0; }
175                     std::vector<std::future<std::pair<double, std::array<↩
                            double, 6>>>> futures;
176                     for (std::int8_t iThread = 0; iThread < nThreads; iThread↩
```

```
                        ++) { futures.push_back(std::async(futureLambda, ↩
                        iThread)); }
177                     for (std::future<std::pair<double, std::array<double, 6>>>↩
                             &future : futures) {
178                         auto integralThread = future.get();
179                         integral += integralThread.first;
180                         for (std::int8_t i = 0; i < 6; i++) { ↩
                                integralTensor[i] += integralThread.second[i]; ↩
                                }
181                     }
182             }
183             return integral;
184     }
185
186     void setup(const Parameters parameters) {
187             params = parameters;
188             partitionRSpace();
189     }
190     IntegralOrientation() {}
191     IntegralOrientation(const Parameters parameters) { setup(parameters); }
192
193     //Boilerplate
194 #ifdef KMGEM_ENABLE_HDF
195     void save(const H5::Group &target) const {
196             params.save(target.createGroup("Parameters"));
197             HardinUtil2::saveScalar<std::int64_t>(&nRodriguesSamples, "↩
                    nRodriguesSamples", target);
198             HardinUtil2::saveArray<double, 2>(
199                     orientationRodriguesSamples.get()->data(),
200                     { { hsize_t(nRodriguesSamples),hsize_t(3) } },
201                     "orientationRodriguesSamples",
202                     target
203                     );
204             HardinUtil2::saveScalar<double>(&orientationRodriguesSampleVolume,↩
                         "orientationRodriguesSampleVolume", target);
205     }
```

```
206
207         void load(const H5::Group &target) {
208                 params.load(target.openGroup("Parameters"));
209                 nRodriguesSamples = HardinUtil2::loadScalar<std::int64_t>("↩
                        nRodriguesSamples", target);
210                 orientationRodriguesSamples.reset(new std::array<double, 3>[↩
                        nRodriguesSamples], std::default_delete<std::array<double, ↩
                        3>[]>());
211                 HardinUtil2::loadArray<double, 2>(
212                         target,
213                         "orientationRodriguesSamples",
214                         orientationRodriguesSamples.get()->data()
215                         );
216                 orientationRodriguesSampleVolume = HardinUtil2::loadScalar<double↩
                        >("orientationRodriguesSampleVolume", target);
217         }
218 #endif
219         std::size_t hash() const { return params.hash(); }
220         bool operator==(const IntegralOrientation &other) const { return other.↩
                params == params; }
221         static IntegralOrientation vanilla() { return IntegralOrientation(↩
                Parameters::vanilla()); }
222         static IntegralOrientation fromIni(const HardinUtil2::Ini &ini) { return ↩
                IntegralOrientation(Parameters::fromIni(ini)); }
223 };
224
225 class TableIntegralOrientation {
226 public:
227         class Parameters {
228         public:
229                 IntegralOrientation integralOrientation;
230
231                 double xiMax;
232                 std::int64_t nXi;
233                 std::int64_t nPsiInitial;
234                 std::int64_t nThetaInitial;
```

```
235
236             static Parameters vanilla(const IntegralOrientation &↩
                    integralOrientation) {
237                 Parameters params;
238                 params.integralOrientation = integralOrientation;
239
240                 params.xiMax = 100;
241                 params.nXi = 500;
242                 params.nPsiInitial = 180;
243                 params.nThetaInitial = 180;
244
245                 return params;
246             }
247
248             static Parameters fromIni(
249                 const HardinUtil2::Ini &ini,
250                 const IntegralOrientation &integralOrientation
251             ) {
252                 Parameters params = Parameters::vanilla(↩
                        integralOrientation);
253
254                 ini.getVar<double>("TableIntegralOrientation/xiMax", &(↩
                        params.xiMax));
255                 ini.getVar<std::int64_t>("TableIntegralOrientation/nXi", ↩
                        &(params.nXi));
256                 ini.getVar<std::int64_t>("TableIntegralOrientation/↩
                        nPsiInitial", &(params.nPsiInitial));
257                 ini.getVar<std::int64_t>("TableIntegralOrientation/↩
                        nThetaInitial", &(params.nThetaInitial));
258
259                 return params;
260             }
261
262             std::size_t hash() const {
263                 std::size_t ret = 0;
264                 HardinUtil2::hashCombine(&ret,
```

```
265                     integralOrientation.hash(),
266                     xiMax,
267                     nXi,
268                     nPsiInitial,
269                     nThetaInitial
270             );
271             return ret;
272         }
273
274         std::string getCacheFilepath(const std::string cacheDirectory) ↩
                const {
275             std::size_t hashCode = hash();
276             std::string hashString = HardinUtil2::intToHexPadded(↩
                    hashCode);
277             std::string filePath = HardinUtil2::cleanupDirectoryPath(↩
                    cacheDirectory)
278                     + "tableIntegralOrientation_"
279                     + hashString
280                     + ".h5";
281             return filePath;
282         }
283
284         bool operator==(const Parameters &other) const {
285             bool ret = true;
286
287             ret = ret && (integralOrientation == other.↩
                    integralOrientation);
288
289             ret = ret && (xiMax == other.xiMax);
290             ret = ret && (nXi == other.nXi);
291             ret = ret && (nPsiInitial == other.nPsiInitial);
292             ret = ret && (nThetaInitial == other.nThetaInitial);
293
294             return ret;
295         }
296
```

```
297 #ifdef KMGEM_ENABLE_HDF
298                 void save(const H5::Group &target) const {
299                         integralOrientation.save(target.createGroup("↩
                            IntegralOrientation"));
300
301                         HardinUtil2::saveScalar<double>(&xiMax, "xiMax", target);
302                         HardinUtil2::saveScalar<std::int64_t>(&nXi, "nXi", target)↩
                            ;
303                         HardinUtil2::saveScalar<std::int64_t>(&nPsiInitial, "↩
                            nPsiInitial", target);
304                         HardinUtil2::saveScalar<std::int64_t>(&nThetaInitial, "↩
                            nThetaInitial", target);
305                 }
306
307                 void load(const H5::Group &target) {
308                         integralOrientation.load(target.openGroup("↩
                            IntegralOrientation"));
309
310                         xiMax = HardinUtil2::loadScalar<double>("xiMax", target);
311                         nXi = HardinUtil2::loadScalar<std::int64_t>("nXi", target)↩
                            ;
312                         nPsiInitial = HardinUtil2::loadScalar<std::int64_t>("↩
                            nPsiInitial", target);
313                         nThetaInitial = HardinUtil2::loadScalar<std::int64_t>("↩
                            nThetaInitial", target);
314                 }
315 #endif
316         };
317
318         Parameters params;
319         IntegralOrientation integralOrientation; //Alias
320         //Discretization and tabulated data
321         std::int64_t nPsi;
322         std::shared_ptr<double> psiVector;
323         std::shared_ptr<double> scalarTableBuffer;
324         std::shared_ptr<std::array<double,6>> tensorTableBuffer;
```

```
325         Eigen::Matrix<double, 2, 2> xiToPsi;
326         Eigen::Matrix<double, 2, 2> psiToXi;
327
328         std::size_t hash() const { return params.hash(); }
329         bool operator==(const TableIntegralOrientation &other) const { return ↩
                other.params == params; }
330
331         double getIntegralPsi(
332                 const double* psi,
333                 double* integralTensor) const {
334
335                 Eigen::Map<Eigen::Matrix<double, -1, -1, Eigen::RowMajor>> ↩
                        scalarTable
336                         = Eigen::Map<Eigen::Matrix<double, -1, -1, Eigen::RowMajor↩
                                >>(scalarTableBuffer.get(), nPsi, nPsi);;
337                 Eigen::Map<Eigen::Matrix<std::array<double, 6>, -1, -1, Eigen::↩
                        RowMajor>> tensorTable
338                         = Eigen::Map<Eigen::Matrix<std::array<double, 6>, -1, -1, ↩
                                Eigen::RowMajor>>(tensorTableBuffer.get(), nPsi, nPsi);
339                 std::int64_t iPsiX = std::upper_bound(psiVector.get(), psiVector.↩
                        get() + nPsi, psi[0]) - psiVector.get() - 1;
340                 std::int64_t iPsiY = std::upper_bound(psiVector.get(), psiVector.↩
                        get() + nPsi, psi[1]) - psiVector.get() - 1;
341                 double tX = (psi[0] - psiVector.get()[iPsiX]) / (psiVector.get()[↩
                        iPsiX + 1] - psiVector.get()[iPsiX]);
342                 double tY = (psi[1] - psiVector.get()[iPsiY]) / (psiVector.get()[↩
                        iPsiY + 1] - psiVector.get()[iPsiY]);
343                 double integralPsi = (1 - tX)*(1 - tY)*scalarTable(iPsiX, iPsiY) +
344                         (tX)*(1 - tY)*scalarTable(iPsiX + 1, iPsiY) +
345                         (tX)*(tY)*scalarTable(iPsiX + 1, iPsiY + 1) +
346                         (1 - tX)*(tY)*scalarTable(iPsiX, iPsiY + 1);
347                 for (std::int8_t i = 0; i < 6; i++) {
348                         integralTensor[i] = (1 - tX)*(1 - tY)*tensorTable(iPsiX, ↩
                                iPsiY)[i] +
349                                 (tX)*(1 - tY)*tensorTable(iPsiX + 1, iPsiY)[i] +
350                                 (tX)*(tY)*tensorTable(iPsiX + 1, iPsiY + 1)[i] +
```

```
351                         (1 - tX)*(tY)*tensorTable(iPsiX, iPsiY + 1)[i];
352             }
353             return integralPsi;
354         }
355
356         double getIntegral(
357                 const double* xi,
358                 double* integralTensor) const {
359                 Eigen::Map<const Eigen::Vector2d> xiEigen(xi);
360                 Eigen::Vector2d psi = xiToPsi * xiEigen;
361                 //If we're querying a point in-bounds then interpolate
362                 double integralScalar;
363                 if (psi[0] > psiVector.get()[0] && psi[0] < psiVector.get()[nPsi↩
                    -1] &&
364                     psi[1] > psiVector.get()[0] && psi[1] < psiVector.get()[↩
                        nPsi-1]) {
365                     integralScalar = getIntegralPsi(psi.data(), integralTensor↩
                        );
366                 }
367                 else { //If we're querying a point out-of-bounds, then integrate ↩
                    from scratch
368                     integralScalar = params.integralOrientation.integrate(
369                             xi,
370                             integralTensor,
371                             1,
372                             false
373                     );
374                 }
375                 return integralScalar;
376         }
377
378         void allocateMemory() {
379                 nPsi = 2 * params.nXi + 1;
380
381                 scalarTableBuffer.reset(new double[nPsi*nPsi], std::default_delete↩
                    <double[]>());
```

```
382             tensorTableBuffer.reset(new std::array<double, 6>[nPsi*nPsi], std↩
                    ::default_delete<std::array<double,6>[]>());
383             psiVector.reset(new double[nPsi], std::default_delete<double[]>())↩
                    ;
384         }
385
386         void calculatePsiXiConversions() {
387             psiToXi(0, 0) = -0.70710678118654752440084436210484 9;
388             psiToXi(0, 1) = -0.40824829046386301636621401245098 2;
389             psiToXi(1, 0) = 0.70710678118654752440084436210484 9;
390             psiToXi(1, 1) = -0.40824829046386301636621401245098 2;
391
392             xiToPsi(0, 0) = -0.70710678118654752440084436210484 9;
393             xiToPsi(0, 1) = 0.70710678118654752440084436210484 9;
394             xiToPsi(1, 0) = -1.22474487139158904909864203735295;
395             xiToPsi(1, 1) = -1.22474487139158904909864203735295;
396         }
397
398         void discretizePsi() {
399             double sinhPrefactor = params.xiMax / std::sinh(8.0);
400             for (std::int64_t iPsi = -params.nXi; iPsi <= params.nXi; iPsi++) ↩
                    { psiVector.get()[iPsi + params.nXi] = sinhPrefactor * std::↩
                    sinh(8 * double(iPsi) / double(params.nXi)); }
401         }
402
403         void discretizeInitial(
404             std::vector<double> &rhoVector,
405             std::vector<double> &thetaVector
406         ) const {
407             rhoVector.resize(params.nPsiInitial);
408             double sinhPrefactor = std::sqrt(2.0)*params.xiMax / std::sinh↩
                    (8.0);
409             for (std::int64_t iPsi = 0; iPsi < params.nPsiInitial; iPsi++) { ↩
                    rhoVector[iPsi] = sinhPrefactor * std::sinh(8 * double(iPsi) /↩
                     double(params.nPsiInitial - 1)); }
410
```

```
411             thetaVector.resize(params.nThetaInitial + 1);
412             double thetaStep = Constants::TWOPI / params.nThetaInitial;
413             for (std::int64_t i = 0; i < params.nThetaInitial; i++) { ↩
                    thetaVector[i] = thetaStep * i; }
414             thetaVector.back() = Constants::TWOPI;
415         }
416
417         void tabulateInitial(
418             const std::vector<double> &rhoVector,
419             const std::vector<double> &thetaVector,
420             Eigen::Matrix<double, -1, -1, Eigen::RowMajor> &scalarTableInitial↩
                    ,
421             Eigen::Matrix<std::array<double, 6>, -1, -1, Eigen::RowMajor> &↩
                    tensorTableInitial,
422             const std::int8_t nThreads,
423             const bool verbose
424         ) const {
425             auto tabulationLambda = [&](const std::int8_t iThread) {
426                 for (std::int64_t iTable = iThread; iTable < ↩
                        scalarTableInitial.size(); iTable += nThreads) {
427                     //iTable = iTheta + nTheta * iPsi;
428                     std::int64_t iTheta = iTable % thetaVector.size();
429                     std::int64_t iPsi = iTable / thetaVector.size();
430
431                     double theta = thetaVector[iTheta];
432                     double rho = rhoVector[iPsi];
433
434                     Eigen::Vector2d psi;
435                     psi[0] = std::cos(theta) * rho;
436                     psi[1] = std::sin(theta) * rho;
437                     Eigen::Vector2d xi = psiToXi * psi;
438
439                     scalarTableInitial(iPsi, iTheta) = params.↩
                            integralOrientation.integrate(
440                                 xi.data(),
441                                 tensorTableInitial(iPsi, iTheta).data(),
```

```
442                                 1,
443                                 false
444                             );
445
446                             scalarTableInitial(iPsi, iTheta) = std::asinh(↩
                                    scalarTableInitial(iPsi, iTheta));
447                             for (std::int8_t i = 0; i < 6; i++) { ↩
                                    tensorTableInitial(iPsi, iTheta)[i] = std::↩
                                    asinh(tensorTableInitial(iPsi, iTheta)[i]); }
448
449                             if (verbose && iThread == 0 && iTable % (nThreads *↩
                                     100) == 0) { std::cout << "↩
                                    TableIntegralOrientation:tabulateInitial:␣" << ↩
                                    (iTable + 1) << "/" << scalarTableInitial.size↩
                                    () << std::endl; }
450                     }
451             };
452
453             if (nThreads == 1) { tabulationLambda(0); }
454             else {
455                     std::vector<std::thread> threads;
456                     for (std::int8_t iThread = 0; iThread < nThreads; iThread↩
                            ++) { threads.push_back(std::thread(tabulationLambda, ↩
                            iThread)); }
457                     for (std::thread &thread : threads) { thread.join(); }
458             }
459     }
460
461     void interpolateInitial(
462             const std::vector<double> &rhoVector,
463             const std::vector<double> &thetaVector,
464             const Eigen::Matrix<double, -1, -1, Eigen::RowMajor> &↩
                    scalarTableInitial,
465             const Eigen::Matrix<std::array<double, 6>, -1, -1, Eigen::RowMajor↩
                    > &tensorTableInitial,
466             const std::int8_t nThreads,
```

