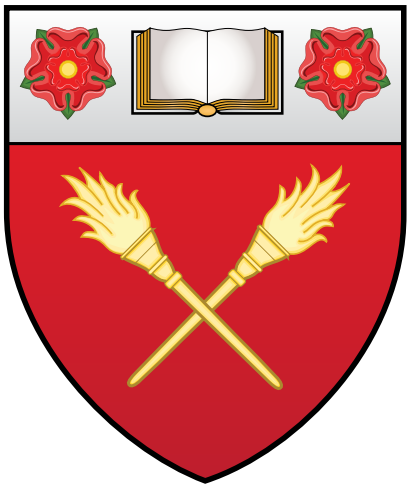
Dislocation Based Modelling of Fusion Relevant Materials



Daniel Celis Garza

University of Oxford

Harris-Manchester College

Department of Materials

Supervisors: Edmund Tarleton & Angus Wilkinson

A thesis submitted for the degree of

Doctor in Philosophy

Hilary Term 2021

Dedication

"Ohana significa familia, y tu familia nunca te abandona, ni te olvida."
— Stitch, Lilo & Stitch

Malle santa, cuando te hablé desde el trabajo para darte la noticia me dijiste que siempre lo supiste. Recuerdo que lloraste en el teléfono, se te cerró la garganta y a mí también. A diferencia de otras veces, no pudimos platicar mucho porque nos quedamos sin aliento. Así que le hablaste a abuelita Raquel mientras le hablé a papá. Madre, David y yo te debemos tanto que no podemos pagar en mil vidas, pero ten por seguro que eres nuestro ejemplo a seguir. Como nuestra madre chingona y chambeadora, solo hay una y como ella no hay ninguna. Sin ti, el mundo sería un lugar más cruel y pobre, no merece un ángel tan grande y puro como tú.

Dad, "Igualito que tu jefe, wey." con tu risa característica fue lo primero que dijiste cuando te llamé para darte la noticia "Sí dad, igualito que mi jefe." Entre risa respondí. Como siempre, no platicamos mucho, pero esa vez porque le querías decir a abuelita Teté y a mis tíos, y yo le quería avisar a David. Pensé que seguirías aquí para carcajearte al verme en la túnica ridícula, así como lo hicimos nosotros cuando te vimos en la tuya. Escribo esta dedicatoria antes de acabar porque lo prometido es deuda y esta madre la voy a acabar. No te tendremos para pedirte consejos y contarte de nuestros avances y tropiezos, pero a veces escucho tu voz cuando veo la luna y las estrellas me siguen a donde voy. Te queremos y extrañamos muchísimo. Buenas noches, dad.

David, "Te mamaste we." Me dijiste cuando me abrazaste al llegar a casa después del trabajo el día que me aceptaron. Me has hecho falta, te extraño mucho. Perdón por no hablar tan seguido, pero me duele colgar. No estoy tan chisqueado como mamá, pero siento feito cuando terminamos de platicar. Me gusta mucho ver tus streams porque me recuerda un poco a sentarme a tu lado para verte jugar, a cuando veíamos a Forsen o TB "missing legal", el hype de ver a Mang0 irse off-stage contra HBox y ver los torneos de CSGO.

Esto es para mi Ohana por sangre y por elección. Ustedes creyeron en mí cuando yo no lo hacía. Me empujaron a ser mejor. Me extendieron la mano cuando nadie más lo hizo. Me hicieron reír cuando solo sabía llorar. Gracias por hacerme quien soy.

Acknowledgements

I did not get here alone. Truly I don't have much idea what I did to deserve the help and encouragement of such an eclectic mix of incredible people. The list is long, but like the proverbial beating of a butterfly's wings, there is no telling where I'd be without the aid and support of these people.

First and foremost, my supervisors Ed Tarleton and Angus Wilkinson. Thank you for believing in me; thank you for the knowledge and advice; but most of all, thank you for your kindness when my life fell apart.

Adrian Taylor, thank you for stepping in when my own country turned its back on me when I needed them most.

Kathryn Harvey, Kate Lancaster, Roddy Vann, Howard Wilson, Ruth Lowman, Donna Cook and the rest of the Fusion CDT, thank you for such an incredible experience. I had so much fun, made a ridiculous amount of life-long friends, and carry with me so many awesome memories. We have you to thank for bringing an eclectic, yet oddly harmonious group of people that I now consider some of my closest friends.

"It's later than you think… But it's never too late" I never thought the unofficial Harris-Manchester motto would apply to my DPhil but here we are. Vicky Lill, Annette Duffel, Sue Killoran and Ralph Waller, thank you for everything. And very special thank you to the man, the legend, the gamer, and college MVP, John Carter—I cannot think of anyone who embodies the spirit of HMC more than you, mate.

My friends and colleagues of the Tarleton Research Group, Haiyang Yu, Fenxian Liu and Daniel Hortelano-Roig for all the fruitful discussions, ideas, advice and help along the way. A special thank you to Bruce Bromage—also of the Tarleton Research Group—whose DPhil ran parallel to mine; your penchant for antagonising my less-than-great ideas, as well as for code archaeology, led to a much improved EasyDD.

Damian Rouson of Sourcery Institute fame. I could not have chosen a better collaboratory. Thank you for the hospitality, the sights, and the experiences.

My former supervisors and mentors, Marcelo F. Videa, Anatoly B. Kolomeisky and John F. Stanton, thank you for giving me a chance when I was green and unproven. I hope to have made you proud.

"OUPLC" thank you for the memes, the hype, the gains, the friendships, and most of all, for showing me that perhaps the heaviest things that we lift are not our weights, but our feels.

“Kopse Lane Krump v1.0 & v2.0” thank you for being such great friends and housemates. Thank you for having me over for Christmas so I wouldn’t be alone. But most of all, thank you for being there when my life turned to dust.

“Goosehood”, si algo bueno salió del fever dream que compartimos es nuestra amistad. Hicieron que aquellos días fueran tolerables. Gracias amigoes.

“Clan del Rano Sentado”, no podría haber elegido mejores compañeros para este viaje místico repleto de manómetros rotos, desacoplados y conectados a tierra.

“La Isla”, ~~mis amigos~~ mi Ohana elegida desde mi retorno a México en el 2008. Para bien o para mal, hemos compartido una buena parte del espectro de experiencias y emociones que ofrece la vida. Witness me!

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 0

Preface

0.1 Notation and abbreviation

For the sake of clarity the following notation and abbreviation conventions have been used:

- GPU: Graphics Processing Unit.
- CPU: Central Processing Unit.
- Tensors are denoted by sans serif bold italics, $\boldsymbol{\mathsf{T}}$.
- Matrices are denoted by serif bold Roman, $\mathbf{M}$.
- Vectors are denoted by serif bold italics, $\boldsymbol{V}$.
- Scalars are denoted by serif italics, S.
- Operators and special functions are Roman, $\exp(x)$, $\det \mathbf{J}$.
- Angles are in radians unless otherwise stated.
- Einstein notation for tensor components is used unless otherwise stated.
- Individual items, be they nodes, surface elements, dislocation segments or indices are denoted by a lower-case subscript to the right, a_n.
- Ensembles are denoted by an upper-case subscript to the right, $a_N \coloneqq \sum a_n$.
- Host variables (CPU variables in parallelisation) are denoted by a Roman h-superscript on the left, ${}^{\mathrm{h}}a$.
- Global device variables (global variables visible only to the GPU) are denoted by a Roman d-superscript on the left, ${}^{\mathrm{d}}a$.

- Thread variables (variables visible only to a GPU thread) are denoted by a Roman t-superscript on the left, ${}^{\mathrm{t}}a$.
- Serial indices/counters are the traditional i, j, k, etc.
- Parallel indices are denoted as Roman variables, a.
- Pseudo-code is C-style—row-major order, 0-based indexing—because it provides a more direct translation of the C-code.

0.2 Typesetting

The repository https://github.com/dcelisgarza/latex_templates contains the custom document class used to typeset this document. It was compiled with XeLaTeX and BibTeX. We recommend compiling the source with XeLaTeX or LuaLaTeX. Compilation requires `minted`, `https://ctan.org/pkg/minted?lang=en`, and its dependencies.

0.3 Diagrams

All diagrams were drawn with Inkscape: Open Source Scalable Vector Graphics Editor, `https://inkscape.org/`, "Draw Freely."

Chapter 1

Introduction

1.1 Materials science and history

Metals and alloys are of such importance to humankind that entire eras of our history have been defined by and named after discoveries and advances in metallurgy [1, 2]. A civilisation's ability to gain mastery of the properties and usage of materials is often strongly correlated with its success in history [3]. The historical influence of metals and alloys has touched all areas of human existence, from fashion to warfare [4].

It should come as no surprise that a material's use is often linked to its properties. When a civilisation learned to harness the properties of a novel material, its influence often grew as a result [5, 6]. Whether building a woodcutter's axe or a fusion reactor, the mechanical properties of materials—metals and alloys in particular—are among the most important considerations for using a material in engineering applications.

Traditionally, such properties have been largely studied via empirical methods such as tensile and compressive tests, beam bending, and indentation [7–9]. Such tests have mainly focused on macroscopic scales, thus offering bulk-averaged properties valuable to engineers. Unfortunately, such methods only describe observed behaviours without truly elucidating the fundamental mechanisms behind them [10]. Advancing technology has allowed progressively smaller scale tests. Micro- and nano-scale testing, where tests can be performed on carefully controlled samples such as single crystals, thin films, crystal boundaries, etc. For example, it is possible to probe specific slip systems to see how they behave under various loading scenarios [11, 12]. The increased resolution and more thorough parameter control goes a long way in providing a window through which more fundamental behaviours can be observed and hopefully explained; thus providing a way of deconvolving macroscopic observations into their constituent phenomena [13, 14].

However, understanding the underlying mechanisms behind empirical results through experimental means has proven difficult at every scale [15, 16]. Fortunately modelling and simulation can be powerful allies in this task. In particular, the study of dislocations is experimentally challenging [17–19]. By virtue of being such an important player in the mechanical properties of materials, dislocations demand careful study from both empirical and in-silico avenues, where insights from one can inform the other, and form a feedback loop that continually advances our understanding.

The story of humanity is the story of our tools. Every leap in technological capability has come as a result of better understanding the materials available to us; from the hand-axes of *homo habilis* to fusion reactors, our increasingly detailed understanding of the properties of materials has enabled us to reach previously unimaginable goals. As we push the boundaries of what is possible, the need for knowledge becomes more important. Progress is made by subjecting our tools to increasingly extreme conditions and working to solve the problems that arise. Some of the most extreme conditions we have ever dreamt of achieving, are those found inside fusion reactors, where radiation damage, large temperature gradients, gas diffusion and plasma instabilities provide the means for properties to change drastically over a components' lifetime [20–22]. In-silico research is an increasingly key component of scientific research. For our purposes, it can aid in the analysis and interpretation of micromechanical tests, as well as provide new fundamental insights that cannot be obtained via traditional methods.

1.2 Fusion energy production

There currently exist two major branches of research for large scale fusion energy production [23]: *1*) magnetic confinement fusion (MCF) [24] which confines the plasma using magnetic fields and *2*) inertial confinement fusion (ICF) [25] which uses a frozen fuel pellet which is either directly or indirectly compressed by arrays of powerful lasers. The physics and engineering challenges vary greatly between both approaches but the fundamental materials problems remain largely the same—with a few exceptions such as divertors [26, 27].

1.2.1 Operating environment

A constant feature of nuclear energy production is the exposure of reactor materials to large loads of ionising and non-ionising radiation. In the case of fission, this is mostly in the form of low energy neutrons and residual radiation from fission products. Fusion on the other hand, deals with the sparsely explored 14 MeV neu-

Location	Radiation Type	MCF (ITER)	ICF (LMJ)
1st Wall	Neutron flux	$3 \times 10^{18}\,\mathrm{m^{-2}\,s^{-1}}$	$1.5 \times 10^{25}\,\mathrm{m^{-2}\,s^{-1}}$
	Neutron fluence*	$3 \times 10^{25}\,\mathrm{m^{-2}}$	$3 \times 10^{18}\,\mathrm{m^{-2}}$
	γ-ray dose rate	$2 \times 10^{3}\,\mathrm{Gy\,s^{-1}}$	$\sim 1 \times 10^{10}\,\mathrm{Gy\,s^{-1}}$
	Energetic ion/atom flux	$5 \times 10^{19}\,\mathrm{m^{-2}\,s^{1}}$	…
1st Diagnostic	Neutron flux	$1 \times 10^{17}\,\mathrm{m^{-2}\,s^{-1}}$	$1 \times 10^{26}\,\mathrm{m^{-2}\,s^{-1}}$
	Neutron damage rate	$6 \times 10^{-9}\,\mathrm{dpa\,s^{-1}}$	negligible
	Neutron fluence*	$2 \times 10^{24}\,\mathrm{m^{-2}}$	$\sim 1 \times 10^{19}\,\mathrm{m^{-2}}$
	Neutron damage	$1 \times 10^{-1}\,\mathrm{dpa}$	negligible
	γ-ray dose rate	$\sim 1 \times 10^{2}\,\mathrm{Gy\,s^{-1}}$	$\sim 1 \times 10^{10}\,\mathrm{Gy\,s^{-1}}$
	Energetic ion/atom flux	$\sim 1 \times 10^{18}\,\mathrm{m^{-2}}$	…
	Nuclear heating	$1\,\mathrm{MW\,m^{-3}}$	0
	Operating temperature	520 K	293 K
	Atomosphere	Vacuum	Air
Other	EM pulse	…	10 to $500\,\mathrm{kV\,m^{-1}}$ @ 1 GHz
	Shrapnel	…	1 to $10\,\mathrm{km\,s^{-1}}$ @ $\sim 30\,\mu\mathrm{m}$

Table 1.1: Estimated operating environment comparison between MCF (ITER) and ICF (LMJ). Reproduced from [31]. Note these numbers are unrepresentative of actual fusion power plants as both ITER and the LMJ are experiments—it is likely power plants will be subject to much harsher operating conditions.
* End of life.

tron spectrum [28]. The lack of appropriate sources of suitably energetic neutrons [29] has meant that modelling, as well as searching for experimental analogues for the true damage cascades have become a crucial part of materials research [30].

When it comes to fusion, the operating environments change vastly between ICF and MCF, as described by table 1.1 [31]. The table should be read with a healthy dose of scepticism, given that the numbers and conditions for commercial reactors, or even reactors capable of producing a net positive amount of energy, are not yet fully known because they do not exist. This is especially true for ICF, where we are extremely far from achieving the operating frequencies required for a commercial reactor. However, the table provides a fairly reasonable first approximation of the demands placed on reactor materials for both mainstream types of fusion energy production. That said, the demands on the components needed for energy harvesting—a divertor in the case of MCF, and an open problem for ICF—are conspicuous by their absence.

Table 1.1 shows that dosages and dosage rates vary wildly between approaches. For the most part, MCF receives higher doses of neutrons and ions in both the first wall and diagnostic equipment. This seems to indicate that the material requirements for MCF are stricter than for ICF. However, shrapnel production is a strong possibility in ICF, especially in indirect drive reactors; where a specially made container called a "hohlraum" holds the fuel pellet and is subsequently

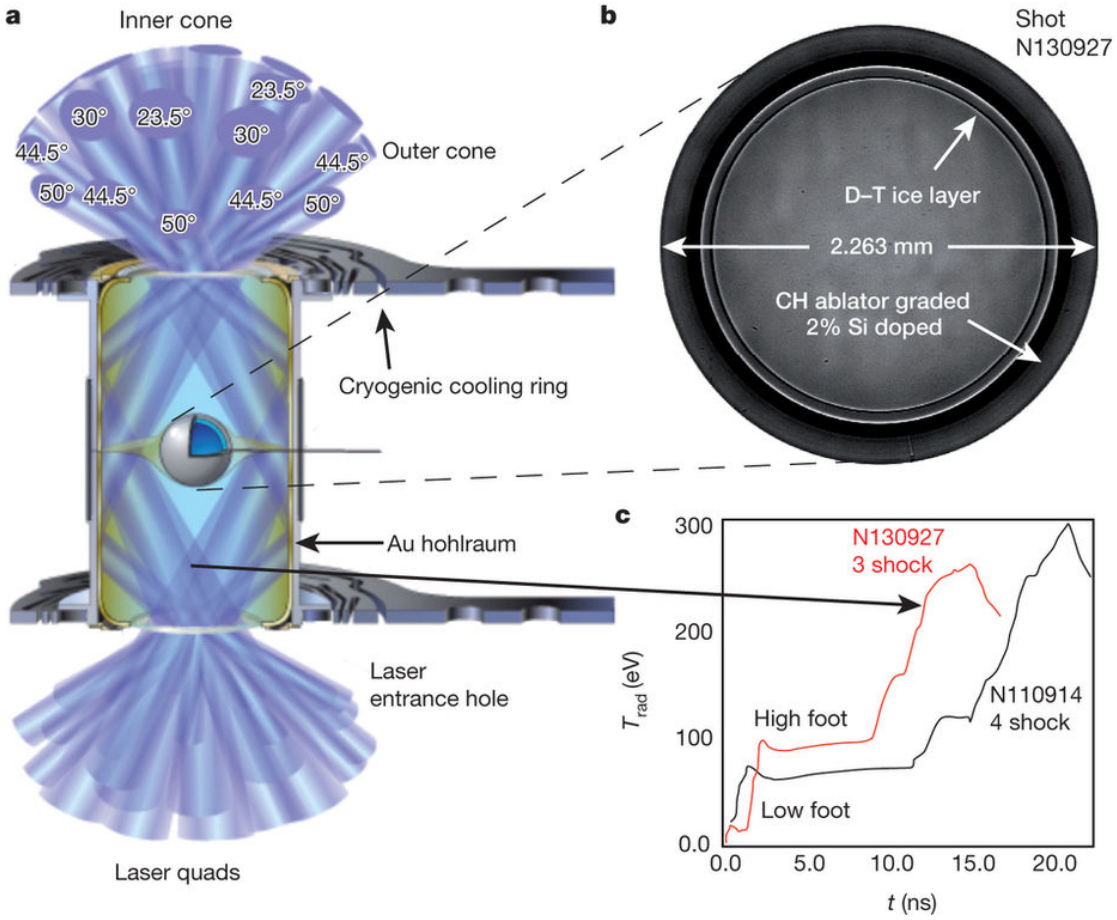


Figure 1.1: **a**, Cross section of the National Ignition Facility's ICF target design showing the gold hohlraum and plastic capsule, fuel pellet, and incident laser bundles. **b**, X-ray image of the actual capsule for N130927 with DT fuel layer and surrounding CH (carbon–hydrogen) plastic ablator. **c**, X-ray radiation drive temperature as a function of time for the National Ignition Campaign (NIC) low-foot implosion and the post-NIC high-foot implosion. Image taken from [32].

obliterated when the pellet undergoes fusion (see fig. 1.1 [32]). This is a huge challenge, as such high energy shrapnel would be capable of destroying diagnostic equipment and damaging the first wall [33–35].

Overall, the table is unrepresentative of potential operating environments in large-scale power stations, but sets lower limits on the demands of the materials involved in both mainstream proposals for fusion energy production.

1.2.2 Effects of radiation on reactor materials

1.2.2.1 Radiation Damage

One of the bigger problems in radiation damage research is the lack of a truly standardised way of measuring damage on materials [36]. The most common unit is displacements per atom (dpa) [37]. It is defined as the average number of

displacements undergone by each atom in a material as a result of being irradiated. The fundamental unit of measurement for this, is the number of displacements per unit volume per unit time, R

$$R = N \int_{E_{\min}}^{E_{\max}} \int_{T_{\min}}^{T_{\max}} \phi(E)\, \sigma(E,T)\, \upsilon(T)\, \mathrm{d}T\, \mathrm{d}E\,, \tag{1.1}$$

where N is the atom number density (no. of atoms per unit volume); E the incoming particle's energy; T the energy transferred in a collision of a particle of energy E and a lattice atom; $\phi(E)$ the energy-dependent particle flux; $\sigma(E,T)$ the cross section for the collision of a particle with energy E resulting in a transfer of energy T to the struck atom; and $\upsilon(T)$ the number of displacements per primary knock on atom as a function of transferred energy T. DPA can be calculated by naïvely multiplying R by the sample volume and total exposure time (or use fluence, $\Phi(E)$, rather than flux). This ignores the fact that $\sigma(E,T)$ and $\upsilon(T)$ will be locally perturbed in the neighbourhood of damage cascades, but since the bulk volume is much greater than that of the damage cascades', the functions are assumed to remain globally unchanged.

In principle, this is a rather good measure of damage [37]. The catch is that, generalised, analytical expressions for $\sigma(E,T)$ and $\upsilon(T)$ depend on a slew of parameters and are therefore incredibly hard, if not impossible to derive. That said, they can be discretised and roughly approximated via Monte Carlo (MC) approaches [36–38]. However, damage cascade modelling falls squarely in the realm of pico- to nanosecond timescales, and as such is only tractable with Molecular Dynamics (MD) and Kinetic Monte Carlo (KMC) [39, 40] approaches. At the end of such cascades, we are often left with dislocation sources or prismatic dislocation loops [41]. Such loops can be used as inputs for Dislocation Dynamics (DD) simulations, which can explore greater temporal and spatial scales [30].

Chapter 2

EasyDD v2.0

EasyDD can be found in https://github.com/TarletonGroup/EasyDD.

As mentioned in **??**, this project aims to integrate multidisciplinary skills for creating increasingly faithful recreations of reality in a user friendly way. This chapter details the software engineering that moved us closer to this goal and yielded a much improved version of EasyDD.

2.1 Bug fixes

2.1.1 Correct time-adaptive integrator

The first obvious hurdle to overcome when making more complex simulations possible was the tremendously long computational time taken for a dislocation plasticity simulation to sufficiently advance past the elastic regime. A simple microcantilever bending simulation with a few frank reed sources would take a whole night to reach the plastic regime, some early simulations by Bruce Bromage took days to get there.

The unacceptable slowness prompted a closer examination of the adaptive-time integration method found in algorithm 10.2 and described by equations (10.42, 10.45 and 10.46) found in [42, p. 214–216], an explicit Euler-trapezoid predictor-corrector solver. These are algorithm 2.1 and eq. (2.1),

$$\boldsymbol{r}_i^{\mathrm{P}}(t+\Delta t) = \boldsymbol{r}_i(t) + \boldsymbol{g}_i\left(\{\boldsymbol{r}_j(t)\}\right)\Delta t\,, \tag{2.1}$$

$$\boldsymbol{r}_i(t+\Delta t) = \boldsymbol{r}_i + \frac{\boldsymbol{g}_i(\{\boldsymbol{r}_j(t)\}) + \boldsymbol{g}_i\left(\left\{\boldsymbol{r}_i^{\mathrm{P}}(t+\Delta t)\right\}\right)}{2}\Delta t\,, \tag{2.2}$$

$$\boldsymbol{v}_i \coloneqq \frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} = \boldsymbol{g}_i\left(\{\boldsymbol{r}_j\}\right)\,, \tag{2.3}$$

where $\boldsymbol{r}_i$ are nodal coordinates, $\boldsymbol{v}_i$ are nodal velocities, $\boldsymbol{g}_i$ is the function that uses the mobility model and nodal forces to compute the nodal velocities, and the

superscript P denotes the predictor. This solver is fast and accurate for a small enough timestep Δt. Two free parameters $\Delta t_{\max}$ and ϵ denote the maximum allowed timestep and accuracy.

Algorithm 2.1 Adaptive Euler-trapezoid predictor-corrector algorithm.

1. Initialise time step $\Delta t \leftarrow \Delta t_{\max}$.
2. $\Delta t_0 \leftarrow \Delta t$.
3. Compute $\boldsymbol{r}_i^{\mathrm{P}}(t + \Delta t)$ and corrector $\boldsymbol{r}_i(t + \Delta t)$ from eq. (2.1).
4. If $\max_i \left(\| \boldsymbol{r}_i^{\mathrm{P}}(t + \Delta t) - \boldsymbol{r}_i(t + \Delta t) \| \right) > \epsilon$, reduce time step $\Delta t \leftarrow \Delta t/2$ and go to 3.
5. $t \leftarrow t + \Delta t$.
6. If $\Delta t = \Delta t_0$, increase time step to $\Delta t \leftarrow \min(1.2\Delta t, \Delta t_{\max})$.
7. Return to 2, unless total number of cycles is reached.

The algorithm being used was not exactly this algorithm 2.1, as it used a different way of calculating the error but other than that they were the same. Unfortunately, this only increases the time step after converging to a satisfactory answer and advancing the time. So if anything caused the timestep to decrease, such as a collision or dislocation reaction, the time step would only increase very little every subsequent step, thus eading to unecessarily small time steps in most cases. The original implementation of this is found in commit `65907b0` under the name `int_trapezoid.m`. An improved algorithm is described in algorithm 2.2. This algorithm takes close to the maximum allowed time step for the maximum allowable accuracy, ($\epsilon_{\max}$, $\upsilon_{\max}$), maximum and minimum timestep, ($\Delta t_{\max}$, $\Delta t_{\min}$) and number of iterations, $\mathrm{iter}_{\max}$. It also uses a better error heuristic and more conservative timestep increase. It can be found in the most current version of `int_trapezoid.m`.

Algorithm 2.2 is more computationally expensive per iteration, but can increase the timestep much more rapidly than algorithm 2.1 without going through the rest of the simulation, which involves computationally expensive procedures. This improvement by itself let the early version of the software move through the elastic regime of our simulations orders of magnitude times faster than before. Simulations that took hours to days to reach the plastic regime, started doing so in minutes to hours. It also narrowed the gap between using different loading conditions, as the simulations could now effectively adjust the timestep to match the error tolerances much more easily than before. Both of which were a great

Algorithm 2.2 Improved adaptive timestep algorithm.

Convergent $\leftarrow$ false, $\Delta t_{\text{valid}} \leftarrow 0$, iter $\leftarrow 0$, flag $\leftarrow$ false
while Convergent **do**
  Compute $\boldsymbol{r}_i^{\text{P}}(t+\Delta t)$ and corrector $\boldsymbol{r}_i(t+\Delta t)$ from eq. (2.1).
  $\Delta \boldsymbol{r}_i \leftarrow \boldsymbol{r}_i(t+\Delta t) - \boldsymbol{r}_i^{\text{P}}(t+\Delta t)$
  $\overline{\boldsymbol{r}}_i \leftarrow \dfrac{\boldsymbol{g}_i(\{\boldsymbol{r}_j(t)\}) + \boldsymbol{g}_i\left(\left\{\boldsymbol{r}_i^{\text{P}}(t+\Delta t)\right\}\right)}{2}\Delta t$
  $\epsilon \leftarrow \max_i\left(\|\Delta \boldsymbol{r}_i\|\right)$
  $\upsilon \leftarrow \max_i\left(\|\Delta \boldsymbol{r}_i - \overline{\boldsymbol{r}}_i\|\right)$
  if $\epsilon < \epsilon_{\max}$ and $\upsilon < \upsilon_{\max}$ **then**
    $\Delta t_{\text{valid}} = \Delta t$
    $\gamma \leftarrow 1.2\left(\dfrac{1}{[1+(1.2^{20}-1)(\epsilon/\epsilon_{\max})]}\right)^{1/20}$
    flag $\leftarrow$ true
    iter $\leftarrow$ iter $+ 1$
    $\Delta t \leftarrow \max\left(\gamma \Delta t,\, \Delta t_{\max}\right)$
  else
    if flag == true **then**
      $\Delta t \leftarrow \Delta t_{\text{valid}}$
      iter $\leftarrow$ iter$_{\max}$
    else
      $\Delta t \leftarrow \Delta t/2$
    end if
  end if
  if iter $>$ iter$_{\max}$ or $\Delta t == \Delta t_{\max}$ or $\Delta t < \Delta t_{\min}$ **then**
    Convergent $\leftarrow$ true
  end if
end while
$t \leftarrow t + \Delta t$
Proceed with the rest of the simulation.

boon to the usability of the code and to the research group's output capacity. This was the first major bottleneck to be fixed, which allowed us to identify and resolve previously unknown issues downstream.

2.1.2 Matrix conditioning

A common problem with dislocation dynamics is the tendancy for some networks to have a large degree of variation in the speed of dislocation nodes [42–44]. This is a consequence of the fact that in discrete dislocation dynamics, nodes are simply the discretised representation of a dislocation line. As such, nodal motion must be constrained to the motion of dislocations, i.e. nodal movement must be consistent with how a dislocation glides, climbs and cross lips accounting for the dislocation character and orientation in the crystal. Moreover, dislocation motion can be assumed to follow a linear, anisotropic viscous drag model [42],

$$\boldsymbol{F}_{\text{drag}}(\boldsymbol{x}) = -\mathcal{B}(\boldsymbol{\xi}(\boldsymbol{x})) \cdot \boldsymbol{v}(\boldsymbol{x}) \,, \tag{2.4}$$

where $\mathcal{B}$ is a tensor constructed from the anisotropic drag that a dislocation node, $\boldsymbol{x}$, experiences as a result of its constrained velocity $\boldsymbol{v}(\boldsymbol{x})$ due to being part of a discritesed segment $\boldsymbol{\xi}(\boldsymbol{x})$. In simple terms, this means a node experiences very different resitances to moving along its allowed directions.