```
467             const bool verbose
468         ) {
469
470             Eigen::Map<Eigen::Matrix<double, -1, -1, Eigen::RowMajor>> ↩
                    scalarTable
471                 = Eigen::Map<Eigen::Matrix<double, -1, -1, Eigen::RowMajor↩
                        >>(scalarTableBuffer.get(), nPsi, nPsi);;
472             Eigen::Map<Eigen::Matrix<std::array<double, 6>, -1, -1, Eigen::↩
                    RowMajor>> tensorTable
473                 = Eigen::Map<Eigen::Matrix<std::array<double, 6>, -1, -1, ↩
                        Eigen::RowMajor>>(tensorTableBuffer.get(), nPsi, nPsi);
474
475             auto tabulationLambda = [&](const std::int8_t iThread) {
476                 for (std::int64_t iTable = iThread; iTable < scalarTable.↩
                        size(); iTable += nThreads) {
477                     //iTable = iPsiY + nXi * iPsiX;
478                     std::int64_t iPsiX = iTable / nPsi;
479                     std::int64_t iPsiY = iTable % nPsi;
480                     double psiX = psiVector.get()[iPsiX];
481                     double psiY = psiVector.get()[iPsiY];
482
483                     double theta = std::atan2(psiY, psiX);
484                     double rho = std::sqrt(psiX*psiX + psiY*psiY);
485                     while (theta >= Constants::TWOPI) { theta -= ↩
                            Constants::TWOPI; }
486                     while (theta < 0) { theta += Constants::TWOPI; }
487                     //OK, so theta is in [0,2Pi)
488                     //Figure out which samples in theta bracket the ↩
                            desired sample
489                     double angleStep = thetaVector[1];
490                     double iThetaDbl = theta / angleStep;
491                     std::int64_t iTheta = std::int64_t(iThetaDbl);
492                     if (iTheta < 0) { iTheta = 0; }
493                     if (iTheta > thetaVector.size() - 2) { iTheta = ↩
                            thetaVector.size() - 2; }
494                     double xTheta = iThetaDbl - iTheta;
```

```
495                     //Figure out which samples in rho bracket the ↩
                            desired sample
496                     std::int64_t iRho = std::upper_bound(rhoVector.↩
                            begin(), rhoVector.end(), rho) - rhoVector.↩
                            begin() - 1;
497                     if (iRho > rhoVector.size() - 2) { iRho = rhoVector↩
                            .size() - 2; }
498                     double xRho = (rho - rhoVector[iRho]) / (rhoVector[↩
                            iRho + 1] - rhoVector[iRho]);
499
500                     scalarTable(iPsiX, iPsiY) = std::sinh(
501                             (1 - xRho)*(1 - xTheta)*scalarTableInitial(↩
                                iRho, iTheta)
502                             + (xRho)*(1 - xTheta)*scalarTableInitial(↩
                                iRho + 1, iTheta)
503                             + (xRho)*(xTheta)*scalarTableInitial(iRho + ↩
                                1, iTheta + 1)
504                             + (1 - xRho)*(xTheta)*scalarTableInitial(↩
                                iRho, iTheta + 1)
505                     );
506
507                     for (std::int8_t i = 0; i < 6; i++) {
508                             tensorTable(iPsiX, iPsiY)[i] = std::sinh(
509                                     (1 - xRho)*(1 - xTheta)*↩
                                        tensorTableInitial(iRho, iTheta)[↩
                                        i]
510                                     + (xRho)*(1 - xTheta)*↩
                                        tensorTableInitial(iRho + 1, ↩
                                        iTheta)[i]
511                                     + (xRho)*(xTheta)*tensorTableInitial(↩
                                        iRho + 1, iTheta + 1)[i]
512                                     + (1 - xRho)*(xTheta)*↩
                                        tensorTableInitial(iRho, iTheta +↩
                                         1)[i]
513                             );
514                     }
```

```
515
516                         if (verbose && iThread == 0 && iTable % (nThreads *↩
                                10000) == 0) { std::cout << "↩
                               TableIntegralOrientation:interpolateInitial:␣" ↩
                               << (iTable + 1) << "/" << scalarTable.size() <<↩
                                std::endl; }
517                     }
518             };
519
520             if (nThreads == 1) { tabulationLambda(0); }
521             else {
522                     std::vector<std::thread> threads;
523                     for (std::int8_t iThread = 0; iThread < nThreads; iThread↩
                            ++) { threads.push_back(std::thread(tabulationLambda, ↩
                            iThread)); }
524                     for (std::thread &thread : threads) { thread.join(); }
525             }
526     }
527
528     void alias() { integralOrientation = params.integralOrientation; }
529
530     void tabulateFromScratch(
531             const Parameters parameters,
532             const std::int8_t nThreads = 1,
533             const bool verbose = false
534     ) {
535             params = parameters;
536             alias();
537             allocateMemory();
538             calculatePsiXiConversions();
539             discretizePsi();
540
541             std::vector<double> rho(params.nPsiInitial);
542             std::vector<double> theta(params.nThetaInitial);
543             discretizeInitial(rho, theta);
544
```

```
545                 Eigen::Matrix<double, -1, -1, Eigen::RowMajor> scalarTableInitial(↩
                        rho.size(), theta.size());
546                 Eigen::Matrix<std::array<double, 6>, -1, -1, Eigen::RowMajor> ↩
                        tensorTableInitial(rho.size(), theta.size());
547                 tabulateInitial(
548                         rho,
549                         theta,
550                         scalarTableInitial,
551                         tensorTableInitial,
552                         nThreads,
553                         verbose
554                 );
555
556                 interpolateInitial(
557                         rho,
558                         theta,
559                         scalarTableInitial,
560                         tensorTableInitial,
561                         nThreads,
562                         verbose
563                 );
564         }
565
566 #ifdef KMGEM_ENABLE_HDF
567
568         void save(const H5::Group &target) const {
569                 params.save(target.createGroup("Parameters"));
570
571                 HardinUtil2::saveArray<double, 1>(
572                         psiVector.get(),
573                         { {hsize_t(nPsi)} },
574                         "psiVector",
575                         target
576                         );
577
578                 HardinUtil2::saveArray<double, 2>(
```

```
579                     scalarTableBuffer.get(),
580                     { {hsize_t(nPsi), hsize_t(nPsi)} },
581                     "scalarTable",
582                     target
583                     );
584
585             HardinUtil2::saveArray<double, 3>(
586                     tensorTableBuffer.get()->data(),
587                     { { hsize_t(nPsi), hsize_t(nPsi), hsize_t(6) } },
588                     "tensorTable",
589                     target
590                     );
591
592             HardinUtil2::saveArray<double, 1>(
593                     xiToPsi.data(),
594                     { { hsize_t(4) } },
595                     "xiToPsi",
596                     target
597                     );
598
599             HardinUtil2::saveArray<double, 1>(
600                     psiToXi.data(),
601                     { { hsize_t(4) } },
602                     "psiToXi",
603                     target
604                     );
605     }
606
607     bool load(const H5::Group &target) {
608             params.load(target.openGroup("Parameters"));
609             alias();
610             allocateMemory();
611
612             HardinUtil2::loadArray<double, 1>(target, "psiVector", psiVector.↩
                    get());
613             HardinUtil2::loadArray<double, 2>(target, "scalarTable", ↩
```

```
                scalarTableBuffer.get());
614             HardinUtil2::loadArray<double, 3>(target, "tensorTable", ↩
                    tensorTableBuffer.get()->data());
615             HardinUtil2::loadArray<double, 1>(target, "xiToPsi", xiToPsi.data↩
                    ());
616             HardinUtil2::loadArray<double, 1>(target, "psiToXi", psiToXi.data↩
                    ());
617
618             return true;
619         }
620
621         bool load(const std::string filepath) {
622             try {
623                 H5::H5File inFile(filepath, H5F_ACC_RDONLY);
624                 bool ret = load(inFile);
625                 inFile.close();
626                 return ret;
627             }
628             catch (H5::FileIException &fileIException) { return false; }
629         }
630
631         bool loadCached(const Parameters params_, const std::string ↩
                cacheDirectory) {
632             std::string cacheFilepath = params_.getCacheFilepath(↩
                    cacheDirectory);
633             return load(cacheFilepath);
634         }
635
636         void save(const std::string filepath) const {
637             H5::H5File outFile(filepath, H5F_ACC_TRUNC);
638             save(outFile);
639             outFile.close();
640         }
641
642         void saveToCache(const std::string cacheDirectory) const {
643             std::string cacheFilepath = params.getCacheFilepath(cacheDirectory↩
```

```
                    );
644                 save(cacheFilepath);
645         }
646
647         void tabulateAndCache(
648                 const Parameters params_,
649                 const std::string cacheDirectory,
650                 const std::int8_t nThreads = 1,
651                 const bool verbose = false
652         ) {
653                 tabulateFromScratch(params_, nThreads, verbose);
654                 saveToCache(cacheDirectory);
655         }
656
657         void loadOrTabulateAndCache(
658                 const Parameters params_,
659                 const std::string cacheDirectory,
660                 const std::int8_t nThreads = 1,
661                 const bool verbose = false
662         ) {
663                 if (!loadCached(params_, cacheDirectory)) { tabulateAndCache(↩
                        params_, cacheDirectory, nThreads, verbose); }
664         }
665
666 #endif //KMGEM_ENABLE_HDF
667 };
668
669 } //namespace Kinetics
670 } //namespace KMGEM
671
672 #endif /* KMGEM_KINETICS_INTEGRALORIENTATION_H_ */
```

Listing B.11: KMGEM/Homogeneous/EvolveStress.h

```
#ifndef KMGEM_HOMOGENEOUS_EVOLVESTRESS_H_
#define KMGEM_HOMOGENEOUS_EVOLVESTRESS_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "KMGEM/Homogeneous/RateMatrixRelaxation.h"
#include "KMGEM/Homogeneous/RateMatricesShear.h"

namespace KMGEM {
namespace Homogeneous {

class EvolveStress {
public:
	class Parameters {
	public:
		RateMatrixRelaxationFactory rateMatrixRelaxationFactory;
		RateMatrixShearFactory rateMatrixShearFactory;
		std::function<double(const double)> temperatureFunction;
		std::function<void(const double, double*)> stressFunction;
	};

	class Workspace {
	public:
		double timeStep;
		Eigen::MatrixXd rateMatrixRelaxation;
		Eigen::MatrixXd rateMatrixShear;
		Eigen::MatrixXd rateMatrixTotal;
		Eigen::MatrixXd stepMatrix;
		Eigen::MatrixXd strainRateMatrixShear;
		HardinUtil2::StressPrincipalWorkspace stressPrincipalWorkspace;
		std::array<double, 3> stressPrincipal;
		Eigen::Matrix3d principalFrame;

		void setup(const std::int64_t nLevels) {
```

```
36                      timeStep = 0;
37                      rateMatrixRelaxation = Eigen::MatrixXd::Zero(nLevels, ↩
                            nLevels);
38                      rateMatrixShear = Eigen::MatrixXd::Zero(nLevels, nLevels);
39                      strainRateMatrixShear = Eigen::MatrixXd::Zero(6, nLevels);
40                      rateMatrixTotal = Eigen::MatrixXd::Zero(nLevels, nLevels);
41                      stepMatrix = Eigen::MatrixXd::Zero(nLevels, nLevels);
42              }
43
44              Workspace() {}
45              Workspace(const std::int64_t nLevels) { setup(nLevels); }
46      };
47
48      class State {
49      public:
50              //Abscissae
51              std::int64_t iStep;
52              double time;
53              //Loading
54              double temperature;
55              std::array<double, 6> stress;
56              //Ordinates
57              Eigen::VectorXd occupationLevels;
58              Eigen::Matrix<double, 6, 1> strainRate;
59      };
60
61      Parameters params;
62      Kinetics::Levels levels;
63      RateMatrixRelaxationFactory rateMatrixRelaxationFactory;
64      RateMatrixShearFactory rateMatrixShearFactory;
65      std::function<double(const double)> temperatureFunction;
66      std::function<void(const double, double*)> stressFunction;
67
68      void alias() {
69              rateMatrixRelaxationFactory = params.rateMatrixRelaxationFactory;
70              rateMatrixShearFactory = params.rateMatrixShearFactory;
```

```
71              levels = rateMatrixShearFactory.levels;
72              temperatureFunction = params.temperatureFunction;
73              stressFunction = params.stressFunction;
74      }
75      void setup(const Parameters &parameters) { params = parameters; alias(); ↩
            }
76      EvolveStress() {}
77      EvolveStress(const Parameters &parameters) { setup(parameters); }
78
79      void getOrdinateDerivatives(
80              Workspace* workspace,
81              State* state,
82              const std::int8_t nThreads = 1
83      )const {
84              HardinUtil2::getStressPrincipalVoigt(
85                      state->stress.data(),
86                      &(workspace->stressPrincipalWorkspace),
87                      workspace->stressPrincipal.data(),
88                      workspace->principalFrame.data()
89              );
90              rateMatrixRelaxationFactory.buildRateMatrixRelaxation(
91                      state->temperature,
92                      workspace->rateMatrixRelaxation.data(),
93                      nThreads
94              );
95              rateMatrixShearFactory.buildRateMatrixShear(
96                      state->temperature,
97                      workspace->stressPrincipal.data(),
98                      workspace->rateMatrixShear.data(),
99                      workspace->strainRateMatrixShear.data(),
100                     nThreads
101             );
102             workspace->rateMatrixTotal = workspace->rateMatrixRelaxation + 0 *↩
                    workspace->rateMatrixShear;
103     }
104
```

```
105     void initializeSteadyState(
106             const double temperature,
107             const double* stressVoigt,
108             Workspace* workspace,
109             State* state,
110             const std::int8_t nThreads = 1
111     ) const {
112             state->temperature = temperature;
113             for (std::int8_t i = 0; i < 6; i++) { state->stress[i] = ↩
                    stressVoigt[i]; }
114             getOrdinateDerivatives(workspace, state, nThreads);
115
116             auto rightPreconditioner = workspace->rateMatrixTotal.colwise().↩
                    norm().asDiagonal().inverse();
117             Eigen::MatrixXd rateMatrixPreconditioned = workspace->↩
                    rateMatrixTotal * rightPreconditioner;
118             Eigen::VectorXd nullSpacePreconditioned = rateMatrixPreconditioned↩
                    .fullPivLu().kernel();
119             state->occupationLevels = rightPreconditioner * ↩
                    nullSpacePreconditioned;
120             state->occupationLevels /= state->occupationLevels.sum();
121
122             state->strainRate = workspace->strainRateMatrixShear * state->↩
                    occupationLevels;
123     }
124
125     void step(
126             const double timeStepMin,
127             const double timeStepMax,
128             const double temperatureStepMax,
129             Workspace* workspace,
130             State* state,
131             const std::int8_t nThreads = 1
132     ) const {
133             //Get the timestep & loading conditions at the step end
134             double timeStep = timeStepMax;
```

```
135             double time = state->time + timeStep;
136             double temperature = params.temperatureFunction(time);
137             double temperatureStep = temperature - state->temperature;
138             while (std::abs(temperatureStep) > temperatureStepMax && timeStep ↩
                    >= 2 * timeStepMin) {
139                 timeStep *= 0.5;
140                 time = state->time + timeStep;
141                 temperature = params.temperatureFunction(time);
142                 temperatureStep = temperature - state->temperature;
143             }
144             if (std::abs(temperatureStep) > temperatureStepMax) {
145                 timeStep = timeStepMin;
146                 time = state->time + timeStep;
147                 temperature = params.temperatureFunction(time);
148             }
149             state->time = time;
150             state->temperature = temperature;
151             stressFunction(time, state->stress.data());
152 
153             //Get the derivatives of the ordinates
154             getOrdinateDerivatives(workspace, state, nThreads);
155 
156             workspace->stepMatrix
157                     = (Eigen::MatrixXd::Identity(levels.nLevels, levels.↩
                            nLevels) - timeStep * workspace->rateMatrixTotal).↩
                            inverse();
158 
159             state->occupationLevels = workspace->stepMatrix * state->↩
                    occupationLevels;
160             state->strainRate = workspace->strainRateMatrixShear * state->↩
                    occupationLevels;
161             HardinUtil2::counterRotateVoigtStrain(state->strainRate.data(), ↩
                    workspace->principalFrame.data(), state->strainRate.data());
162             state->iStep++;
163         }
164 
```

```
165 };
166
167 } //namespace Homogeneous
168 } //namespace KMGEM
169
170 #endif /* KMGEM_HOMOGENEOUS_EVOLVESTRESS_H_ */
```

Listing B.12: KMGEM/Homogeneous/RateMatricesShear.h

```
#ifndef KMGEM_HOMOGENEOUS_RATEMATRIXSHEAR_H_
#define KMGEM_HOMOGENEOUS_RATEMATRIXSHEAR_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

namespace KMGEM {
namespace Homogeneous {

class RateMatrixShearFactory {
public:
        class Parameters {
        public:
                Kinetics::RateShearEvolution rateShearEvolution;
        };

        Parameters params;
        Kinetics::RateShearEvolution rateShearEvolution; //Alias
        Kinetics::Levels levels; //Alias

        void alias() {
                rateShearEvolution = params.rateShearEvolution;
                levels = rateShearEvolution.levels;
        }
        void setup(const Parameters parameters) { params = parameters; alias(); }
        RateMatrixShearFactory() {}
        RateMatrixShearFactory(const Parameters parameters) { setup(parameters); }
            }

        //PDot = Matrix * P
        void buildRateMatrixShear(
                const double temperature,
                const double* stressPrincipal,
                double* rateMatrixShear, //[nLevels, nLevels]{COL-Major}
                double* strainRateMatrixShear, //[6, nLevels]{COL-Major}
```

```
35              const std::int8_t nThreads = 1
36          ) const {
37              std::vector<double> levelRates(levels.nLevels);
38              std::vector<double> levelProbabilities(levels.nLevels);
39              rateShearEvolution.getRateShearEvolution(temperature, ↩
                    stressPrincipal, levelRates.data(), strainRateMatrixShear, ↩
                    levelProbabilities.data(), nThreads);
40
41              Eigen::Map<Eigen::Matrix<double, -1, -1, Eigen::ColMajor>> ↩
                    rateMatrix(
42                      rateMatrixShear,
43                      levels.nLevels,
44                      levels.nLevels
45              );
46              rateMatrix.setZero();
47
48              auto threadLambda = [&](std::int8_t iThread) {
49                      std::array<double, 6> strainRate;
50                      //Element r,c of the matrix represents the contribution of↩
                            the population in level c to the rate of change of ↩
                            level r
51                      for (std::int64_t iLevelFrom = iThread; iLevelFrom < ↩
                            levels.nLevels; iLevelFrom += nThreads) {
52                          for (std::int64_t iLevelTo = 0; iLevelTo < levels.↩
                                nLevels; iLevelTo++) {
53                              double flowPerPopulationFrom = levelRates[↩
                                    iLevelFrom] * levelProbabilities[↩
                                    iLevelTo];
54                              rateMatrix(iLevelFrom, iLevelFrom) -= ↩
                                    flowPerPopulationFrom;
55                              rateMatrix(iLevelTo, iLevelFrom) += ↩
                                    flowPerPopulationFrom;
56                          }
57                      }
58              };
59
```

```
60                if (nThreads == 1) { threadLambda(0); }
61                else {
62                        std::vector<std::thread> threads;
63                        for (std::int8_t iThread = 0; iThread < nThreads; iThread↩
                              ++) { threads.push_back(std::thread(threadLambda, ↩
                              iThread)); }
64                        for (std::thread &thread : threads) { thread.join(); }
65                }
66        }
67 };
68
69 } //namespace Homogeneous
70 } //namespace KMGEM
71
72 #endif /* KMGEM_HOMOGENEOUS_RATEMATRIXSHEAR_H_ */
```

Listing B.13: KMGEM/Homogeneous/RateMatrixRelaxation.h

```
#ifndef KMGEM_HOMOGENEOUS_RATEMATRIXRELAXATION_H_
#define KMGEM_HOMOGENEOUS_RATEMATRIXRELAXATION_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

namespace KMGEM {
namespace Homogeneous {

class RateMatrixRelaxationFactory {
public:
        class Parameters {
        public:
                Kinetics::RateRelaxation rateRelaxation;
                Kinetics::Levels levels;
        };

        Parameters params;
        Kinetics::RateRelaxation rateRelaxation;
        Kinetics::Levels levels;

        void alias() {
                rateRelaxation = params.rateRelaxation;
                levels = params.levels;
        }
        void setup(const Parameters parameters) { params = parameters; alias(); }
        RateMatrixRelaxationFactory() {}
        RateMatrixRelaxationFactory(const Parameters parameters) { setup(↩
            parameters); }

        //PDot = Matrix * P
        void buildRateMatrixRelaxation(
                const double temperature,
                double* rateMatrixRelaxation, //COL-Major
                const std::int8_t nThreads = 1
```

```
35      ) const {
36              Eigen::Map<Eigen::Matrix<double, -1, -1, Eigen::ColMajor>> ↩
                    rateMatrix(
37                      rateMatrixRelaxation,
38                      levels.nLevels,
39                      levels.nLevels
40              );
41              rateMatrix.setZero();
42
43              auto threadLambda = [&](std::int8_t iThread) {
44                      //Element r,c of the matrix represents the contribution of↩
                            the population in level c to the rate of change of ↩
                            level r
45                      for (std::int64_t iLevelFrom = iThread; iLevelFrom < ↩
                            levels.nLevels; iLevelFrom += nThreads) {
46                          double cpeIsochoricInitial = levels.↩
                                cpeIsochoricLevels(iLevelFrom);
47                          double cpeDilatativeInitial = levels.↩
                                cpeDilatativeLevels(iLevelFrom);
48
49                          for (std::int64_t iLevelTo = 0; iLevelTo < levels.↩
                                nLevels; iLevelTo++) {
50                              if (iLevelFrom == iLevelTo) { continue; }
51
52                              double cpeIsochoricFinal = levels.↩
                                    cpeIsochoricLevels(iLevelTo);
53                              double cpeDilatativeFinal = levels.↩
                                    cpeDilatativeLevels(iLevelTo);
54
55                              double rate = rateRelaxation.getRate(
56                                      temperature,
57                                      cpeIsochoricInitial,
58                                      cpeDilatativeInitial,
59                                      cpeIsochoricFinal,
60                                      cpeDilatativeFinal
61                              );
```

```
62
63                                      double flowPerPopulationFrom = rate * levels↩
                                            .degeneracies(iLevelTo);
64
65                                      rateMatrix(iLevelFrom, iLevelFrom) -= ↩
                                            flowPerPopulationFrom;
66                                      rateMatrix(iLevelTo, iLevelFrom) += ↩
                                            flowPerPopulationFrom;
67                                  }
68                          }
69                  };
70
71                  if (nThreads == 1) { threadLambda(0); }
72                  else {
73                          std::vector<std::thread> threads;
74                          for (std::int8_t iThread = 0; iThread < nThreads; iThread↩
                                ++) { threads.push_back(std::thread(threadLambda, ↩
                                iThread)); }
75                          for (std::thread &thread : threads) { thread.join(); }
76                  }
77          }
78  };
79
80  } //namespace Homogeneous
81  } //namespace KMGEM
82
83  #endif /* KMGEM_HOMOGENEOUS_RATEMATRIXRELAXATION_H_ */
```

Listing B.14: KMGEM/Homogeneous/State.h

```
#ifndef KMGEM_HOMOGENEOUS_STATE_H_
#define KMGEM_HOMOGENEOUS_STATE_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

namespace KMGEM {
namespace Homogeneous {

class State {
public:
	//Time state
	std::int64_t iStep;
	double time;
	double temperature;
	//Statistical nanostructural state
	Eigen::VectorXd occupationLevels;
	//Macroscopic mechanical state
	Eigen::Matrix<double, 6, 1> stress;
	Eigen::Matrix<double, 6, 1> strainPlastic;
	Eigen::Matrix<double, 6, 1> strainPlasticRate;
	//Homogenized properties
	double shearModulus;
	double bulkModulus;
	double volumeMolar;
	double cpem;
	double cpemIsochoric;
	double cpemVolumetric;

	void setup(const std::int64_t nLevels) {
		iStep = 0;
		time = 0;
		temperature = 0;

		occupationLevels = Eigen::VectorXd::Zero(nLevels);
```

```
36
37                stress.setZero();
38                strainPlastic.setZero();
39                strainPlasticRate.setZero();
40        }
41        State() {}
42        State(const std::int64_t nLevels) { setup(nLevels); }
43
44 #ifdef KMGEM_ENABLE_HDF
45        void save(const H5::Group &target) const {
46                HardinUtil2::saveScalar<std::int64_t>(&iStep, "iStep", target);
47                HardinUtil2::saveScalar<double>(&time, "time", target);
48                HardinUtil2::saveScalar<double>(&temperature, "temperature", ↩
                      target);
49
50                HardinUtil2::saveArray<double, 1>(
51                        occupationLevels.data(),
52                        { { hsize_t(occupationLevels.size()) } },
53                        "occupationLevels",
54                        target
55                        );
56
57                HardinUtil2::saveArray<double, 1>(
58                        stress.data(),
59                        { { hsize_t(stress.size()) } },
60                        "stress",
61                        target
62                        );
63                HardinUtil2::saveArray<double, 1>(
64                        strainPlastic.data(),
65                        { { hsize_t(strainPlastic.size()) } },
66                        "strainPlastic",
67                        target
68                        );
69                HardinUtil2::saveArray<double, 1>(
70                        strainPlasticRate.data(),
```

```
71                      { { hsize_t(strainPlasticRate.size()) } },
72                      "strainPlasticRate",
73                      target
74                      );
75
76              HardinUtil2::saveScalar<double>(&shearModulus, "shearModulus", ↩
                    target);
77              HardinUtil2::saveScalar<double>(&bulkModulus, "bulkModulus", ↩
                    target);
78              HardinUtil2::saveScalar<double>(&volumeMolar, "volumeMolar", ↩
                    target);
79              HardinUtil2::saveScalar<double>(&cpem, "cpem", target);
80              HardinUtil2::saveScalar<double>(&cpemIsochoric, "cpemIsochoric", ↩
                    target);
81              HardinUtil2::saveScalar<double>(&cpemVolumetric, "cpemVolumetric",↩
                     target);
82      }
83  #endif
84  };
85
86  //A low-level utility that fills in a matrix in the pattern of an isotropic ↩
        stiffness matrix
87  static void fillMatrixPattern(const double a, const double b, const double c, ↩
        double* matrix) {
88      //Row 0
89      matrix[0 * 6 + 0] = a;
90      matrix[0 * 6 + 1] = b;
91      matrix[0 * 6 + 2] = b;
92      matrix[0 * 6 + 3] = 0;
93      matrix[0 * 6 + 4] = 0;
94      matrix[0 * 6 + 5] = 0;
95
96      //Row 1
97      matrix[1 * 6 + 0] = b;
98      matrix[1 * 6 + 1] = a;
99      matrix[1 * 6 + 2] = b;
```

```
100         matrix[1 * 6 + 3] = 0;
101         matrix[1 * 6 + 4] = 0;
102         matrix[1 * 6 + 5] = 0;
103
104         //Row 2
105         matrix[2 * 6 + 0] = b;
106         matrix[2 * 6 + 1] = b;
107         matrix[2 * 6 + 2] = a;
108         matrix[2 * 6 + 3] = 0;
109         matrix[2 * 6 + 4] = 0;
110         matrix[2 * 6 + 5] = 0;
111
112         //Row 3
113         matrix[3 * 6 + 0] = 0;
114         matrix[3 * 6 + 1] = 0;
115         matrix[3 * 6 + 2] = 0;
116         matrix[3 * 6 + 3] = c;
117         matrix[3 * 6 + 4] = 0;
118         matrix[3 * 6 + 5] = 0;
119
120         //Row 4
121         matrix[4 * 6 + 0] = 0;
122         matrix[4 * 6 + 1] = 0;
123         matrix[4 * 6 + 2] = 0;
124         matrix[4 * 6 + 3] = 0;
125         matrix[4 * 6 + 4] = c;
126         matrix[4 * 6 + 5] = 0;
127
128         //Row 5
129         matrix[5 * 6 + 0] = 0;
130         matrix[5 * 6 + 1] = 0;
131         matrix[5 * 6 + 2] = 0;
132         matrix[5 * 6 + 3] = 0;
133         matrix[5 * 6 + 4] = 0;
134         matrix[5 * 6 + 5] = c;
135 }
```

```
136
137 //Pa[36] <- (Pa,Pa)
138 //Get the 6x6 isotropic stiffness matrix associated with the given shear and ↩
        bulk moduli
139 //Assumes Voigt stress and strain notation
140 static void getStiffnessMatrix(const double shearModulus, const double ↩
        bulkModulus, double* stiffnessMatrix) {
141         double a = bulkModulus + (4.0 / 3.0)*shearModulus;
142         double b = bulkModulus - (2.0 / 3.0)*shearModulus;
143         double c = shearModulus;
144         fillMatrixPattern(a, b, c, stiffnessMatrix);
145 }
146
147 } //namespace Homogeneous
148 } //namespace KMGEM
149
150 #endif /* KMGEM_HOMOGENEOUS_EVOLVESTRESSCONTROLLED_H_ */
```

Listing B.15: KMGEM/Mesoscale/BuildFeForceVector.h

```
#ifndef KMGEM_MESOSCALE_BUILDFEFORCEVECTOR_H_
#define KMGEM_MESOSCALE_BUILDFEFORCEVECTOR_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "metis.h"
#include "mpi.h"
#include "H5Cpp.h"
#include "tetgen.h"

#include "Eigen/Core"
#include "Eigen/Sparse"

#include "HardinFE3/FiniteElements/TetrahedronLinear.h"
#include "HardinFE3/FiniteElements/Elasticity3.h"

#include "KMGEM/Mesoscale/IdMatrix.h"
#include "KMGEM/Mesoscale/Mesh.h"

namespace KMGEM {
namespace Mesoscale {

template <
        class Scalar,
        class GlobalOrdinal,
        std::int8_t CARDINALITY
> void buildFeForceVector(
        const Mesh<Scalar, GlobalOrdinal, CARDINALITY> &mesh,
        const IdMatrix<GlobalOrdinal> &idMatrix,
        const std::array<std::array<Scalar, 6>, 6>* elasticStiffnessMatrices,
        Eigen::VectorXd &feForceVector
) {
        feForceVector.setZero(idMatrix.nEqns);
        HardinFE3::FiniteElements::TetrahedronLinear<4, double, HardinFE3::↩
```

```
        FiniteElements::QuadratureTetrahedronQuadratic<double>> ↩
        tetrahedronLinear;
36      std::array<const std::array<double, 3>*, CARDINALITY> nodesElt;
37      for (GlobalOrdinal elementIndex = 0; elementIndex < mesh.elements.size();↩
            elementIndex++) {
38          //Nail down the quadrature business
39          const std::array<GlobalOrdinal, CARDINALITY> &element = mesh.↩
                elements[elementIndex];
40          for (std::int8_t a = 0; a < CARDINALITY; a++) { nodesElt[a] = &(↩
                mesh.nodes[element[a]]); }
41          std::array<double, 4> quadratureWeights;
42          std::array<std::array<std::array<double, 3>, 4>, 4> ↩
                shapeFunctionGradientsQuadrature;
43          std::array<std::array<double, 3>, 3> jacobianWorkspace;
44          std::array<std::array<double, 3>, 3> jacobianInverseWorkspace;
45          tetrahedronLinear.↩
                computeChildQuadratureWeightsAndShapeFunctionGradients(
46                  nodesElt,
47                  &quadratureWeights,
48                  &shapeFunctionGradientsQuadrature,
49                  jacobianWorkspace,
50                  jacobianInverseWorkspace
51          );
52
53          std::array<std::array<std::array<std::array<double, 3>, 3>, 4>, 4>↩
                elementStiffnessMatrix;
54          HardinFE3::FiniteElements::↩
                computeElementStiffnessMatrixElasticity3<4, 4, double>(
55                  quadratureWeights,
56                  shapeFunctionGradientsQuadrature,
57                  elasticStiffnessMatrices[elementIndex],
58                  &elementStiffnessMatrix
59                  );
60
61          for (std::int8_t a = 0; a < 4; a++) {
62                  for (std::int8_t i = 0; i < 3; i++) {
```