Different mobility laws have different parametrisations for these directions. A problem that often plagues traditional mobility laws[1] is the movement of a node along a line segment itself. Having a node move along a single line segment has no effect on the energy of a network, as seen in section 2.1.2. Both structures are equivalent because the line—and thus the dislocation—does not change from one to the other, merely the limits of the line integral change. Therefore nodal movement along a line should not contribute to the total drag, and therefore have a zero line drag coefficient. On the other end of the spectrum we have climb, which is orders of magnitude less favourable than glide. But this can make eq. (2.7) singular, so its value is the careful balancing act of keeping the matrix invertible whilst keeping the associated energy cost of moving along a line low. This also causes nodal speeds along line directions to be much higher than other directions, which often limits the step size an integrator can take.

In the overdamped regime, $\dfrac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t}(\boldsymbol{x}) \to \boldsymbol{0}$, known driving forces (Peach-Köhler

[1] Bruce Bromage, of [45] has developed a new BCC mobility law, `mobbcc1_bb.m` which can be found in the most current version of EasyDD. It solves many of the issues with traditional laws and is our new go-to for BCC materials. Among the things it fixes are: unreasonable cross-slip, pencil glide, and the problematic line-direction parametrisation. The drag matrix, $\mathbf{B}$, can still be singular however, so this law also includes the scale-averaged Marquardt-Levenberg regularisation.

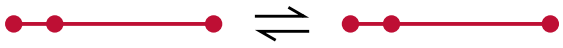


Figure 2.1: Node movement along dislocation line should have no drag contribution.

force + segment-segment forces + self-force) and the unkown drag force sum to give,

$$\boldsymbol{F}_{\text{drag}}(\boldsymbol{x}) + \boldsymbol{F}_{\text{drive}}(\boldsymbol{x}) = \boldsymbol{0}\,. \tag{2.5}$$

Substituting eq. (2.4) into eq. (2.5) and rearranging gives,

$$\mathcal{B}(\boldsymbol{\xi}(\boldsymbol{x})) \cdot \boldsymbol{v}(\boldsymbol{x}) = \boldsymbol{F}_{\text{drive}}(\boldsymbol{x})\,. \tag{2.6}$$

Strictly speaking, solving eq. (2.4) requires global knowledge of the network. The discretisation makes it possible to distribute $\mathcal{B}$ along a line segment $i - j$,

$$\mathbf{B}_{ij} \leftarrow \oint_C N_i(\boldsymbol{x})\mathcal{B}(\boldsymbol{\xi}(\boldsymbol{x}))N_j(\boldsymbol{x})\mathrm{d}L\,, \tag{2.7}$$

where C is the whole network, N_i and N_j are simply shape functions that interpolate quantities given the relative position of a node to a segment, and $\mathrm{d}L$ is the infinitesimal line segment. Thus giving individual expressions for node i connected to node j. Using this discretisation and assuming the nodes connected to i are subject to similar conditions, eq. (2.6) can be broken up into local segments,

$$\boldsymbol{v}_i \approx \left(\sum_j \mathbf{B}_{ij}\right)^{-1} \cdot \boldsymbol{f}_i, \tag{2.8}$$

where i is the node in question, j are the nodes it shares a segment with, and $\boldsymbol{f}_i$ is the local force on the node. From now on, we will drop the Einstein notation and refer to the local frame as $\boldsymbol{v} = \mathbf{B}^{-1}\boldsymbol{f}$.

The immediate implication of the orders of magnitude difference between the drag coefficients used to construct $\mathcal{B}$—and therefore $\mathbf{B}$—means the condition number of $\sum_j \mathbf{B}$ can be very large. As $\mathbf{B}$ is symmetric and therefore normal, its condition number is given by,

$$\kappa(\mathbf{B}) = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}\,, \tag{2.9}$$

where λ_i is the i^{th} eigenvalue.

The condition number of a matrix dictates how much collinearity exists in its basis. The larger the condition number, the larger the collinearity and the

more sensitive the sytem is to small perturbations. Such systems are said to be ill-conditioned. As a general rule, a condition number $\kappa(\mathbf{A}) = 10^k$ represents a loss of up to k digits of precision on top of what is already lost from arithmetic methods under limited precision [46].

True to Murphy's law, $\mathbf{B}$ is frequently ill-conditioned—even in the simplest simulations. The way these cases were handled relied on the integrator reducing the timestep enough to sidestep the issue. We call this "strongly coupled" behaviour. Strong coupling between functionally independent functions is a very bad practice in software engineering.

A function $\mathcal{X}$ with a set of $\mathbb{X}$ bugs with phase space $\chi(x)$ that is, $\forall x \in \mathbb{X}\ \chi(x)$. Bugs in production code are often either low impact with broad phase spaces, or high impact with narrow phase spaces. Both of which can be avoided with a combination of proper software design and testing protocols. All this goes out the window when functions are not properly decoupled. The multiplicative product of the corresponding phase spaces is often non-trivial, and can sometimes cancel out in some cases and blow up in others. Complicating the diagnosis and solution of errors.

The way the mobility functions dealt with ill-conditioned $\mathbf{B}$, relied on the integrator picking up the slack. There are several flaws with the approach found in algorithm 2.3, the actual mobility function can also be found under commit `65907b0` under the name `mobbcc1.m`. Every mobility law in that commit has this issue.

Algorithm 2.3 Avoiding singular matrix by making $\mathbf{B}$ extremely wrong and hoping the integrator error bounds pick it up and the timestep is decreased.

Compute eigenvalue matrix, $\mathbf{D} \leftarrow \mathbf{P}^{-1}\mathbf{B}\mathbf{P}$
$\kappa(\mathbf{B}) \leftarrow |\lambda_{\max}|/|\lambda_{\min}|$
if $\kappa(\mathbf{B}) < \kappa_{\min}$ **then**
    $\mathbf{D}_{\text{norm}} \leftarrow \mathbf{D}/\lambda_{\max}$
    $\boldsymbol{f} \leftarrow \boldsymbol{f}/\lambda_{\max}$
                                ▷ Invert $\mathbf{D}_{\text{norm}}$.
    for all $(\lambda_k \in \mathbf{D}_{\text{norm}}) \neq \lambda_{\max}$ **do**
        if $\lambda_k > \lambda_{\text{crit}}$ **then**
            $\lambda_k \leftarrow 1/\lambda_k$
        else
                                ▷ The inverse of 0 is not 0.
            $\lambda_k \leftarrow 0$
        end if
    end for
    $\boldsymbol{v} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}\boldsymbol{f}$
else
    $\boldsymbol{v} = \mathbf{B}^{-1}\boldsymbol{f}$
end if

The first is assuming that an ill-conditioned matrix can be fixed by diagonalising. This is simply not true, perturbing the smallest eigenvalue in a direction such that κ improves is a good way of doing so without adding much error. However, perturbations of this type are often not enough when the condition number is too large. Furthermore, one wants to keep the overall behaviour of the system, i.e. maintain the relative sizes of the eigenvalues with respect to one another. If the multiple eigenvalues are close in magnitude to the smallest eigenvalue, one can change the dynamics of the system if the perturbation is too large. Not to mention the fact that κ may still be too large. In a system as stiff and dynamic as discrete dislocation dynamics, this approach is not enough for some cases.

The second flaw is the assumption that the case where $\lambda_k < \lambda_{\text{crit}}$ for non-maximal eigenvalues, is so rare, it won't have much of an effect. Unfortunately, it's common in junctions where nodal mobility is limited, as well as in volumes with large stress gradients. Although to be fair, these are also consequence of the assumption of locality, yielding section 3.2. Moreover, when every non-maximal eigenvalue is smaller than λ_{crit}, the error much greater. For a every eigenvalue under the threshold of of $\lambda_{\text{crit}} = 10^{-k}$, the upper bound of the error is k digits in the corresponding mobility component.

The third is the assumption that the integrator will not allow obviously wrong solutions because it will fail the error check and reduce the time step. This assumption breaks down whenever dislocation velocities are calculated outside the context of time integration, such as every topological operation where a new node is spawned.

The regularisation of ill-conditioned systems in an active field of research [47–49]. However, they require some knowledge of the matrix structure or follow a well-known statistical distribution to thier values/eigenvalues. Regularlisation is most often applied to high rank matrices because poorly conditioned low rank matrices ones are often very sensitive to perturbations, thus requiring specialised methods. In this case, $\mathbf{B}$ is of rank equal to the number of orthogonal crystallographic directions, so it is always of low rank.

One of the most effective regularisations for low rank matrices is dampening the inversion by adding a small value, c—called the Marquardt-Levenberg coefficient—to the diagonal of the matrix. This is roughly equivalent to perturbing the smallest eigenvalue as previously mentioned. Luckily, if this factor is added the whole diagonal, the inversion can be dampened sufficiently whilst keeping the main contributions from each basis roughly equal in relation to each other. This amounts to adding a multiple of the identity matrix to an ill-conditioned matrix. For a more accurate solution, one can iteratively refine c to find the smallest value that keeps the matrix inversion from blowing up. A good heuristic for the value

of the Marquardt-Levenberg coefficient is,

$$c = \max\left(|\mathbf{A}|\right)\sqrt(\epsilon)\,, \tag{2.10}$$

where ϵ is machine precision. This ensures the perturbation is scaled to a number that will not underflow during arithmetic operations. Simply put, it ensures the perturbation is small but does not disappear during matrix inversion.

It is worth noting that doing this and inverting is still a source of error, adding noise even if small and along the diagonal, does move the system away from its true behaviour. However, the degree to which it is wrong is many orders of magnitude smaller than what is done in algorithm 2.3. Indeed, algorithm 2.4 uses $\pm c$ and both perturbed solutions are averaged to give a much more accurate and simpler solution even under the traditionally problematic scenarios discussed above. This new algorithm is now present in the mobility laws we now use, such as the most current version of `mobbcc_bb.m`.

Algorithm 2.4 Improved regularisation of $\mathbf{B}$ by way of perturbing the diagonal.

if $\|\mathbf{B}\| < \epsilon$ **then**
  $\boldsymbol{v} \leftarrow \mathbf{0}$
else if $|\lambda_{\max}|/|\lambda_{\min}| < \epsilon$ **then**
  $\mathbf{B}_+ \leftarrow \mathbf{B} + \sqrt{\epsilon}\max(\mathbf{B})$
  $\mathbf{B}_- \leftarrow \mathbf{B} - \sqrt{\epsilon}\max(\mathbf{B})$
  $\boldsymbol{v}_+ \leftarrow \mathbf{B}_+^{-1}\boldsymbol{f}$
  $\boldsymbol{v}_- \leftarrow \mathbf{B}_-^{-1}\boldsymbol{f}$
  $\boldsymbol{v} \leftarrow (\boldsymbol{v}_- + \boldsymbol{v}_+)/2$
else
  $\boldsymbol{v} \leftarrow \mathbf{B}^{-1}\boldsymbol{f}$
end if

This change enabled other team members, namely Haiyang Yu and Fengxian Liu to get further with their simulations. Haiyang's nanoindentation simulations have particularly complex dislocation structures with a substantial amount of junction formation; while Fengxian's have large localised stress gradients as a result of inclusions. Both require nodal velocieties to be accurately calculated, else the dislocations move erratically, readily cross slip, and cause massive slowdowns as simulations advance.

2.2 Research software engineering

Many scientists and researchers spare no thought for good software. However, with the presently on-going SARS-CoV 2 pandemic, the importance of research software has been made clear. Modelling has played a large role in informing government

policy in the UK and the world [50–52]. With modelling thrust into the spotlight and the public release of the Imperial College modelling code used to inform so much policy [52], `https://github.com/mrc-ide/covid-sim`, criticism was levied at the lack of care taken to ensure the correctness of the code [53]. In experimental research, a lot of care is take to ensure a methodoogy used is reproducible, robust and sound. There are standards, both external and internal, which ensure the quality of the results. But even with these measures in place, there is already a common issue with irreproducibility in science [54–56]. In obtaining results and analysing them, software is ubiquitous. It should be good, especially if the science is modelling-based.

Along this vein, one of the primary goals of this project was to develop a competent codebase that non-experts can take advantage of with minimal preparation. There were multiple sub-objectives that help us achieve this.

1. Provide a generic framework upon which the software can be modularly expanded with minimal change to the source: prevents increasingly divergent and incompatible source code between researchers.

2. Provide an easy way to autocomplete inputs with sensible default parameters: prevents the software from crashing when—inevitably—one of the required arguments for a function is undefined.

3. Improve readability and organisation: makes it easy to identify, fix bugs and know what the code does.

4. Compartmentalise variables: makes the code more generic and increases usability and readability by reducing the number of function arguments.

5. Optimise memory and computation: improves computational speed and reduces resource use.

Akin to the Ship of Theseus, it is hard to tell whether the new version of the code is the same code as when this project began. While `EasyDD v2.0` is "complete", the process of improving the code is on-going. The amount of initial technical debt since its inception as the infinite-domain discrete dislocation dynamics code, `DDLab` [42], plus what it accrued as the Tarleton Group expanded its functionality and turned it into `EasyDD`, is quite large and needs to be carefully chipped away so as not to introduce regressions. However, while the quality and capabilities of the code can be greatly improved, there is only so much that can be done without fundamentally redesigning it completely. This is in part the subject of chapter 6.

Other team members have made significant contributions to the release of `EasyDD v2.0`. Some of which remain to be added, but should slowly make their way in. Given the code is the amalgam of the research group's work, we will credit other members' contributions where relevant but will leave the details for them to explain.

This is actively evolving on research software. Therefore the work described herein will continue past the end of this project, for there are still many improvements to be made to the old codebase on top of whatever functionality is added by future researchers.

2.2.1 Organisation

An oft-overlooked aspect of code useability is the organisational aspect. However as a codebase grows, it becomes more important for both a code be properly organised and source controlled. This makes it easier to browse, understand, is less overwhelming for users and developers and crucially provides a safey blanket against things going wrong.

2.2.1.1 Source control

Source control is a crucial aspect of industrial software development. It is an industry standard and the first thing to set up when starting a software development project. Academia is very far behind in this regard. Not only does it provide a history of restore points in case things break; but keeps a history of changes, who and when they made them; can perform automated checks such as testing or conflict detection; lets developers and users (if the repository is open-sourced) make changes independently and request their changes be added to the main code; among many other tasks.

In regards to reproducibility, source control is a crucial tool. In both sections 2.1.1 and 2.1.2 we took advantage of this to provide a specific commit and a file name, which lets anyone with access to a repository, the ability to see the entire history of the code. In the digital version of this document, those link commits are links directly to the file—and where relevant—specific lines. The advantages of such a system are obvious. For example, when submitting a manuscript for publication one can also submit a link to the specific version of the software used to obtain the results. This is a gigantic stride towards open, reproducible science.

2.2.1.2 Folder structure

An on-going task is improving the folder structure. Which in the past was almost non-existant. Having a default place to place inputs, outputs and source code is

of great import if a code is to be used by non-experts. It is trivial to do and a boon to the quality of life of developers and users.

2.2.2 Modularisation

Modern software practices usually make heavy use of some form of variable encapsulation and code modulirisation to make code safer, simpler to expand and easier to use. This enables users and developers to take advantage of a set of toolboxes that can be chosen according to their needs without having to directly modify the source code, this concept is called "moduliration".

2.2.2.1 Encapsulation

The act of keeping variables neatly stored in a way that fits a purpose is called encapsulation. By encapsulating variableswe greatly improve the usability of software by increasing readability, reducingthe risk of human error by reducing the number of things to keep track of, and if done properly can even improve performance. This concept is tightly bound to the design philosophy of software as described in the following list.

- Object-Oriented Programming (OOP): where variables are viewed as objects upon which the logic acts. Functions can take these objects and operate on them. This design pattern keeps related variables as part of a single entity and reduces the number of things users and developers have to keep track of, thus reducing the chance of implementation and user error. There are many advanced concepts and pitfalls in OOP that unecessarily increase complexity, so it is best to apply the KISS (keep it simple, stupid) approach for best results.

- Data-Oriven Programming (DDP): where the data dictates the structure of the programme. It is essentially a modified version of OOP where the object is made to fit the data, not the data made to fit the object. It often leads to more performant code and avoids many of the issues that arise from OOP because it forces developers to really think about how to manage the data.

- Functional programming: where everything is a function. There is no state, but also no side effects. This is the strongest form of encapsulation but also the most inflexible.

- Procedural programming: this is the most ancient form of programming where there are only functions and variables. It is the form most scientific software packages take and it is the most difficult to work with both, as

a developer, user. Even modern computers which can make use of branch prediction and deletion, cache pre-fetching, and compiler optimisations can be hindered by this approach.

Procedural programming is what gives us so-called spagghetti code[2]. Unfortunately, education in programming or software engineering in science is sorely lacking. As a result, a large amount of scientific software tends to be written in this way (particularly legacy software). Thus we strive to separate ourselves from it as much as possible.

Functional programming is impractical for most use cases so it is seldom used outside of computer science and very specific industrial and academic applications [57, 58].

Object-oriented programming can be very useful if used properly. But it has the potential to unecessarily increase complexity and reduce performance. Fortunately, `MATLAB` does not truly have object-oriented capabilities, so it forces either a procedural or data-driven approach. Unfortunately once more, it leads to software being written in a procedural manner.

This can be remedied by using `struct` and being sensible about, i.e. by taking a data-driven approach, in building them. While they are more of an ordered dictionary (Hash table), than a true object, they are sufficient for our use-case. They have allowed us to make use of generic functions for different mobility functions, loading conditions, finite element meshes, the dislocation network itself, etc.All of which have a massively simplifying effect on the code without sacrifing performance. While this project pioneered this task, Daniel Hortelano-Roig has been doing much in the way of increasing its scope. His changes greatly simplify the code and will slowly be ported over to the main branch.

2.2.2.2 Generic functions

Generic functions are implementation-agnostic functions that can be given by the user as any other variable. Only the variable in this case is a function. `MATLAB` has an old method of doing so using `feval()`, where the first argument is the name of the file whose function is being called, and subsequent arguments are passed on to the function. However this has some limitations, mainly their performance, and function handles are now preferred, e.g.

```
% myFunction is defined in 'myFunction.m' and is called
% like so myFunction(foo, bar)
```

[2]Code with many branches and disparate procedures without a clear purpose. It is difficult to work with as a user and developer. It also tends to interfere with branch prediction and deletion as well as compiler optimisations.

```
    genericFunction = 'myFunction.m';
    foo = 1;
    bar = 2;
    feval('genericFunction', foo, bar);
```

as opposed to,

```
    % myFunction is defined in 'myFunction.m' and is called
    % like so myFunction(foo, bar)
    genericFunction = @myFunction;
    foo = 1;
    bar = 2;
    genericFunction(foo, bar);
```

which are not dissimilar, but the latter is more performant and easier to follow. Generic functions were already in use for dislocation mobility but there are now ones for calculating numeric and analytic tractions (see chapter 4), loading conditions (multi-stage loading, constant loading, cyclic loading), various post-processing functions, boundary conditions and various miscelaneous functions that support different simulation types.

The main advantage of generic functions is reducing the need for direct source code modification. Changing source code, especially when there are no automated tests to speak of, is a dangerous proposition. It greatly increases the probability of introducing regressions (new bugs) and code divergeance between researchers. Regressions are always problematic, but code divergeance is a big issue that still affects the Tarleton Group. If experts within the same research group find it difficult to incorporate each others' additions to our work, there is little hope for the general user.

The introduction of regressions makes it so code must be thoroughly and exhaustively tested before merging two people's work. The use of generic functions makes it so if there is a problem with a new addition, the problem is localised and can be quickly identified and fixed because it is isolated. We can be sure the changes are only found in the new piece of code. And since it is generic, we can chose whether or not to use it by changing an input file rather than messing with the source code.

If on the other hand, one were to add a new branch to an `if` statement (or a brand new `if` statement) with new computations, likely the function needs new arguments to cover this new case, we may or may not need to make up- or downstream modifications so the function can accommodate the additions. Then very

quickly we will find ourselves procedurally programming and fixing the innevitable problems that come with hard to understand spaghetti code—which is most likely also less performant now than it used to be. The problem worsens as we add more functionality. Which can all be avoided by intelligently designing a generic approach, that can be easily expanded if the need arises.

2.2.3 Optimisation

> We should forget about small efficiencies, say about 97% of the time: pre-mature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
>
> — *Donald E. Knuth [59, p. 268]*

2.2.3.1 Spurious memory allocation

`MATLAB` is infamous for the way it will happily dynamically grow an array when going out of bounds. This terrible practice is even fundmental to the operation of `EasyDD`. However, changing this fundamental part of the codebase requires a complete rewrite and not deemed worthwhile, there are worse bottlenecks. However, there were many instances of memory reallocated or arrays grown every cycle that were completely unecessary.

For example, the amount of memory unecessarily reallocated in the function that couples DDD and FEM was $12(N_x + 1)(N_y + 1)(N_z + 1)$ double precision floating point numbers, where N_i is the number of elements in dimension i of the finite domain. The cubic scaling is a performance killer at larger mesh sizes. In a simulation with hundreds or thousands of steps and a mesh with a few thousand FE nodes, the cost added up.

It is not only the cost of allocating, but also the cost of garbage collection [60] i.e. automatically freeing memory based on a set of criteria can be significantly higher. Timing the equivalent of allocating a $20 \times 20 \times 20$ mesh in `MATLAB` 2020b takes on the order of 70 µsec (for x86 CPU architectures), which is in-line to what it costs in `C`. Deallocating memory takes about 5× longer. In a function that otherwise takes a few m sec, adding another 400 µsec of spurious allocation is a significant overhead that can be easily remedied.

2.2.3.2 Finite element optimisation

The use of sparse arrays is a must when working with finite elements, where matrices often take structured sparse forms will significantly reduce memory footprint

and computational expense. The code already used these special data structures and Cholesky factorisation for faster solution of linear systems. However, `MATLAB` has a special form of it that exploits sparsity. This reduced the time required for the factorisation by around 50×, reduced the memory footprint of the final sparse arrays by a similar amount, and reduced the memory footprint of the actual factorisation procedure by approximately 10×. The whole point of this factorisation is to make solving linear systems faster, in this case the calculation of the corrective displacements, $\hat{\boldsymbol{u}}$ on the free boundaries, S_U of freedom as a result of the forces acting upon them,

$$\hat{\boldsymbol{u}} = \mathbf{K}^{-1}\boldsymbol{f} \quad \text{on } S_U \tag{2.11}$$

It improved solution speed by a factor of 3, thus reducing the cost of coupling DDD to FEM to be dominated by the computation of dislocation-induced tractions and displacements.

The final two optimisations also deal with memory. The first, removes unecessary allocations in the calculation of the global stiffness matrix and the introduction of a reduced stiffness matrix such that it only includes relevant degrees of freedom.

In aggregate these changes have given us the ability to use 15× more nodes as well as letting us build meshes over two orders of magnitude faster, mostly down to improved memory access patterns inside the mesh building loops.

2.2.4 Misc quality of life improvements

Formerly, initialising a simulation would involve a series of non-obvious steps, particularly when initially compiling the C and CUDA code used for accelerating simulations. This is now done automatically on a need-to-compile basis and has a fallback in case no `CUDA` compiler is found.

There was also a recurring problem with simulations suddenly failing when a function was called with an undefined argument. This extremely common scenario is unbecoming of good software, especially if the failure happened a few minutes after starting a simulation. This has been remedied with the automatic calculation of a set of reasonable defaults for each and every undefined variable, aside from the arrays defining a dislocation network. Any new developments get added to this very simple, yet much needed script.

2.3 Conclusions

Scientific modelling is inherently an interdisciplinary skill. EasyDD is now faster, more efficient and more capable than ever before, in large part thanks to the work—ongoing or otherwise—described within this chapter. Subsequent chapters explore aspects more specific to discrete dislocation dynamics. However, it is important not to overlook the work that enables the obtention of results. Modern academia deems the final result to be paramount. But what is a result if it cannot be scrutinised because nobody can understand how it came about? Moreover, by developing good software, we allow others to partake in the obtention of results. We deem this to be one of the main contributions of the present project. We feel safe our ability to claim that we now have the capacity for providing more faithful recreations of reality in a less user-unfriendly way. And without so much as a glancing touch on dislocation dynamics.

Chapter 3

Improved tolopogical operations

In 2D, dislocations are parametrised as points on a surface. Therefore aside from annihilation, interactions with grain boundaries [61, 62], and perhaps the creation of superdislocations, topological changes in the dislocation network can be mostly ignored. However, in 3D, dislocations are parametrised as lines in a volume. This gives rise to extra complexities that result from multiple dislocations coming into contact with one another. These reactions ultimately lead to the formation of many secondary structures with their own properties that affect the rest of the network.

The local dislocation density at these interaction sites, particularly arround sessile (immobile) junctions, tends to increase with time. Therefore so do the local stress gradients, and with increasing stress gradients come increasing velocity gradients, which lead to global decreases in timestep. But a smaller timestep is not the only consequence, increases in dislocation density also mean higher probabilities of dislocation-dislocation interactions, so the phenomenon is autocatalytic and self-perpetuating. So properly accounting for these changes is of the utmost importance, particularly for simulations with high dislocation densities such as nanoindentation. This chapter details these improvements.

3.1 Collision

3.1.1 Hinges

The lowest hanging fruit and most obvious problem were collisions that share a lot of similarities with the remeshing criteria that checks for a minimum area enclosed between two segments connected by a single node, as shown in fig. 3.1. A single dislocation hinge can lead to three different final topologies, the conditions for which can be simultaneously met in any combination

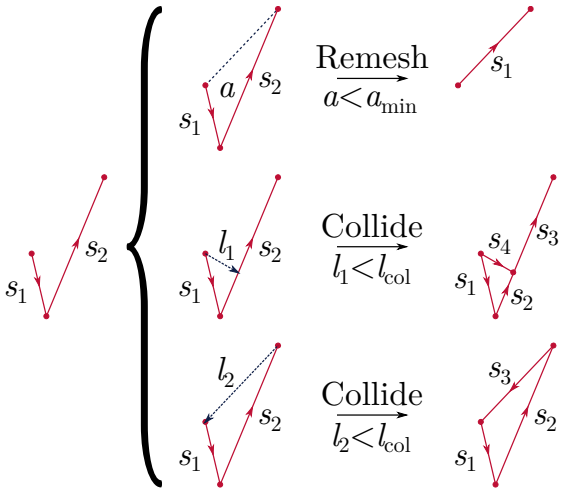


Figure 3.1: A single dislocation hinge can lead to three different final topologies, the conditions for which can be simultaneously met in any combination.

1. Remeshing to eliminate the middle node because the area, a, enclosed by the triangle created by segments s_1 and s_2 is less than the minimum allowable area, $a_{\min}$.

2. Colliding s_1 into s_2 if the minimum distance between the non-hinge node of s_1 (the node s_1 does not share with s_2) and segment s_2 is less than the collision distance, l_{col}.

3. Colliding s_2 into s_1 if the minimum distance between the non-hinge node of s_2 and segment s_1 is less than the collision distance, l_{col}.

The combination in which these conditions can be met, and the order in which they are checked will have drastic downstream consequences (see section 3.2). Especially at high dislocation densities where multiple collisions happen at every timestep. This had not been an issue in prior to fixing the integrator and matrix inversion, but simulations could now advance to a point where this was having a significant impact, particularly in Haiyang Yu's nanoindentation simulations and the microtensile simulations in section 5.1.

This was a multi-stage fix, where each fix progressively uncovered more and more shortcomings resulting from the unavoidable coupling between topological operations. These progressive fixes are detailed in section 3.2. Here we will only focus on problems with collisions.

The first fix involved prioritising hinge collisions above others, which we call two-line collisions. This meant detecting and forming the new connection. The rationale for this is that a segments in a hinge are close enough to and connected with each other at a node already that they will interact more strongly than two unconnected segments approaching one another.

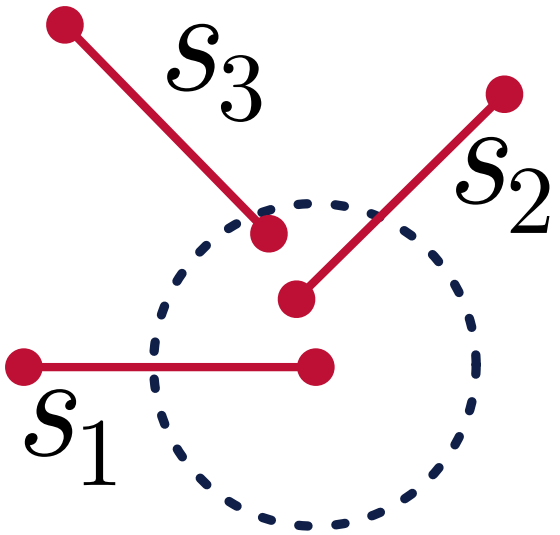


Figure 3.2: Multiple dislocation segments inside the same collision radius.

The second was much more subtle. Figure 3.1 has two options for colliding a hinge. And they both result in differently sized segments $l_1 < l_2$. The one the code calculated depended on which segment, s_1 or s_2 it came across. If s_1 was first, then the answer would be l_1, otherwise it would be l_2. At the very least, the answer should be the same for a given network regardless of how it is represented. So the first step is to ensure one of these is always chosen, but once this is done it is a simple case of calculating the distance from the short segment to the long one.

Remeshing used to be called after finding a single collision. This worked well when the number of collisions at a given time step was small. The problems with this are detailed in section 3.2. But before moving on to that, it became apparent that there was another issue with the collision detection.

3.1.2 Collision distance

In the same vein as section 3.1.1, two-line collisions were being improperly prioritised. They occurred arbitrarily as the code came across them. Again yielding different answers for different representations of the same network. Only this time it had to do with the collision radius as seen in fig. 3.2.

The actual algorithm is more sophisticated because it accounts for both complete segments and finds the two points along them with the minimum distance between each other. But the problem was that as soon as a viable collision was found, the function returned. So in a scenario like fig. 3.2, if the code were looking for a collision on s_1 and it came across s_3 first it would flag that as the collision disregarding the fact that not only was s_2 closer, but s_2 and s_3 are also closer. The same fix had to be applied to hinge collisions.

Despite the fact that checking all possible collisions for the minimum distance is a more expensive task, it results in faster simulations because the resulting structures are guaranteed to either produce the shortest segments, which can be cleaned up by remeshing, or the highest energy structures, which can be cleaned up by separation. Otherwise the structures often end up in a no-man's land which is energetically unfavourable to separate, or cannot be remeshed; leading to an accumulation of highly immobile, highly energetic structures that cause more and more collisions as simulations advance and create sharp stress gradients that cause the timestep to drop significantly, preventing simulations from advancing very much after hitting a critical point.

3.2 Non-commutativity of topological changes

For a given iteration the time-evolution of the network used to be performed before any topological changes. This was very sub-optimal for a couple of reasons.

The fundamental reason is that by doing so, the time evolution erroneously accounts for complex structures that should have dissipated into lower energy ones as per the quasi-static principle our model is based on. In other words, high energy structures in the network mean large stress gradients. These break the assumption that a node and its neighbours experience similar conditions. These locality assumptions lead to , which is what all our mobility laws are based on.

The practical reason is that by leaving these structures around when they should have been removed, the computational complexity increases by virtue of having more nodes and segments that would otherwise not be there.

Moreover, the order in which topological changes were carried out was also very sub-optimal. It used to be the case that separation, collision and remeshing were performed in that order. This would result in nodes being separated only to immediately collide again. Furthermore, remeshing at the end meant having segments that should not have been there in the first place, or not having segments that should have. Meaning the network was not properly discretised before collisions and separations were performed, leading to improper structures.

One of the larger problems with higher dislocation densities is the computational complexity of the separation function. Its role is to take high energy, highly connected structures and break them apart into lower energy, lower connection ones. Separation is a very poorly scaling operation $\mathcal{O}\left(\sum_i \sum_{j=2}^{\lfloor C_i/2 \rfloor} \frac{C_i!}{(C_i-j)!j!}\right)$. Where the sum over i is done over the nodes with more connections than is allowed; C_i is the number of connections for node i; and j is the number connections C_i connections can be split into. Mirror symmetries where $2j = C_i$ half the num-

ber of possible splitting configurations for that case. However at best, this puts the lower bound at approximately Sterling's factorial formula $N! \approx \times(N \log N)$ for large N. All of these configurations must be generated—an $\mathcal{O}(M)$ operation where M are the number of nodes to be added—in order to check their power dissipation. Unfortunately, this is a relatively expensive operation because the code dynamically expands the relevant arrays. The nodal velocities and segment forces must then be calcualted for every configuration. Whichever configuration represents the and highest energy dissipation is picked as the final configuration. Locality is assumed, so only the forces and mobilities of the segments directly involved in the split are calculated. It is therefore extremely important that the number of separations each step be kept to a minimum.

The solution may appear to be as simple as calling separation after resolving a single collision, and looping through all collisions in a given timestep. However, this can cause collisions to be undone, leading to an infinite loop. If this happens, and there is no change in the number of nodes and links pre- and post-collision-separation, the offending segments are stored in a buffer whose entries are skipped by the collision detection. Once the segments have been added to the buffer, the iteration is run again without updating the network, and the collision detection will skip any pairs of colliding segments in the buffer. Once a successful collision-separation loop is performed, the buffers are deleted and the cycle repeats itself until there are no more collisions that cause the network to change. This means collisions can be checked more than once, but in keeping with the quasi-static conditions, ensures all first-order[1] energetically favourable topological operations are performed every timestep while preventing infinite loops. This, together with the fixes made to the collision detection, ensures subsequent iterations of the collision-separation loop utilise minimal energy structures. An added benefit is that it also keeps the number of possible separations to a minimum because single collisions get resolved every iteration.

It is worth noting that nodes labelled as immobile (an artificial construct to mimic pinning) are not split by separation. Which means they sometimes become very highly connected and as a result can create an impassable obstacle for other nodes. As a result, they get 'unpinned' when they reach a critical level of connectivity. Separation is performed after this check to break them apart, and also because nodes may need separating despite there being no collisions during the timestep.

The role of remeshing is to ensure the resolution of the network is appropriate. It does so by ensuring there is an appropriate number and distribution of nodes

[1] A true minimal energy structure would require all collisions to be performed and a full traversal of every splitting mode tree.

throughout the network to keep discretisation accuracy high, whilst minimising computational cost. Therefore, the final piece of the puzzle was remeshing the network pre- and post-collision-separation loop, so the next iteration starts with a clean network.

The new order of topological operations in a single iteration is as follows,

1. remesh,
2. collision-separation loop,
3. mobilise highly connected immobile nodes,
4. separation,
5. remesh.

For a given iteration, the first remeshing is called after the network is evolved in time, as the nodes can move into positions that necessitate a remeshing to keep the resolution appropriate; it is then called again at the end of the iteration because collision and separation could have created structures that don't meet the network resolution criteria.

3.3 Conclusions

Discrete dislocation dynamics is a stiff and chaotic dynamical system. It is very sensitive to small changes, as we will see in chapter 4. Having a correct topology is of the utmost importance in this regard. Something as small as a collision that was not performed when it should have, can have dramatic downstream consequences. As dislocation density increases, the degree of correctness when resolving their interactions increases in importance. Not only does it increase the accuracy of a simulation, but improves its running speed.

The increase in accuracy is obvious; detecting all the collsions that should be detected; resolving them in a manner more keeping with the quasi-static assumption yielding the mobility law; and ensuring the network resolution is what it should be; make for more accurate simulations.

Part of the increase in running speed is down to the fact that quasi-static conditions are better met than before, so the timsetep can be greater. The other huge benefit to running speed is down to the asymptotic scaling and high proportionality constant of the algorithms we use; $\mathcal{O}(N^2)$ for seg-seg forces and collisions; $\mathcal{O}(C_i!)$ for a single separation; $\mathcal{O}(MN)$ for tractions. Where N is the number of segments, C_i the number of connections, and M the number of surface elements. Simply put, the fewer things to calculate have a disproportionately large impact

on how many operations need to be performed every time any of these functions is run.

With these improvements, simulations that used to be computationally intractible within a reasonable timeframe are now very doable. Large, complex simulations with high dislocation densities are still slow by virtue of how topologically active they are. They are simultaneously more accurate and faster by multiple orders of magnitude (sometimes 5 or more) than before.

Chapter 4

Dislocation-induced surface tractions

Section 4.1 have been adapted from a manuscript prepared for publication. Section 4.2 outlines the work done in parallelising the analytic traction calculation on GPUs.

4.1 Numeric v.s. analytic tractions

4.1.1 Introduction

In-silico experiments are essential for interpreting experimental data and relating the measured mechanical response to dislocation mechanisms [63–66]. Experimentally, we are limited by the electron transparency of the material and our capacity for subjecting samples to representative loads whilst simultaneously keeping dislocations in focus. Modelling, albeit imperfect and idealised, can greatly inform our understanding and provide insight into the dislocation dynamics responsible for our observations.

Simulating micromechanical tests with explicit dislocation interactions necessitates the coupling of discrete dislocation dynamics (DDD) to the finite element method (FEM) [67]. One of the simples and most popular ways of doing so is the superposition method [68–70]. Which works by decomposing the problem into separate DDD and FE problems as illustrated by fig. 4.1.

Specifically, a linear-elastic solid V bounded by a surface S is subjected to traction boundary conditions, $\boldsymbol{T}$, on S_T and displacement boundary conditions, $\boldsymbol{U}$, on S_U. The $(\tilde{})$ fields are those generated by the dislocations in an infinite solid and in our case are obtained by evaluating analytic solutions in a DDD simulation

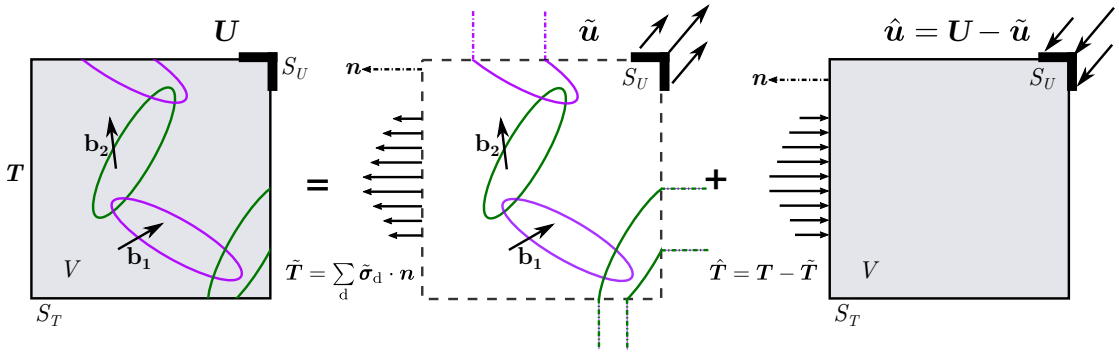


Figure 4.1: The superposition used to couple DDD and FEM. The volume V is bounded by a surface $S = S_T \cup S_U$ and contains a dislocation ensemble and is subjected to tractions $\boldsymbol{T}$ on S_T and $\boldsymbol{u}$ on S_u. First, the traction, $\tilde{\boldsymbol{T}}$, and displacement, $\tilde{\boldsymbol{u}}$, fields due to the dislocations in the infinite domain (DDD) are evaluated on the boundaries S_T and S_U respectively. Then an elastic boundary value problem can be solved with FEM to calculate the corrective elastic fields required to satisfy the boundary conditions $\hat{\boldsymbol{T}} = \boldsymbol{T} - \tilde{\boldsymbol{T}}$ and $\hat{\boldsymbol{u}} = \boldsymbol{U} - \tilde{\boldsymbol{U}}$.

[71]. Formally, the dislocation field satisfies,

$$\left.\begin{aligned} &\nabla \cdot \tilde{\boldsymbol{\sigma}} = \boldsymbol{0} \\ &\tilde{\boldsymbol{\sigma}} = \boldsymbol{C} : \tilde{\boldsymbol{\varepsilon}} \\ &\tilde{\boldsymbol{\varepsilon}} = \tfrac{1}{2}\left(\nabla\tilde{\boldsymbol{u}} + (\nabla\tilde{\boldsymbol{u}})^T\right) \end{aligned}\right\} \quad \text{in V} \tag{4.1}$$

$$\tilde{\boldsymbol{\sigma}} \cdot \boldsymbol{n} = \tilde{\boldsymbol{T}} \quad \text{on } S_T \tag{4.2}$$

$$\left.\begin{aligned} &\tilde{\boldsymbol{u}} = U, \quad t > 0 \\ &\tilde{\boldsymbol{u}} = \boldsymbol{0}, \quad t = 0 \end{aligned}\right\} \quad \text{on } S_U\,. \tag{4.3}$$

As the dislocation fields do not vanish on S, the dislocations load the volume by generating tractions, $\tilde{\boldsymbol{T}}$, on S_T and displacements, $\tilde{\boldsymbol{u}}$, on S_U. This additional loading deforms V, generating an additional "image" stress which the dislocations then feel. Therefore corrective ($\hat{}$) fields must be superimposed to satisfy the desired boundary conditions. The corrective field which accounts for both the applied and image stress is obtained numerically by solving the elastic boundary value problem,

$$\left.\begin{aligned} &\nabla \cdot \hat{\boldsymbol{\sigma}} = \boldsymbol{0} \\ &\hat{\boldsymbol{\sigma}} = \boldsymbol{C} : \hat{\boldsymbol{\varepsilon}} \\ &\hat{\boldsymbol{\varepsilon}} = \tfrac{1}{2}\left(\nabla\hat{\boldsymbol{u}} + (\nabla\hat{\boldsymbol{u}})^T\right) \end{aligned}\right\} \quad \text{in V} \tag{4.4}$$

$$\hat{\boldsymbol{\sigma}} \cdot \boldsymbol{n} = \boldsymbol{T} - \tilde{\boldsymbol{T}} \quad \text{on } S_T \tag{4.5}$$

$$\hat{\boldsymbol{u}} = \boldsymbol{U} - \tilde{\boldsymbol{u}} \quad \text{on } S_U\,. \tag{4.6}$$

Once the solutions to both problems are known, their superposition solves the

desired mixed boundary value problem,

$$\left.\begin{aligned} &\nabla \cdot \boldsymbol{\sigma} = \nabla \cdot (\hat{\boldsymbol{\sigma}} + \tilde{\boldsymbol{\sigma}}) = \mathbf{0} \\ &\boldsymbol{\sigma} = \hat{\boldsymbol{\sigma}} + \tilde{\boldsymbol{\sigma}} = \boldsymbol{C} : (\hat{\boldsymbol{\varepsilon}} + \tilde{\boldsymbol{\varepsilon}}) = \boldsymbol{C} : \boldsymbol{\varepsilon} \\ &\boldsymbol{\varepsilon} = \hat{\boldsymbol{\varepsilon}} + \tilde{\boldsymbol{\varepsilon}} = \tfrac{1}{2}\left(\nabla(\hat{\boldsymbol{u}} + \tilde{\boldsymbol{u}}) + [\nabla(\hat{\boldsymbol{u}} + \tilde{\boldsymbol{u}})]^T\right) = \tfrac{1}{2}\left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right) \end{aligned}\right\} \quad \text{in V} \tag{4.7}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{T} \quad \text{on } S_T \tag{4.8}$$

$$\left.\begin{aligned} &\boldsymbol{u} = \boldsymbol{U}, \quad t > 0 \\ &\boldsymbol{u} = \mathbf{0}, \quad t = 0 \end{aligned}\right\} \quad \text{on } S_U \tag{4.9}$$

For all its simplicity and elegance, the method is not without issue. As the distance between dislocation and surface decreases, $\tilde{\boldsymbol{\sigma}}$ diverges [72]. This can be partially solved by using a non-singular formulation for $\tilde{\boldsymbol{\sigma}}$, such as that proposed by Cai et al. [71]. Regardless, the steep gradients in the dislocation field are difficult to accurately capture as the dislocation approaches S. Another problem with this method is the large computational cost when simulating a heterogeneous solid, as this requires calculating polarisation stresses due to the difference in the elastic constants between phases [68, 72, 73].

A modified superposition scheme [74] can overcome this by dividing the problem into separate DDD-FEM problems coupled through a elastic FE problem. This requires accurate evaluation of $\tilde{\boldsymbol{T}}$ on the domain boundaries—which can only be captured with a fine FE mesh—increasing the computational cost. Therefore, simulating polycrystalline or composite materials using superposition necesitates methods to accurately evaluate both $\tilde{\boldsymbol{u}}$ and $\tilde{\boldsymbol{T}}$. The displacements can be evaluated analytically as shown by [45] and this paper investigates the evaluation of the dislocation tractions analytically.

As previously mentioned, the relative simplicity of the superposition method has made it a popular choice for coupling DDD and FEM as all it needs is the evaluation of FE nodal forces and displacements on the boundary. Furthermore, the analytic expression for the stress field produced by a finite, straight dislocation line segment has allowed Queyreau et al. [75] to use the non-singular formulation [71] to obtain closed-form solutions for the tractions generated by the segment on the surface of a finite element.

4.1.2 Theory

The force exerted by a dislocation ensemble on a node, a, belonging to element, e, is given by,

$$\boldsymbol{F}_a = \int_{S_e} [\tilde{\boldsymbol{\sigma}}(\boldsymbol{x}) \cdot \boldsymbol{n}]\, N_a(\boldsymbol{x})\, \mathrm{d}S_e \,. \tag{4.10}$$

Where $\mathrm{d}S_e$ is the surface of element e with surface area S_e, and N_a is the finite element shape function for node a. The solutions are for linear rectangular surface elements and as such the shape functions are,

$$\begin{aligned} N_1 &= \frac{1}{4}(1-s_1)(1-s_2) \\ N_2 &= \frac{1}{4}(1+s_1)(1-s_2) \\ N_3 &= \frac{1}{4}(1+s_1)(1+s_2) \\ N_4 &= \frac{1}{4}(1-s_1)(1+s_2)\,. \end{aligned} \tag{4.11}$$

Where s_1 and s_2 are the orthogonal coordinates local to the element.

Gauss quadrature is usually used to numerically evaluate the surface integral in eq. (4.10). In 1D this is,

$$\int_{-1}^{1} f(s)\,\mathrm{d}s \approx \sum_{i=1}^{n} w_i f(s_i) \quad \text{where} \quad w_i = \frac{2}{(1-s_i^2)\left[P_n'(s_i)\right]^2}\,, \tag{4.12}$$

is the weighting of the Gauss point, s_i, which is the i^{th} root of the n^{th} normalised Legendre polynomial, $P_n(1) = 1$. P_n' is the first order derivative of P_n. This method is very accurate for functions that can be accurately approximated by polynomials. In fact, for a polynomial of degree n, one needs $n-1$ points to obtain an exact numerical solution. However, this quadrature is well-known for being unsuitable for integrating functions with poles or near-poles [76, 77].

We avoid the strict pole in $\tilde{\boldsymbol{\sigma}}$ by using the non-singular formulation described in [71], where the true singularity is avoided by adding a small cut-off radius to account for the dislocation core. However, if an integration point falls close to, or within the dislocation core, Gauss quadrature can still produce large errors.

For rectangular surface elements, we must transform from the parent element in $(s_1,\, s_2)$ in eq. (4.12) to the real element coordinate system $(x\,, y\,, z)$. Evaluating eq. (4.10) in the parent element and mapping to the real element gives the force on node a,

$$\boldsymbol{F}_a \approx \sum_{i=1}^{Q} w_i \sum_{j=1}^{Q} w_j [\tilde{\boldsymbol{\sigma}}(r_i, s_j) \cdot \boldsymbol{n}] N_a(r_i, s_j) \det(\mathbf{J})\,. \tag{4.13}$$

Where the sum is over the Q quadrature points and $J_{ij} = dx_i/ds_j$, is the element Jacobian defining the transformation from $(s_1,\, s_2) \mapsto (x,\, y)$. Since the real element is rectangular with surface area S_e, then $\det(\mathbf{J}) = S_e/4$. Figure 4.2 contains examples of how the Gauss points are distributed on the surface.

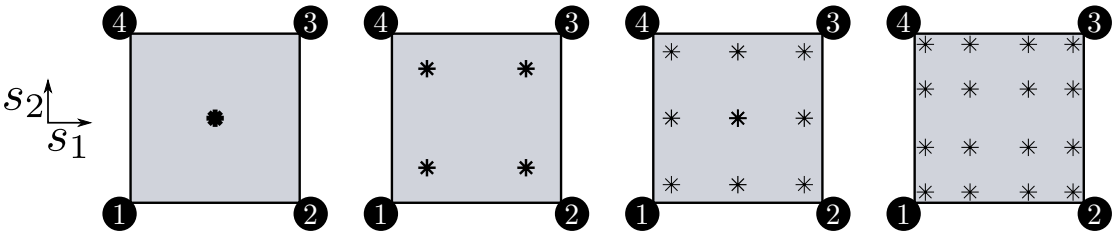


Figure 4.2: Examples of 2D Gauss-Legendre quadrature of the parent element with $Q = 1, 2, 3, 4$. The point size represents the weight w of the integration point. The parent elements are centred at the origin and s_1, $s_2 \in [-1, 1]$. We use an anticlockwise node numbering scheme.

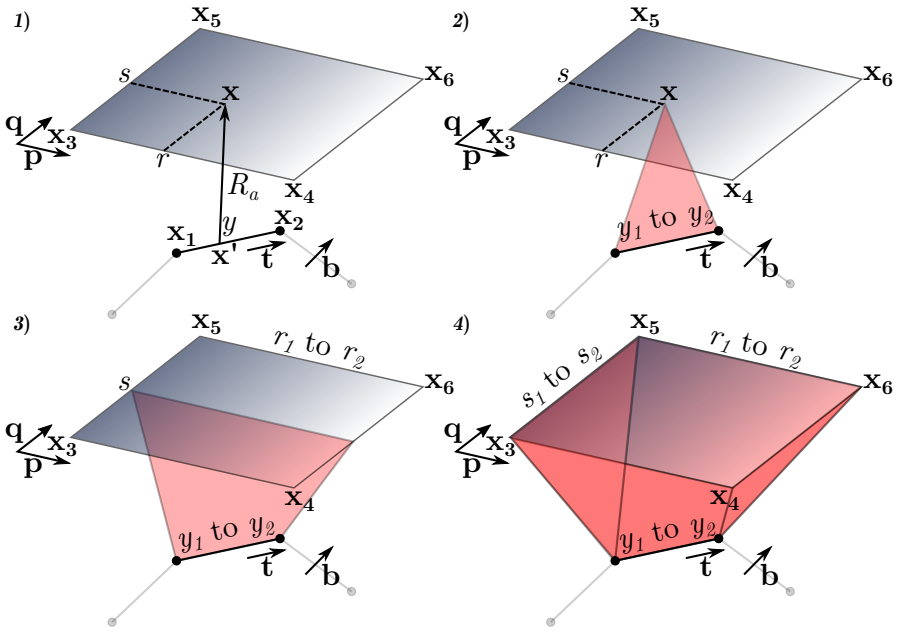


Figure 4.3: Diagram of the parametric line integrals solved by Queyreau et al. [75] to find the forces on linear rectangular surface elements.

Explicitly, eq. (4.10) is actually a triple vector integral as shown in fig. 4.3. This is because given the isotropic Burgers vector distribution proposed in [71], the dyadic form of the stress tensor produced by a straight, finite dislocation segment bounded by nodes at $\mathbf{x_1}$ and $\mathbf{x_2}$ [75] is,

$$\begin{aligned}\tilde{\boldsymbol{\sigma}}(\boldsymbol{x}) = &-\frac{\mu}{8\pi}\int_{\mathbf{x_1}}^{\mathbf{x_2}}\left(\frac{2}{R_a^3}+\frac{3a^2}{R_a^5}\right)\left[(\boldsymbol{R}\times\boldsymbol{b})\otimes \mathrm{d}\boldsymbol{x}' + \mathrm{d}\boldsymbol{x}'\otimes(\boldsymbol{R}\times\boldsymbol{b})\right] \\ &+\frac{\mu}{4\pi(1-\nu)}\int_{\mathbf{x_1}}^{\mathbf{x_2}}\left(\frac{1}{R_a^3}+\frac{3a^2}{R_a^5}\right)\left[(\boldsymbol{R}\times\boldsymbol{b})\cdot\mathrm{d}\boldsymbol{x}'\right]\boldsymbol{I_2} \\ &-\frac{\mu}{4\pi(1-\nu)}\int_{\mathbf{x_1}}^{\mathbf{x_2}}\frac{1}{R_a^3}\left[(\boldsymbol{b}\times\mathrm{d}\boldsymbol{x}')\otimes\boldsymbol{R}+\boldsymbol{R}\otimes(\boldsymbol{b}\times\mathrm{d}\boldsymbol{x}')\right] \\ &+\frac{\mu}{4\pi(1-\nu)}\int_{\mathbf{x_1}}^{\mathbf{x_2}}\frac{3}{R_a^5}\left[(\boldsymbol{R}\times\boldsymbol{b})\cdot\mathrm{d}\boldsymbol{x}'\right]\boldsymbol{R}\otimes\boldsymbol{R},\end{aligned} \tag{4.14}$$

where,

$$\boldsymbol{R} = \boldsymbol{x}-\boldsymbol{x}' = y\boldsymbol{l}+r\boldsymbol{p}+s\boldsymbol{q} \tag{4.15}$$

$$R_a = \sqrt{\boldsymbol{R}\cdot\boldsymbol{R}+a^2} \tag{4.16}$$

$$\mathrm{d}\boldsymbol{x}' = -\mathrm{d}y\boldsymbol{l}\,. \tag{4.17}$$

The vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ are aligned with the edges of the rectangular finite element, $\boldsymbol{n}=\boldsymbol{p}\times\boldsymbol{q}$ is the element surface normal (pointing away from the dislocation), and $\boldsymbol{l}$ is parallel to the dislocation line segment as shown in fig. 4.3. Then (provided $\boldsymbol{l}$ is not parallel to $\boldsymbol{p}$ or $\boldsymbol{q}$) $\boldsymbol{R}$ can be expressed in terms of $(\boldsymbol{l},\ \boldsymbol{p},\ \boldsymbol{q})$ with coefficients,

$$y=\frac{\boldsymbol{R}\cdot\boldsymbol{n}}{\boldsymbol{l}\cdot\boldsymbol{n}},\quad r=\frac{\boldsymbol{R}\cdot(\boldsymbol{q}\times\boldsymbol{l})}{\boldsymbol{p}\cdot(\boldsymbol{q}\times\boldsymbol{l})},\quad s=\frac{\boldsymbol{R}\cdot(\boldsymbol{p}\times\boldsymbol{l})}{\boldsymbol{q}\cdot(\boldsymbol{p}\times\boldsymbol{l})}\,. \tag{4.18}$$

Substituting eq. (4.14) and eq. (4.11) into eq. (4.10) yields four long and messy equations (one for each FE node) that were elegantly solved by Queyreau et al. [75] by utilising the fact that the triple integrals all had the form,

$$H_{ijkl} = \int_{r_1}^{r_2}\int_{s_1}^{s_2}\int_{y_1}^{y_2}\frac{r^i s^j y^k}{R_a^m} \tag{4.19}$$

$$\begin{aligned}&\text{when } m=5 \text{ then } i,\ j\in[0,\,3],\ k\in[0,\,2]\\ &\text{when } m=3 \text{ then } i,\ j\in[0,\,2],\ k\in[0,\,1]\\ &\text{when } m=1 \text{ then } i=j=k=0\,.\end{aligned} \tag{4.20}$$

Using partial differentiation and integration by parts, they found a series of recurrence relations that lead to double and single integrals of similar form to eq. (4.19). All of which are used to construct a full, exact solution. The recurrence relations stop working when $i = j = k = 0$ and $m = 1, 3$. At which point, direct integration of the remaining single and double integrals (the last triple integrals all cancel out in the global calculation) yields six seed functions that are used as the starting point for the recurrence relations. Three of them are logarithms and three either arctangents or—if a discriminant is negative—hyperbolic arctangents. The details of the procedure can be found in [75].

Although exact, the use of arctangents, hyperbolic arctangents and logarithmic functions, compounded by the large number of recurrence relations is prime territory for error propagation and numerical problems (see section 4.1.3). The problem is particularly egregious when using general purpose compilers instead of high-performance or scientific computing compilers where mathematical functions are implemented more precisely. Such issues must be taken into account when using analytic tractions, which can be done by using numerical tolerances as described in section 4.1.3.

In simulations, tractions are manifested as image stresses calculated by the FE solver at FE nodes. In order to validate and compare the practical differences between analytic and numeric methods, we compare the resulting image stresses from both methods to the analytic expressions for infinite dislocations in inhomogenous media for edge dislocations [78], as well as those for screw dislocations [79, p. 59, 64]. We keep the same nomenclature and coordinate system as both infinite-domain solutions. Where the traction surface is the line $x = 0$, the dislocation line direction is the positive z-direction (pointing out of the page), the dislocation coordinates are represented by $(a,\ c)$, and points in the xy-plane described by their $(x,\ y)$ coordinates.