```
63                                  char roleA = idMatrix.roles[element[a]][i];
64                                  GlobalOrdinal equationNumberA = idMatrix.indices[↩
                                        element[a]][i];
65                                  if (roleA != 'i') { continue; }
66
67                                  for (std::int8_t b = 0; b < 4; b++) {
68                                        for (std::int8_t j = 0; j < 3; j++) {
69                                              char roleB = idMatrix.roles[element[b↩
                                                    ]][j];
70                                              GlobalOrdinal equationNumberB = ↩
                                                    idMatrix.indices[element[b]][j];
71                                              if (roleB != 'd') { continue; }
72                                              feForceVector[equationNumberA] -= ↩
                                                    elementStiffnessMatrix[a][b][i][j↩
                                                    ];
73                                        }
74                                  }
75                            }
76                      }
77              }
78  }
79
80  } //namespace Mesoscale
81  } //namespace KMGEM
82
83  #endif /* KMGEM_MESOSCALE_BUILDFEFORCEVECTOR_H_ */
```

Listing B.16: KMGEM/Mesoscale/BuildFeStiffnessMatrix.h

```
#ifndef KMGEM_MESOSCALE_BUILDFESTIFFNESSMATRIX_H_
#define KMGEM_MESOSCALE_BUILDFESTIFFNESSMATRIX_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "metis.h"
#include "mpi.h"
#include "H5Cpp.h"
#include "tetgen.h"

#include "Eigen/Sparse"

#include "HardinFE3/FiniteElements/TetrahedronLinear.h"
#include "HardinFE3/FiniteElements/Elasticity3.h"

#include "KMGEM/Mesoscale/IdMatrix.h"
#include "KMGEM/Mesoscale/Mesh.h"

namespace KMGEM {
namespace Mesoscale {

template <
        class Scalar,
        class GlobalOrdinal,
        std::int8_t CARDINALITY
> void buildFeStiffnessMatrix(
        const Mesh<Scalar,GlobalOrdinal,CARDINALITY> &mesh,
        const IdMatrix<GlobalOrdinal> &idMatrix,
        const std::array<std::array<Scalar,6>,6>* elasticStiffnessMatrices,
        Eigen::SparseMatrix<Scalar, 0, GlobalOrdinal> &feStiffnessMatrix
) {
        HardinFE3::FiniteElements::TetrahedronLinear<4, double, HardinFE3::↩
            FiniteElements::QuadratureTetrahedronQuadratic<double>> ↩
            tetrahedronLinear;
```

```
34          std::vector<Eigen::Triplet<Scalar, GlobalOrdinal>> triplets;
35          std::array<const std::array<double, 3>*, CARDINALITY> nodesElt;
36          for (GlobalOrdinal elementIndex = 0; elementIndex < mesh.elements.size();↩
                elementIndex++) {
37              //Nail down the quadrature business
38              const std::array<GlobalOrdinal, CARDINALITY> &element = mesh.↩
                    elements[elementIndex];
39              for (std::int8_t a = 0; a < CARDINALITY; a++) { nodesElt[a] = &(↩
                    mesh.nodes[element[a]]); }
40              std::array<double, 4> quadratureWeights;
41              std::array<std::array<std::array<double, 3>, 4>, 4> ↩
                    shapeFunctionGradientsQuadrature;
42              std::array<std::array<double, 3>, 3> jacobianWorkspace;
43              std::array<std::array<double, 3>, 3> jacobianInverseWorkspace;
44              tetrahedronLinear.↩
                    computeChildQuadratureWeightsAndShapeFunctionGradients(
45                      nodesElt,
46                      &quadratureWeights,
47                      &shapeFunctionGradientsQuadrature,
48                      jacobianWorkspace,
49                      jacobianInverseWorkspace
50              );
51
52              std::array<std::array<std::array<std::array<double, 3>, 3>, 4>, 4>↩
                     elementStiffnessMatrix;
53              HardinFE3::FiniteElements::↩
                    computeElementStiffnessMatrixElasticity3<4, 4, double>(
54                      quadratureWeights,
55                      shapeFunctionGradientsQuadrature,
56                      elasticStiffnessMatrices[elementIndex],
57                      &elementStiffnessMatrix
58                      );
59
60              for (std::int8_t a = 0; a < 4; a++) {
61                      for (std::int8_t i = 0; i < 3; i++) {
62                              char roleA = idMatrix.roles[element[a]][i];
```

```
63                                  GlobalOrdinal equationNumberA = idMatrix.indices[↩
                                        element[a]][i];
64                                  if (roleA != 'i') { continue; }
65
66                                  for (std::int8_t b = 0; b < 4; b++) {
67                                          for (std::int8_t j = 0; j < 3; j++) {
68                                                  char roleB = idMatrix.roles[element[b↩
                                                        ]][j];
69                                                  GlobalOrdinal equationNumberB = ↩
                                                        idMatrix.indices[element[b]][j];
70                                                  if (roleB != 'i') { continue; }
71                                                  triplets.push_back(Eigen::Triplet<↩
                                                        Scalar, GlobalOrdinal>(↩
                                                        equationNumberA, equationNumberB,↩
                                                         elementStiffnessMatrix[a][b][i][↩
                                                        j]));
72                                          }
73                                  }
74                          }
75                  }
76          }
77          feStiffnessMatrix.resize(idMatrix.nEqns, idMatrix.nEqns);
78          feStiffnessMatrix.setFromTriplets(triplets.begin(), triplets.end());
79  }
80
81  } //namespace Mesoscale
82  } //namespace KMGEM
83
84  #endif /* KMGEM_MESOSCALE_BUILDFESTIFFNESSMATRIX_H_ */
```

Listing B.17: KMGEM/Mesoscale/CubeMesh.h

```
#ifndef KMGEM_MESOSCALE_CUBEMESH_H_
#define KMGEM_MESOSCALE_CUBEMESH_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "KMGEM/Mesoscale/IdMatrix.h"
#include "KMGEM/Mesoscale/Mesh.h"


namespace KMGEM {
namespace Mesoscale {

template <
        class Scalar,
        class GlobalOrdinal
> class CubeMesh {
public:
        std::array<Scalar, 3> origin;
        Scalar unitLength;
        std::array<GlobalOrdinal, 3> sideUnits;

        GlobalOrdinal size() const { return (sideUnits[0] + 1) * (sideUnits[1] + 1) *(sideUnits[2] + 1); }

        GlobalOrdinal indexNodes(const std::array<GlobalOrdinal, 3> &triple) const {
                GlobalOrdinal ret = triple[2] + (sideUnits[2] + 1) * (triple[1] + (sideUnits[1] + 1) * (triple[0]));
                return ret;
        }

        void deIndexNodes(GlobalOrdinal index, std::array<GlobalOrdinal, 3>* triple) const {
                (*triple)[2] = index % (sideUnits[2] + 1);
```

```
32              index /= (sideUnits[2] + 1);
33              (*triple)[1] = index % (sideUnits[1] + 1);
34              (*triple)[0] = index / (sideUnits[1] + 1);
35          }
36
37          void interpolate(const std::array<Scalar, 3> &node, std::array<↩
                GlobalOrdinal, 8> &cubeNodes, std::array<Scalar, 8> &cubeWeights) ↩
                const {
38              std::array<Scalar, 3> normalizedNode;
39              for (std::int8_t dim = 0; dim < 3; dim++) { normalizedNode[dim] = ↩
                    (node[dim] - origin[dim]) / unitLength; }
40              std::array<GlobalOrdinal, 3> rootNode;
41              for (std::int8_t dim = 0; dim < 3; dim++) { rootNode[dim] = ↩
                    normalizedNode[dim]; }
42
43              std::int8_t iNodeLocal = 0;
44              for (std::int8_t i = 0; i < 2; i++) {
45                      Scalar weightX = normalizedNode[0] - rootNode[0];
46                      if (i == 0) { weightX = 1 - weightX; }
47
48                      for (std::int8_t j = 0; j < 2; j++) {
49                              Scalar weightY = normalizedNode[1] - rootNode[1];
50                              if (j == 0) { weightY = 1 - weightY; }
51
52                              for (std::int8_t k = 0; k < 2; k++) {
53                                      Scalar weightZ = normalizedNode[2] - ↩
                                            rootNode[2];
54                                      if (k == 0) { weightZ = 1 - weightZ; }
55
56                                      std::array<GlobalOrdinal, 3> triple{ ↩
                                            rootNode[0] + i, rootNode[1] + j, ↩
                                            rootNode[2] + k };
57                                      cubeNodes[iNodeLocal] = indexNodes(triple);
58                                      cubeWeights[iNodeLocal] = weightX * weightY ↩
                                            * weightZ;
59
```

```
60                                      iNodeLocal++;
61                                  }
62                              }
63                          }
64                  }
65  };
66
67  template <
68          class Scalar,
69          class GlobalOrdinal,
70          std::int8_t CARDINALITY
71  >
72  void buildCubeMeshHeirarchy(
73          const Mesh<Scalar,GlobalOrdinal,CARDINALITY> &mesh,
74          std::vector<CubeMesh<Scalar,GlobalOrdinal>> &cubeMeshes
75  ) {
76          //Info from the last level down
77          Scalar unitLength = std::cbrt(mesh.volumeFinest);
78          std::array<Scalar, 3> boundsLo = mesh.boundsLo;
79          std::array<Scalar, 3> boundsHi = mesh.boundsHi;
80
81          cubeMeshes.clear();
82          while (true) {
83                  CubeMesh<Scalar, GlobalOrdinal> cubeMeshNew;
84                  cubeMeshNew.unitLength = unitLength * 2;
85                  for (std::int8_t dim = 0; dim < 3; dim++) { cubeMeshNew.sideUnits[↩
                        dim] = std::ceil((boundsHi[dim] - boundsLo[dim]) / cubeMeshNew↩
                        .unitLength); }
86                  for (std::int8_t dim = 0; dim < 3; dim++) { cubeMeshNew.origin[dim↩
                        ] = 0.5 * (boundsHi[dim] + boundsLo[dim]) - 0.5 * cubeMeshNew.↩
                        sideUnits[dim] * cubeMeshNew.unitLength; }
87                  cubeMeshes.push_back(cubeMeshNew);
88
89                  auto minmaxSideSize = std::minmax_element(cubeMeshNew.sideUnits.↩
                        begin(), cubeMeshNew.sideUnits.end());
90                  auto smallestSideSize = *(minmaxSideSize.first);
```

```
91                 auto largestSideSize = *(minmaxSideSize.second);
92                 if (smallestSideSize <= 2 && largestSideSize <= 8) { break; }
93
94                 boundsLo = cubeMeshNew.origin;
95                 for (std::int8_t dim = 0; dim < 3; dim++) { boundsHi[dim] = ↩
                       cubeMeshNew.origin[dim] + cubeMeshNew.unitLength * cubeMeshNew↩
                       .sideUnits[dim]; }
96                 unitLength = cubeMeshNew.unitLength;
97         }
98
99 }
100
101
102 } //namespace Mesoscale
103 } //namespace KMGEM
104
105 #endif /* KMGEM_MESOSCALE_CUBEMESH_H_ */
```

Listing B.18: KMGEM/Mesoscale/CylinderPLC.h

```
#ifndef KMGEM_MESOSCALE_CYLINDERPLC_H_
#define KMGEM_MESOSCALE_CYLINDERPLC_H_

#include <cstddef>

#include "tetgen.h"

namespace KMGEM {
namespace Mesoscale {

void getCylinderPlc(const double radius,
        const double length,
        const std::int64_t nTheta,
        tetgenio* tetGenPLC) {
        const double PI = 3.14159265358979324;
        double dTheta = 2 * PI / nTheta;

        tetGenPLC->firstnumber = 0;
        tetGenPLC->mesh_dim = 3;

        tetGenPLC->pointattributelist = NULL;
        tetGenPLC->pointmtrlist = NULL;
        tetGenPLC->pointmarkerlist = NULL;
        tetGenPLC->numberofpointattributes = 0;
        tetGenPLC->numberofpointmtrs = 0;

        //Nail down the point indices
        std::int64_t nPoints = 0;
        std::array<std::vector<std::int64_t>, 2> pointIndices;
        for (std::int64_t i = 0; i < nTheta; i++) { pointIndices[0].push_back(↩
            nPoints++); }
        pointIndices[0].push_back(pointIndices[0].front());
        for (std::int64_t i = 0; i < nTheta; i++) { pointIndices[1].push_back(↩
            nPoints++); }
        pointIndices[1].push_back(pointIndices[1].front());
```

```
34
35          //Construct the points
36          tetGenPLC->numberofpoints = nPoints;
37          tetGenPLC->pointlist = new REAL[tetGenPLC->numberofpoints * 3];
38          std::array<REAL, 3>* pointList = (std::array<double, 3>*)(tetGenPLC->↩
                pointlist);
39          for (std::int64_t iTheta = 0; iTheta < nTheta; iTheta++) {
40                  double theta = iTheta * dTheta;
41                  double cosTheta = std::cos(theta);
42                  double sinTheta = std::sin(theta);
43
44                  pointList[pointIndices[0][iTheta]] = { { radius * cosTheta, radius↩
                        *sinTheta, 0 } };
45                  pointList[pointIndices[1][iTheta]] = pointList[pointIndices[0][↩
                        iTheta]];
46                  pointList[pointIndices[1][iTheta]][2] += length;
47          }
48
49          //Construct the facets
50          tetGenPLC->numberoffacets = nTheta + 2;
51          tetGenPLC->facetlist = new tetgenio::facet[nTheta + 2];
52          tetGenPLC->facetmarkerlist = new int[tetGenPLC->numberoffacets];
53          //Side facets
54          for (std::int64_t iTheta = 0; iTheta < nTheta; iTheta++) {
55                  tetGenPLC->facetmarkerlist[iTheta] = 2;
56                  tetgenio::facet &f = tetGenPLC->facetlist[iTheta];
57                  f.numberofholes = 0;
58                  f.holelist = NULL;
59                  f.numberofpolygons = 1;
60                  f.polygonlist = new tetgenio::polygon[f.numberofpolygons];
61
62                  tetgenio::polygon &p = f.polygonlist[0];
63                  p.numberofvertices = 4;
64                  p.vertexlist = new int[p.numberofvertices];
65                  p.vertexlist[0] = pointIndices[0][iTheta];
66                  p.vertexlist[1] = pointIndices[0][iTheta+1];
```

```
67                p.vertexlist[2] = pointIndices[1][iTheta+1];
68                p.vertexlist[3] = pointIndices[1][iTheta];
69        }
70        //Bottom facet
71        {
72                tetGenPLC->facetmarkerlist[nTheta] = 0;
73                tetgenio::facet &f = tetGenPLC->facetlist[nTheta];
74                f.numberofholes = 0;
75                f.holelist = NULL;
76                f.numberofpolygons = 1;
77                f.polygonlist = new tetgenio::polygon[f.numberofpolygons];
78
79                tetgenio::polygon &p = f.polygonlist[0];
80                p.numberofvertices = nTheta;
81                p.vertexlist = new int[p.numberofvertices];
82                for (std::int64_t i = 0; i < nTheta; i++) { p.vertexlist[i] = ↩
                      pointIndices[0][nTheta - i - 1]; }
83        }
84        //Top facets
85        {
86                tetGenPLC->facetmarkerlist[nTheta+1] = 1;
87                tetgenio::facet &f = tetGenPLC->facetlist[nTheta+1];
88                f.numberofholes = 0;
89                f.holelist = NULL;
90                f.numberofpolygons = 1;
91                f.polygonlist = new tetgenio::polygon[f.numberofpolygons];
92
93                tetgenio::polygon &p = f.polygonlist[0];
94                p.numberofvertices = nTheta;
95                p.vertexlist = new int[p.numberofvertices];
96                for (std::int64_t i = 0; i < nTheta; i++) { p.vertexlist[i] = ↩
                      pointIndices[1][i]; }
97        }
98 }
99
100 } //namespace Mesoscale
```

```
101 } //namespace KMGEM
102
103 #endif /* KMGEM_MESOSCALE_CYLINDERPLC_H_ */
```

Listing B.19: KMGEM/Mesoscale/GaussSeidel.h

```
#ifndef KMGEM_MESOSCALE_GAUSSSEIDEL_H_
#define KMGEM_MESOSCALE_GAUSSSEIDEL_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "Eigen/Core"