The original paper by Head [78] has a few typos that have been replicated in other sources. We therefore include the complete and correct expressions in eqs. (4.21) to (4.23). Head [78] gives two basic cases. Equation (4.21) corresponds to the case where $\boldsymbol{b}$ is perpendicular to the surface and positive b means it points

in the positive x-direction,

$$\sigma_{xx} = D(y-c)\left\{-\frac{3(x-a)^2+(y-c)^2}{[(x-a)^2+(y-c)^2]^2}+\frac{3(x+a)^2+(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.21a}$$
$$\left.+4ax\frac{3(x+a)^2-(y-c)^2}{[(x+a)^2+(y-c)^2]^3}\right\},$$

$$\sigma_{yy} = D(y-c)\left\{\frac{(x-a)^2-(y-c)^2}{[(x-a)^2+(y-c)^2]^2}-\frac{(x+a)^2-(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.21b}$$
$$\left.+4a(2a-x)\frac{(x+a)^2+(3x+2a)(y-c)^2}{[(x+a)^2+(y-c)^2]^3}\right\},$$

$$\sigma_{xy} = D\left\{(x-a)\frac{(x-a)^2-(y-c)^2}{[(x-a)^2+(y-c)^2]^2}-(x+a)\frac{(x+a)^2-(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.21c}$$
$$\left.+2a\frac{6x(x+a)(y-c)^2-(x-a)(x+a)^3-(y-c)^4}{[(x+a)^2+(y-c)^2]^3}\right\}.$$

Equation (4.22) corresponds to the case where the $\boldsymbol{b}$ lies parallel to the surface and positive b means it points in the positive y-direction,

$$\sigma_{xx} = D\left\{(x-a)\frac{(x-a)^2-(y-c)^2}{[(x-a)^2+(y-c)^2]^2}-(x+a)\frac{(x+a)^2-(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.22a}$$
$$\left.+2a\frac{(3x+a)(x+a)^3-6x(x+a)(y-c)^2-(y-c)^4}{[(x+a)^2+(y-c)^2]^3}\right\},$$

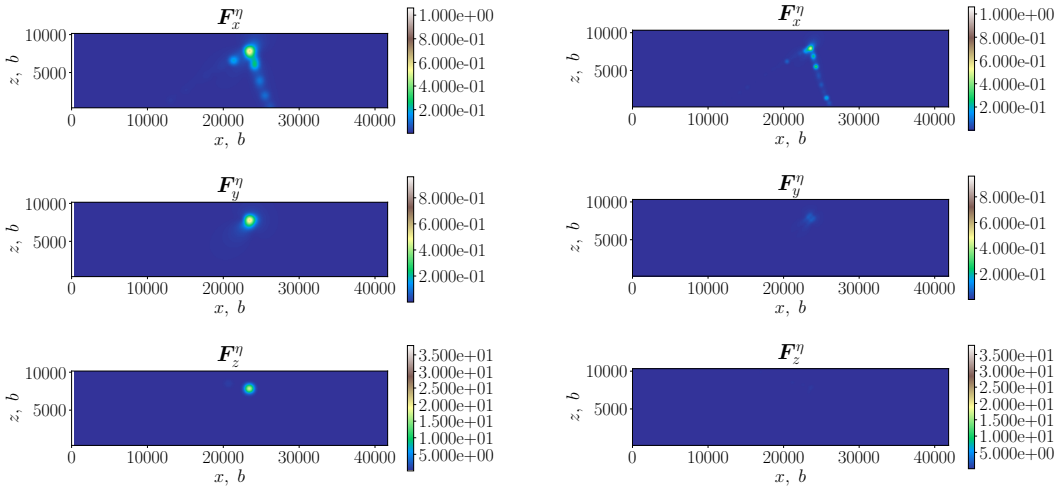
$$\sigma_{yy} = D\left\{(x-a)\frac{(x-a)^2+3(y-c)^2}{[(x-a)^2+(y-c)^2]^2}-(x+a)\frac{(x+a)^2+3(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.22b}$$
$$\left.-2a\frac{(x-a)(x+a)^3-6x(x+a)(y-c)^2+(y-c)^4}{[(x+a)^2+(y-c)^2]^3}\right\},$$

$$\sigma_{xy} = D(y-c)\left\{\frac{(x-a)^2-(y-c)^2}{[(x-a)^2+(y-c)^2]^2}-\frac{(x+a)^2-(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.22c}$$
$$\left.+4ax\frac{3(x+a)^2-(y-c)^2}{[(x+a)^2+(y-c)^2]^3}\right\}.$$

Equation (4.23) corresponds to screw dislocations, which are markedly simpler as only the shear components are non-zero. Here $\boldsymbol{b} = \boldsymbol{l}$ so positive b means it points in the positive z-direction,

$$\sigma_{xz} = -D\left(\frac{y-c}{(x-a)^2+(y-c)^2}-\frac{y-c}{(x+a)^2+(y-c)^2}\right) \tag{4.23a}$$

$$\sigma_{yz} = D\left(\frac{x-a}{(x-a)^2+(y-c)^2}-\frac{x+a}{(x+a)^2+(y-c)^2}\right). \tag{4.23b}$$



(a) 60×16 elements ($x \times y$). (b) 120×36 elements ($x \times y$).

Figure 4.4: Relative error ($\boldsymbol{F}^\eta$) in the nodal force obtained using numerical integration with one quadrature point $Q = 1$, and the analytic solution. The xz-face of a rectangular cantilever with plane normal, $\boldsymbol{n} = [0\,\bar{1}\,0]$. The dislocation is of pure edge character with $\boldsymbol{b} = [1\,0\,\bar{1}]$, and line direction, $\boldsymbol{l} = [\bar{1}\,2\,\bar{1}]\,/\sqrt{6}$ which pierces both xz-faces. The dislocation has its centre at the centroid of the cantilever.

In every case, the constant D is defined by eq. (4.24),

$$D = \frac{\mu}{2\pi} \cdot \frac{1+\nu}{1-\nu^2} \cdot b\,. \tag{4.24}$$

Note the first terms of eqs. (4.21) to (4.23) all correspond to the stress field generated by the dislocation itself. The following terms are the corrective terms required to make the boundary conditions on the surface equal to zero. Therefore, we can split these equations and only look at the real or corrective terms independently, which we do in order to only visualise the effect of the different traction calculations. Furthermore, eqs. (4.21) to (4.23) are all singular at the dislocation coordinates. Our simulation code uses the non singular expressions found by Cai et al. [71], which smooth out drastic increases in stresses and avoid numerical blow up as we near the dislocation core.

4.1.3 Methodology

Numerical integration of tractions can produces unexpected behaviour such as force hot spots and sign inversions as a dislocation approaches a surface. During a large simulation, these effects are hard to spot. Figure 4.4 has a quick example of the relative errors for an idealised system not disimilar to what can be found with in a simple cantilever bending simulation with a single dislocation loop. As expected, the errors decrease as the mesh gets finer.

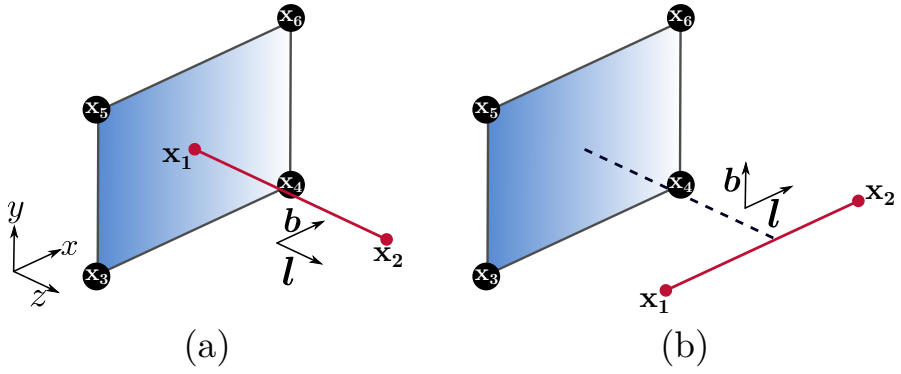


Figure 4.5: Simple test cases for an edge segment and surface element perpendicular (a) and parallel (b). The perpendicular dislocation is centered at the midpoint of the surface element, node $\mathbf{x_1}$ is separated by a perpendicular distance $\mathbf{x_1}^z$ to prevent the dislocation from intersecting the surface. On the right, the parallel dislocation runs along the x-axis at half the height of the surface element. The nodes of each dislocation line segment are kept at a perpendicular distance of at least one core radius away from the surface element.

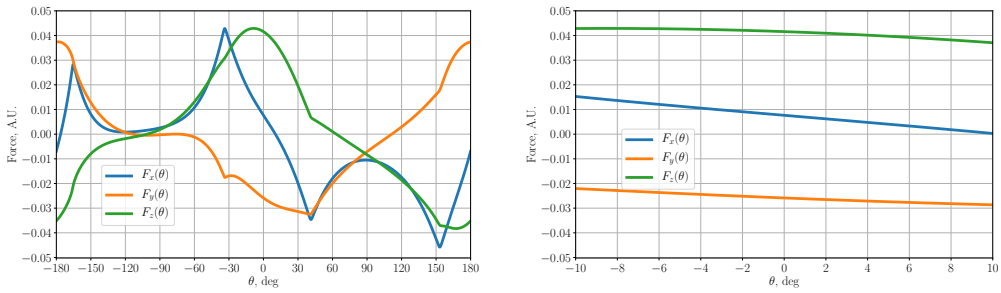
Queyreau et al. [75] identified that for a given number of quadrature points, the error is dependent on the dislocation character but always increases rapidly as the distance between the segment and element surface decrease (see figs. 4.9 and 4.10).

Expanding the test cases reveals just how problematic numerical integration of the tractions can be when a dislocation approaches a surface. The two basic test cases, an orthogonal and parallel edge segment, are shown in fig. 4.5.

The symmetry of these simple test cases benefits the accuracy of the numerical solutions because the stress fields exhibit symmetries about the dislocation line. If under ideal conditions for error cancellation, it can be proven that the numerical method is inferior to the analytic one, it can be more effectively argued that the analytic one should always be used instead.

Queyreau et al. [75] found the analytic solution is approximately 10 times more computationally expensive than its numerical counterpart for 1 quadrature point, our findings agree with this result but the implications for a whole simulation are favourable (see section 4.1.4).

One serious disadvantage of the analytic tractions is that the implementation of the analytic solution is also much more involved and full of snags. One issue is the calculation of the y-coordinate in the local coordinate frame as shown in fig. 4.3 and eq. (4.18). If $\boldsymbol{l} \perp \boldsymbol{n}$, we get a singularity. As mentioned in [75], an easy fix is to rotate the line segment. We do this about its midpoint and around the $\boldsymbol{l} \times \boldsymbol{n}$ axis in both clockwise and anticlockwise directions. We use the mean of the values as the answer for $\theta = 0$. An example of what this rotation looks like in terms of the forces on a surface element can be seen in fig. 4.6 (avoiding the



(a) (b)

Figure 4.6: (a) Example of the components of the total force on a surface element (the force summed over all four FE nodes) as a dislocation segment parallel to the surface element ($\theta = 0$) is rotated about its midpoint around the axis defined by $\boldsymbol{l} \times \boldsymbol{n}$. (b) is zomms into $\pm 10\,\text{deg}$, the force is smooth but not necessarily antisymmetric about the neighbourhood of $\theta = 0$.

singularity at $\theta = 0$). The specific shape of the curves will vary depending on the element-segment configuration, but is smooth and well-behaved about singularity. For our purposes, we use a total of 8 perturbations of 1 deg each (4 clockwise and 4 anti-clockwise). We found this worked well, but can be changed if desired.

However, under finite-precision arithmetic, the check for orthogonality is dependent on the length scales involved in the simulation. This is particularly important considering the aforementioned use of arctangents, hyperbolic arctangents, logarithms and the large number of recurrence relations. Causing unexpected and rampant error propagation and numerical blow up is not difficult to achieve. When it happens, finding the root cause can lead one on a wild goose chase that is best avoided and often leads to a single segment in a single step of a long simulation that is slightly too small compared to its distance to a surface element to which it is slightly too parallel. To avoid these rare but high impact scenarios, we created a heuristic that dictates how strict the tolerance should be in order for the code to consider a segment to be parallel,

$$|\boldsymbol{l} \cdot \boldsymbol{n}| \lesssim \frac{\max\left(|\boldsymbol{R} \cdot \boldsymbol{n}|\right)}{10^8}\,. \tag{4.25}$$

In our case the numerator on the RHS is simply the FEM domain's largest dimension. 10^8 is used instead of actual machine precision $\sim 10^{15}$ because the seed functions and large number of recurrence relations of the solution propagate errors if the value of $\boldsymbol{l} \cdot \boldsymbol{n}$ is too close to zero. Ironically, tolerances which are too large can cause the perturbations to rotate the dislocation segment closer to the singularity, producing erroneous results. Larger than necessary tolerances can also slow down the calculation by detecting dislocations that are far enough from the special case that they can be treated like non-parallel segments. This heuristic is

a good general purpose rule that keeps the tolerance in a goldilocks zone.

Another issue with the rotation is that one does not want a dislocation segment to intersect the surface when it is being rotated. Naively one would calculate the maximum rotational angle, $\theta_{\max}$, to be,

$$\theta_{\max} = \arctan\left(\frac{2d}{|\mathbf{x_2} - \mathbf{x_1}|}\right) . \tag{4.26}$$

Where d is the minimum orthogonal distance from the dislocation to the surface element i.e. the collision distance—which in our case is a function of the dislocation core radius—and $\mathbf{x_1}$, $\mathbf{x_2}$ are the dislocation segment node coordinates—whose maximum and minimum lengths are also functions of the dislocation core radius. However, $\theta_{\max}$ might be too small in cases where the segment length is too small compared to the distance to a surface element, or when the segment length is much greater than d. Fine-tuning the angle is a task that involves knowing the minimum collision distance, minimum segment length, dislocation core radius, and the compiler's implementation of mathematical functions. Given the rarity of such cases and their comparatively low impact, we chose our 1 deg perturbation such that we safely avoid this problem while keeping within the bounds of the non-singular model.

Furthermore, the chirality and self-consistency of the FE nodes must be accounted for such that they are in the proper order regardless of the element face they belong to. Here we use 8-node linear hexahedral (brick) elements. The node ordering for the various surfaces is that for which the calculated normals point out from the the domain, this is dependant on the specific FE mesh implementation.

The total force on a given node must include the force contributions from every element in which said node appears, see fig. 4.7. The specifics of the mapping depend on the global FE node numbering. Using fig. 4.7 as our reference labels for elements and nodes, e and n respectively, we can give a concrete example of

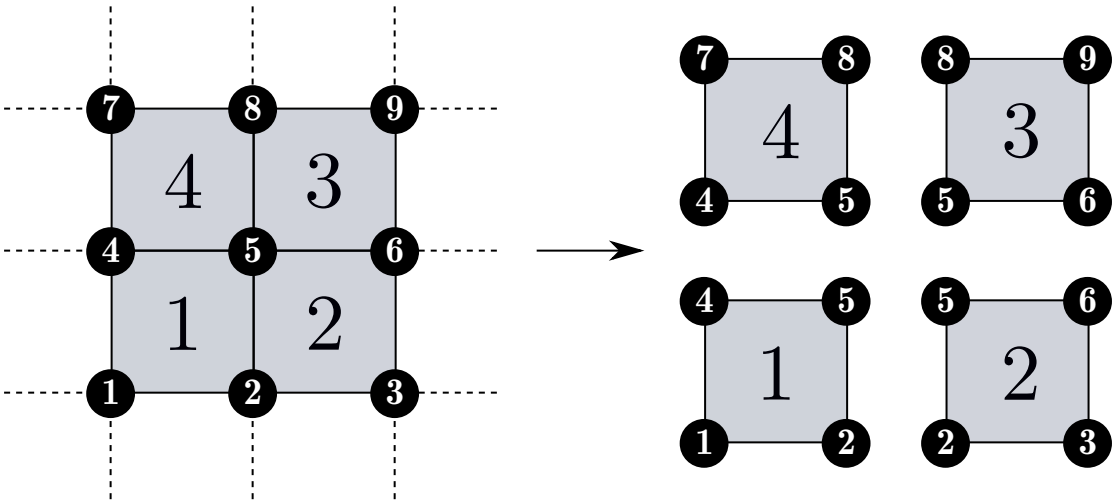


Figure 4.7: FE nodes are shared by either 4 element faces or 3 if it is a corner node. The total force on a given node is the summation of the force contributions from each element it belongs to.

how this is done by defining,

$$\boldsymbol{x_{e,n}} \equiv \begin{bmatrix} x_{e,n} & y_{e,n} & z_{e,n} \end{bmatrix}^{\mathsf{T}}, \quad \boldsymbol{x_n} \equiv \begin{bmatrix} x_n & y_n & z_n \end{bmatrix}^{\mathsf{T}} \tag{4.27a}$$

$$\mathbf{N_L} = \begin{bmatrix} l_{1,1} & l_{1,2} & l_{1,4} & l_{1,5} \\ l_{2,2} & l_{2,3} & l_{2,5} & l_{2,6} \\ l_{3,5} & l_{3,6} & l_{3,8} & l_{3,9} \\ l_{4,4} & l_{4,5} & l_{4,7} & l_{4,8} \end{bmatrix}, \quad \boldsymbol{\gamma} = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_9 \end{bmatrix} \tag{4.27b}$$

$$\mathbf{F_e} = \begin{bmatrix} \boldsymbol{x_{1,1}} & \boldsymbol{x_{1,2}} & \boldsymbol{x_{1,4}} & \boldsymbol{x_{1,5}} \\ \boldsymbol{x_{2,2}} & \boldsymbol{x_{2,3}} & \boldsymbol{x_{2,5}} & \boldsymbol{x_{2,6}} \\ \boldsymbol{x_{3,5}} & \boldsymbol{x_{3,6}} & \boldsymbol{x_{3,8}} & \boldsymbol{x_{3,9}} \\ \boldsymbol{x_{4,4}} & \boldsymbol{x_{4,5}} & \boldsymbol{x_{4,7}} & \boldsymbol{x_{4,8}} \end{bmatrix}, \quad \tilde{\boldsymbol{F}} = \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_9} \end{bmatrix}. \tag{4.27c}$$

Where $\boldsymbol{x_{e,n}}$ is a 3×1 column vector corresponding to the $(x\,, y\,, z)$ dislocation induced forces on node, n, on the surface element, e. There are four of these per rectangular surface element, where a given node, n, can appear in multiple surface elements (e.g. node 5 in fig. 4.7 is shared by all 4 surface elements), all of which independently contribute to the total force on said node. $\boldsymbol{x_n}$ is a 3×1 column vector corresponding to the total $(x\,, y\,, z)$ dislocation induced forces on node n. These are used to shorten the definition of eq. (4.27c) and are not explicitly defined in the implementation, rather they give the force matrices $\mathbf{F_e}$ and $\tilde{\boldsymbol{F}}$ a specific row order. $\mathbf{N_L}$ is crucial for the correct implementation of this analytical solution in traditional FE codes. It is the $E \times 4$ matrix corresponding to the global label of each node in a given surface element. Each row of the matrix represents a surface element and each column represents a node in the surface element. We cannot

naïvely add the columns together as that would give the total force acting on the element as a whole, not each FE node individually. We chose to arrange the columns in accordance to fig. 4.3 as it makes it easier to implement the solution, but the only thing that matters is that the basis vectors $\boldsymbol{n}$, $\boldsymbol{p}$, $\boldsymbol{q}$ are calculated appropriately. $\boldsymbol{\gamma}$ is the vector with the FE node labels, which makes mapping force to node possible. $\mathbf{F_e}$ is the $3E \times 4$ matrix where the forces acting on each of the four nodes (column) in a particular surface element (each element corresponds to three consecutive rows because there are three dimensions) are stored. $\tilde{\boldsymbol{F}}$ is the $3N \times 1$ column vector where the total forces on each node are stored (each node has three rows because there are three dimensions). This is easily generalisable to E elements and N nodes.

Algorithm 4.1 illustrates how the total force on each node is obtained. However, our implementation does not strictly follow it because we memoise a generalised version of $\boldsymbol{L}$ upon simulation initialisation instead of finding one at every iteration, reducing computational time but requiring us to account for nodes without traction boundary conditions. Our indexing also starts at 1, but zero indexing makes the algorithm easier to follow.

Algorithm 4.1 Assuming $\tilde{\boldsymbol{F}}$ is arranged the same way as $\boldsymbol{\gamma}$ and indexing starts at 0.

▷ Loop through the array containing the node labels of the relevant surface nodes.
for $i = 0$; $i <$ length($\boldsymbol{\gamma}$); $i++$ **do**
    ▷ Save the global node label for the current iteration.
    $n \leftarrow \boldsymbol{\gamma}[i]$
    ▷ Use the node label to find a vector, $\boldsymbol{L}$, with the linearised indices in $\mathbf{N_L}$ where node n appears as part of a surface element whose tractions we are calculating.
    $\boldsymbol{L} \leftarrow \text{find}(\mathbf{N_L} == n)$
    ▷ Loop over coordinates.
    for $k = 0$; $k < 3$; $k++$ **do**
        ▷ Use global node label vector to index the force array from the analytical force calculation. Multiplied by 3 because there are three coordinates per node. We sum the forces from the analytical calculation because the same global node can be part of multiple surface elements. We add k because the x, y, z coordinates are consecutively stored in $\mathbf{F_e}$.
        $\tilde{\boldsymbol{F}}[3n + k] \leftarrow \tilde{\boldsymbol{F}}[3n + k] + \sum \mathbf{F_e}[3\boldsymbol{L} + k]$
    end for
end for

The resulting force vector is then used in eq. (4.5) to calculate $\hat{\boldsymbol{\sigma}}$, since $\tilde{\boldsymbol{T}} \equiv \tilde{\boldsymbol{F}}$. Figure 4.8 shows the simple system we used to compare the image stresses calculated by our FE solver using numeric tractions v.s. analytic tractions v.s.

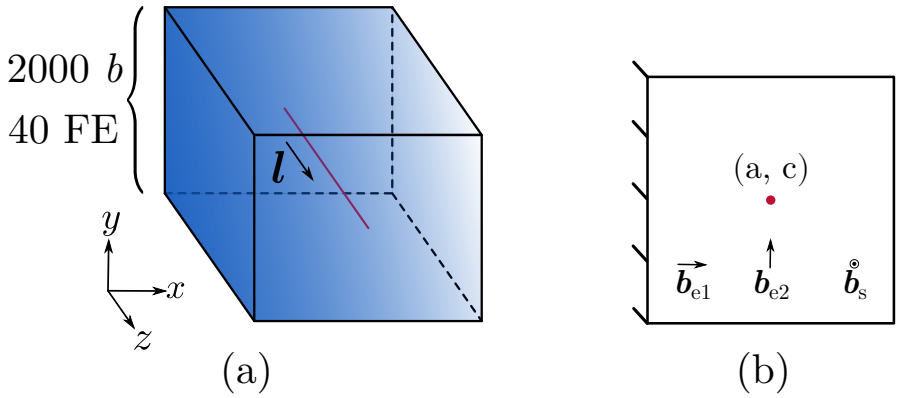


Figure 4.8: Dislocation parallel to a surface described by the line $x = 0$, where the (x, y) dislocation coordinates are (a, c) and the dislocation line going in the positive z-direction. (a) describes the box we used for our comparison, a $40\times40\times40$ element cubic box with side lengths equal to 2000 with $\boldsymbol{T} = \boldsymbol{0}$, traction boundary conditions only on the nodes along $x = 0$, and $\boldsymbol{U} = \boldsymbol{0}$, displacement conditions everywhere else. (b) is the 2D view with the dislocation coordinates $(a,\ c)$, as well as the two edge Burgers vectors in [78], $\boldsymbol{b}_{\text{e1}} \equiv \boldsymbol{b} = [1\,0\,0]$, $\boldsymbol{b}_{\text{e2}} \equiv \boldsymbol{b} = [0\,1\,0]$ and the screw Burgers vector $\boldsymbol{b}_{\text{s}} \equiv \boldsymbol{b} = \boldsymbol{l} = [0\,0\,1]$ in [79, p. 59, 64].

infinite-domain, singular image stresses in eqs. (4.21) to (4.23). The three test cases are: two edge dislocations and one screw, all of which have line direction $\boldsymbol{l} = [0\,0\,1]$. The Burgers vectors for the different scenarios are $\boldsymbol{b}_{\text{e1}} \equiv \boldsymbol{b} = [1\,0\,0]$, $\boldsymbol{b}_{\text{e2}} \equiv \boldsymbol{b} = [0\,1\,0]$, as defined in [78]; and $\boldsymbol{b}_{\text{s}} \equiv \boldsymbol{b} = \boldsymbol{l} = [0\,0\,1]$, as defined in [79, p. 59, 64].

The units in all our examples are normalised to lattice parameter, a is the dislocation core radius for the non-singular formulation discussed in [71], and $b \equiv \|\boldsymbol{b}\|$.

The slices we took for our contour plots in section 4.1.4 are on the middle plane of the domain at $z = 1000$.

We also ran a very simple simulation comparing the results between using analytic tractions as opposed to numeric ones. Finding a simple yet clear case of tractions producing catastrophically wrong behaviours can be difficult. Large differences in image forces are relatively rare and other factors often dominate. Such factors include the core radius—which affects the dislocation line tension and therefore works against the deformation of a dislocation line—mobility law and mobility parameters used, external loading, etc.

However, we found a simple and realistic configuration that is close to the setup in the infinite domain examples. The differences between those comparisons and the simulation are described in table 4.1. We use a mobility law developed in-house by B. Bromage [45], that corrects common issues found in other laws. We also rotate the domain such that the $[1\,1\,1]$ and $[1\,\bar{1}\,0]$ crystallographic directions respectively correspond to the simulation's x and y-directions, for this we use the

Table 4.1: Parameters for our simulation. Where $\mathbf{R}$ is a rotation matrix such that the $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ basis vectors of our simulation correspond to the $\boldsymbol{b}$, $\boldsymbol{n}$, $\boldsymbol{l}$ crystallographic directions, i.e. $\mathbf{R}\boldsymbol{x} = \boldsymbol{b}$. Everything else was kept just as the previous comparisons. All values are in units of lattice parameters.

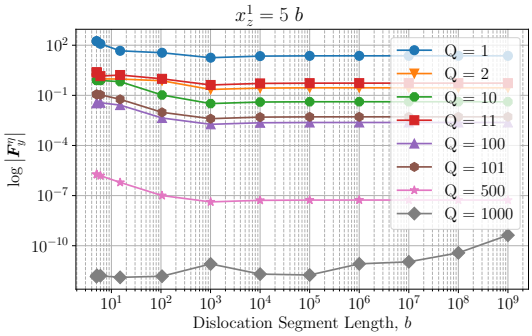
Parameter	Value
Crystal Structure	BCC
$\boldsymbol{b}$	$[1\,1\,1]$
$\boldsymbol{n}$	$[1\,\bar{1}\,0]$
$\boldsymbol{l}$	$[1\,1\,\bar{2}]$
b	$\sqrt{3}/2$
a	5
Grid Size	(20, 20, 20)
Domain Size	(2000, 2000, 2000)
Lattice size	3.18×10^{-4} μm
$(x_0,\, y_0)$	(62.5, 1000)
Min segment length	50
Max segment length	125
$\mathbf{R}$	$[\hat{\boldsymbol{b}}\ \hat{\boldsymbol{n}}\ \hat{\boldsymbol{l}}]$

same rotation technique as [66]. The dislocation was allowed to move under no external loads or displacements, with $\boldsymbol{T} = \mathbf{0}$ boundary conditions only on the yz-plane and $\boldsymbol{U} = \mathbf{0}$ everywhere else.

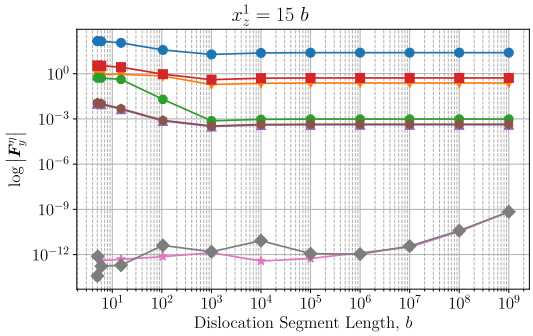
4.1.4 Results and discussion

Figure 4.9 shows that even for dislocations only one dislocation core radius (5) away from the surface element, the force can be obtained, up to numerical precision, with 1000 Gauss quadrature points Q for all segment lengths tested. It also shows a very peculiar issue Gauss quadrature has when computing integrals of rational functions when the Gauss nodes are close poles/maximal values. This undesirable behaviour is observed in the case where $Q = 11$. Where the highest weighted Gauss node is closest to the point where $1/R_a$ is maximal, resulting in lower accuracy when compared to $Q = 2, 10$ in fig. 4.9 (a) and (b). It is worth noting however that the relative errors for small numbers of quadrature points don't really start decreasing until relatively large distances. And when close to a surface, these can be quite large even under highly symmetric circumstances.

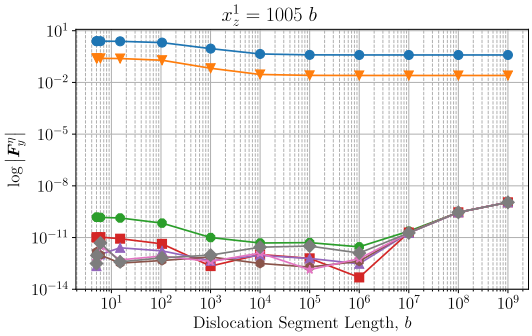
The limitations become even more evident when the dislocation line segment is parallel to a surface element. In fig. 4.10, we observe the relative errors are quite large when close to the surface. At distances larger than 10^4 , loss of significance causes the relative errors to converge at ~ 1 which is expected as two finite precision floating point numbers get closer to zero. Of particular note is how large the



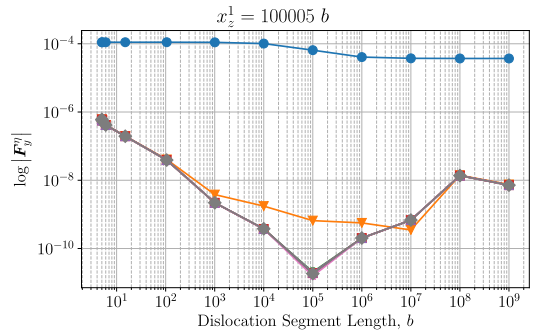
(a) Segment starts a single dislocation core radius away.



(b) Segment starts 3 dislocation core radii away.



(c) Segment starts 201 dislocation core radii away.



(d) Segment starts 20001 dislocation core radii away.

Figure 4.9: Log-log plot of the relative error as a function of dislocation segment length for a perpendicular edge dislocation (fig. 4.5a). x_z^1 is the z-coordinate of node $\mathbf{x_1}$. $\boldsymbol{b} = [1\,0\,0]$, line direction, $\boldsymbol{l} = [0\,0\,1]$, and dislocation core radius, $a = 5$, the surface element's normal and size are, $\boldsymbol{n} = [0\,0\,1]$, $L = 1000$, respectively. Q is the number of quadrature points per dimension.

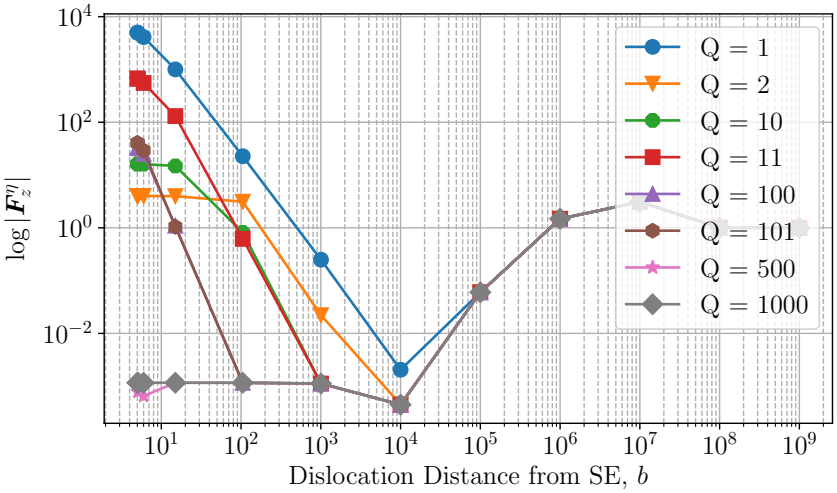


Figure 4.10: Log-log plot of the relative error as a function of distance from the surface element for a parallel edge dislocation (fig. 4.5b). Burgers vector, $\boldsymbol{b} = [0\,1\,0]$, line direction, $\boldsymbol{l} = [1\,0\,0]$, dislocation core and surface element parameters are the same as fig. 4.9. The dislocation length is fixed to 10^6, $x \in [-0.5 \times 10^6, 0.5 \times 10^6]$ and bisects the surface element along the $[1\,0\,0]$ direction. The whole dislocation was segmented into 10^4 pieces of length 100 to prevent the dislocation from intersecting the surface element when they were rotated to avoid the singularity. The relative error at large distances ($> 10^6$) converges to one due to loss of significance.

relative errors are even at 100 lattice units away from the surface, even for large numbers of quadrature points. This figure backs up the earlier point regarding Gauss nodes close to maximal values of rational functions. It can be shocking to see that as $Q = 2$ performs significantly better than $Q = 10, 11, 100, 100$ at distances from the surface as having nodes near the maximal values is a large source of error. Consequently, different configurations have different optimal numbers of points. The numerical instability of this method, which when coupled to the chaotic nature of dislocation dynamics and stiffness of the equations of motion, can lead to large deviations between simulations.

From figs. 4.9 and 4.10 one might be tempted to say that for a segment parallel to a surface, Gauss quadrature performs far worse than for a perpendicular one. However there is a further wrinkle in this problem: symmetry. To exemplify this we plot the relevant components of the stress tensor for the arrangement found in fig. 4.5(a) in figs. 4.11a to 4.11c. The σ_{xz} and σ_{zz} components are antisymmetric about the centre of the element. If we use Gauss quadrature on them, we sample equivalent but oppositely valued points that are equally weighted, thus the sum vanishes and therefore do not contribute to fig. 4.9. However, σ_{yz} does not vanish, but can be accurately integrated with sufficiently large Q. If we were to move the dislocation off-centre such that these symmetries are broken, the errors would

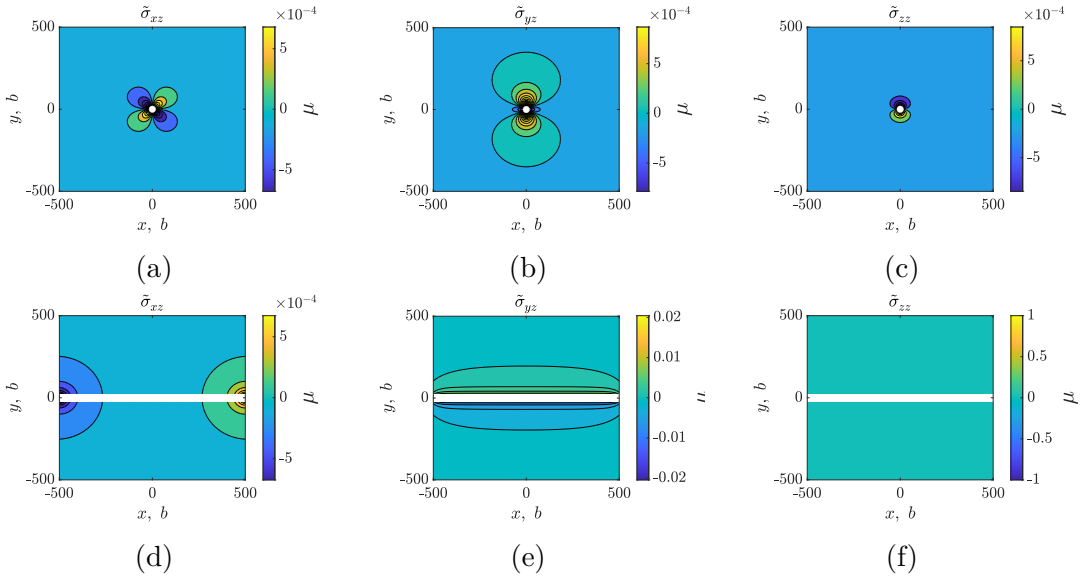


Figure 4.11: (a) to (c) show the real stress fields from a dislocation in the same configuration that yields the plots found in fig. 4.9 as shown in fig. 4.5(a), where the closest node is a dislocation core radius away from the surface, $a = 5$. (e) to (f) does the same for the configuration that yields fig. 4.10 as shown in fig. 4.5(b), where the whole dislocation is a core radius away from the surface, $a = 5$. The white line or dot is the dislocation. Units are in terms of lattice parameters.

increase.

We also plot the relevant stresses for the arrangement described by fig. 4.5(b) in figs. 4.11d to 4.11f. From fig. 4.11d and fig. 4.11e, it is immediately apparent why gauss quadrature fails so spectacularly in fig. 4.10. At low numbers of quadrature points, it fails to appropriately sample the rapidly changing value of $\tilde{\sigma}_{xz}$ at both ends of the dislocation, as well those in the neighbourhood of the dislocation in $\tilde{\sigma}_{yz}$. Moreover, the further away the quadrature points are from the midpoint of the domain, the lower their relative weighting. So even with a relatively large number of them, there can still be large errors. Which is a particularly egregious problem in fig. 4.11e, and explains why such a large number of quadrature points is required to accurately compute the integral.

Despite these being artificially idealised examples that illustrate the failings of Gauss quadrature, other problematic scenarios commonly show up in simulations. These tend to worsen with smaller core radii a, fewer Gauss nodes, higher dislocation densities near surfaces, more permissive mobility functions, and coarser FE meshes. The $\mathcal{O}(1/R)$ decay rate of $\tilde{\boldsymbol{\sigma}}$ and chaotic nature of dislocation dynamics, means these errors may result in unwarranted topological changes that cascade as the simulation advances. This is particularly deleterious when doing simulations with higher dislocation densities and/or where a large number of dislocations are close to the surface, such as nanoindentation simulations.

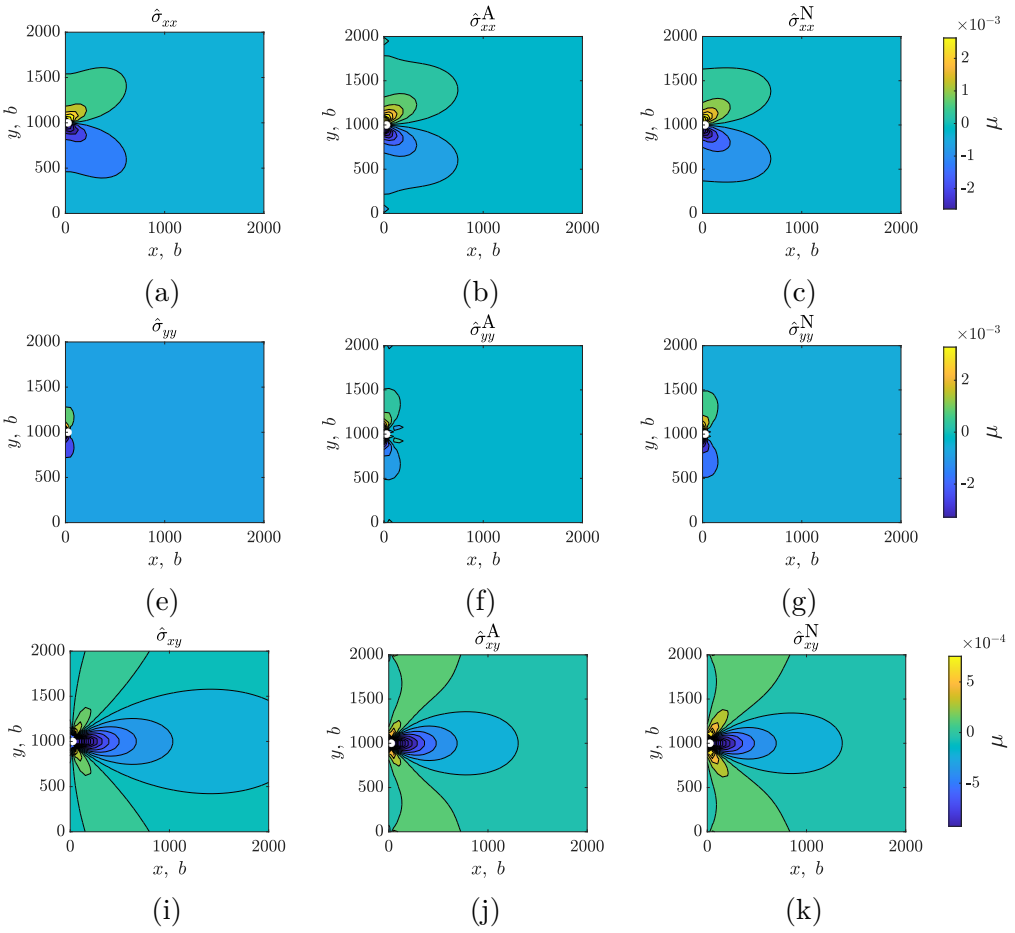


Figure 4.12: Image stresses for an edge dislocation with $\boldsymbol{l} = [001]$, $\boldsymbol{b} = [100]$, where $a = 5$, with coordinates (26, 1000), i.e. the centre of the first FE from the surface at $x = 0$, and in the centre of the simulation box along the y-direction. (a), (e) and (i) are the stress fields for the infinite-domain solution, $\hat{\boldsymbol{\sigma}}$; (b), (f) and (j) are those obtaine from analytic tractions + FEM, $\hat{\boldsymbol{\sigma}}^{\mathrm{A}}$; (c), (g) and (k) are those obtained using numeric tractions ($Q = 1$) + FEM, $\hat{\boldsymbol{\sigma}}^{\mathrm{N}}$. (a) to (c) represent the xx; (e) to (g) the yy; and (i) to (k) the xy components of the stress tensor.

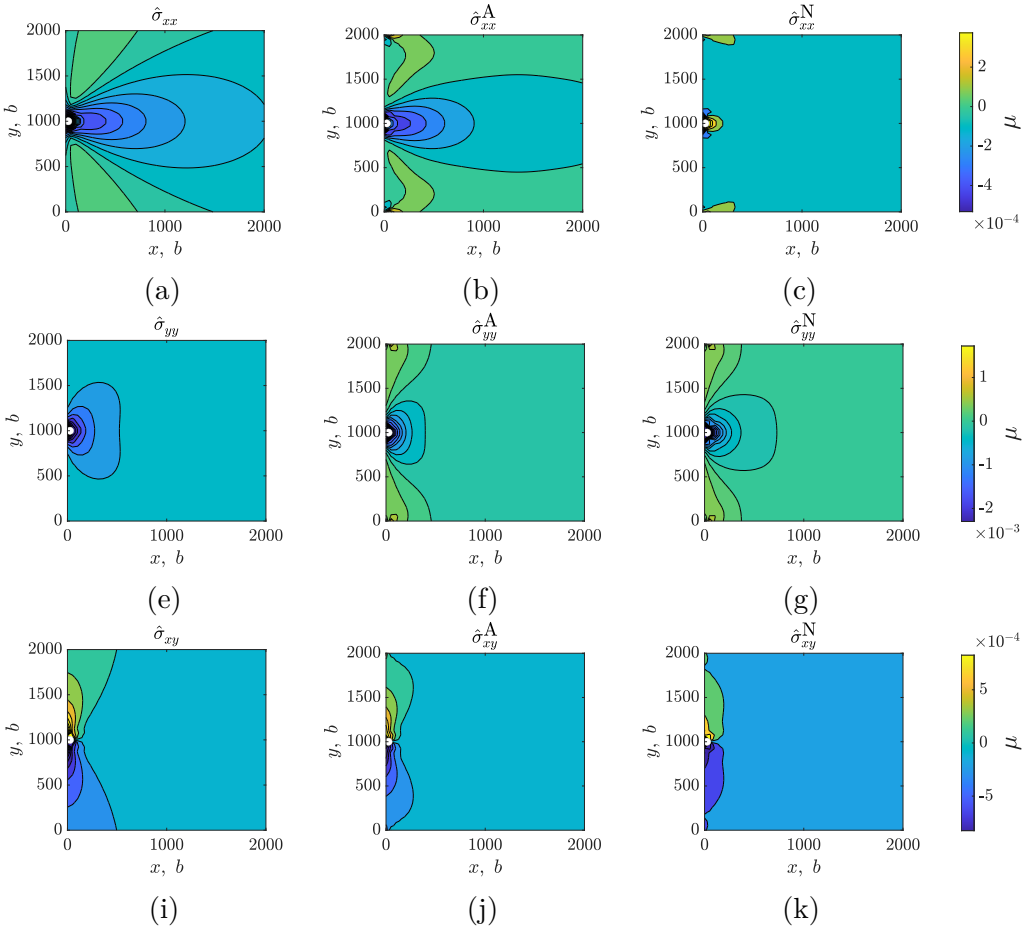


Figure 4.13: Image stresses for an edge dislocation with $\boldsymbol{l} = [0\,0\,1]$, $\boldsymbol{b} = [0\,1\,0]$, where $a = 5$, with coordinates $(26,\ 1000)$, i.e. the centre of the first FE from the surface at $x = 0$, and in the centre of the simulation box from top to bottom. (a), (e) and (i) are the stress fields for the infinite-domain solution, $\hat{\boldsymbol{\sigma}}$; (b), (f) and (j) are those obtaine from analytic tractions + FEM, $\hat{\boldsymbol{\sigma}}^{\mathrm{A}}$; (c), (g) and (k) are those obtained using numeric tractions ($Q = 1$) + FEM, $\hat{\boldsymbol{\sigma}}^{\mathrm{N}}$. (a) to (c) represent the xx; (e) to (g) the yy; and (i) to (k) the xy components of the stress tensor.

As stated in section 4.1.3, tractions are used to calculate the image stresses resulting from the boundary conditions. We therefore compare the differences in image stresses resulting from numeric ($Q = 1$) and analytic traction calculations of both our implementations and how they compare to the infinite-domain, singular expressions in eqs. (4.21) to (4.23)[1]. The stress field comparisons for all three cases are found in figs. 4.12 to 4.14, where the dislocation is denoted by a white dot.

Figure 4.12 shows the stress fields corresponding to analytic expressions for image stress components, $\hat{\sigma}_{ij}$ in eq. (4.21) where no superscript denotes the infinite-domain singular expressions, the A superscript are the stresses calculated from

[1]Since the first term in each equation corresponds to the real stresses, we omit them to view the image stresses.

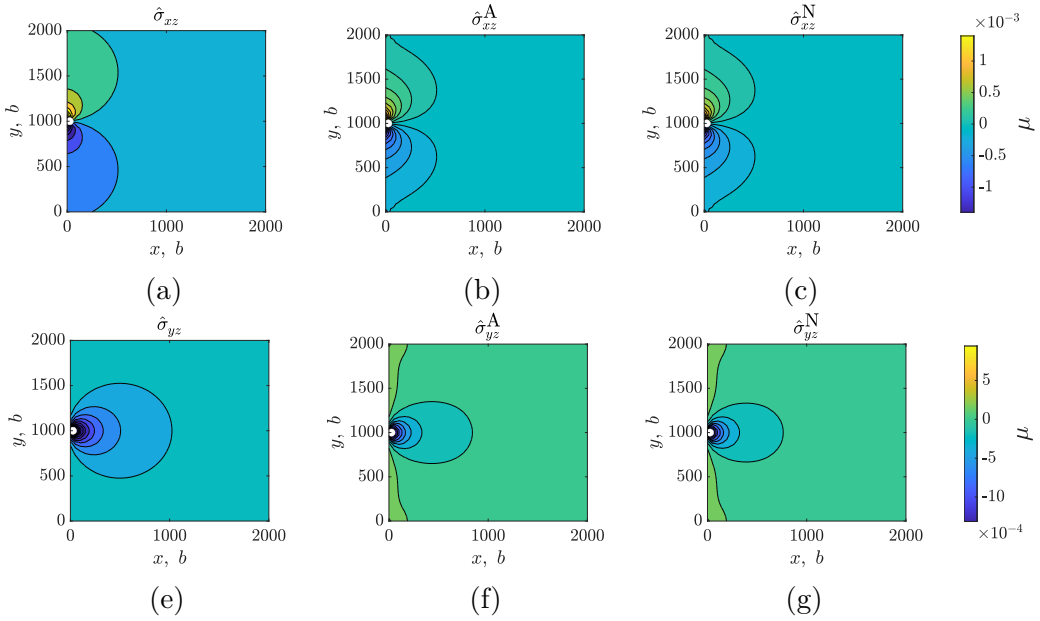


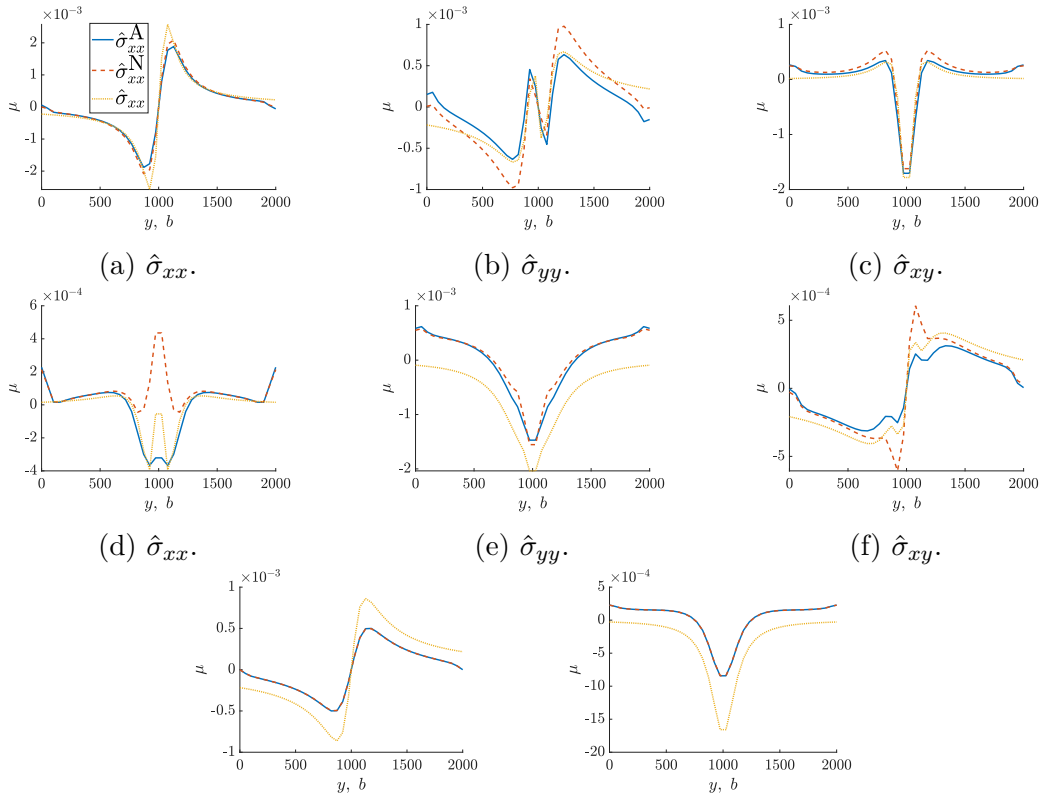
Figure 4.14: Image stresses for a screw dislocation with $\boldsymbol{l} = [0\,0\,1]$, $\boldsymbol{b} = [0\,0\,1]$, where $a = 5$, with coordinates $(26,\ 1000)$, i.e. the centre of the first FE from the surface at $x = 0$, and in the centre of the simulation box from top to bottom. (a) and (e) are the stress fields for the infinite-domain solution, $\hat{\boldsymbol{\sigma}}$; (b) and (f) are those obtaine from analytic tractions + FEM, $\hat{\boldsymbol{\sigma}}^{\mathrm{A}}$; (c) and (g) are those obtained using numeric tractions ($Q = 1$) + FEM, $\hat{\boldsymbol{\sigma}}^{\mathrm{N}}$. (a) to (c) represent the xz; (e) to (g) the yz.

analytic tractions and the N superscript are those coming from numeric tractions where $Q = 1$ (same nomenclature in figs. 4.13 and 4.14). The setup corresponds to the one described in fig. 4.8 where $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}1} = [1\,0\,0]$ and the dislocation is found at $(26, 1000)$.

It is clear edge effects play a role in the generated stresses. All three components have notable differences from the infinite-domain solutions resulting from the finite constraints, but the overall agreement between them all is quite good.

Things get markedly more interesting when looking at $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}2} = [0\,1\,0]$ in fig. 4.13 for a dislocation in the same place, $(26, 1000)$. Of particular note is $\hat{\sigma}_{xx}$, where a comparison between figs. 4.13a and 4.13b and fig. 4.13c reveals one of the major issues with numeric tractions. If we look at the neighbourhood of the dislocation (just to the right), we will find a sign inversion i.e. yellow and green contours as opposed to purple and blue. As well as a drastically different isosurface shape. Image stresses like those can lead dislocations to behave quite differently than they should, particularly if the sign inversion also has a significantly different magnitude than the correct solution. Specifically, there is a region in the positive x-direction away from the dislocation where $\hat{\sigma}_{xx}$ is tensile rather than compressive. It is as absurd as a ship that floats by lowering the water level.

Perhaps the most evident deformation resulting from the displacement bound-



(a) $\hat{\sigma}_{xx}$. (b) $\hat{\sigma}_{yy}$. (c) $\hat{\sigma}_{xy}$.

(d) $\hat{\sigma}_{xx}$. (e) $\hat{\sigma}_{yy}$. (f) $\hat{\sigma}_{xy}$.

(g) $\hat{\sigma}_{xz}$. (h) $\hat{\sigma}_{yz}$.

Figure 4.15: Line plots of the image stresses of a dislocation at $(26, 1000)$ for a line going through $x = 103$ (start of the third element) for the analytic image stresses, as well as those calculated with numeric and analytic tractions. (a) to (c) correspond to $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}1}$, (d) to (f) $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}2}$ and (g) to (h) to $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{s}}$.

aries can be seen when $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{s}}$ in fig. 4.14, where the lobes of the isolines are highly deformed when compared to the infinite-domain solutions. Though deformed, their familiar shape is still recognisable and both analytic and numeric tractions yield fairly similar fields.

From figs. 4.12 to 4.14 it seems like both analytic and numeric tractions are appropriate in most cases. At least at these scales, there is only one instance where numeric tractions yield very incorrect results. That said, these can have a significant impact on simulations, particularly those where multiple dislocations interact with surfaces in close proximity with one another.

We also produced line plots to better show how the stress fields deviate from one another. Figure 4.15 shows line plots through the line $x = 103$ for a dislocation at $(26, 1000)$. Figures 4.15a to 4.15c correspond to $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}1}$, figs. 4.15d to 4.15f to $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}2}$ and figs. 4.15g and 4.15h to $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{s}}$. Here, the singular nature of the infinite-domain solutions is evidenced by the sharp spikes in its stresses. In general, the non-singular formulation smoothes out the stress line plots. However, there are a few instances where the numeric tractions lead to larger spikes than

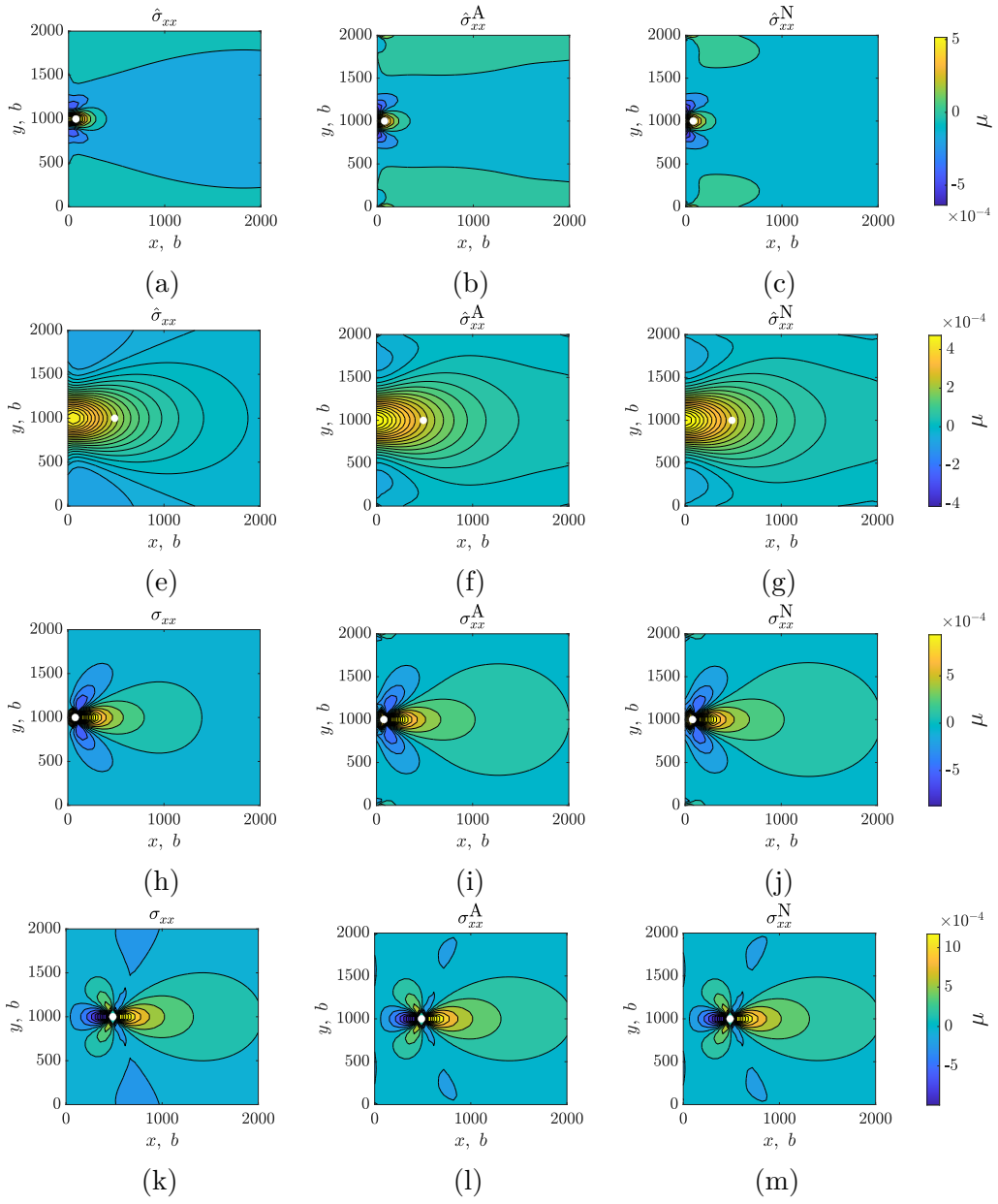
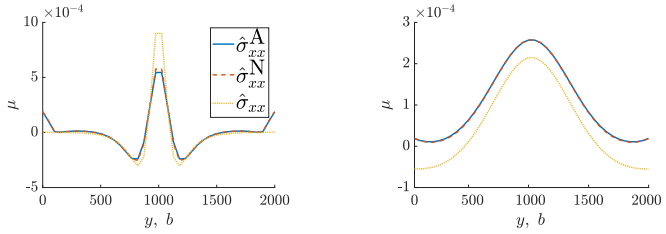
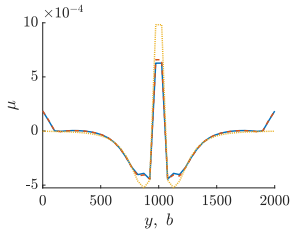


Figure 4.16: Stresses for an edge dislocation with $\boldsymbol{l} = [0\,0\,1]$, $\boldsymbol{b} = [0\,1\,0]$. Subfigures (a) to (g) show image stresses; (h) to (m) show total stresses. In subfigures (a) to (c) and (h) to (j) the dislocation is found at (77, 1000); in (e) to (g) and (k) to (m) the dislocation is at (487, 1000).