#include "metis.h"
#include "mpi.h"
#include "H5Cpp.h"
#include "tetgen.h"

namespace KMGEM {
namespace Mesoscale {

template <
        class SparseMat,
        class Vect
>
void gaussSeidelDualSweep(
        const SparseMat &lhsMatrix,
        const Vect &rhsVector,
        Vect &x
) {
        x = lhsMatrix.triangularView<Eigen::Lower>().solve(rhsVector - lhsMatrix.↩
            triangularView<Eigen::StrictlyUpper>() * x);
        x = lhsMatrix.triangularView<Eigen::Upper>().solve(rhsVector - lhsMatrix.↩
            triangularView<Eigen::StrictlyLower>() * x);
}

template <
        class SparseMat,
        class Vect
>
```

```
34 void gaussSeidelDual(
35         const int nGaussSeidelDualSweeps,
36         const SparseMat &lhsMatrix,
37         const Vect &rhsVector,
38         Vect &x
39 ) {
40         for (int i = 0; i < nGaussSeidelDualSweeps; i++) { gaussSeidelDualSweep(↩
               lhsMatrix,rhsVector,x); }
41 }
42
43 } //namespace Mesoscale
44 } //namespace KMGEM
45
46 #endif /* KMGEM_MESOSCALE_GAUSSSEIDEL_H_ */
```

Listing B.20: KMGEM/Mesoscale/IdMatrix.h

```
#ifndef KMGEM_MESOSCALE_IDMATRIX_H_
#define KMGEM_MESOSCALE_IDMATRIX_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "metis.h"
#include "mpi.h"
#include "H5Cpp.h"
#include "tetgen.h"

#include "Mesh.h"

namespace KMGEM {
namespace Mesoscale {

template <class GlobalOrdinal>
class IdMatrix {
public:
        GlobalOrdinal nEqns;
        std::vector<std::array<char, 3>> roles;
        std::vector<std::array<GlobalOrdinal, 3>> indices;
};

template <
        class Scalar,
        class GlobalOrdinal,
        std::int8_t CARDINALITY
> void buildIdMatrix(
        const Mesh<Scalar, GlobalOrdinal, CARDINALITY> &mesh,
        IdMatrix<GlobalOrdinal> &idMatrix
) {
        idMatrix.roles.resize(mesh.nodes.size(), { { 'i','i','i' } });
        idMatrix.indices.resize(mesh.nodes.size(), { { -1,-1,-1 } });

```

```
36          //Identify the top and bottom nodes
37          std::vector<GlobalOrdinal> bottomNodes;
38          std::vector<GlobalOrdinal> topNodes;
39          for (GlobalOrdinal iNode = 0; iNode < mesh.nodes.size(); iNode++) {
40                  if (mesh.nodes[iNode][2] == mesh.boundsLo[2]) { bottomNodes.↩
                        push_back(iNode); }
41                  if (mesh.nodes[iNode][2] == mesh.boundsHi[2]) { topNodes.push_back↩
                        (iNode); }
42          }
43
44
45          for (GlobalOrdinal iNode : bottomNodes) { idMatrix.roles[iNode] = { 'z','↩
                z','z' }; }
46          for (GlobalOrdinal iNode : topNodes) { idMatrix.roles[iNode] = { 'i','d',↩
                'd' }; }
47
48          GlobalOrdinal iIndex = 0;
49          for (GlobalOrdinal iNode = 0; iNode < mesh.nodes.size(); iNode++) {
50                  for (std::int8_t dim = 0; dim < 3; dim++) {
51                          char role = idMatrix.roles[iNode][dim];
52                          if (role == 'i') { idMatrix.indices[iNode][dim] = iIndex↩
                                ++; }
53                  }
54          }
55
56          idMatrix.nEqns = iIndex;
57  }
58
59
60  } //namespace Mesoscale
61  } //namespace KMGEM
62
63  #endif /* KMGEM_MESOSCALE_IDMATRIX_H_ */
```

Listing B.21: KMGEM/Mesoscale/IdMatrixCubeMesh.h

```
#ifndef KMGEM_MESOSCALE_IDMATRIXCUBEMESH_H_
#define KMGEM_MESOSCALE_IDMATRIXCUBEMESH_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "Eigen/Sparse"

#include "KMGEM/Mesoscale/CubeMesh.h"
#include "KMGEM/Mesoscale/IdMatrix.h"
#include "KMGEM/Mesoscale/Mesh.h"


namespace KMGEM {
namespace Mesoscale {

template <
        class Scalar,
        class GlobalOrdinal,
        std::int8_t CARDINALITY
> void _buildInterpolation(
        const std::function<std::array<Scalar, 3>(GlobalOrdinal)> &nodesFine,
        const IdMatrix<GlobalOrdinal> &idMatrixFine,
        const CubeMesh<Scalar, GlobalOrdinal> &cubeMeshCoarse,
        IdMatrix<GlobalOrdinal> &idMatrixCoarse,
        Eigen::SparseMatrix<Scalar, 0, GlobalOrdinal> &interpolationMatrix
) {
        GlobalOrdinal nNodesFine = idMatrixFine.indices.size();
        GlobalOrdinal nNodesCoarse = cubeMeshCoarse.size();

        idMatrixCoarse.nEqns = 0;

        idMatrixCoarse.roles.clear();
        idMatrixCoarse.roles.resize(nNodesCoarse, { {-1,-1,-1} });

```

```
36          idMatrixCoarse.indices.clear();
37          idMatrixCoarse.indices.resize(nNodesCoarse, { {-1,-1,-1} });
38
39          std::vector<Eigen::Triplet<Scalar, GlobalOrdinal>> triplets;
40
41          for (GlobalOrdinal iNodeFine = 0; iNodeFine < nNodesFine; iNodeFine++) {
42                  std::array<Scalar, 3> nodeFine = nodesFine(iNodeFine);
43                  const std::array<char, 3> &rolesFine = idMatrixFine.roles[↩
                        iNodeFine];
44                  const std::array<GlobalOrdinal, 3> &equationIndicesFine = ↩
                        idMatrixFine.indices[iNodeFine];
45
46                  std::array<Scalar, 8> interpolationWeightsCoarse;
47                  std::array<GlobalOrdinal, 8> interpolationNodesCoarse;
48                  cubeMeshCoarse.interpolate(nodeFine, interpolationNodesCoarse, ↩
                        interpolationWeightsCoarse);
49
50                  for (std::int8_t dim = 0; dim < 3; dim++) {
51                          GlobalOrdinal equationIndexFine = equationIndicesFine[dim↩
                                ];
52                          if (rolesFine[dim] == 'i') {
53                                  for (std::int8_t iInterpolation = 0; iInterpolation↩
                                        < 8; iInterpolation++) {
54                                          GlobalOrdinal interpolationNodeCoarse = ↩
                                                interpolationNodesCoarse[iInterpolation↩
                                                ];
55                                          Scalar interpolationWeightCoarse = ↩
                                                interpolationWeightsCoarse[↩
                                                iInterpolation];
56                                          if (std::abs(interpolationWeightCoarse) < ↩
                                                std::numeric_limits<Scalar>::epsilon()) ↩
                                                { continue; }
57
58                                          GlobalOrdinal &equationIndexCoarse = ↩
                                                idMatrixCoarse.indices[↩
                                                interpolationNodeCoarse][dim];
```

```
59                                      char &roleCoarse = idMatrixCoarse.roles[↩
                                            interpolationNodeCoarse][dim];
60
61                                      if (roleCoarse != 'i') {
62                                              roleCoarse = 'i';
63                                              equationIndexCoarse = idMatrixCoarse.↩
                                                    nEqns++;
64                                      }
65                                      triplets.push_back(Eigen::Triplet<Scalar, ↩
                                            GlobalOrdinal>(equationIndexFine, ↩
                                            equationIndexCoarse, ↩
                                            interpolationWeightCoarse));
66                              }
67                      }
68              }
69      }
70
71      interpolationMatrix.resize(idMatrixFine.nEqns, idMatrixCoarse.nEqns);
72      interpolationMatrix.setFromTriplets(triplets.begin(), triplets.end());
73  }
74
75  template <
76          class Scalar,
77          class GlobalOrdinal,
78          std::int8_t CARDINALITY
79  > void buildCubeMeshIdMatrices(
80          const Mesh<Scalar, GlobalOrdinal, CARDINALITY> &mesh,
81          const IdMatrix<GlobalOrdinal> &meshIdMatrix,
82          const std::vector<CubeMesh<Scalar, GlobalOrdinal>> &cubeMeshes,
83          std::vector<IdMatrix<GlobalOrdinal>> &idMatrices,
84          std::vector<Eigen::SparseMatrix<Scalar,0,GlobalOrdinal>> &↩
              interpolationMatrices
85  ) {
86          idMatrices.clear();
87          idMatrices.resize(cubeMeshes.size());
88          {
```

```
89              const std::function<std::array<Scalar, 3>(GlobalOrdinal)> ↩
                    nodesFine = [&mesh](GlobalOrdinal i) { return mesh.nodes[i]; ↩
                    };
90              const IdMatrix<GlobalOrdinal> &idMatrixFine = meshIdMatrix;
91              const CubeMesh<Scalar, GlobalOrdinal> &cubeMeshCoarse = cubeMeshes↩
                    [0];
92              IdMatrix<GlobalOrdinal> &idMatrixCoarse = idMatrices[0];
93              Eigen::SparseMatrix<Scalar, 0, GlobalOrdinal> &interpolationMatrix↩
                     = interpolationMatrices[0];
94
95              _buildInterpolation<Scalar,GlobalOrdinal,CARDINALITY>(
96                      nodesFine,
97                      idMatrixFine,
98                      cubeMeshCoarse,
99                      idMatrixCoarse,
100                     interpolationMatrix
101             );
102     }
103
104     for (int iHeir = 1; iHeir < cubeMeshes.size(); iHeir++) {
105             const CubeMesh<Scalar, GlobalOrdinal> &cubeMeshFine = cubeMeshes[↩
                    iHeir - 1];
106             std::function<std::array<Scalar, 3>(const GlobalOrdinal)> ↩
                    nodesFine
107                     = [&cubeMeshFine](const GlobalOrdinal i) {
108                     std::array<GlobalOrdinal, 3> triple;
109                     cubeMeshFine.deIndexNodes(i, &triple);
110                     std::array<Scalar, 3> ret = cubeMeshFine.origin;
111                     for (std::int8_t dim = 0; dim < 3; dim++) { ret[dim] += ↩
                            cubeMeshFine.unitLength * triple[dim]; }
112                     return ret;
113             };
114             const IdMatrix<GlobalOrdinal> &idMatrixFine = idMatrices[iHeir-1];
115             const CubeMesh<Scalar, GlobalOrdinal> &cubeMeshCoarse = cubeMeshes↩
                    [iHeir];
116             IdMatrix<GlobalOrdinal> &idMatrixCoarse = idMatrices[iHeir];
```

```
117                 Eigen::SparseMatrix<Scalar, 0, GlobalOrdinal> &interpolationMatrix↩
                        = interpolationMatrices[iHeir];
118
119                 _buildInterpolation<Scalar, GlobalOrdinal, CARDINALITY>(
120                         nodesFine,
121                         idMatrixFine,
122                         cubeMeshCoarse,
123                         idMatrixCoarse,
124                         interpolationMatrix
125                 );
126         }
127 }
128
129 } //namespace Mesoscale
130 } //namespace KMGEM
131
132 #endif /* KMGEM_MESOSCALE_IDMATRIXCUBEMESH_H_ */
```

Listing B.22: KMGEM/Mesoscale/Mesh.h

```
#ifndef KMGEM_MESOSCALE_MESH_H_
#define KMGEM_MESOSCALE_MESH_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "metis.h"
#include "mpi.h"
#include "H5Cpp.h"
#include "tetgen.h"

#include "HardinUtil2/PermuteInPlace.h"

#include "KMGEM/Mesoscale/CylinderPLC.h"

namespace KMGEM {
namespace Mesoscale {

template <
        class Scalar,
        class GlobalOrdinal,
        std::int8_t CARDINALITY
> class Mesh {
public:
        std::vector<std::array<GlobalOrdinal, CARDINALITY>> elements;
        std::vector<std::array<GlobalOrdinal, 3>> boundaries;
        std::vector<std::array<Scalar, 3>> nodes;
        std::vector<std::vector<GlobalOrdinal>> nodeElements;
        std::vector<GlobalOrdinal> boundaryElements;
        std::array<Scalar, 3> boundsLo;
        std::array<Scalar, 3> boundsHi;
        double volumeFinest;
};

template <
```

```
36          class Scalar,
37          class GlobalOrdinal,
38          std::int8_t CARDINALITY
39  > void buildmesh(Mesh<Scalar, GlobalOrdinal, CARDINALITY> &mesh) {
40          double cylinderRadius = 5;
41          double cylinderLength = 30;//ini.getVar<double>("geometry/length");
42          GlobalOrdinal cylinderNTheta = 12;//ini.getVar<GlobalOrdinal>("geometry/↩
                nTheta");
43          double volumeFinest = .25;
44
45          mesh.volumeFinest = volumeFinest;
46
47          //Get PLC corresponding to a cylinder
48          tetgenio tetgenPLC;
49          KMGEM::Mesoscale::getCylinderPlc(cylinderRadius, cylinderLength, ↩
                cylinderNTheta, &tetgenPLC);
50
51          //Tetrahedralize
52          std::string commandString = "pq1.4a" + std::to_string(volumeFinest) + "09↩
                /7zQ";
53          if (CARDINALITY == 10) { commandString += "o2"; }
54          std::vector<char> commandLineVector(commandString.begin(), commandString.↩
                end());
55          commandLineVector.push_back('\0');
56          tetgenbehavior behavior;
57          behavior.parse_commandline(commandLineVector.data());
58          tetgenio tetgenMesh;
59          tetrahedralize(&behavior, &tetgenPLC, &tetgenMesh);
60
61          //Copy the mesh
62          GlobalOrdinal nElements = tetgenMesh.numberoftetrahedra;
63          mesh.elements.resize(nElements);
64          for (GlobalOrdinal e = 0; e < nElements; e++) for (std::int8_t a = 0; a <↩
                CARDINALITY; a++) {
65              mesh.elements[e][a] = tetgenMesh.tetrahedronlist[e*CARDINALITY + a↩
                    ];
```

```
66          }
67
68          GlobalOrdinal nBoundaries = tetgenMesh.numberoftrifaces;
69          mesh.boundaries.resize(nBoundaries);
70          for (GlobalOrdinal b = 0; b < nBoundaries; b++) for (std::int8_t a = 0; a<↩
                 < 3; a++) {
71              mesh.boundaries[b][a] = tetgenMesh.trifacelist[b * 3 + a];
72          }
73
74          GlobalOrdinal nNodes = tetgenMesh.numberofpoints;
75          mesh.nodes.resize(nNodes);
76          for (GlobalOrdinal i = 0; i < nNodes; i++) for (std::int8_t dim = 0; dim ↩
                < 3; dim++) {
77              mesh.nodes[i][dim] = tetgenMesh.pointlist[i * 3 + dim];
78          }
79
80          //Sort nodes
81          static_assert(sizeof(GlobalOrdinal) == sizeof(idx_t), "Metis␣index␣type␣↩
                is␣incompatible␣with␣mesh");
82          {
83              std::vector<idx_t> perm(mesh.nodes.size());
84              std::vector<idx_t> iperm(mesh.nodes.size());
85              {
86                  int err;
87
88                  idx_t ne = mesh.elements.size();
89                  idx_t nn = mesh.nodes.size();
90
91                  idx_t* eind = mesh.elements.data()->data();
92
93                  std::vector<idx_t> eptrVector(ne + 1);
94                  for (idx_t e = 0; e < ne + 1; e++) { eptrVector[e] = ↩
                        CARDINALITY * e; }
95                  idx_t* eptr = eptrVector.data();
96
97                  idx_t numflag = 0;
```

```
98
99                      idx_t* xadj;
100                     idx_t* xadjncy;
101
102                     err = METIS_MeshToNodal(&ne, &nn, eptr, eind, &numflag, &↩
                            xadj, &xadjncy); assert(err == METIS_OK);
103
104                     err = METIS_NodeND(&nn, xadj, xadjncy, NULL, NULL, perm.↩
                            data(), iperm.data()); assert(err == METIS_OK);
105
106                     err = METIS_Free(xadj); assert(err == METIS_OK);
107                     err = METIS_Free(xadjncy); assert(err == METIS_OK);
108             }
109
110             //Reorder the nodes
111             std::vector<bool> workspace;
112             HardinUtil2::permuteInverseInPlace<idx_t, std::array<double, 3>>(↩
                    mesh.nodes.size(), perm.data(), mesh.nodes.data(), workspace);
113
114             //Update node numbers in tetrahedra
115             for (auto &element : mesh.elements) for (auto &iNode : element) { ↩
                    iNode = iperm[iNode]; }
116
117             //Update node numbers in boundaries
118             for (auto &boundary : mesh.boundaries) for (auto &iNode : boundary↩
                    ) { iNode = iperm[iNode]; }
119     }
120
121     //Reorder the elements using lexographic ordering
122     std::sort(mesh.elements.begin(), mesh.elements.end(),
123             [](const std::array<GlobalOrdinal, CARDINALITY> & lhs, const std::↩
                    array<GlobalOrdinal, CARDINALITY> & rhs) {
124             std::array<GlobalOrdinal, CARDINALITY> lhsSorted = lhs; std::sort(↩
                    lhsSorted.begin(), lhsSorted.end());
125             std::array<GlobalOrdinal, CARDINALITY> rhsSorted = rhs; std::sort(↩
                    rhsSorted.begin(), rhsSorted.end());
```