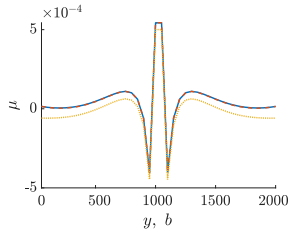


(a) $\hat{\sigma}_{xx}$.

(b) $\hat{\sigma}_{xx}$.



(c) σ_{xx}.



(d) σ_{xx}.

Figure 4.17: Line plots corresponding to fig. 4.16. (a) and (c) are of the image stresses; (b) and (d) are of total stresses. (a) and (b) are of a dislocation at (77, 1000), taken along the line $x = 103$. (b) and (d) are of a dislocation at (487, 1000), taken along the line $x = 513$.

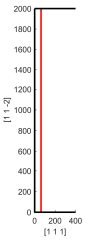
even the infinite-domain solutions such as in fig. 4.15d. Again, the general shape of the line plots is the same but the tendancy for numerical tractions to spike under specific circumstances is evident in almost every case.

To show the convergence in methods we also show stress fields in fig. 4.17 and line plots fig. 4.17 of the image and total stress fields as an edge dislocation with $\boldsymbol{b} = \boldsymbol{b}_{e2}$ moves from (77, 1000) to (487, 1000). The line plots are taken from the second closest set of nodes in the positive x-direction i.e. at $x = 103, 513$.

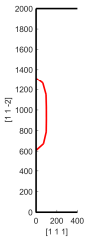
Notice that the image stresses in figs. 4.16a to 4.16c are all very close to fig. 4.13c, which is the stress field calculated via numerical tractions for a dislocation that is slightly closer to the surface. Essentially, the numerical tractions over or underestimated a set of forces something that become the dominant contributors as the dislocation moves away from the surface.

From figs. 4.16 and 4.17 it can be observed that both analytic and numeric tractions converge to similar shapes to each other as expected. However it also becomes clear that the infinite-domain solution is not totally accurate for image stresses in finite domains due to edge effects.

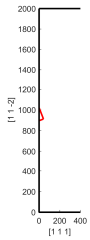
Lastly, to show how much numeric tractions can affect a result, fig. 4.18 shows a few snapshots of the simulation described in section 4.1.3. The figures shown are not at equivalent times because the simulations using numeric tractions ran indefenitely. The line tension equilibrated with the image forces on the traction-free surface, preventing the dislocation from exiting and keeping it oscillating in a local energy minimum. The simulation using analytic tractions ended when the



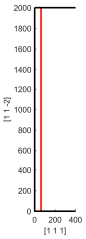
(a) Analytic tractions $t = 0$.



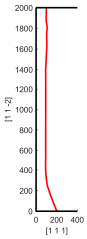
(b) Analytic tractions mid simulation.



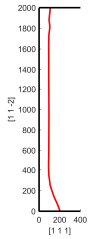
(c) Analytic tractions end.



(d) Numeric tractions $t = 0$.



(e) Numeric tractions mid simulation.



(f) Numeric tractions $t \to \infty$.

Figure 4.18: Progression of simulations using both traction calculation methods.

dislocation completely exited the surface as expected. It is feasable that at some point, the simulation using numeric tractions could jump out of the local minimum due to some spike in the tractions, but the last image was taken for a simulation time approximately 20 times greater than it took the one using analytic tractions to end when the dislocation fully exited the box.

4.1.5 Conclusions

Often the effects of tractions on a simulation can be quite subtle if things do not go catastrophically wrong, but a quick and easy way of seeing how the numeric tractions have a tendancy to spike and invert sign (as in fig. 4.15d) is to pause a simulation while its running and scatter plot the numeric tractions v.s. analytic ones. The plot will show a positive correlation—as the sign inversions in numeric tractions are relatively rare events—but whichever axis corresponds to the numeric tractions will have the largest range. Using proportional axes makes the differences quite evident even at a quick glance as the aspect ratio will be far from 1:1.

In other parts of our work, we have noted that despite the analytic tractions taking approximately 10 times longer to compute than numerically calculated tractions with $Q = 1$ (which agrees with the findings in [75]), yield more stable and ultimately faster simulations. Even simple simulations are typically faster when using analytic tractions over numeric ones. This was not the case with the simulation we show here, but the numeric tractions led to a hung simulation, which we have found is quite common. The margin by which using analytic tractions

leads to faster overall simulations grows as simulation complexity increases. Only the simplest simulations initially benefit from numeric tractions but this often decreases and reverses as the simulation advances. Another fortunate side effect of more accurate tractions is the fact that fewer dislocation segments are generated during simulations, which Bromage and Tarleton [45] found when correctly accounting for the displacements generated by dislocations.

Given our findings, we cannot recommend using numeric tractions when analytic ones are available. The losses in computational speed that result from moving from one to the other are more than made up for by the fewer topological operations, fewer generated segments, and more accurate velocities. All of which result in fewer calculations of segment-segment interactions, fewer collisions, larger timesteps and ultimately cleaner simulations that run into fewer snags along the way.

There is a case however, for combining both approaches in larger scale simulations. Since both numeric and analytic tractions converge to the same value as the dislocation-surface distance increases, a hybrid approach may possibly be undertaken without many negatives. Furthermore, [75] also derived a Taylor series expansion of the solutions which can be used in cases where the dislocation-surface distance is large. These may be worth exploring in larger scale simulatios. However, in the types of systems we model, other parts of our model tend to be much more rate-limiting than tractions. As such, we have all but ceased to use numeric tractions in our work.

4.2 GPU parallelisation of analytic tractions

4.2.1 Introduction

Graphics Processing Units (GPUs) have been leveraged by digital art, animation studios and gaming companies since the 1990s to offload repetitive, grid-based mathematical computations away from the Central Processing Units (CPUs). These operations are usually very computationally cheap, require minimal logic, and do not need more than single precision (32-bit) arithmetic [80–82], in fact, increases in accuracy would get lost in the RGB channel regardless. As such, GPUs evolved primarily to efficiently perform lightweight operations on vast amounts of 32-bit data. As such, their memory buses are extremely fast and and their processors very minimalistic.

It wasn't until much later that engineers and scientists picked up on the potential of GPUs [83–85]. Fields where lower precision is advantageous—because it increases the signal to noise ratio, or real-world tolerances exceed the in-silico

precision—such as data science, civil engineering and image/signal processing, first leveraged this technology. In fact, many big data applications operate on half and some even go as low as quarter precision simply due to the low sensitivity, large amount of noise and vast quantities of data [86].

NVidia has spearheaded the development and adoption of scientific-computing and high performance computing (HPC) GPUs with the development of their propriatory Compute Unified Device Architecture, CUDA technology. They now produce a wide range of GPUs tailored for scientific applications. These have specialised architecture [87].

- Tensor modules for fast, piecewise matrix multiplication.
- Larger but slower memory buses mean higher concurrent resource use because scientific computations are expensive, so it is better to have them work through a lot of data while more memory is being fetched, rather than fecthing small amounts of memory really quickly but having to wait for the processors to finish their computation.
- Expanded precision capabilities.
- Specialised registers, specialised heirarchical memory architecture, etc.

For all their advantages, parallel algorithms are fundamentally different to serial ones. In particular the way GPUs differ from CPUs makes it a non-trivial task to properly parallelise an algorithm. Doing so poorly can result in decreased performance, particularly when using high performace/scientific computing cards. There is also the issue of scaling, whereby the cost of utilising the GPU may not be worth paying until a certain computation size. In this section, we detail the work, results and conclusions of parallelising the analytic traction calculation.

4.2.2 Methodology

4.2.2.1 Data mapping

A cache is a store of memory used to reduce the cost of accessing data from the main memory. CPUs and GPUs have cache heirarchies designed to progressively get smaller and faster the closer they are to the processor. This reduces the average cost of accessing data because the missing data can be looked for in progressively higher cache levels. It is important to know that caches operate as single units, meaning that every time a processor needs data that is not found in a cache (cache miss), the whole cache needs to be updated. This means that every time there is a cache miss, any operations dependent on the missing data have to wait until the

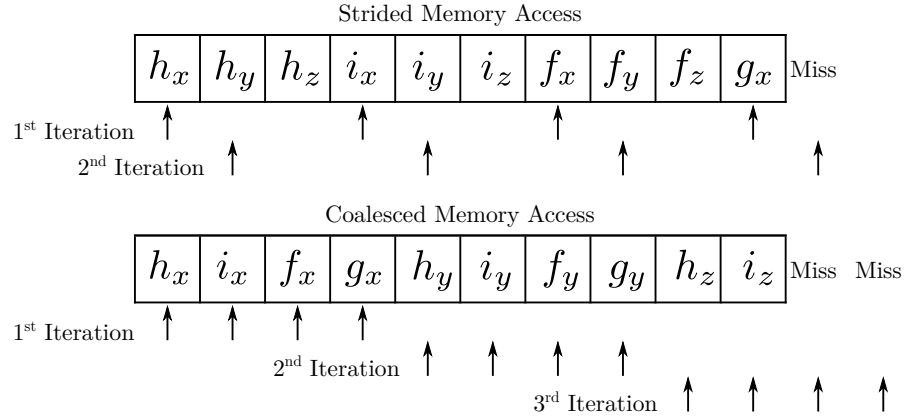


Figure 4.19: Memory access patterns. Arrows are threads in a warp simultaneously requesting access to memory from the cache. All processors work simultaneously, so they can only proceed until every one of them has the necessary data.

new data arrives. As processors have gotten faster, memory speeds have not kept up. As a result, minimising the ratio of cache misses to cache hits has become an important aspect of optimising software applications [88, 89]. Optimising cache use is of particular importance in GPUs.

The increasingly divergent speeds of processors and memory, are the reason why scientific computing GPUs opt for larger but slower memory buses. In properly written software, this feature decreses the time spent waiting for a processor to finish whilst the data is ready [90–92]. Tnstead the infrastructure and cost associated with faster memory can be better utilised on more processing power. However, in improperly written code, the cache miss to hit ratio will be large. Non-scientific applications can get away with this to an extent, because memory buses are fast and computational cost is low. But for scientific applications—and in particular when using specialised GPUs—this can be catastrophic for performance.

In NVidia GPUs, "coalesced memory access" is when, a single cache line contains all the data needed by a "warp"—a collection of 32 threads which operate "simultaneously" as a single unit on separate pieces of data—to carry out an instruction/process. This is not always possible because it is problem-dependent, so usually the best we can do is ensure there are no unecessary memory fetches by arranging simultaneously-used data in a contiguous manner. Figure 4.19 is a simple example of this.

Since threads in a warp behave as a single unit, they hang until every other thread in the warp has the data it needs to execute the next instruction. Under a strided memory access pattern, this happens more often than if the data were contiguously accessed. Furthermore, cache usage optimisation heuristics have the task of deciding what memory to bring in order to minimise future cache misses.

Under a contiguous memory access model, this done by simply placing the first missing item as the first item in the next cache line. However, even in the simple example of strided memory access shown in fig. 4.19, it's easy to see how this is suboptimal. To start with, a new cache line is needed for the for the second iteration instead of the third, as it would be if the access pattern were contiguous. Then, the new cache line could start with h_y and still work for the second iteration, but then a new cache line would be needed for the third. However, grabbing the first missing item as the first item of the cache line used in the second iteration means going back down for the third iteration. In fact, the optimum strategy in this case would be to start the cache line for the second iteration on the third item, h_z, in the cache line for the first iteration.

As a result, we have to be extremely careful when mapping CPU (host) memory, arranged to be advantageous for serial access (and therefore human intuition), to device (GPU) memory, arranged for parallel access.

4.2.2.2 Parallelisation schemes

The traction calculation is $\mathcal{O}(MN)$, where M is the number of surface elements and N the number of segments. As such, there are two first order parallelisation strategies. Parallelising over elements and looping over segments; parallelising over segments and looping over elements. And two second order parallelisation strategies. Parallelising over elements and subparallelising over segments and vice-versa. However, higher order parallelisations are usually disadvantageous unless the problem is very computationally cheap because there is more competition for parallel resource use. We only implemented both first order parallelisations. There cannot be a parallelisation over nodes as the unit problem is a single surface element and dislocation segment.

Both first order parallelisations have similar performances, but parallelising over dislocations have better scaling as simulations advance because the number of surface elements is constant while the number of dislocation segments tends to increase. The choice of parallelisation will depend on the relative amounts of surface elements to dislocation segments.

The algorithm for mapping from the objects to be parallelised over from host (CPU) to device (GPU) memory is the same in either case. The only thing that changes is that surface elements have 4 nodes and dislocation segments have 2. If parallelising over dislocation segments, the Burgers vector also needs to be mapped in the same manner, but it is equivalent to having a single 'node'. The mapping

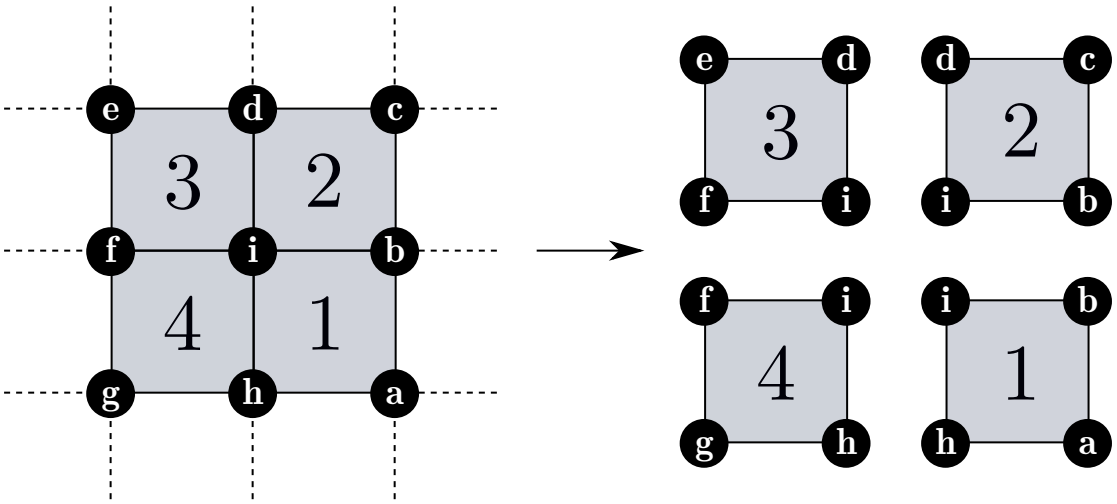


Figure 4.20: Each linear rectangular surface element is mapped to one thread.

is as follows,

$$\boldsymbol{X}_{en} \coloneqq [x_{en},\, y_{en},\, z_{en}] \tag{4.28a}$$

$$\boldsymbol{X}_{(1\to E)n} \mapsto \boldsymbol{X}_n$$

$$\boldsymbol{X}_n \coloneqq [x_{1n},\, y_{1n},\, z_{1n}, \ldots, x_{En},\, y_{En},\, z_{En}] \tag{4.28b}$$

$$\boldsymbol{X}_{1\to N} \mapsto \boldsymbol{X}$$

$$\boldsymbol{X} \coloneqq \begin{matrix} [x_{11},\, \ldots,\, x_{E1},\, x_{12},\, \ldots,\, x_{E2},\, \ldots,\, x_{1N},\, \ldots,\, x_{EN}, \\ y_{11},\, \ldots,\, y_{E1},\, y_{12},\, \ldots,\, y_{E2},\, \ldots,\, y_{1N},\, \ldots,\, y_{EN}, \\ z_{11},\, \ldots,\, z_{E1},\, z_{12},\, \ldots,\, z_{E2},\, \ldots,\, z_{1N},\, \ldots,\, z_{EN}] \,. \end{matrix} \tag{4.28c}$$

where e is the element to be mapped, n the node number, E the total number of elements and N the total number of nodes per element.

For a set up like fig. 4.20, parallelising over the surface elements with node labels corresponding to fig. 4.3, $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $\boldsymbol{x_3}$ and $\boldsymbol{x_4}$ yields,

$$\boldsymbol{X} = \begin{bmatrix} \underbrace{h_x,\, i_x,\, f_x,\, g_x}_{\boldsymbol{x_1}},\, \underbrace{a_x,\, b_x,\, i_x,\, h_x}_{\boldsymbol{x_2}},\, \underbrace{i_x,\, d_x,\, e_x,\, f_x}_{\boldsymbol{x_3}},\, \underbrace{b_x,\, c_x,\, d_x,\, i_x}_{\boldsymbol{x_4}}, \\ \ldots\ y\text{-coord}\ \ldots, \\ \ldots\ z\text{-coord}\ \ldots \end{bmatrix}. \tag{4.29}$$

Whether the parallelisation is done over dislocation line segments or surface elements, the item that wasn't parallelised over must also be mapped such that it can be contiguously accessed by the GPU. However this just means merging

arrays in a very simple manner,

$$\boldsymbol{X}_{en} \coloneqq [x_{en},\, y_{en},\, z_{en}] \tag{4.30a}$$

$$\boldsymbol{X}_{(1\to E)n} \mapsto \boldsymbol{X}_n$$

$$\boldsymbol{X}_n \coloneqq [x_{1n}, \ldots,\, x_{En},\, y_{1n}, \ldots,\, y_{En},\, z_{1n}, \ldots,\, z_{En}] \tag{4.30b}$$

$$\boldsymbol{X}_{1\to N} \mapsto \boldsymbol{X}$$

$$\boldsymbol{X} \coloneqq \begin{matrix} [x_{11}, \ldots,\, x_{E1},\, y_{11},\, \ldots,\, y_{E1},\, z_{11}, \ldots,\, z_{E1} \\ , \ldots, \\ x_{1M}, \ldots,\, x_{EN},\, y_{1N}, \ldots,\, y_{EN},\, z_{1N}, \ldots,\, z_{EN}] \end{matrix} \tag{4.30c}$$

In a similar vein to eq. (4.29), using eq. (4.30) to map the data yields,

$$\boldsymbol{X} = \begin{matrix} \Big[\underbrace{h_x,\, i_x,\, f_x,\, g_x,\, h_y,\, i_y,\, f_y,\, g_y,\, h_z,\, i_z,\, f_z,\, g_z}_{\mathbf{x_0}}\,, \\ \ldots\ \mathbf{x_1},\, xyz\text{-coords}\ \ldots, \\ \ldots\ \mathbf{x_2},\, xyz\text{-coords}\ \ldots, \\ \ldots\ \mathbf{x_3},\, xyz\text{-coords}\ \ldots\Big]\,. \end{matrix} \tag{4.31}$$

4.2.2.3 Maximising performance

In order to maximise performance after properly mapping memory to the GPU we have to minimise hang time. A good rule of thumb is to try optimising cache use. Full cache line utilisation (coalesced memory access) is achieved if cache lines can accomodate l entries given by eq. (4.32),

$$l = \begin{cases} a \times N \times E\,, & a > 0 \in \mathbb{N} \\ \text{or} & \\ \dfrac{1}{2^a} \times N \times E\,, & a \geq 0 \in \mathbb{N},\, N \times E \equiv 0 \mod 2^a\,, \end{cases} \tag{4.32}$$

where a is the number of entries, N the number of nodes and E the number of elements. This can be achieved by making use of the flexibility afforded by the NVidida architecture. Every GPU has various collections of warps called Streaming Microprocessors (SMs). Each SM is independent of other SMs, however, warps within an SM all share certain resources such as various levels of caches and schedulers. Each SM can be split into blocks of threads, all of which share some resources. The optimal usage entails finding the best block size (preferably a multiple of 32 to maximise thread usage, as the smallest unit of processing is a warp) for a particular problem. Each block as a dimension (the number of threads

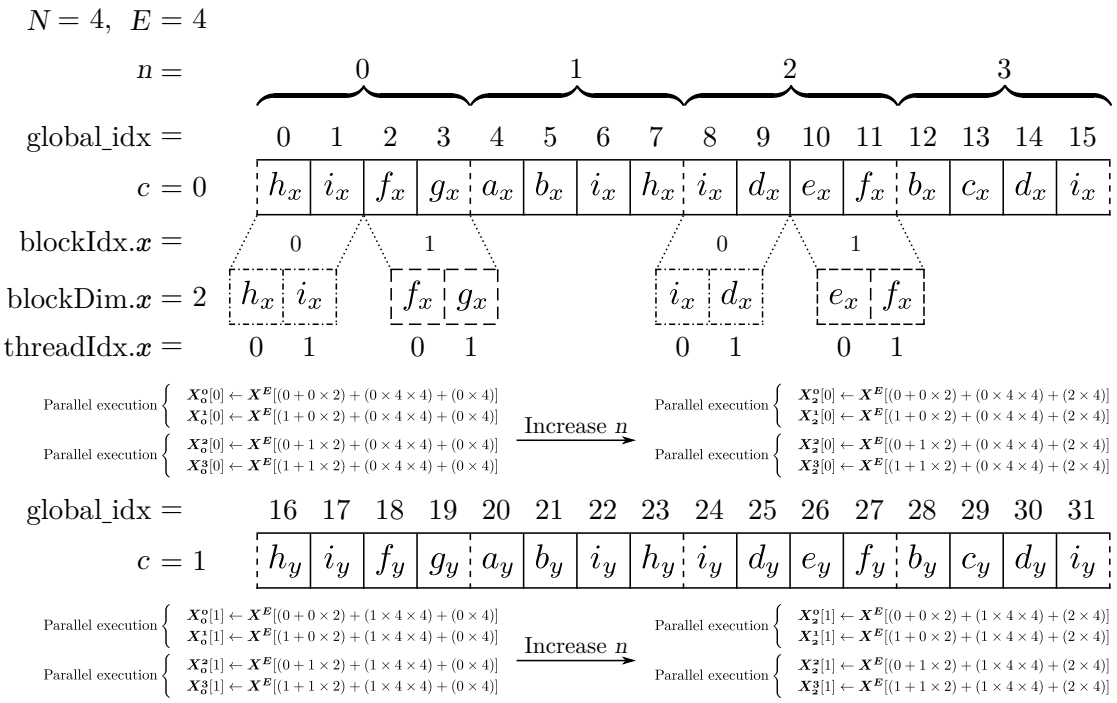


Figure 4.21: Minimum working complete example of a thread block obtaining data from global memory, from parallelising over the four surface elements in fig. 4.20 and mapped by eq. (4.28). $\boldsymbol{X}_{\boldsymbol{a}}^{\boldsymbol{b}}$ denotes a 1D array of length three containing the xyz-coordinates of node a of element b. Each thread concerns itself with only one element at a time. The dash-dot and dashed boxes represent cache lines for a thread block, the dotted line represents a memory fetch, and the dashed lines in $\boldsymbol{X}^{\boldsymbol{E}}$ represent steps of E entries (the start of the data for the next node of the element type we're dealing with). The memory operations are not shown twice to minimuse redundancy.

inside it), an index (there can be multiple blocks in a single SM), and each thread has an index that identifies it within its block. These values are used to index the global device memory. Figure 4.21 has an example of how this works for a block of two threads and a set up as described by fig. 4.20 and mapping by eq. (4.28).

There are 6*3*8 bytes of information in each unit problem (6 nodes with 3 coordinates each, all double precision numbers that take up 8 bytes each), so we must consider the maximum shared memory in a GPU.

Realistically, this is extremely difficult to achieve, particularly in dynamic, non-conservative simulations such as discrete dislocation dynamics. So we also came up with two heuristics that aim to utilise as many computational resources as possible.

$$T = 32 \left\lceil (n \mod T_{\max})/32 \right\rceil \tag{4.33}$$

$$G = \left\lceil (n + T - 1)/T \right\rceil \ , \tag{4.34}$$

where n is the number of items to be parallelised (either surface elements or

dislocation line segments), T is the number of threads per block, $T_{\max}$ is the maximum number of threads allowed in a block (GPU dependent) and G is the total number of blocks. G ensures the load gets distributed evenly across as many blocks as possible, and T ensures each thread block is busy for the longest time possible between memory fetches.

4.2.2.4 Solving parallelisation problems

One of the things that can be missed during parallelisation is the fact that writing to global memory is asyncronous. This means that threads can race each other to write to global memory. Fortunately we avoided some of the more egregious issues because each unit problem is independent of the rest. However, the forces calculated independently accross threads have to be sequentially added to the overall force on each node. This is done with atomic operations, that force the threads to sequentially write their contributions to the global memory.

The second issue is that of the singularity when a dislocation segment is parallel to a surface, see eq. (4.18).

The reason this is such an issue is because GPUs cannot branch. That is to say, every case of an if or case statement is executed regardless of whether a condition is met or not. The only difference is that the values are only stored if a condition is met. This means that cases as rare yet costly as these, disproportionatley impact performance. Parallelising over special cases is not worth the time because they are not common enough, so the best solution use the fact that GPUs and CPUs work asyncronously and have the GPU only store the data for non-special cases, while the CPU only calculated the contributions from the special cases. Both contributions are added back at the end.

4.2.3 Results, discussion and conclusions

One of the greatest disapointments of this project can be seen in fig. 4.22. It was produced using a linux workstation with an NVidia GTX 750 and an Intel Core i5-8500 @ 3.0 GHz. Both GPU and CPU are equally outdated so the comparison remains relatively fair insofar as other non-HPC set ups are concerned. That said, using a high performance GPU would help a great deal as CPUs have somewhat stagnated in comparison. Regardless it wouldn't help the root cause. The x-axis is scaled to 10^5 dislocation segments. We have nowhere near the capacity to run a simulation with 10^5 segments. The $(O)(N^2)$ collision and segment-segment force algorithms and $(O)(N!)$ separation prevent us from going much past 10^4 segments. Moreover, fig. 4.22 was obtained using only non-parallel segments, the code checking for parallel ones in the CPU was not even compiled. Therefore,

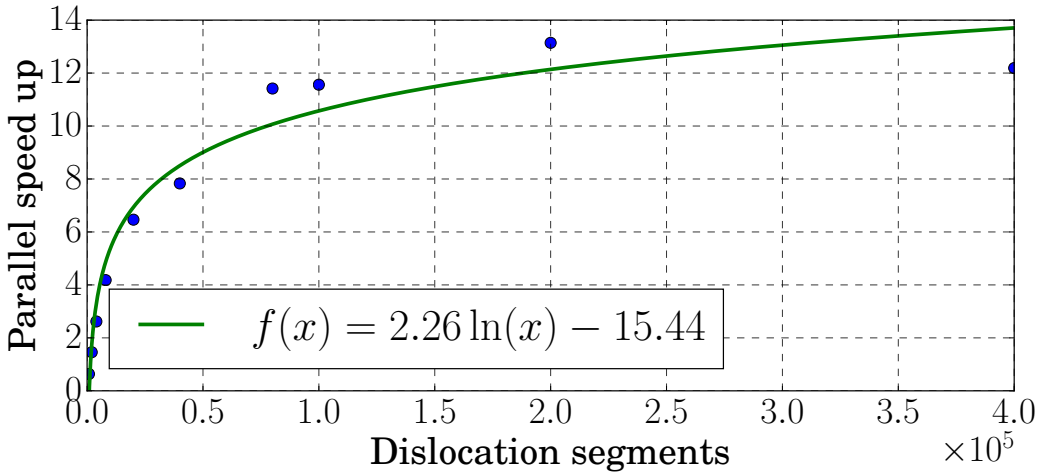


Figure 4.22: Parallel speedup on an NVidia GTX 750 and an Intel Core i5-8500 @ 3.0 GHz.

the speed up is purely from the GPU. Unfortunately, checking for the special case means the CPU can lag behind the GPU in some scenarios, reducing the true gains by a noticeable margin.

While using the GPU-accelerated code does not actively hinder most simulations, it doesn't help much either. Larger simulations on more advanced hadware are sped up, but other processes slow them down before the computational gains can be realised.

The analytic traction calculation is particularly troublesome for parallelisation. Not only is it much more computationally intensive than what GPUs are normally used for, but also has disproportionately expensive and rare corner cases, and has an awkward unit problem that necesitates a lot of data. It is too far removed from the relatively simpler layouts of "trivially" parallelisable grid-based problems found in fluid dynamics, civil engineering and image processing that get so much out of GPU parallelisation.

The silver lining to this is that the parallelised code is close to optimal. So when the rest of the software and hardware catch up to the complexity presented by the analytic tractions, it'll be there to shine like it deserves.

Chapter 5

Simulations

In this chapter we present three different simulations. Two of which have an accompaning experimental data for micro tensile loading in the [1 0 0] and [1 1 0] directions of a single crystal Zn micropillar provided by Alan Xu from ANSTO Sidney. The third is is a proof of concept for a techinque developed by Jicheng Gong to cyclicly load a cantilever in opposing directions. However, this technique was developed on Ti microcantilevers and the HCP mobility law requires some adaptation before it is ready for general use. Instead we use the BCC law developed in-house by Bruce Bromage that as mentioned in the footnotes of section 2.1.2.

5.1 Nickel tensile micropillar

10 dislocations per square micron.

Square cross-section.

5.1.1 Introduction

5.1.2 Methodology

We define surface node sets $\left\{\forall(x, y, z) \in [0,\, 1] | S_{xyz} \in \partial\hat{V}\right\}$, where $\hat{V}$ is a unit volume such that S_{000} denotes the node at the origin, S_{x00} the x-axis spanning edge at $y,\, z = 0$, and S_{xy0} the xy-plane at $z = 0$. We use these node sets to define our neuman (displacement) boundary conditions as follows.

$$S_{0yz},\, S_{0y0},\, S_{0y1},\, S_{00z},\, S_{01z},\, S_{000},\, S_{001},\, S_{010},\, S_{011} \leftarrow u_x = 0 \tag{5.1a}$$

$$S_{01z},\, S_{010},\, S_{011} \leftarrow u_y = 0 \tag{5.1b}$$

$$S_{0y0},\, S_{010},\, S_{000} \leftarrow u_z = 0 \tag{5.1c}$$

$$S_{1yz},\, S_{1y0},\, S_{1y1},\, S_{10z},\, S_{11z},\, S_{100},\, S_{101},\, S_{110},\, S_{111} \leftarrow u_x = U \neq 0\,. \tag{5.1d}$$

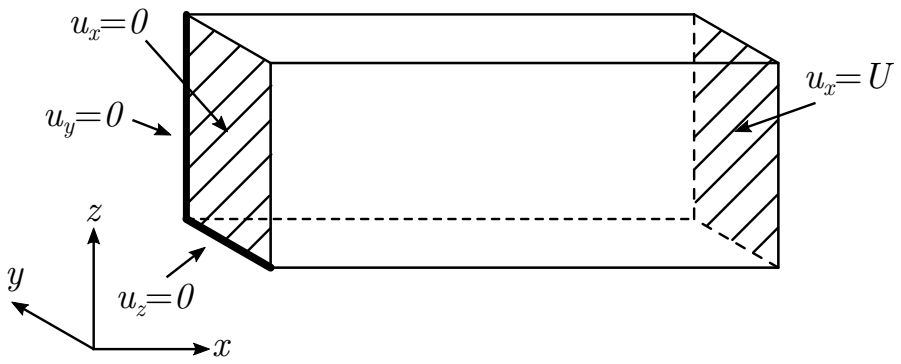


Figure 5.1: Displacement boundary conditions for dislocation plasticity modelling of single crystal micro-tensile tests.

In simple terms, the whole yz-plane at $x = 0$ (including corner nodes and edges) is fixed in x; the whole y-edge at x, $z = 0$ (including corner nodes) is also fixed in z; the whole z-edge at $x = 0$ $y = 1$ (including corner nodes) is fixed in y; and the nodes at $x = 1$ have a displacement U applied in the x-direction. All other degrees of freedom are free to move as necessary. Once mapped to our simulated cuboid geometry, it looks like fig. 5.1.

In EasyDD, time is defined in units of shear modulus eq. (5.2),

$$t_{\text{real}} = t_{\text{sim}} \frac{B}{\|\mu\|}\,, \tag{5.2}$$

where $B \coloneqq 1 \times 10^{-4}$[Pa][s] is the dislocation mobility and $\|\mu\| \coloneqq$ [Pa] is the magnitude of the shear modulus (the shear modulus we use for the simulations is normalised to 1, its magnitude is used to scale parameters). The dislocation mobilities are normalised such that edge dislocations have mobility 1, other mobilities are defined from these. So the time conversion to real time is a matter of dividing the simulation time by the shear modulus.

The experimental loading rate provided was 5 nm/s, which when converted to simulation time, gives a loading rate that is far too low for the timescales we can model. So we defined $\Delta t_0 \equiv 5 \times 10^{-9}\|\mu\|$ and found the maximum loading rate of the form $\dot{u} \equiv a\dfrac{\Delta l}{\Delta t_0}$, for which the quasi-static condition held true, where Δl is the length of the beam in the loading direction and a is a constant.

Chapter 6

Future work

During the course of this project, it became increasingly clear there are some fundamental issues preventing us from exploring the full breadth of what discrete dislocation dynamics has to offer. DDLab [42], the code that EasyDD is based on, made some design decisions that ultimately limit its potential. Namely, the lack of attention paid to the cost of dynamic memory allocation, procedural nature of the code, and choice of programming language. These choices were reasonable upon inception, but as the Tarleton group has increased in size and maturity, it has become evident that it is not enough. We are reaching the upper limits of what can be done with `MATLAB`. There are more things we can do, more things we are doing. But a constant obstacle in obtaining results is how long they take to get and how difficult it is to add new functionality, even changing boundary conditions presents a non-trivial obstacle that requires intimate knowledge of how the finite element mesh is generated.

The work described in chapters 2 to 4 has done much in not only producing more accurate simulations, but doing so faster. Fengxian Liu's work on modelling climb, diffusion and inclusions would greatly benefit from better designed code. Haiyang Yu's work on modelling hydrogen [66] and nanoindentation would also stand to benefit a great deal from a parallelised and faster dislocation separation algorithm. Potential work relevant to modelling dislocation dynamics in nuclear reactors such as post-damage cascade dislocation dynamics [93], necessitates spontaneous generation of dislocations as a network evolves. Including stochasticity in the form of Langevin dynamics [94] is also highly relevant, especially at higher temperatures.

Many of these require a complete overhaul to some of the most fundamental parts of EasyDD. Even so, its potential would be hindered by the fact that `MATLAB` is a scripting language not designed for scientific computing. It lacks performance, flexibility and many of the modern features that make life easy when dealing with large codebases. Furthermore, the need for licences can represent a prohibitative

barrier for users and researchers. Not only this, but the cost of distributed licences that allow `MATLAB` to run on clusters severely limits scalability and potential for collaboration—something that at one point, inhibited this project.

But there is a better way. Prompted by the public release of the Imperial College COVID-19 source code `https://github.com/mrc-ide/covid-sim`, and the recurring issues with computational efficiency, compiler compatibility and availability, and fundamental technical debt inherited from `DDLab` [42]. We started a weekend project that has turned out to be extremely promising.

It is our belief that we are currently in a renaissance of programming languages. There are so many exciting modern languages that build and improve upon the lessons of the past. The most promising for scientific computing is widely regarded to be `Julia` [95]. An paradigm-shifting, open-source JIT (just in time) compiled language, explicitly designed for scientific computing. It has a number of features that make it an excellent candidate for a new generation of scientific software.

1. Multiple dispatch: methods are dispatched and compiled based on the types of their arguments. Types propagate their way through child functions. This lets the compiler generate optimal code for the platform being used.

2. Type system: its type system is based on set theory, where types are arranged in a genealogical tree with broader types giving way to narrower ones, e.g. `Float64 <: AbstractFloat <: Real <: Number <: Any == true`. Structures can be parametric on type. These features let developers create generic code without bothering with type annotations, and the behaviour will be appropriate for whatever types are used. This kind of code greatly improves modularity and allows for "magic" such as automatic differentiation by only defining the basic arithmetic rules for dual numbers. It also allows for zero-cost abstractions, letting users represent mathematical equations as they are written without incurring a performance cost.

3. CPU and GPU Parallelisation: parallelising loops is trivial. The `LLVM` compiler is platform agnostic and creates custom, optimal code for almost all CPU and GPU architectures. This means users get platform-specific optimisations for free, and it means the software can remain a single language with no detriment.

4. Metaprogramming: one of the most powerful and esoteric features of `Julia`. Much like `Lisp`, code is an object that can be manipulated by the language itself. This allows code to write and modify code. This can be used to autoparallelise functions on different without rewriting. It lets developers precompute values like an extremely powerful C-preprocessor. It even lets

developers design custom syntax and perform mathematical transformations on code itself.

5. Modern features: `Julia` has all the features of a modern programming language. From a booming and quickly growing fully-integrated package ecosystem with standard and registered libraries, some of which are now best in class. Integrated testing and documentation generation systems. Extremely powerful code introspection tools, e.g. one can check type stability, call trees, and different stages of compilation all the way down to machine instructions. Plus it sports all the interactivity and unicode features we have come to expect from modern interpreted languages.

6. Interoperability: calling other languages from `Julia` is seamless, so the use of external libraries or languages is as easy as calling a `Julia` function. However, `Julia` is so powerful and performant that most of its packages are mono-language. A feature often shared by other modern languages such as `Go`, `Rust`, `Swift` and `Elixir`.

The code can be found here `https://github.com/dcelisgarza/DDD.jl`. Of note are the banners at the top of the readme, those point to the documentation; automated, remote testing; and test coverage i.e. which lines of code have been hit during testing and how many times. It is the result of the lessons learned while working on EasyDD. Its design avoids the same pitfalls and has a clear vision for usability, modularity, and extensibility, without sacrificing performance.

As this has been a didactic, after hours project to escape the anxieties of the pandemic and resolve the frustrations with some fundamental aspects of EasyDD, it is nowhere near feature complete. Nevertheless, we can provide a small show of current capabilities.

6.1 Next-Gen 3D discrete dislocation dynamics

As with every simulation, there are parameters that need to be defined. For dislocation plasticity simulations, the number of parameters is quite large, which is why encapsulation (see section 2.2.2.1) can be so helpful. However, there are also certain relationships between parameters that must be maintained. This is a problem for usability, as it is very easy for one to use non-sensical parameters and be completely unaware of it until a simulation presents unexpected behaviour. We have solved this issue in EasyDD (see section 2.2.4), but the solution is clunky, unelegant and opaque. Here we solve both issues in a single stroke and offer greater flexibility. Setting up a simulations is quite easy and there are multiple options.

1. Load the data from external files generated by any file extension supported by the `FileIO`. In some cases the data has to be used to manually create the structures, but if `JLD2` is used, it works like `MATLAB`'s `save()` and `load()`. Since these files are usually generated from actual variables, their values are already sensical.

2. Use the built-in serialiser. This works like `MATLAB`'s built-in `save()` and `load()` functions.

3. Use the specially defined functions that load `JSON`[1] files and create the structures automatically. Since `JSON` is a human readable format, this is the preferred method for creating sets of parameters that can be used in multiple simulations and loaded from the same file. The functions called internally which create the data structures also validate the provided values and calculate sensible default values from them.

4. Create the structures manually with keyword functions that validate the data, calculate derived values and provide sensible defaults of derived values.

Here is how easy creating all the parameters needed for a simulation can be.

```
# Dislocation parameters.
dlnParams = DislocationParameters(; mobility = mobBCC())
# Material parameters.
matParams = MaterialParameters(; crystalStruct = BCC())
# Finite element parameters.
femParams = FEMParameters(;
                type = DispatchRegularCuboidMesh(),
                order = LinearElement(),
                model = CantileverLoad(),
                dx = 1013.0, dy = 1987.0, dz = 2999.0,
                mx = 3, my = 5, mz = 7
            )
# Slip system data.
slipSystems = SlipSystem(;
                crystalStruct = BCC(),
                slipPlane = Float64[-1;1;0],
                bVec = Float64[1;1;1]
```

[1] `JSON` is an open standard for representing objects as dictionaries in a file. It is widely used by web develpers because it is language agnostic, has high compressibility ratios, and is human readable.

```
            )
# Integration parameters.
intParams = IntegrationParameters(;
                method = AdaptiveEulerTrapezoid()
            )
# Integration time, time step and iteration step
intTime = IntegrationTime()
```

We're using the as many default values as we can, so a simulation using these might not be very good. But the key thing is that one can provide the values for crucial parameters and the rest will be calculated off those. If one were to provide those derived values, there are checks in place that ensure what the user is providing is viable. All together, these represent over 50 constants that control all aspects of a simulation.

Generating dislocation sources is also quite easy. With EasyDD these are made bespoke for a simulation. That means generating a script that creates the sources as need be. But using multiple dispatch it is quite easy to reuse code by simply dispatching functions based on the types of their variables.

```
# Dimensions of the FE domain.
dx, dy, dz = femParams.dx, femParams.dy, femParams.dz

# Length of each dislocation segment.
# We only use one segment length, but one can also
# provide the lengths of every segment in the loop.
segLen = (dx + dy + dz) / 30

# One octagonal prismatic loop with one node per side.
# All nodes are mobile.
# It will be placed in the middle of the FE domain.
prismOct = DislocationLoop(;
    loopType = loopPrism(),
    numSides = 8,
    nodeSide = 1,
    numLoops = 1,
    segLen = segLen * ones(8),
    slipSystemIdx = 1,
    slipSystem = slipSystems,
    label = nodeTypeDln.(ones(Int, 8)),
```

```
    buffer = 0,
    range = [
                dx/2 dx/2;
                dy/2 dy/2;
                dz/2 dz/2
            ],
    dist = Zeros(),
)
```

This generates a prismatic loop with 8 sides whose nodes are all mobile. It will be centered in the middle of the FE domain with a zero distribution.

The buffer is defined relative to the average segment length and is only used to space the loops out when calculating their positions from the distribution and range. Currently there are zero, random uniform and random normal distributions, but new ones can be quickly and easily added by subtyping the distribution type and making a function that dispatches on it. The loop generation source code does not have to be modified.

The type `nodeTypeDln` only has a limited number of values, each of them with a specific meaning. This provides a safeguard against regressions and useful information to users and developers. For example, by looking at the loop's labels we see that they are `intMobDln`, i.e. internal mobile dislocation nodes. Each node type behaves differently and the values are limited to the types of nodes the code supports. A similar technique is used to denote node sets on the finite element side.

```
julia> prismOct.label
8-element Array{nodeTypeDln,1}:
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
 intMobDln::nodeTypeDln = 1
```

We can visualise the loop using a myriad of plotting backends but we will use an interactive one similar to MATLAB's[2]

[2] MATLAB uses a modified version of this backend.

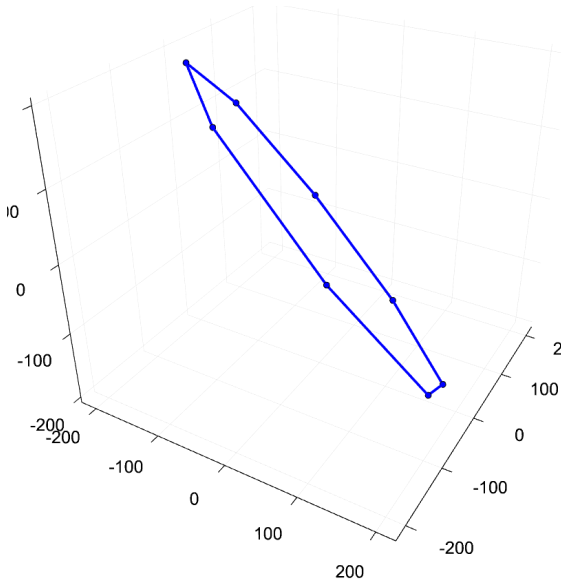


Figure 6.1: Sample 8-sided prismatic BCC dislocation loop with slip plane $\boldsymbol{n} = [\bar{1}\,1\,0]$, and Burgers vector $\boldsymbol{b} = [1\,1\,1]$.

```
using Plots
plotlyjs()

fig = plotNodes(
    prismOct,
    m = 1,
    l = 3,
    linecolor = :blue,
    marker = :circle,
    markercolor = :blue,
    legend = false,
)
```

We can create a prismatic loop in the same way by changing the `loopType` to the shear type, and visualise in the same space by mutating[3] the figure. This time we will create 10 and distribute them random normally in the domain.

```
shearOct = DislocationLoop(;
    loopType = loopShear(),
    numSides = 8,
    nodeSide = 1,
```

[3]By convention, functions that mutate their arguments are denoted by adding an `!` at the end of the function name.

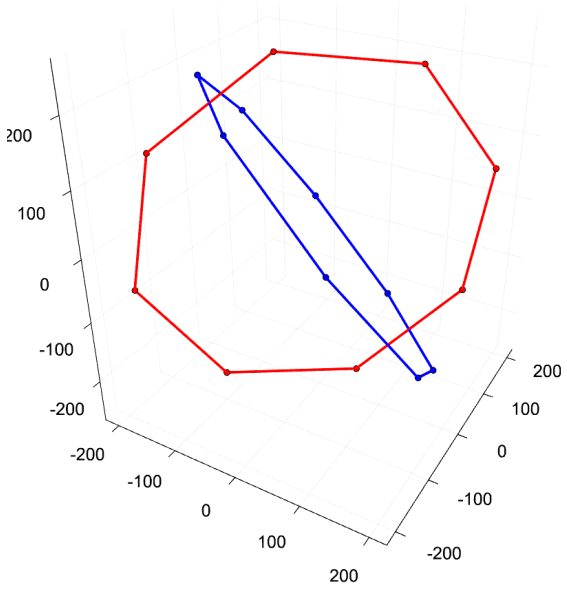


Figure 6.2: Adding 10, 8-sided shear BCC dislocation loop with slip plane $\boldsymbol{n} = [\bar{1}\,1\,0]$, and Burgers vector $\boldsymbol{b} = [1\,1\,1]$.

```
    numLoops = 10,
    segLen = segLen * ones(8),
    slipSystemIdx = 1,
    slipSystem = slipSystems,
    label = nodeTypeDln.(ones(Int, 8)),
    buffer = 0,
    range = [
                0 dx;
                0 dy;
                0 dz
            ],
    dist = Rand(),
)
plotNodes!(
    fig,
    shearOct,
    m = 1,
    l = 3,
    linecolor = :red,
    marker = :star,
    markercolor = :red,
    legend = false,
)
```

As both loops have the same BCC slip system, they are orthogonal to one another.

The `DislocationLoop` structure only denotes dislocation loops, it does not represent the network. The number of loops in the structure, `numLoops`, is what determines how many will be generated. The values of `buffer`, `range` and `dist` determine how they are distributed.

Dislocation networks can be generated manually by specifying all their relevant fields, giving flexibility for the generation of bespoke networks or by using the `DislocationLoop` structure. The latter is easier and more convenient. If certain types of bespoke structures, such as mixed-slip system loops, jogs or kinks, it would be prudent to utilise multiple dispatch and define a specific `DislocationLoop()` function for the purpose. Making the network is quick and painless. There are also default arguments that change things such as how much memory to allocate to allow for the network to expand and the maximum number of connections a node in the network is allowed to have. The default is to allocate $N \log_2 N$ places for N total nodes and segments in the network. The topological operations also use this heuristic when the network needs to be expanded further.

```
network = DislocationNetwork([prismOct, shearOct])
```

We could view the network like this, but where's the fun in that? We'll first generate the mesh. And while we're at it we can make the boundary conditions according to `matParams` and `femParams`. We specified in the creation of `femParams` that we would like to model a `CantileverLoad()` dispatching on a `DispatchRegularCuboidMesh()` with elements of order `LinearElement()`. All this ensures the mesh is built of the correct order and type. Adding mesh types only requires the definition of the appropriate types and the mesh generation function. Users do not need to worry about calling different functions.

```
regularCuboidMesh = buildMesh(matParams, femParams)
cantileverBC, forceDisplacement = Boundaries(
                                      femParams,
                                      regularCuboidMesh
                                  )
```

In fig. 6.3 we can visualise the node sets generated by `buildMesh()`. These are used to very easily generate the boundary conditions without knowing the details of how the FE mesh builder works. This greatly expedites the process of defining the boundary conditions for different scenarios. They define which finite element nodes belong to corners, edges and faces. Those not labelled are internal. These

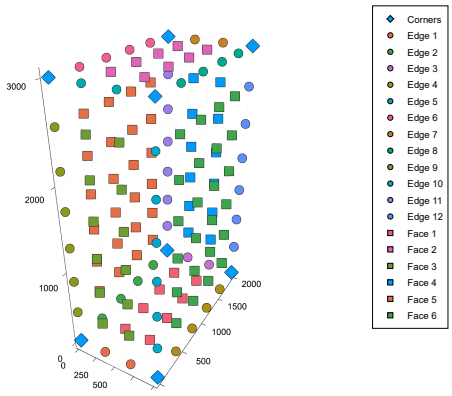


Figure 6.3: Finite element node sets.

are all generated by the mesh builder and is a requirement that forces developers to make the sensible decision of creating node sets so their work is usable by others simply by looking at which nodes belong to which set.

```
plotFEDomain(regularCuboidMesh)
```

Now that we have our FE mesh and network we can see what they look like.

```
plotNodes(
    regularCuboidMesh,
    network,
    m = 1,
    l = 3,
    linecolor = :blue,
    marker = :circle,
    markercolor = :blue,
    legend = false,
)
```

Since we decided to randomly distribute 10 relatively large loops in the domain, there are some segments that are poking out. We can fix that by using the surface remeshing. However, the function requires nodal velocities to work because we need a vector to define the intersection with the surface of the volume.

As of the time of writing, the surface remeshing has not been thoroughly tested and the currently implemented mobility function is out-dated. So instead, we use a random velocity as a stand-in for the real and updated velocities. However, this surface remeshing works for any convex hull regardless of mesh type. It also does not require a surface tasselation because defining the convex hull (using the

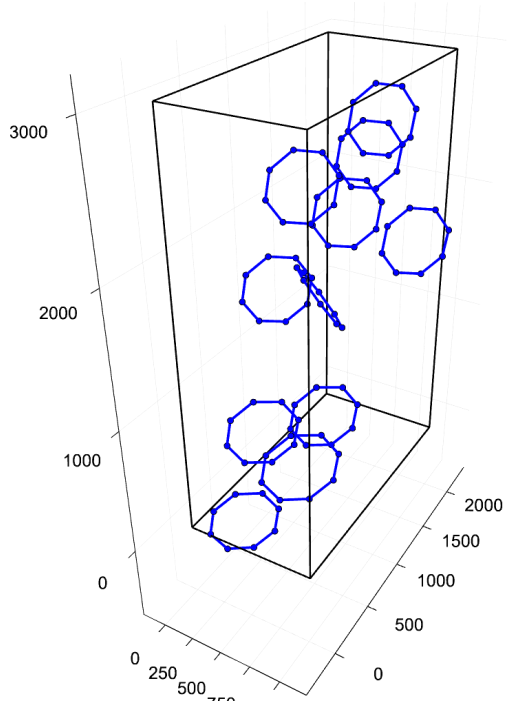


Figure 6.4: Dislocation network and domain.

vertices and a `JuliaGeometry` library), surface normals, and mid-points is enough to exactly determine how to project external nodes. Moreover, the value used to decide how far the nodes should be projected is not hard-coded and is in fact the scale of the FE domain ($\sqrt[3]{\Delta x \Delta y \Delta z}$), ensuring there is always an appropriate closure point for calculating dislocation displacements no matter the domain's aspect ratio.

```
numSeg = network.numSeg[1]
network.nodeVel[:, 1:numSeg] = rand(3, numSeg)
network = remeshSurfaceNetwork!(regularCuboidMesh,  network)
```

This showcases one of the quirks of immutable structures in `julia`. Immutable structures are faster and more easily optimised. But arrays inside them cannot be expanded and the values of non-array fields cannot be changed. However, the values of elements inside arrays can be changed. So to keep track of the number of nodes and segments they are vectors of length 1. Also, any function with the potential for resizing the arrays of an immutable structure, has to return the structure, else the new array entries get lost and garbage collected later. Since `remeshSurfaceNetwork!()` can change the values inside the arrays of `network` as well as resize them, its return value gets reassigned to `network`. After running this we get a properly internal network with a bunch of surface nodes.

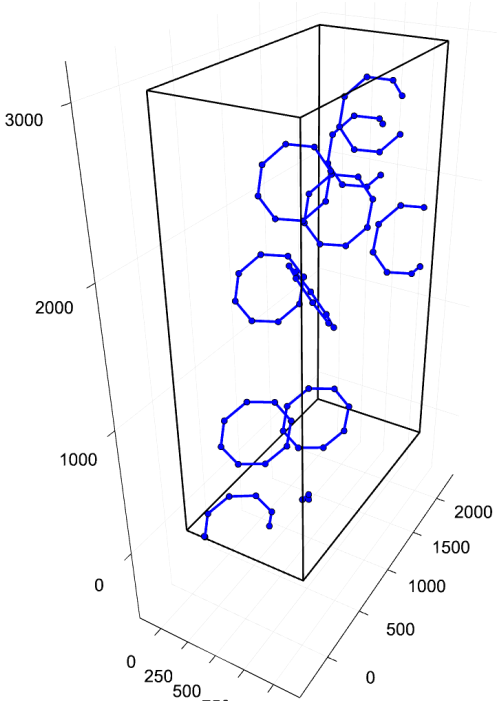


Figure 6.5: Sample network and domain with remeshed surface.

As previously stated, this has been mostly a didactic exercise. However, as of the time of writing, the code has the following capabilities.

1. IO,
2. loop and network generation,
3. regular cuboid mesh generation,
4. boundary conditions for cantilever bending,
5. internal remeshing,
6. surface remeshing,
7. dislocation displacements,
8. forces on segments,
9. field point stresses,
10. outdated BCC mobility function,
11. time integration.

All of which are faster than what is offered by EasyDD—even those accelerated by `C`. The $\mathcal{O}(N^2)$ segment-segment calculation is 10% faster than its equivalent `C` function and has already been CPU-parallelised; for large numbers of dislocations it offers linear scaling with number of threads. The self-forces and Peach-Köhler forces as a result of the dislocations being inside a cuboid mesh are both 10× faster than their `MATLAB` counterparts. Since the mesh refinement uses a better heuristic than dynamically sizing arrays every time a new node is added to the network, it is about as slow as `MATLAB`'s when it needs to allocate memory, but when it does not it is $>$ 100× faster. Mesh coarsening is 3× faster. The calculation of field point stresses for numeric tractions is 30% faster than its `C` counterpart. Building the FE mesh is $>$ 10× faster. The outdated mobility law is $>$ 30× faster and in all the tests has not required dampening the matrix inversion. The dislocation displacement calculation is $>$ 10× faster. The cholesky factorisation is 20× faster and uses a substantially smaller amount of memory.

The memory footprint is also a lot lower. Care has been taken to do things as optimally as possible and `Julia`'s typesystem lets sparse arrays be treated in exactly the same way as dense ones, which is not always the case in `MATLAB`, which has thwarted efforts at sparsifying various arrays that would benefit from it.

The total number of lines of code so far, $\sim$ 3200 are the equivalent of $\sim$ 4500 without counting input files and the many different loop generation functions. However this accounts for mutating and non-mutating functions, which are essentially the same implementation only non mutating functions allocate a return value while mutating ones change their arguments. Mutating functions tend to be faster because of the lack of heap allocation, but we decided to make both as only the indexing changes. Accounting only for single versions of mutating or non-mutating functions yields $\sim$ 1800 unique lines of code, which is about the same number of lines in the segment-segment `C` function.

Moreover, doing the equivalent of what we have shown on EasyDD requires a substantially larger amount of code and a not insignificant amount of knowledge of its inner workings. From generating the network, to defining values for its parameters—which, despite the fact that EasyDD provides a reasonable set of defaults, users can easily define derived parameters incorrectly and be none the wiser. The barrier to entry and potential for error are quite high. Even us, who use the code all the time and have worked on it, sometimes miss things only to catch them after a simulation does something unexpected.

The one knock on `Julia` is its JIT-compiled nature. Which leads to a notoriously long time to first plot and means functions are slow the first time they are run during a session because they need to be compiled. However this is quickly improving and can be solved by keeping a precompiled system image of the of-

fending packages. There is also the conspicuously poor performance of the `spy()` function, even with the `UnicodePlots` backend. Having said that, the power, flexibility and ease of use of `Julia` greatly outweighs such small issues.

6.2 Conclusions and proposal

What started as a side project to learn and escape from the world during the last year has resulted in quite a promising avenue for future research. One that lets us have our cake and eat it too. Not only is it performant, it is almost trivially parallelisable on both CPUs and GPUs; while staying easy to use and develop. What `Julia` offers is almost incomprehensible. The performance of `C` and `Fortran`; the syntax and interactivity of `Python` and `R`; metaprogramming of `Lisp`; and fully integrated testing and package environments of `Rust` and `Go`.

A tool is only as good as the craftsman who weilds it. We aspire to be among those elite craftsmen. Why then, should we not use as good a tool as we can? During the course of this project, the whole Tarleton group have made gigantic strides with EasyDD. It has indeed come a very long way, but we can do better.

Some issues are so deeply intertwined with how the software was originally designed that they cannot be separated from how it fundamentally operates. To fix them, a fresh new start is needed. Given the extremely encouraging results of what amounts to a pandemic hobby, we think it worthwhile to continue this as a research software engineering project post-DPhil. We think it is better to do so now rather than wait until there is no more juice left to squeeze out of `MATLAB`. Not only so the shackles of proprietary software can be broken; but also so the full, ambitious vision of EasyDD can be realised.

Chapter 7

Conclusions

Bibliography

[1] Ellen G Howe and Ulrich Petersen. Silver and lead in the late prehistory of the mantaro valley, peru. Archaeometry of pre-Columbian sites and artifacts, pages 183–198, 1994.