```
126             return std::lexicographical_compare(lhsSorted.begin(), lhsSorted.↩
                    end(), rhsSorted.begin(), rhsSorted.end());
127         });
128
129         //Generate a node->elements map
130         mesh.nodeElements.resize(nNodes);
131         for (GlobalOrdinal e = 0; e < nElements; e++) for (auto i : mesh.elements↩
                [e]) { mesh.nodeElements[i].push_back(e); }
132
133         //Generate a boundary->elements map
134         {
135             mesh.boundaryElements.resize(nBoundaries);
136             for (GlobalOrdinal b = 0; b < nBoundaries; b++) {
137                 std::array<GlobalOrdinal, 3> boundary = mesh.boundaries[b↩
                        ];
138                 std::sort(boundary.begin(), boundary.end());
139                 GlobalOrdinal boundaryNode = boundary[0];
140                 std::vector<GlobalOrdinal> &possibleElements = mesh.↩
                        nodeElements[boundaryNode];
141
142                 for (GlobalOrdinal e : possibleElements) {
143                     std::array<GlobalOrdinal, CARDINALITY> element = ↩
                            mesh.elements[e];
144                     std::sort(element.begin(), element.end());
145
146                     bool includes = std::includes(
147                             element.begin(), element.end(),
148                             boundary.begin(), boundary.end()
149                     );
150
151                     if (includes) {
152                         mesh.boundaryElements[b] = e;
153                         break;
154                     }
155                 }
156             }
```

```
157         }
158
159         //Reorder the boundaries by element number
160         //Partition the boundaries according to elements
161         {
162                 std::vector<GlobalOrdinal> boundaryPermutation(nBoundaries);
163                 for (GlobalOrdinal b = 0; b < nBoundaries; b++) { ↩
                        boundaryPermutation[b] = b; }
164                 std::sort(boundaryPermutation.begin(), boundaryPermutation.end(), ↩
                        [&mesh](const GlobalOrdinal a, const GlobalOrdinal b) {return ↩
                        mesh.boundaryElements[a] < mesh.boundaryElements[b]; });
165
166                 std::vector<bool> workspace;
167                 HardinUtil2::permuteInverseInPlace<GlobalOrdinal, std::array<↩
                        GlobalOrdinal, 3>>(nBoundaries, boundaryPermutation.data(), ↩
                        mesh.boundaries.data(), workspace);
168                 HardinUtil2::permuteInverseInPlace<GlobalOrdinal, GlobalOrdinal>(↩
                        nBoundaries, boundaryPermutation.data(), mesh.boundaryElements↩
                        .data(), workspace);
169         }
170
171         for (std::int8_t dim = 0; dim < 3; dim++) {
172                 mesh.boundsLo[dim] = std::numeric_limits<Scalar>::infinity();
173                 mesh.boundsHi[dim] = -std::numeric_limits<Scalar>::infinity();
174         }
175         for (GlobalOrdinal iNode = 0; iNode < nNodes; iNode++) {
176                 for (std::int8_t dim = 0; dim < 3; dim++) {
177                         double val = mesh.nodes[iNode][dim];
178                         if (val < mesh.boundsLo[dim]) { mesh.boundsLo[dim] = val; ↩
                                }
179                         if (val > mesh.boundsHi[dim]) { mesh.boundsHi[dim] = val; ↩
                                }
180                 }
181         }
182 }
183
```

```
184
185 } //namespace Mesoscale
186 } //namespace KMGEM
187
188 #endif /* KMGEM_MESOSCALE_MESH_H_ */
```

Listing B.23: KMGEM/Mesoscale/Multigrid.h

```
#ifndef KMGEM_MESOSCALE_MULTIGRID_H_
#define KMGEM_MESOSCALE_MULTIGRID_H_

#include "KMGEM/Core.h"
#include "KMGEM/Kinetics.h"

#include "Eigen/Core"

#include "metis.h"
#include "mpi.h"
#include "H5Cpp.h"
#include "tetgen.h"

#include "KMGEM/Mesoscale/GaussSeidel.h"

namespace KMGEM {
namespace Mesoscale {

template <
        class SparseMat,
        class Vect
>
void multigrid(
        const std::vector<const SparseMat*> lhsMatrixHeirarchy,
        const Vect rhsFinest,
        const std::vector<const SparseMat*> interpolationMatrices,
        const std::vector<int> &multigridPlan,
        const int nGaussSeidelDualSweeps,
        Vect &x
        ) {
        if (x.rows() != rhsFinest.rows()) { x.setZero(rhsFinest.rows()); }

        std::vector<Vect> rhsHeirarchy(lhsMatrixHeirarchy.size());
        rhsHeirarchy[0] = rhsFinest;
        std::vector<Vect> xHeirarchy(lhsMatrixHeirarchy.size());
```

```
36          xHeirarchy[0] = x;
37
38          //Step 1. Tunnel down to the starting level, building up RHS and x
39          int startingLevel = multigridPlan.front();
40          for (int iHeir = 1; iHeir <= startingLevel; iHeir++) {
41                  //Get the interpolation matrix from iHeir to iHeir-1
42                  const SparseMat &interpolationMatrix = *(interpolationMatrices[↩
                        iHeir-1]);
43
44                  //Get the residual from the finer level
45                  Vect &rhsFiner = rhsHeirarchy[iHeir - 1];
46                  Vect &xFiner = xHeirarchy[iHeir - 1];
47                  const SparseMat &lhsMatrixFiner = *(lhsMatrixHeirarchy[iHeir - 1])↩
                        ;
48                  Vect residualFiner = rhsFiner - lhsMatrixFiner * xFiner;
49
50                  //Restrict the finer residual to the coarser level
51                  rhsHeirarchy[iHeir] = interpolationMatrix.transpose() * ↩
                        residualFiner;
52                  xHeirarchy[iHeir] = Eigen::VectorXd::Zero(rhsHeirarchy[iHeir].rows↩
                        ());
53          }
54
55          //Step 2. Execute the multigrid plan
56          //Smooth the first level
57          gaussSeidelDual(nGaussSeidelDualSweeps, *(lhsMatrixHeirarchy[↩
                startingLevel]), rhsHeirarchy[startingLevel], xHeirarchy[↩
                startingLevel]);
58          {
59                  double res = (rhsHeirarchy[startingLevel] - *(lhsMatrixHeirarchy[↩
                        startingLevel]) * xHeirarchy[startingLevel]).norm();
60                  std::cout << startingLevel << ',' << res << std::endl;
61          }
62          for (int iMultigrid = 1; iMultigrid < multigridPlan.size(); iMultigrid++)↩
                {
63                  int iHeirPrev = multigridPlan[iMultigrid - 1];
```

```
64              int iHeir = multigridPlan[iMultigrid];
65              if (iHeir == iHeirPrev + 1) { //If we're coarsening, restrict the ↩
                    residual from the previous level to solve against
66                  //Get the interpolation matrix
67                  const SparseMat &interpolationMatrix = *(↩
                        interpolationMatrices[iHeir - 1]);
68
69                  //Get the residual from the finer level
70                  Vect &rhsFiner = rhsHeirarchy[iHeir - 1];
71                  Vect &xFiner = xHeirarchy[iHeir - 1];
72                  const SparseMat &lhsMatrixFiner = *(lhsMatrixHeirarchy[↩
                        iHeir - 1]);
73                  Vect residualFiner = rhsFiner - lhsMatrixFiner * xFiner;
74
75                  //Restrict the finer residual to the coarser level
76                  rhsHeirarchy[iHeir] = interpolationMatrix.transpose() * ↩
                        residualFiner;
77                  xHeirarchy[iHeir] = Eigen::VectorXd::Zero(rhsHeirarchy[↩
                        iHeir].rows());
78              }
79              else if (iHeir == iHeirPrev - 1) { //If we're refining, ↩
                    interpolate the coarser solution and add it in
80                  const SparseMat &interpolationMatrix = *(↩
                        interpolationMatrices[iHeir]);
81                  Vect &xCoarser = xHeirarchy[iHeir+1];
82                  xHeirarchy[iHeir] += interpolationMatrix * xCoarser;
83              }
84              else { throw("ERROR:␣Invalid␣multigrid␣plan."); }
85
86              gaussSeidelDual(nGaussSeidelDualSweeps, *(lhsMatrixHeirarchy[iHeir↩
                    ]), rhsHeirarchy[iHeir], xHeirarchy[iHeir]);
87
88              for (int i = iHeir; i >= 0; i--) {
89
90
91
```

```
92                }
93
94                {
95                        double res = (rhsHeirarchy[iHeir] - *(lhsMatrixHeirarchy[↩
                              iHeir]) * xHeirarchy[iHeir]).norm();
96                        std::cout << iHeir << ',' << res << std::endl;
97                }
98        }
99
100       x = xHeirarchy[0];
101
102 }
103
104 template <
105       class SparseMat,
106       class Vect
107 >
108 void multigrid_FMG(
109       const std::vector<const SparseMat*> lhsMatrixHeirarchy,
110       const Vect rhsFinest,
111       const std::vector<const SparseMat*> interpolationMatrices,
112       const int nGaussSeidelDualSweeps,
113       Vect &x
114 ) {
115       int nHeir = lhsMatrixHeirarchy.size();
116
117       std::vector<int> multigridPlan;
118
119       multigridPlan.push_back(nHeir - 1);
120       for (int i = 0; i < nHeir - 1; i++) {
121               for (int j = nHeir - 2; j > nHeir - 2 - i; j--) {
122                       multigridPlan.push_back(j);
123               }
124               for (int j = nHeir - 2 - i; j < nHeir; j++) {
125                       multigridPlan.push_back(j);
126               }
```