[2] John Chapman. Fragmentation in archaeology: people, places and broken objects in the prehistory of south eastern Europe. Routledge, 2013.

[3] Kris MY Law and Marko Kesti. Yin Yang and Organizational Performance: Five elements for improvement and success. Springer, 2014.

[4] Bryan K Hanks and Katheryn M Linduff. Social complexity in prehistoric Eurasia: Monuments, metals and mobility. Cambridge University Press, 2009.

[5] Jared M Diamond. Guns, germs and steel: a short history of everybody for the last 13,000 years. Random House, 1998.

[6] Arthur Wilson. The living rock: the story of metals since earliest times and their impact on developing civilization. Woodhead Publishing, 1994.

[7] EW Hart. Theory of the tensile test. Acta metallurgica, 15(2):351–355, 1967.

[8] SU Benscoter. A theory of torsion bending for multicell beams. Journal of Applied Mechanics, 21(1):25–34, 1954.

[9] RF Bishop, Rodney Hill, and NF Mott. The theory of indentation and hardness tests. Proceedings of the Physical Society, 57(3):147, 1945.

[10] S Zhandarov, E Pisanova, and B Lauke. Is there any contradiction between the stress and energy failure criteria in micromechanical tests? part i. crack initiation: stress-controlled or energy-controlled? Composite Interfaces, 5(5): 387–404, 1997.

[11] Srinivasa D Thoppul and Ronald F Gibson. Mechanical characterization of spot friction stir welded joints in aluminum alloys by combined experimental/numerical approaches: part i: micromechanical studies. Materials characterization, 60(11):1342–1351, 2009.

[12] Dierk Raabe, M Sachtleber, Zisu Zhao, Franz Roters, and Stefan Zaefferer. Micromechanical and macromechanical effects in grain scale polycrystal plasticity experimentation and simulation. Acta Materialia, 49(17):3433–3441, 2001.

[13] YW Kwon and JM Berner. Micromechanics model for damage and failure analyses of laminated fibrous composites. Engineering Fracture Mechanics, 52(2):231–242, 1995.

[14] Dierk Raabe, M Sachtleber, Zisu Zhao, Franz Roters, and Stefan Zaefferer. Micromechanical and macromechanical effects in grain scale polycrystal plasticity experimentation and simulation. Acta Materialia, 49(17):3433–3441, 2001.

[15] Risto M Nieminen. From atomistic simulation towards multiscale modelling of materials. Journal of Physics: Condensed Matter, 14(11):2859, 2002.

[16] JA Elliott. Novel approaches to multiscale modelling in materials science. International Materials Reviews, 56(4):207–225, 2011.

[17] Lihua Wang, Xiaodong Han, Pan Liu, Yonghai Yue, Ze Zhang, and En Ma. In situ observation of dislocation behavior in nanometer grains. Physical review letters, 105(13):135501, 2010.

[18] T Tabata and HK Birnbaum. Direct observations of the effect of hydrogen on the behavior of dislocations in iron. Scripta Metallurgica, 17(7):947–950, 1983.

[19] M Dao, L Lu, RJ Asaro, J Th M De Hosson, and E Ma. Toward a quantitative understanding of mechanical behavior of nanocrystalline metals. Acta Materialia, 55(12):4041–4065, 2007.

[20] MR Gilbert, SL Dudarev, S Zheng, LW Packer, and J-Ch Sublet. An integrated model for materials in a fusion power plant: transmutation, gas production, and helium embrittlement under neutron irradiation. Nuclear Fusion, 52(8):083019, 2012.

[21] Steven J Zinkle and GS Was. Materials challenges in nuclear energy. Acta Materialia, 61(3):735–758, 2013.

[22] Ian Cook. Materials research for fusion energy. Nature Materials, 5(2):77, 2006.

[23] Heinrich Hora. Developments in inertial fusion energy and beam fusion at magnetic confinement. Laser and Particle Beams, 22(04):439–449, 2004.

[24] Weston M Stacey. Fusion: an introduction to the physics and technology of magnetic confinement fusion. John Wiley & Sons, 2010.

[25] ROBERT L McCrory and CHARLES P Verdon. Inertial confinement fusion. Computer Applications in Plasma Science and Engineering (Springer Science & Business Media, 2012), page 291, 2012.

[26] Mohamed E Sawan. Geometrical, spectral and temporal differences between icf and mcf reactors and their impact on blanket nuclear parameters. Fusion Technology, 10(3P2B):1483–1488, 1986.

[27] GL Kulcinski and ME Sawan. Differences between neutron damage in inertial and magnetic confinement fusion materials test facilities. Journal of nuclear materials, 133:52–57, 1985.

[28] Steven J Zinkle and Jeremy T Busby. Structural materials for fission & fusion energy. Materials Today, 12(11):12–19, 2009.

[29] Alban Mosnier, PY Beauvais, B Branas, M Comunian, A Facco, P Garin, R Gobin, JF Gournay, R Heidinger, A Ibarra, et al. The accelerator prototype of the ifmif/eveda project. IPAC, 10:588, 2010.

[30] T Muroga, M Gasparotto, and SJ Zinkle. Overview of materials research for fusion reactors. Fusion engineering and design, 61:13–25, 2002.

[31] JL Bourgade, AE Costley, R Reichle, ER Hodgson, W Hsing, V Glebov, M Decreton, R Leeper, JL Leray, M Dentan, et al. Diagnostic components in harsh radiation environments: Possible overlap in r&d requirements of inertial confinement and magnetic fusion systemsa). Review of Scientific Instruments, 79(10):10F304, 2008.

[32] OA Hurricane, DA Callahan, DT Casey, PM Celliers, Charles Cerjan, EL Dewald, TR Dittrich, T Döppner, DE Hinkel, LF Berzak Hopkins, et al. Fuel gain exceeding unity in an inertially confined fusion implosion. Nature, 506 (7488):343, 2014.

[33] D Böhne, I Hofmann, G Kessler, GL Kulcinski, J Meyer-ter Vehn, U von Möllendorff, GA Moses, RW Müller, IN Sviatoslavsky, DK Sze, et al. Hiball—a conceptual design study of a heavy-ion driven inertial confinement fusion power plant. Nuclear Engineering and Design, 73(2):195–200, 1982.

[34] John D Lindl, Robert L McCrory, and E Michael Campbell. Progress toward ignition and burn propagation in inertial confinement fusion. Phys. Today, 45(9):32–40, 1992.

[35] RW Moir, RL Bieri, XM Chen, TJ Dolan, MA Hoffman, PA House, RL Leber, JD Lee, YT Lee, JC Liu, et al. Hylife-ii: A molten-salt inertial fusion energy power plant design—final report. Fusion Science and Technology, 25(1):5–25, 1994.

[36] Roger E Stoller, Mychailo B Toloczko, Gary S Was, Alicia G Certain, Shyam Dwaraknath, and Frank A Garner. On the use of SRIM for computing radiation damage exposure. Nuclear instruments and methods in physics research section B: beam interactions with materials and atoms, 310:75–80, 2013.

[37] GH Kinchin and RS Pease. The displacement of atoms in solids by radiation. Reports on progress in physics, 18(1):1, 1955.

[38] James F Ziegler, Matthias D Ziegler, and Jochen P Biersack. SRIM–the stopping and range of ions in matter (2010). Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms, 268(11):1818–1823, 2010.

[39] Daniel Mason. Incorporating non-adiabatic effects in embedded atom potentials for radiation damage cascade simulations. Journal of Physics: Condensed Matter, 27(14):145401, 2015.

[40] AE Sand, SL Dudarev, and K Nordlund. High-energy collision cascades in tungsten: Dislocation loops structure and clustering scaling laws. EPL (Europhysics Letters), 103(4):46003, 2013.

[41] T Diaz De La Rubia and MW Guinan. New mechanism of defect production in metals: A molecular-dynamics study of interstitial-dislocation-loop formation in high-energy displacement cascades. Physical review letters, 66(21):2766, 1991.

[42] Vasily Bulatov and Wei Cai. Computer simulations of dislocations, volume 3. Oxford University Press on Demand, 2006.

[43] N Bertin, S Aubry, A Arsenlis, and W Cai. Gpu-accelerated dislocation dynamics using subcycling time-integration. Modelling and Simulation in Materials Science and Engineering, 27(7):075014, 2019.

[44] Athanasios Arsenlis, Wei Cai, Meijie Tang, Moono Rhee, Tomas Oppelstrup, Gregg Hommes, Tom G Pierce, and Vasily V Bulatov. Enabling strain hardening simulations with dislocation dynamics. Modelling and Simulation in Materials Science and Engineering, 15(6):553, 2007.

[45] B Bromage and E Tarleton. Calculating dislocation displacements on the surface of a volume. Modelling and Simulation in Materials Science and Engineering, 26(8):085007, 2018.

[46] E Ward Cheney and David R Kincaid. Numerical mathematics and computing. Cengage Learning, 2012.

[47] Paul Rodríguez. Total variation regularization algorithms for images corrupted with different noise models: a review. Journal of Electrical and Computer Engineering, 2013, 2013.

[48] Peter J Bickel, Bo Li, Alexandre B Tsybakov, Sara A van de Geer, Bin Yu, Teófilo Valdés, Carlos Rivero, Jianqing Fan, and Aad van der Vaart. Regularization in statistics. Test, 15(2):271–344, 2006.

[49] Zhanxuan Hu, Feiping Nie, Rong Wang, and Xuelong Li. Low rank regularization: A review. Neural Networks, 2020.

[50] Coronavirus (covid-19): modelling the epidemic. URL https://www.gov.scot/collections/coronavirus-covid-19-modelling-the-epidemic/.

[51] Grant Hill-Cawthorne. Models of covid-19: Part 1, Dec 2020. URL https://post.parliament.uk/models-of-covid-19-part-1/.

[52] Grant Hill-Cawthorne. Models of covid-19: Part 2, Dec 2020. URL https://post.parliament.uk/models-of-covid-19-part-2/.

[53] Dalmeet Singh Chawla. Critiqued coronavirus simulation gets thumbs up from code-checking efforts, Jun 2020. URL https://www.nature.com/articles/d41586-020-01685-y.

[54] Niels G Mede, Mike S Schäfer, Ricarda Ziegler, and Markus Weißkopf. The "replication crisis" in the public eye: Germans' awareness and perceptions of the (ir) reproducibility of scientific research. Public Understanding of Science, page 0963662520954370, 2020.

[55] David Randall and Christopher Welser. The Irreproducibility Crisis of Modern Science: Causes, Consequences, and the Road to Reform. ERIC, 2018.

[56] Roberto Bolli. Reflections on the irreproducibility of scientific papers. Circulation research, 117(8):665–666, 2015.

[57] Simon Thompson. Haskell: the craft of functional programming, volume 2. Addison-Wesley, 2011.

[58] Konrad Hinsen. The promises of functional programming. Computing in Science & Engineering, 11(4):86–90, 2009.

[59] Donald E Knuth. Structured programming with go to statements. ACM Computing Surveys (CSUR), 6(4):261–301, 1974.

[60] David R Hanson. Fast allocation and deallocation of memory based on object lifetimes. Software: Practice and Experience, 20(1):5–12, 1990.

[61] Z Shen, RH Wagoner, and WAT Clark. Dislocation pile-up and grain boundary interactions in 304 stainless steel. Scripta metallurgica, 20(6):921–926, 1986.

[62] Z Shen, RH Wagoner, and WAT Clark. Dislocation and grain boundary interactions in metals. Acta metallurgica, 36(12):3231–3242, 1988.

[63] M Verdier, M Fivel, and I Groma. Mesoscopic scale simulation of dislocation dynamics in fcc metals: Principles and applications. Modelling and Simulation in Materials Science and Engineering, 6(6):755, 1998.

[64] C. Déprés, C. F. Robertson, and M. C. Fivel. Low-strain fatigue in 316l steel surface grains: a three dimension discrete dislocation dynamics modelling of the early cycles. part 2: Persistent slip markings and micro-crack nucleation. Philosophical Magazine, 86(1):79–97, 2006. doi: 10.1080/14786430500341250.

[65] E Tarleton, DS Balint, J Gong, and AJ Wilkinson. A discrete dislocation plasticity study of the micro-cantilever size effect. Acta Materialia, 88:271–282, 2015.

[66] Haiyang Yu, Alan Cocks, and Edmund Tarleton. Discrete dislocation plasticity helps understand hydrogen effects in bcc materials. Journal of the Mechanics and Physics of Solids, 2018. ISSN 0022-5096. doi: https://doi.org/10.1016/j.jmps.2018.08.020.

[67] S. Groh and H. M. Zbib. Advances in Discrete Dislocations Dynamics and Multiscale Modeling. Journal of Engineering Materials and Technology, 131 (4):041209, 2009. ISSN 00944289. doi: 10.1115/1.3183783.

[68] Erik Van der Giessen and Alan Needleman. Discrete dislocation plasticity: a simple planar model. Modelling and Simulation in Materials Science and Engineering, 3(5):689, 1995.

[69] MC Fivel and GR Canova. Developing rigorous boundary conditions to simulations of discrete dislocation dynamics. Modelling and Simulation in Materials Science and Engineering, 7(5):753, 1999.

[70] Akiyuki Takahashi and Nasr M Ghoniem. A computational method for dislocation–precipitate interaction. Journal of the Mechanics and Physics of Solids, 56(4):1534–1553, 2008.

[71] Wei Cai, Athanasios Arsenlis, Christopher R Weinberger, and Vasily V Bulatov. A non-singular continuum theory of dislocations. Journal of the Mechanics and Physics of Solids, 54(3):561–587, 2006.

[72] B Devincre, A Roos, and S Groh. Boundary problems in dd simulations. In In: Thermodynamics, Microstructures and Plasticity, A. Finel et al., Nato Sciences Series II: Mathematics, Physics and Chemistry, 108, p. 275, Eds (Kluwer, NL-Dordrecht. Citeseer, 2003.

[73] CS Shin, MC Fivel, and KH Oh. Nucleation and propagation of dislocations near a precipitate using 3d discrete dislocation dynamics simulations. Le Journal de Physique IV, 11(PR5):Pr5–27, 2001.

[74] M. P. O'Day and W. a. Curtin. A Superposition Framework for Discrete Dislocation Plasticity. Journal of Applied Mechanics, 71(November 2004): 805, 2004. ISSN 00218936. doi: 10.1115/1.1794167.

[75] S Queyreau, J Marian, BD Wirth, and A Arsenlis. Analytical integration of the forces induced by dislocations on a surface element. Modelling and Simulation in Materials Science and Engineering, 22(3):035004, 2014.

[76] Gene H Golub and John H Welsch. Calculation of gauss quadrature rules. Mathematics of computation, 23(106):221–230, 1969.

[77] Michael A Khayat and Donald R Wilton. Numerical evaluation of singular and near-singular potential integrals. IEEE Transactions on Antennas and Propagation, 53(10):3180–3190, 2005.

[78] AK Head. Edge dislocations in inhomogeneous media. Proceedings of the Physical Society. Section B, 66(9):793, 1953.

[79] John Price Hirth, Jens Lothe, and T Mura. Theory of dislocations, 1983.

[80] Brian Kelleher and Thomas C Furlong. Software configurable memory architecture for data processing system having graphics capability, August 28 1990. US Patent 4,953,101.

[81] Masayuki Sumi. Image processing devices and methods, March 31 1998. US Patent 5,734,807.

[82] Joseph Celi Jr, Jonathan M Wagner, and Roger Louie. Advanced graphics driver architecture, February 3 1998. US Patent 5,715,459.

[83] John Nickolls and William J Dally. The gpu computing era. IEEE micro, 30 (2), 2010.

[84] Shengquan Wang, Chao Wang, Yong Cai, and Guangyao Li. A novel parallel finite element procedure for nonlinear dynamic problems using gpu and mixed-precision algorithm. Engineering Computations, 2020.

[85] Xun Jia, Peter Ziegenhein, and Steve B Jiang. Gpu-based high-performance computing for radiation therapy. Physics in Medicine & Biology, 59(4):R151, 2014.

[86] Amir Sabbagh Molahosseini and Hans Vandierendonck. Half-precision floating-point formats for pagerank: Opportunities and challenges. In 2020 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7. IEEE, 2020.

[87] NVidia, CUDA. NVidia CUDA C programming guide. NVidia Corporation, 120(18):8, 2011.

[88] Carlos Carvalho. The gap between processor and memory speeds. In Proc. of IEEE International Conference on Control and Automation, 2002.

[89] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. Dacache: Memory divergence-aware gpu cache management. In Proceedings of the 29th ACM on International Conference on Supercomputing, pages 89–98, 2015.

[90] Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, and Henri Bal. A detailed gpu cache model based on reuse distance theory. In 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pages 37–48. IEEE, 2014.

[91] Martin Schindewolf, Jie Tao, Wolfgang Karl, and Marcelo Cintra. A generic tool supporting cache design and optimisation on shared memory systems. In 9th workshop on parallel systems and algorithms–workshop of the GI/ITG

special interest groups PARS and PARVA. Gesellschaft für Informatik e. V., 2008.

[92] Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. Mdm: The gpu memory divergence model. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1009–1021. IEEE, 2020.

[93] Andrea Elisabet Sand, Kai Nordlund, and SL Dudarev. Radiation damage production in massive cascades initiated by fusion neutrons in tungsten. Journal of Nuclear Materials, 455(1-3):207–211, 2014.

[94] Yang Li, Max Boleininger, Christian Robertson, Laurent Dupuy, and Sergei L Dudarev. Diffusion and interaction of prismatic dislocation loops simulated by stochastic discrete dislocation dynamics. Physical Review Materials, 3(7): 073805, 2019.

[95] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. SIAM review, 59(1):65–98, 2017. URL https://doi.org/10.1137/141000671.

[96] Fortran Standards Committee. Contributing to OpenCoarrays. URL https://wg5-fortran.org/.

[97] Sourcery Institute. Home, November 2017. URL http://www.sourceryinstitute.org/.

[98] Fanfarillo, Alessandro and Burnus, Tobias and Cardellini, Valeria and Filippone, Salvatore and Nagle, Dan and Rouson, Damian. OpenCoarrays: open-source transport layers supporting coarray Fortran compilers. In Proceedings of the 8th International Conference on Partitioned Global Address Space Prog page 4. ACM, 2014.

Appendix A

Implementation of the standard C descriptors for C-Fortran interoperability

Code hosted on `https://github.com/`.
Sourcery Institute: `sourceryinstitute`
`iso_fortran_binding`
`GCC`: `https://github.com/gcc-mirror/gcc/blob/master/libgfortran/`
`GCC` definitions: `ISO_Fortran_binding.h`
`GCC` implementation: `/runtime/ISO_Fortran_binding.c`

The Fortran 2018 standard [96] defines a standard set of C structures and functions that give C the ability to access and operate on Fortran variables and objects. The structures contain all the information needed to fully explore most Fortran objects from C programmes. The functions perform unit operations that can be used together to perform more complex ones. These structures and functions standardise and greatly expand current C-Fortran interoperability. This paper summarises their use and implementation.

A.1 Introduction

Fortran has historically been the de facto language of scientific computing. However, C-type languages such as C/C++ have gained much traction since the 90s due to their popularity in software development. Furthermore, the large degree of compatibility between C and higher level languages such as Python, Java, Objective-C and others have made C the center piece of general purpose computing.

The popularity of C-type languages has resulted in an increase of scientific

computing codes written in them, particularly C++ and Python due to their ready-made data structures and algorithms. However, when it comes to high performance computing in numerical applications Fortran still has the edge. In order to get the best of both worlds however, increasing C-Fortran interoperability is of the utmost importance.

C-Fortran interoperability has existed in a rudimentary fashion since the Fortran 2003 standard [96]. Compiler manufacturers are free to add non-standard compliant functionality which expand language features. These expansions are compiler-specific and limit code compatibility between different systems and compilers. Often, these compiler-specific additions are standardised and expanded by the standards committee in future releases.

The Fortran 2018 standard did so with the C-descriptors of Fortran variables as well as basic functions to query and utilise them. They expand and homogenise the capabilities of various compiler-specific expansions, increasing software portability and interoperability.

A.2 C descriptor structure

The C descriptors are C structures which act as non-generic containers for Fortran variables. The C descriptor structure contains various member variables that allow one to fully explore and use Fortran objects within C. The C descriptor structure makes use of a few custom types, among which is another structure `CFI_dim_t`;

- `CFI_index_t lower_bound`: lower bound of the dimension being described;
- `CFI_index_t extent`: number of elements in the dimension being described;
- `CFI_index_t sm`: memory stride for the dimension, i.e. the difference in bytes between the addresses of successive elements in the dimension being described.

The actual structure of the C descriptor is as follows,

- `void *base_address`: pointer to the base address of the fortran object or first element of the array being described;
- `size_t elem_len`: storage size in bytes of the object or element of the array being described;
- `CFI_rank_t rank`: rank of the array described, rank 0 denotes a scalar;
- `CFI_type_t type`: type of the object described, each interoperable type has a specific type identifier;

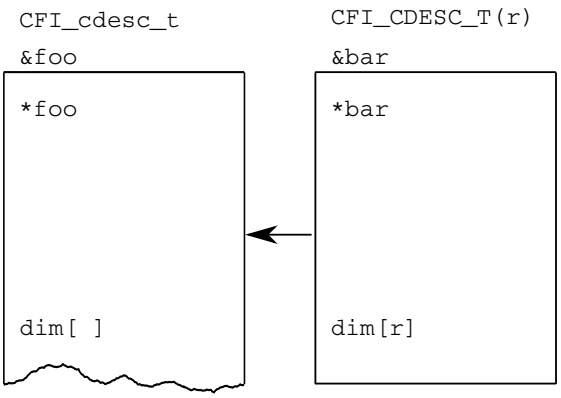


Figure A.1: C descriptor with rank `r` being cast into one with unspecified rank. C looses track of exactly how large the object is, but given the value of the rank element `r`, the size of a single element of `dim` and the information contained within the `dim` member variables, the object can be fully explored.

- `CFI_attribute_t attribute`: denotes whether the object is allocatable, pointer or nonallocatable nonpointer, each has a unique value;
- `CFI_dim_t dim`: dimensional information.

C is a weak, statically typed language; statically typed means that types are checked at compile and runtime, weak means they can be converted into other types via pointer casting. This is often the source of many a headache and bug, but can double as a poor-man's polymorphism[1]. This is readily exploited by the functions which utilise the C descriptors. Figure A.1 illustrates how C looses track of the full size of the structure, but since it knows the size of the `dim` structure as well as the rank of the object being described, one can fully traverse the object. As a result, developers are strongly advised to use the standard functions and—if need be—create their own, more complex ones if they want to manipulate the C object.

A.3 C descriptor functions

As aforementioned, the functions defined by the standard [96] allow the user to use and explore Fortran objects from within C;

- `CFI_establish`: updates the C descriptor variable to establish it as a variable that would allow the program to access fortran variables;
- `CFI_address`: returns the C address of the fortran object or the C address of the corresponding subscripts provided;

[1]Polymorphism is when a variable or function, can be of different types.

- `CFI_allocate`: allocates memory for an object described by a C descriptor;
- `CFI_deallocate`: frees memory allocated to the C descriptor;
- `CFI_is_contiguous`: describe whether the array described by the descriptor is contiguous or not;
- `CFI_section`: updates the base address and dimension information of the C descriptor to describe the array section determined by the function arguments, this can be used to reduce the rank of an array and even invert the order in which the array is traversed;
- `CFI_select_part`: updates the base address, dimensional information and element length parameters of the C descriptor to select a member variable of a Fortran derived-type, a substring or the real/imaginary part of a complex variable, it also works on arrays of such objects.

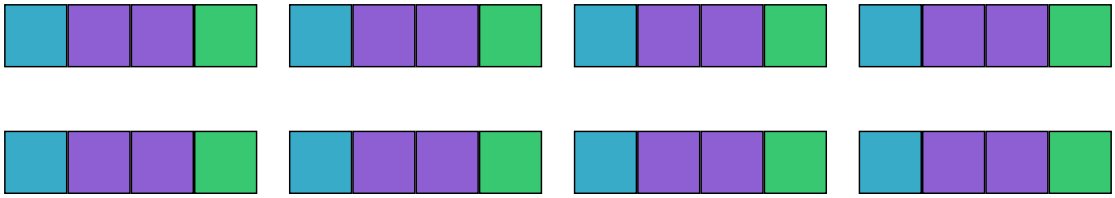
These functions were originally defined by the standard as if they would be used on independent C descriptors. However, when using `CFI_section` and `CFI_select_part` on the same descriptor, the element length and dimensional information can end up being updated in nontrivial ways (see fig. A.2). These affect array contiguity and lead to increased complexity in other functions, none of which had been accounted for by the standard.

It is important to note that the C descriptor functions do no data copying as that would severely impact performance. Instead, what they do is update the information in the C descriptor such that the programme may find the relevant C addresses via pointer arithmetic.

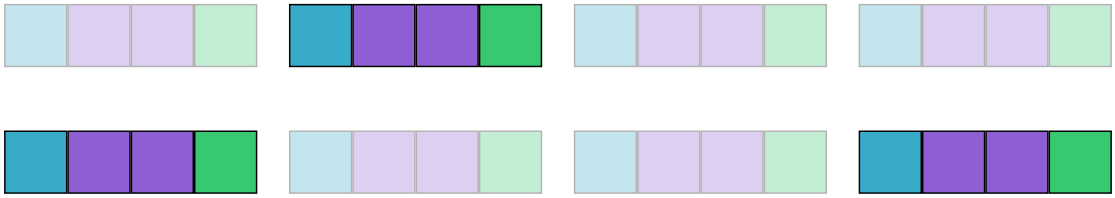
A.4 Results and conclusions

The standard C descriptors and associated functions were successfully implemented and fully tested. They passed all the tests during development. They fail to build with `clang` because they utilise a `GCC` expansion that allows one to define structures with variable length arrays (arrays with no explicit length within structures). However, this is being solved by the Sourcery Institute [97] and is merely a case of "unrolling" the structure with the variable length array into explicit structures for specific ranks.

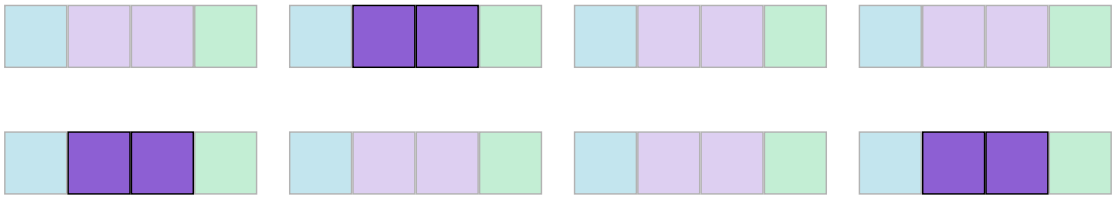
The code can be found in the Github repository /sourceryinstitute/ISO_Fortran_binding. The descriptors are now also being implemented by the `GCC gfortran` developers. They plan on porting the non-standard descriptors and associated programs to use the standard descriptors and



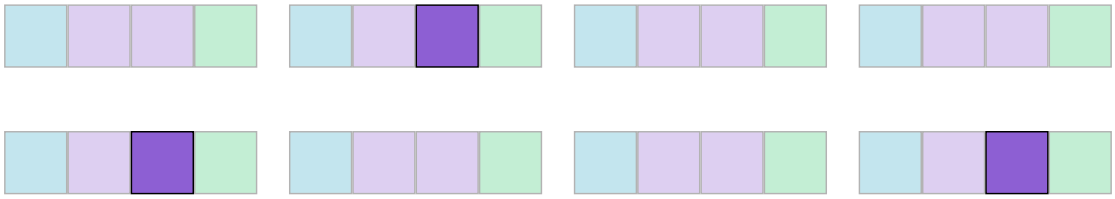
(a) 1D array of length 8 of a derived type containing a scalar, 2 element array, and scalar. The array is contiguous.



(b) Section the array to include only every 2^{nd} item (stride of two) starting from the 2^{nd} element. The array is no longer contiguous.



(c) Select only the arrays within each element of the array section found in fig. A.2b.



(d) Section the array to only select the second item of the array of arrays in fig. A.2c.

Figure A.2: The order can be different, but all these changes must somehow be accounted for within the C descriptor.

functions. They are also being ported into the OpenCoarrays project [98] by the Sourcery Institute.

It is hard to say which future releases will include the full implementation of the C descriptors but the next major `GCC` release will likely provide partial support[2]. The OpenCoarrays project is still mostly largely focused on ironing out bugs with platform support and rare edge cases, so it is unlikely the port will be made within the next release. However, support for them might be present in the release of OpenCoarrays 3.0 within the next few years.

[2]These features are partially supported by early versions of `GCC 7` and fully supported by `GCC 8` and above.

A.5 Acknowledgements

I would like to thank Izaak “Zaak” Beekman, Soren Rasmusen, Jerry Delisle, Paul Thomas and last but certainly not least Damian Rouson for the opportunity and priviledge to work with them. I have been a fan of the Sourcery Institute since my early days as an undergrad. It was a wonderful and fun time full of learning and merriment. Hopefully my work will contribute to the success and popularity of open-source scientific software.