```
127         }
128         for (int i = nHeir - 2; i >= 0; i--) { multigridPlan.push_back(i); }
129
130         multigrid<SparseMat,Vect>(
131                 lhsMatrixHeirarchy,
132                 rhsFinest,
133                 interpolationMatrices,
134                 multigridPlan,
135                 nGaussSeidelDualSweeps,
136                 x);
137
138
139 }
140
141 } //namespace Mesoscale
142 } //namespace KMGEM
143
144 #endif /* KMGEM_MESOSCALE_MULTIGRID_H_ */
```

Bibliography

[1] Christopher A Schuh and Alan C Lund. Atomistic basis for the plastic yield criterion of metallic glass. *Nature materials*, 2(7):449, 2003.

[2] Daniel B Miracle, Takeshi Egami, Katharine M Flores, and Kenneth F Kelton. Structural aspects of metallic glasses. *MRS bulletin*, 32(8):629–634, 2007.

[3] Jun Ding, Sylvain Patinet, Michael L Falk, Yongqiang Cheng, and Evan Ma. Soft spots and their structural signature in a metallic glass. *Proceedings of the National Academy of Sciences*, 111(39):14052–14056, 2014.

[4] Peter Tsai, Kelly Kranjc, and Katharine M Flores. Hierarchical heterogeneity and an elastic microstructure observed in a metallic glass alloy. *Acta Materialia*, 139:11–20, 2017.

[5] SV Ketov, YH Sun, S Nachum, Z Lu, A Checchi, AR Beraldin, HY Bai, WH Wang, DV Louzguine-Luzgin, Michael Allan Carpenter, et al. Rejuvenation of metallic glasses by non-affine thermal strain. *Nature*, 524(7564):200, 2015.

[6] Pengyang Zhao, Ju Li, Jinwoo Hwang, and Yunzhi Wang. Influence of nanoscale structural heterogeneity on shear banding in metallic glasses. *Acta Materialia*, 134:104–115, 2017.

[7] YQ Cheng, HW Sheng, and E Ma. Relationship between structure, dynamics, and mechanical properties in metallic glass-forming alloys. *Physical Review B*, 78(1):014207, 2008.

[8] Weidong Li, Yanfei Gao, and Hongbin Bei. On the correlation between microscopic structural heterogeneity and embrittlement behavior in metallic glasses. *Scientific reports*, 5:14786, 2015.

[9] AI Taub and F Spaepen. The kinetics of structural relaxation of a metallic glass. *Acta Metallurgica*, 28(12):1781–1788, 1980.

[10] Frans Spaepen. A microscopic mechanism for steady state inhomogeneous flow in metallic glasses. *Acta metallurgica*, 25(4):407–415, 1977.

[11] Frans Spaepen. Homogeneous flow of metallic glasses: A free volume perspective. *Scripta materialia*, 54(3):363–367, 2006.

[12] A Slipenyuk and J Eckert. Correlation between enthalpy change and free volume reduction during structural relaxation of zr55cu30al10ni5 metallic glass. *Scripta materialia*, 50(1):39–44, 2004.

[13] YQ Cheng and E Ma. Indicators of internal structural states for metallic glasses: Local order, free volume, and configurational potential energy. *Applied Physics Letters*, 93(5):051910, 2008.

[14] Gang Duan, Mary Laura Lind, Marios D Demetriou, William L Johnson, William A Goddard III, Tahir Çağin, and Konrad Samwer. Strong configurational dependence of elastic properties for a binary model metallic glass. *Applied physics letters*, 89(15):151901, 2006.

[15] John S Harmon, Marios D Demetriou, William L Johnson, and Min Tao. Deformation of glass forming metallic liquids: Configurational changes and their relation to elastic softening. *Applied physics letters*, 90(13):131912, 2007.

[16] DJ Magagnosc, G Kumar, J Schroers, P Felfer, JM Cairney, and DS Gianola. Effect of ion irradiation on tensile ductility, strength and fictive temperature in metallic glass nanowires. *Acta Materialia*, 74:165–182, 2014.

[17] William L Johnson, Marios D Demetriou, John S Harmon, Mary L Lind, and Konrad Samwer. Rheology and ultrasonic properties of metallic glass-forming liquids: A potential energy landscape perspective. *MRS bulletin*, 32(8):644–650, 2007.

[18] Jun Ding, Yong-Qiang Cheng, Howard Sheng, Mark Asta, Robert O Ritchie, and Evan Ma. Universal structural parameter to quantitatively predict metallic glass properties. *Nature communications*, 7:13733, 2016.

[19] Zhao Fan, Jun Ding, Qing-Jie Li, and Evan Ma. Correlating the properties of amorphous silicon with its flexibility volume. *Physical Review B*, 95(14):144211, 2017.

[20] Qiong Zhang and Ken Kamrin. Microscopic description of the granular fluidity field in nonlocal flow modeling. *Physical review letters*, 118(5):058001, 2017.

[21] Jun Ding, Yong Qiang Cheng, and Evan Ma. Quantitative measure of local solidity/liquidity in metallic glasses. *Acta Materialia*, 61(12):4474–4480, 2013.

[22] Evan Ma. Tuning order in disorder. *Nature materials*, 14(6):547, 2015.

[23] YQ Cheng, AJ Cao, and E Ma. Correlation between the elastic modulus and the intrinsic plastic behavior of metallic glasses: The roles of atomic configuration and alloy composition. *Acta materialia*, 57(11):3253–3267, 2009.

[24] SD Feng, KC Chan, and RP Liu. Quantifying the microstructural inhomogeneity of zr46cu46al8 metallic glasses based on change ratio of polyhedral volume. *Journal of Alloys and Compounds*, 731:452–457, 2018.

[25] Yun-Fei Mo, Rang-Su Liu, Ze-An Tian, Yong-Chao Liang, Hai-Tao Zhang, Zhao-Yang Hou, Hai-Rong Liu, Ai-long Zhang, Li-Li Zhou, Ping Peng, et al. Non-linear effects of initial melt temperatures on microstructures and mechanical properties during quenching process of liquid cu46zr54 alloy. *Physica B: Condensed Matter*, 465:81–88, 2015.

[26] J Ashwin, Eran Bouchbinder, and Itamar Procaccia. Cooling-rate dependence of the shear modulus of amorphous solids. *Physical Review E*, 87(4):042310, 2013.

[27] JM Pelletier, S Cardinal, JC Qiao, M Eisenbart, and UE Klotz. Main and secondary relaxations in an au-based bulk metallic glass investigated by mechanical spectroscopy. *Journal of Alloys and Compounds*, 684:530–536, 2016.

[28] Cristian Rodríguez-Tinoco, Joan Ràfols-Ribé, Marta González-Silveira, and Javier Rodríguez-Viejo. Relaxation dynamics of glasses along a wide stability and temperature range. *Scientific reports*, 6:35607, 2016.

[29] LS Huo, JF Zeng, WH Wang, Chain Tsuan Liu, and Yong Yang. The dependence of shear modulus on dynamic relaxation and evolution of local structural heterogeneity in a metallic glass. *Acta Materialia*, 61(12):4329–4338, 2013.

[30] ZF Yao, JC Qiao, JM Pelletier, and Y Yao. Characterization and modeling of dynamic relaxation of a zr-based bulk metallic glass. *Journal of Alloys and Compounds*, 690:212–220, 2017.

[31] Zach Evenson, Tönjes Koschine, Shuai Wei, Oliver Gross, Jozef Bednarcik, Isabella Gallino, Jamie J Kruzic, Klaus Rätzke, Franz Faupel, and Ralf Busch. The effect of low-temperature structural relaxation on free volume and chemical short-range ordering in a au49cu26. 9si16. 3ag5. 5pd2. 3 bulk metallic glass. *Scripta Materialia*, 103:14–17, 2015.

[32] O Haruyama, T Mottate, K Morita, N Yamamoto, H Kato, and T Egami. Volume and enthalpy relaxation in pd42. 5cu30ni7. 5p20 bulk metallic glass. *Materials Transactions*, 55(3):466–472, 2014.

[33] O Haruyama, Y Nakayama, R Wada, H Tokunaga, J Okada, T Ishikawa, and Y Yokoyama. Volume and enthalpy relaxation in zr55cu30ni5al10 bulk metallic glass. *Acta Materialia*, 58(5):1829–1836, 2010.

[34] E Ma and J Ding. Tailoring structural inhomogeneities in metallic glasses to enable tensile ductility at room temperature. *Materials Today*, 19(10):568–579, 2016.

[35] Christopher A Schuh, Todd C Hufnagel, and Upadrasta Ramamurty. Mechanical behavior of amorphous alloys. *Acta Materialia*, 55(12):4067–4109, 2007.

[36] Jamie J Kruzic. Bulk metallic glasses as structural materials: A review. *Advanced Engineering Materials*, 18(8):1308–1331, 2016.

[37] Todd C Hufnagel, Christopher A Schuh, and Michael L Falk. Deformation of metallic glasses: Recent developments in theory, simulations, and experiments. *Acta Materialia*, 109:375–393, 2016.

[38] H-U Künzi. Mechanical properties of metallic glasses. In *Glassy Metal II*, pages 169–216. Springer, 1983.

[39] YQ Cheng and E Ma. Configurational dependence of elastic modulus of metallic glass. *Physical Review B*, 80(6):064104, 2009.

[40] J Ding, YQ Cheng, and E Ma. Correlating local structure with inhomogeneous elastic deformation in a metallic glass. *Applied Physics Letters*, 101(12):121917, 2012.

[41] Mary Laura Lind, Gang Duan, and William L Johnson. Isoconfigurational elastic constants and liquid fragility of a bulk metallic glass forming alloy. *Physical review letters*, 97(1):015501, 2006.

[42] BA Sun, YC Hu, DP Wang, ZG Zhu, P Wen, WH Wang, CT Liu, and Y Yang. Correlation between local elastic heterogeneities and overall elastic properties in metallic glasses. *Acta Materialia*, 121:266–276, 2016.

[43] C Wang, ZZ Yang, T Ma, YT Sun, YY Yin, Y Gong, L Gu, P Wen, PW Zhu, YW Long, et al. High stored energy of metallic glasses induced by high pressure. *Applied Physics Letters*, 110(11):111901, 2017.

[44] Yuan-Chao Hu, Peng-Fei Guan, Qing Wang, Yong Yang, Hai-Yang Bai, and Wei-Hua Wang. Pressure effects on structure and dynamics of metallic glass-forming liquid. *The Journal of chemical physics*, 146(2):024507, 2017.

[45] DV Louzguine-Luzgin, V Yu Zadorozhnyy, SV Ketov, Z Wang, AA Tsarkov, and AL Greer. On room-temperature quasi-elastic mechanical behaviour of bulk metallic glasses. *Acta Materialia*, 129:343–351, 2017.

[46] Alan Lindsay Greer and YH Sun. Stored energy in metallic glasses due to strains within the elastic limit. *Philosophical Magazine*, 96(16):1643–1663, 2016.

[47] AS Argon. Plastic deformation in metallic glasses. *Acta metallurgica*, 27(1):47–58, 1979.

[48] AS Argon and HY Kuo. Plastic flow in a disordered bubble raft (an analog of a metallic glass). *Materials science and Engineering*, 39(1):101–109, 1979.

[49] AS Argon. Plastic deformation in metallic glasses. *Acta metallurgica*, 27(1):47–58, 1979.

[50] AS Argon and HY Kuo. Free energy spectra for inelastic deformation of five metallic glass alloys. *Journal of Non-Crystalline Solids*, 37(2):241–266, 1980.

[51] AS Argon and L To Shi. Development of visco-plastic deformation in metallic glasses. *Acta Metallurgica*, 31(4):499–507, 1983.

[52] C Zhong, H Zhang, QP Cao, XD Wang, DX Zhang, U Ramamurty, and JZ Jiang. Size distribution of shear transformation zones and their evolution towards the formation of shear bands in metallic glasses. *Journal of Non-Crystalline Solids*, 445:61–68, 2016.

[53] Alan C Lund and Christopher A Schuh. Yield surface of a simulated metallic glass. *Acta Materialia*, 51(18):5399–5411, 2003.

[54] AC Lund and CA Schuh. The mohr–coulomb criterion from unit shear processes in metallic glass. *Intermetallics*, 12(10-11):1159–1165, 2004.

[55] David Rodney and Christopher A Schuh. Yield stress in metallic glasses: The jamming-unjamming transition studied through monte carlo simulations based on the activation-relaxation technique. *Physical Review B*, 80(18):184203, 2009.

[56] Pengyang Zhao, Ju Li, and Yunzhi Wang. Heterogeneously randomized stz model of metallic glasses: Softening and extreme value statistics during deformation. *International Journal of Plasticity*, 40:1–22, 2013.

[57] CE Packard and CA Schuh. Initiation of shear bands near a stress concentration in metallic glass. *Acta Materialia*, 55(16):5348–5358, 2007.

[58] GN Yang, BA Sun, SQ Chen, Y Shao, and KF Yao. The multiple shear bands and plasticity in metallic glasses: A possible origin from stress redistribution. *Journal of Alloys and Compounds*, 695:3457–3466, 2017.

[59] M Stolpe, JJ Kruzic, and R Busch. Evolution of shear bands, free volume and hardness during cold rolling of a zr-based bulk metallic glass. *Acta Materialia*, 64:231–240, 2014.

[60] Bo Shi, Yuanli Xu, Chao Li, Wentao Jia, Zhaoqing Li, and Jiangong Li. Evolution of free volume and shear band intersections and its effect on hardness of deformed zr64. 13cu15. 75ni10. 12al10 bulk metallic glass. *Journal of Alloys and Compounds*, 669:167–176, 2016.

[61] Chunguang Tang, Hailong Peng, Yu Chen, and Michael Ferry. Formation and dilatation of shear bands in a cu-zr metallic glass: A free volume perspective. *Journal of Applied Physics*, 120(23):235101, 2016.

[62] Hongbin Bei, Sujing Xie, and Easo P George. Softening caused by profuse shear banding in a bulk metallic glass. *Physical review letters*, 96(10):105503, 2006.

[63] JW Liu, QP Cao, LY Chen, XD Wang, and JZ Jiang. Shear band evolution and hardness change in cold-rolled bulk metallic glasses. *Acta Materialia*, 58(14):4827–4840, 2010.

[64] O Haruyama, K Kisara, A Yamashita, K Kogure, Y Yokoyama, and K Sugiyama. Characterization of free volume in cold-rolled zr55cu30ni5al10 bulk metallic glasses. *Acta Materialia*, 61(9):3224–3232, 2013.

[65] Harald Rösner, Martin Peterlechner, Christian Kübel, Vitalij Schmidt, and Gerhard Wilde. Density changes in shear bands of a metallic glass determined by correlative analytical transmission electron microscopy. *Ultramicroscopy*, 142:1–9, 2014.

[66] MQ Jiang, G Wilde, and LH Dai. Shear band dilatation in amorphous alloys. *Scripta Materialia*, 127:54–57, 2017.

[67] MQ Jiang, G Wilde, F Jiang, and LH Dai. Understanding ductile-to-brittle transition of metallic glasses from shear transformation zone dilatation. *Theoretical and Applied Mechanics Letters*, 5(5):200–204, 2015.

[68] I Binkowski, GP Shrivastav, J Horbach, SV Divinski, and G Wilde. Shear band relaxation in a deformed bulk metallic glass. *Acta Materialia*, 109:330–340, 2016.

[69] Jing Li, F Spaepen, and TC Hufnagel. Nanometre-scale defects in shear bands in a metallic glass. *Philosophical Magazine A*, 82(13):2623–2630, 2002.

[70] Yang Song, Xie Xie, Jiajia Luo, Peter K Liaw, Hairong Qi, and Yanfei Gao. Seeing the unseen: uncover the bulk heterogeneous deformation processes in metallic glasses through surface temperature decoding. *Materials Today*, 20(1):9–15, 2017.

[71] JG Wang, Y Pan, SX Song, BA Sun, G Wang, QJ Zhai, KC Chan, and WH Wang. How hot is a shear band in a metallic glass? *Materials Science and Engineering: A*, 651:321–331, 2016.

[72] Peter Thurnheer, Fabian Haag, and Jörg F Löffler. Time-resolved measurement of shear-band temperature during serrated flow in a zr-based metallic glass. *Acta Materialia*, 115:468–474, 2016.

[73] Michael L Falk and James S Langer. Dynamics of viscoplastic deformation in amorphous solids. *Physical Review E*, 57(6):7192, 1998.

[74] YM Lu, JF Zeng, S Wang, BA Sun, Q Wang, J Lu, S Gravier, JJ Bladin, WH Wang, MX Pan, et al. Structural signature of plasticity unveiled by nanoscale viscoelastic contact in a metallic glass. *Scientific reports*, 6:29357, 2016.

[75] TP Ge, XQ Gao, B Huang, WH Wang, and HY Bai. The role of time in activation of flow units in metallic glasses. *Intermetallics*, 67:47–51, 2015.

[76] TG Nieh, C Schuh, J Wadsworth, and Yi Li. Strain rate-dependent deformation in bulk metallic glasses. *Intermetallics*, 10(11-12):1177–1182, 2002.

[77] Jun Lu, Guruswami Ravichandran, and William L Johnson. Deformation behavior of the zr41. 2ti13. 8cu12. 5ni10be22. 5 bulk metallic glass over a wide range of strain-rates and temperatures. *Acta materialia*, 51(12):3429–3443, 2003.

[78] JC Qiao, Yun-Jiang Wang, JM Pelletier, Leon M Keer, Morris E Fine, and Y Yao. Characteristics of stress relaxation kinetics of la60ni15al25 bulk metallic glass. *Acta Materialia*, 98:43–50, 2015.

[79] ZT Wang, J Pan, Y Li, and CA Schuh. Densification and strain hardening of a metallic glass under tension at room temperature. *Physical review letters*, 111(13):135504, 2013.

[80] CE Packard, LM Witmer, and CA Schuh. Hardening of a metallic glass during cyclic loading in the elastic range. *Applied Physics Letters*, 92(17):171911, 2008.

[81] CE Packard, ER Homer, N Al-Aqeeli, and CA Schuh. Cyclic hardening of metallic glasses under hertzian contacts: Experiments and stz dynamics simulations. *Philosophical Magazine*, 90(10):1373–1390, 2010.

[82] Glenn R Garrett, Marios D Demetriou, Maximilien E Launey, and William L Johnson. Origin of embrittlement in metallic glasses. *Proceedings of the National Academy of Sciences*, 113(37):10257–10262, 2016.

[83] Wojciech Dmowski, Y Yokoyama, A Chuang, Y Ren, M Umemoto, K Tsuchiya, A Inoue, and T Egami. Structural rejuvenation in a bulk metallic glass induced by severe plastic deformation. *Acta Materialia*, 58(2):429–438, 2010.

[84] XL Bian, G Wang, J Yi, YD Jia, J Bednarčík, QJ Zhai, I Kaban, B Sarac, M Mühlbacher, F Spieckermann, et al. Atomic origin for rejuvenation of a zr-based metallic glass at cryogenic temperature. *Journal of Alloys and Compounds*, 718:254–259, 2017.

[85] Perry Ross, Stefan Küchemann, Peter M Derlet, HaiBin Yu, Walter Arnold, Peter Liaw, Konrad Samwer, and Robert Maass. Linking macroscopic rejuvenation to nano-elastic fluctuations in a metallic glass. *Acta Materialia*, 138:111–118, 2017.

[86] YM Wang, M Zhang, and L Liu. Mechanical annealing in the homogeneous deformation of bulk metallic glass under elastostatic compression. *Scripta Materialia*, 102:67–70, 2015.

[87] Junji Saida, Rui Yamada, Masato Wakeda, and Shigenobu Ogata. Thermal rejuvenation in metallic glasses. *Science and Technology of advanced MaTerialS*, 18(1):152–162, 2017.

[88] DJ Magagnosc, R Ehrbar, G Kumar, MR He, J Schroers, and DS Gianola. Tunable tensile ductility in metallic glasses. *Scientific reports*, 3:1096, 2013.

[89] Jun Ding and En Ma. Computational modeling sheds light on structural evolution in metallic glasses and supercooled liquids. *npj Computational Materials*, 3(1):9, 2017.

[90] Alexei Vinogradov, Mikhail Seleznev, and Igor S Yasnikov. Dislocation characteristics of shear bands in metallic glasses. *Scripta Materialia*, 130:138–142, 2017.

[91] John D Eshelby. The determination of the elastic field of an ellipsoidal inclusion, and related problems. *Proc. R. Soc. Lond. A*, 241(1226):376–396, 1957.

[92] Yun-Jiang Wang, MQ Jiang, ZL Tian, and LH Dai. Direct atomic-scale evidence for shear–dilatation correlation in metallic glasses. *Scripta Materialia*, 112:37–41, 2016.

[93] Francesco Delogu. Thermal and mechanical activation of inelastic events in metallic glasses. *Scripta Materialia*, 113:145–149, 2016.

[94] Eric R Homer and Christopher A Schuh. Mesoscale modeling of amorphous metals by shear transformation zone dynamics. *Acta Materialia*, 57(9):2823–2833, 2009.

[95] Ezequiel E Ferrero, Kirsten Martens, and Jean-Louis Barrat. Relaxation in yield stress systems through elastically interacting activated events. *Physical review letters*, 113(24):248301, 2014.

[96] VV Bulatov and AS Argon. A stochastic model for continuum elasto-plastic behavior. i. numerical approach and strain localization. *Modelling and Simulation in Materials Science and Engineering*, 2(2):167, 1994.

[97] VV Bulatov and AS Argon. A stochastic model for continuum elasto-plastic behavior. ii. a study of the glass transition and structural relaxation. *Modelling and Simulation in Materials Science and Engineering*, 2(2):185, 1994.

[98] VV Bulatov and AS Argon. A stochastic model for continuum elasto-plastic behavior. iii. plasticity in ordered versus disordered solids. *Modelling and Simulation in Materials Science and Engineering*, 2(2):203, 1994.

[99] L Li, ER Homer, and CA Schuh. Shear transformation zone dynamics model for metallic glasses incorporating free volume as a state variable. *Acta Materialia*, 61(9):3347–3359, 2013.

[100] Eric R Homer, David Rodney, and Christopher A Schuh. Kinetic monte carlo study of activated states and correlated shear-transformation-zone activity during the deformation of an amorphous metal. *Physical Review B*, 81(6):064204, 2010.

[101] Eric R Homer. Examining the initial stages of shear localization in amorphous metals. *Acta Materialia*, 63:44–53, 2014.

[102] Eric R Homer and Christopher A Schuh. Three-dimensional shear transformation zone dynamics model for amorphous metals. *Modelling and Simulation in Materials Science and Engineering*, 18(6):065009, 2010.

[103] Thomas J Hardin and Eric R Homer. Microstructural factors of strain delocalization in model metallic glass matrix composites. *Acta Materialia*, 83:203–215, 2015.

[104] Lin Li, Neng Wang, and Feng Yan. Transient response in metallic glass deformation: a study based on shear transformation zone dynamics simulations. *Scripta Materialia*, 80:25–28, 2014.

[105] Eric R Homer, Lin Li, and Christopher A Schuh. Kinetic monte carlo modeling of nanomechanics in amorphous systems. In *Multiscale Materials Modeling for Nanomechanics*, pages 441–468. Springer, 2016.

[106] Babak Kondori, A Amine Benzerga, and Alan Needleman. Discrete shear transformation zone plasticity. *Extreme Mechanics Letters*, 9:21–29, 2016.

[107] Babak Kondori, A Amine Benzerga, and Alan Needleman. Discrete shear-transformation-zone plasticity modeling of notched bars. *Journal of the Mechanics and Physics of Solids*, 111:18–42, 2018.

[108] Frank H Stillinger. A topographic view of supercooled liquids and glass formation. *Science*, 267(5206):1935–1939, 1995.

[109] Francesco Sciortino. Potential energy landscape description of supercooled liquids and glasses. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(05):P05015, 2005.

[110] Yue Fan, Takuya Iwashita, and Takeshi Egami. Energy landscape-driven non-equilibrium evolution of inherent structure in disordered material. *Nature communications*, 8:15417, 2017.

[111] Zachary H Aitken, Mehdi Jafary-Zadeh, John J Lewandowski, and Yong-Wei Zhang. Anharmonic model for the elastic constants of bulk metallic glass across the glass transition. *Physical Review B*, 97(1):014101, 2018.

[112] Hillary L Smith, Chen W Li, Andrew Hoff, Glenn R Garrett, Dennis S Kim, Fred C Yang, Matthew S Lucas, Tabitha Swan-Wood, JYY Lin, MB Stone, et al. Separating the configurational and vibrational entropy contributions in metallic glasses. *Nature Physics*, 13(9):900, 2017.

[113] Toshio Mura. *Micromechanics of defects in solids*. Springer Science & Business Media, 2013.

[114] Henry Eyring. The activated complex in chemical reactions. *The Journal of Chemical Physics*, 3(2):107–115, 1935.

[115] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Physical review*, 140(4A):A1133, 1965.

[116] Wolfram Koch and Max C Holthausen. *A chemist's guide to density functional theory.* John Wiley & Sons, 2015.

[117] IE Fermi, P Pasta, S Ulam, and M Tsingou. Studies of the nonlinear problems. Technical report, Los Alamos Scientific Lab., N. Mex., 1955.

[118] Dennis C Rapaport, Robin L Blumberg, Susan R McKay, Wolfgang Christian, et al. The art of molecular dynamics simulation. *Computers in Physics*, 10(5):456–456, 1996.

[119] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[120] Morton E Gurtin. *An introduction to continuum mechanics*, volume 158. Academic press, 1982.

[121] Robert John Amodeo and Nasr Mostafa Ghoniem. Dislocation dynamics. i. a proposed methodology for deformation micromechanics. *Physical Review B*, 41(10):6958, 1990.

[122] AN Gulluoglu, David J Srolovitz, R LeSar, and PS Lomdahl. Dislocation distributions in two dimensions. *Scripta metallurgica*, 23(8):1347–1352, 1989.

[123] Ladislas P Kubin, G Canova, M Condat, Benoit Devincre, V Pontikis, and Yves Bréchet. Dislocation microstructures and plastic flow: a 3d simulation. In *Solid State Phenomena*, volume 23, pages 455–472. Trans Tech Publ, 1992.

[124] Ryan B Sills, William P Kuykendall, Amin Aghaei, and Wei Cai. Fundamentals of dislocation dynamics simulations. In *Multiscale Materials Modeling for Nanomechanics*, pages 53–87. Springer, 2016.

[125] George J Fix. Phase field methods for free boundary problems. 1982.

[126] Long-Qing Chen. Phase-field models for microstructure evolution. *Annual review of materials research*, 32(1):113–140, 2002.

[127] Nele Moelans, Bart Blanpain, and Patrick Wollants. An introduction to phase-field modeling of microstructure evolution. *Calphad*, 32(2):268–294, 2008.

[128] WM Young and EW Elcock. Monte carlo studies of vacancy migration in binary ordered alloys: I. *Proceedings of the Physical Society*, 89(3):735, 1966.

[129] Alfred B Bortz, Malvin H Kalos, and Joel L Lebowitz. A new algorithm for monte carlo simulation of ising spin systems. *Journal of Computational Physics*, 17(1):10–18, 1975.

[130] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of computational physics*, 22(4):403–434, 1976.

[131] Arthur F Voter. Introduction to the kinetic monte carlo method. *Radiation effects in solids*, pages 1–23, 2007.

[132] Lin Li, Peter M Anderson, Myoung-Gyu Lee, Erik Bitzek, Peter Derlet, and Helena Van Swygenhoven. The stress–strain response of nanocrystalline metals: a quantized crystal plasticity approach. *Acta Materialia*, 57(3):812–822, 2009.

[133] Lin Li, Myoung-Gyu Lee, Peter M Anderson, F Barlat, YH Moon, and MG Lee. A quantized crystal plasticity finite element model for nanocrystalline metals: Connecting atomistic simulations and experiments. In *AIP Conference Proceedings*, volume 1252, pages 841–841. AIP, 2010.

[134] Lin Li, Myoung-Gyu Lee, and Peter M Anderson. Critical strengths for slip events in nanocrystalline metals: predictions of quantized crystal plasticity simulations. *Metallurgical and Materials Transactions A*, 42(13):3875–3882, 2011.

[135] Lin Li, Myoung-Gyu Lee, and Peter M Anderson. Probing the relation between dislocation substructure and indentation characteristics using quantized crystal plasticity. *Journal of Applied Mechanics*, 79(3):031009, 2012.

[136] S Van Petegem, L Li, PM Anderson, and H Van Swygenhoven. Deformation mechanisms in nanocrystalline metals: insights from in-situ diffraction and crystal plasticity modelling. *Thin Solid Films*, 530:20–24, 2013.

[137] Rui Yuan, Irene J Beyerlein, and Caizhi Zhou. Emergence of grain-size effects in nanocrystalline metals from statistical activation of discrete dislocation sources. *Acta Materialia*, 90:169–181, 2015.

[138] Lin Li and Peter M Anderson. Quantized crystal plasticity modeling of nanocrystalline metals. In *Multiscale Materials Modeling for Nanomechanics*, pages 413–440. Springer, 2016.

[139] Ying Chen and Christopher A Schuh. A coupled kinetic monte carlo–finite element mesoscale model for thermoelastic martensitic phase transformations in shape memory alloys. *Acta Materialia*, 83:431–447, 2015.

[140] MP O'day and William A Curtin. A superposition framework for discrete dislocation plasticity. *Transactions of the ASME-E-Journal of Applied Mechanics*, 71(6):805–815, 2004.

[141] LE Shilkrot, William A Curtin, and Ronald E Miller. A coupled atomistic/-continuum model of defects in solids. *Journal of the Mechanics and Physics of Solids*, 50(10):2085–2106, 2002.

[142] Svante Arrhenius. Über die reaktionsgeschwindigkeit bei der inversion von rohrzucker durch säuren. *Zeitschrift für physikalische Chemie*, 4(1):226–248, 1889.

[143] Lin Tian, Xiao-Lei Wang, and Zhi-Wei Shan. Mechanical behavior of micro-nanoscaled metallic glasses. *Materials Research Letters*, 4(2):63–74, 2016.

[144] OV Kuzmin, YT Pei, and J Th M De Hosson. In situ compression study of taper-free metallic glass nanopillars. *Applied Physics Letters*, 98(23):233104, 2011.

[145] Chang Qiang Chen, Yu Tao Pei, Oleksii Kuzmin, Zhe Feng Zhang, Evan Ma, and Jeff Th M De Hosson. Intrinsic size effects in the mechanical response of taper-free nanopillars of metallic glass. *Physical Review B*, 83(18):180201, 2011.

[146] CQ Chen, YT Pei, and J Th M De Hosson. Effects of size on the mechanical response of metallic glasses investigated through in situ tem bending and compression experiments. *Acta Materialia*, 58(1):189–200, 2010.

[147] DZ Chen, D Jang, KM Guan, Q An, WA Goddard III, and JR Greer. Nanometallic glasses: size reduction brings ductility, surface state drives its extent. *Nano letters*, 13(9):4462–4468, 2013.

[148] Lin Tian, Zhi-Wei Shan, and Evan Ma. Ductile necking behavior of nanoscale metallic glasses under uniaxial tension at room temperature. *Acta Materialia*, 61(13):4823–4830, 2013.

[149] Matthew B Harris, Lars S Watts, and Eric R Homer. Competition between shear band nucleation and propagation across rate-dependent flow transitions in a model metallic glass. *Acta Materialia*, 111:273–282, 2016.

[150] CA Schuh and TG Nieh. A nanoindentation study of serrated flow in bulk metallic glasses. *Acta Materialia*, 51(1):87–99, 2003.

[151] Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[152] Commandant Benoıt. Note sur une méthode de résolution des équations normales provenant de lâĂŹapplication de la méthode des moindres carrésa un systeme dâĂŹéquations linéaires en nombre inférieura celui des inconnues. application de la méthodea la résolution dâĂŹun systeme défini dâĂŹéquations linéaires (procédé du commandant cholesky). *Bulletin géodésique*, 2(1):67–77, 1924.

[153] Leslie Fox, Harry D Huskey, and James Hardy Wilkinson. Notes on the solution of algebraic linear simultaneous equations. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):149–173, 1948.

[154] André-Louis Cholesky. Sur la résolution numérique des systèmes dâĂŹéquations linéaires. *Bulletin de la Sabix. Société des amis de la Bibliothèque et de l'Histoire de l'École polytechnique*, (39):81–95, 2005.

[155] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[156] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):22, 2008.

[157] The HDF Group. Hierarchical data format version 5, 2000-2017.

[158] Pieter Wesseling. Introduction to multigrid methods. Technical report, INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING HAMPTON VA, 1995.

[159] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid.* Elsevier, 2000.

[160] Are Magnus Bruaset Aslak Tveito and Are Magnus Bruaset. *Numerical solution of partial differential equations on parallel computers*. Springer, 2006.

[161] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[162] Robert W Balluffi, Sam Allen, and W Craig Carter. *Kinetics of materials.* John Wiley & Sons, 2005.

[163] David Gilbarg and Neil S Trudinger. *Elliptic partial differential equations of second order.* springer, 2015.

[164] Ronald B Guenther and John W Lee. *Partial differential equations of mathematical physics and integral equations.* Courier Corporation, 1988.

[165] N Benjamin Murphy, Elena Cherkaev, Christel Hohenegger, and Kenneth M Golden. Spectral measure computations for composite materials. *Communications in Mathematical Sciences*, 13(4):825–862, 2015.

[166] K Golden and G Papanicolaou. Bounds for effective parameters of heterogeneous media by analytic continuation. *Communications in Mathematical Physics*, 90(4):473–491, 1983.

[167] Graeme W Milton. The theory of composites (cambridge monographs on applied and computational mathematics). 2002.

[168] NB Murphy and KM Golden. The ising model and critical behavior of transport in binary composite media. *Journal of Mathematical Physics*, 53(6):063506, 2012.

[169] Johan Helsing. The effective conductivity of arrays of squares: large random unit cells and extreme contrast ratios. *Journal of Computational Physics*, 230(20):7533–7547, 2011.

[170] Johan Helsing. The effective conductivity of random checkerboards. *Journal of Computational Physics*, 230(4):1171–1181, 2011.

[171] Johan Helsing, Ross C McPhedran, and Graeme W Milton. Spectral superresolution in metamaterial composites. *New Journal of Physics*, 13(11):115005, 2011.

[172] Leslie Greengard and Monique Moura. On the numerical evaluation of electrostatic fields in composite materials. *Acta numerica*, 3:379–410, 1994.

[173] Leslie Greengard and June-Yub Lee. Electrostatics and heat conduction in high contrast composite materials. *Journal of Computational Physics*, 211(1):64–76, 2006.

[174] Gilbert Strang and George J Fix. *An analysis of the finite element method*, volume 212. Prentice-Hall Englewood Cliffs, NJ, 1973.

[175] Franco Brezzi and Michel Fortin. *Mixed and hybrid finite element methods*, volume 15. Springer Science & Business Media, 2012.

[176] Philippe G Ciarlet. *The finite element method for elliptic problems*, volume 40. Siam, 2002.

[177] Vivette Girault and Pierre-Arnaud Raviart. *Finite element methods for Navier-Stokes equations: theory and algorithms*, volume 5. Springer Science & Business Media, 2012.

[178] Solomon GrigorÊźevich Mikhlin and Trevor Boddington. *Variational methods in mathematical physics*, volume 1. Pergamon Press Oxford, 1964.

[179] Aleksandr A Samarskij and Evgenii S Nikolaev. *Numerical Methods for Grid Equations: Volume II Iterative Methods*. Birkhäuser, 2012.

[180] Alfio Quarteroni and Alberto Valli. *Numerical approximation of partial differential equations*, volume 23. Springer Science & Business Media, 2008.

[181] Wolfgang Hackbusch. *Multi-grid methods and applications*, volume 4. Springer Science & Business Media, 2013.

[182] Susanne Brenner and Ridgway Scott. *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media, 2007.

[183] Boris N Khoromskij and Gabriel Wittum. *Numerical solution of elliptic differential equations by reduction to the interface*, volume 36. Springer Science & Business Media, 2012.

[184] James H Bramble. *Multigrid methods*, volume 294. CRC Press, 1993.

[185] Barry Smith, Petter Bjorstad, and William Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge university press, 2004.

[186] Andrea Toselli and Olof Widlund. *Domain decomposition methods: algorithms and theory*, volume 34. Springer, 2005.

[187] Daniel Ben-Avraham and Shlomo Havlin. Diffusion on percolation clusters at criticality. *Journal of Physics A: Mathematical and General*, 15(12):L691, 1982.

[188] Bernard Sapoval, Michel Rosso, and Jean-François Gouyet. The fractal nature of a diffusion front and the relation to percolation. *Journal de Physique Lettres*, 46(4):149–156, 1985.

[189] Henri Poincaré. La méthode de neumann et le probleme de dirichlet. *Acta mathematica*, 20(1):59–142, 1897.

[190] VA Steklov. General methods for the solution of principal mathematical physics problems. *Kharkov Mathematical Society Publishing, Kharkov*, 1901.

[191] Emilie V Haynsworth. On the schur complement. Technical report, DTIC Document, 1968.

[192] Emilie V Haynsworth. Determination of the inertia of a partitioned hermitian matrix. *Linear algebra and its applications*, 1(1):73–81, 1968.

[193] J Schur. Über potenzreihen, die im innern des einheitskreises beschränkt sind. *Journal für die reine und angewandte Mathematik*, 147:205–232, 1917.

[194] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[195] James W Demmel. *Applied numerical linear algebra*. Siam, 1997.

[196] DS McLachlan. Measurement and analysis of a model dual conductivity medium using a generalized effective medium theory. *Physica A: Statistical Mechanics and its Applications*, 157(1):188–191, 1989.

[197] David S McLachlan, Michael Blaszkiewicz, and Robert E Newnham. Electrical resistivity of composites. *Journal of the American Ceramic Society*, 73(8):2187–2203, 1990.

[198] Victor E Brunini, Christopher A Schuh, and W Craig Carter. Percolation of diffusionally evolved two-phase systems. *Physical Review E*, 83(2):021119, 2011.