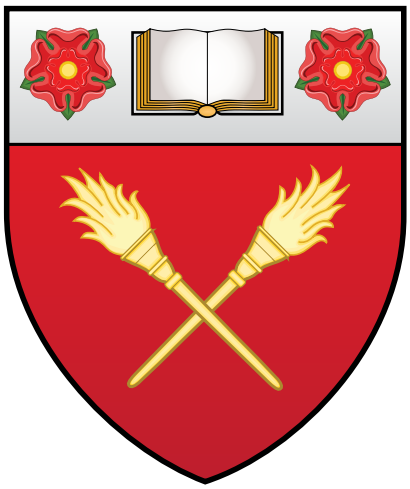
Dislocation Based Modelling of Fusion Relevant Materials



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

In the digital version of this document, URLs, commits, and filenames are hyperlinks to the relevant websites.

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. Requires `minted`, `https://ctan.org/pkg/minted?lang=en` and its dependencies to compile.

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 [2, 3]. A civilisation's ability to gain mastery of the properties and usage of materials is often strongly correlated with its success in history [4]. The historical influence of metals and alloys has touched all areas of human existence, from fashion to warfare [5].

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 [6, 7]. 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 [8–10]. 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 [11]. 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 [12, 13]. 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 [14, 15].

However, understanding the underlying mechanisms behind empirical results through experimental means has proven difficult at every scale [16, 17]. Fortunately modelling and simulation can be powerful allies in this task. In particular, the study of dislocations is experimentally challenging [18–20]. 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 [21–23]. 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 [24]: *1*) magnetic confinement fusion (MCF) [25] which confines the plasma using magnetic fields and *2*) inertial confinement fusion (ICF) [26] 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 [27, 28].

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
	Atmosphere	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 [32]. 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 [29]. The lack of appropriate sources of suitably energetic neutrons [30] has meant that modelling, as well as searching for experimental analogues for the true damage cascades have become a crucial part of materials research [31].

When it comes to fusion, the operating environments change vastly between ICF and MCF, as described by table 1.1 [32]. 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 obliter-

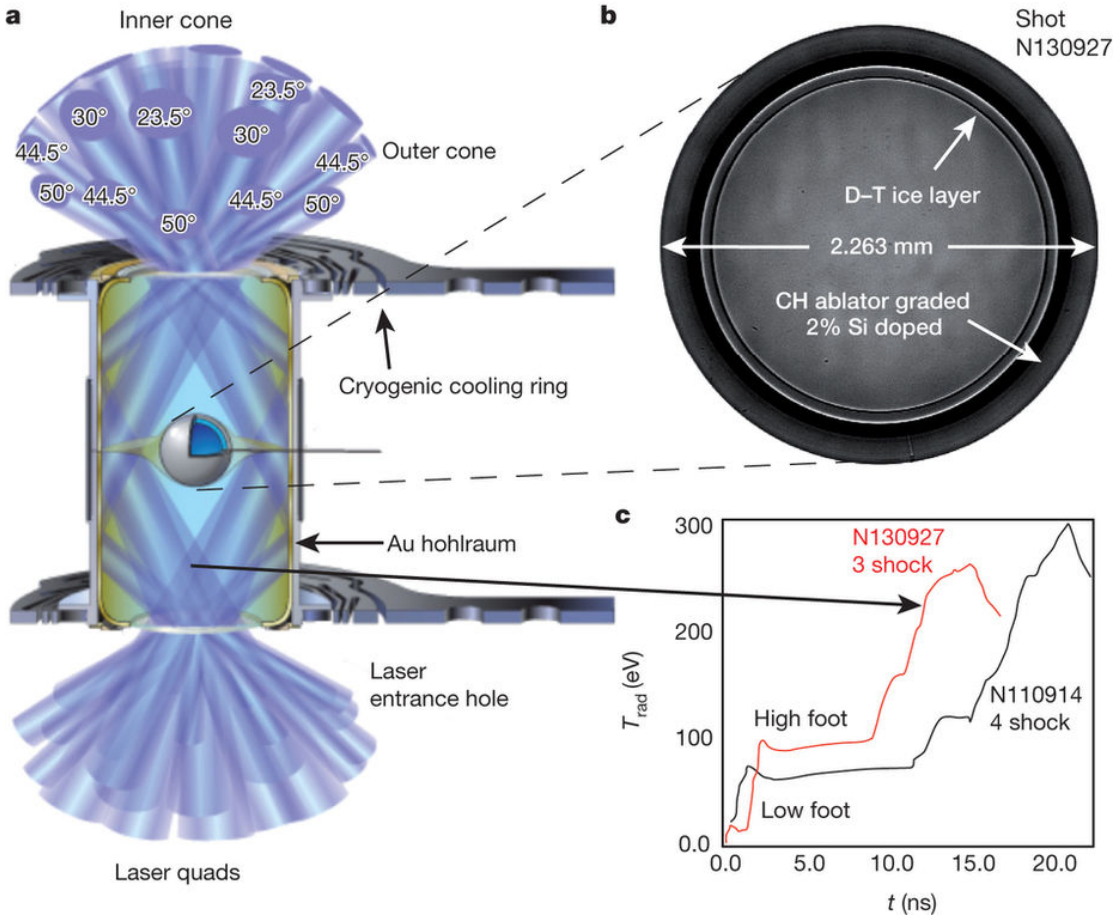


Figure 1.1: **a** Cross-section of the National Ignition Facility's ICF target design showing the gold hohlraum and plastic capsule, fuel pellet, and incident laser bundles. **b** X-ray image of the actual capsule for N130927 with DT fuel layer and surrounding CH (carbon–hydrogen) plastic ablator. **c** 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 [33].

ated when the pellet undergoes fusion (see fig. 1.1 [33]). This is a huge challenge, as such high energy shrapnel would be capable of destroying diagnostic equipment and damaging the first wall [34–36].

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 [37]. The most common unit is displacements per atom (dpa) [38]. 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 [38]. 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 [37–39]. 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) [40, 41] approaches. At the end of such cascades, we are often left with dislocation sources or prismatic dislocation loops [42]. Such loops can be used as inputs for Dislocation Dynamics (DD) simulations, which can explore greater temporal and spatial scales [31].

1.2.2.2 Transmutation

Transmutation products are one of the biggest sources of problems for materials in fusion applications [21, 22]. Not only do they tend to embrittle materials, they often also reduce their thermal conductivity [43]. The former presents significant challenges for structural materials [44]; the latter is especially egregious for energy extraction by limiting the divertor's ability to conduct heat, thus lowering the reactor's efficiency and causing thermal stresses due to the generation of hot spots that cannot be easily dissipated. As a result, understanding the mechanical and thermal behaviour of transmutation alloys is crucial for moving forward [45–47], but doing so requires knowledge of the time evolution of a reactor's components.

Since the chemical composition changes in a non-trivial way, so do the mechanical properties. Worse still are the long timescales over which this occurs—on the order of years or tens of years [48]. Even if we were able to experimentally irradiate fusion-relevant materials with appropriately energetic neutrons, it would take years before we could characterise their behaviour, and doing so would be problematic due to radioactive decay.

The way we go about addressing the time-dependent compositional change of a material is to model it. Culham Centre for Fusion Energy's (CCFE), FISPACT [49], takes an MC approach at calculating transmutation and decay products of a sample given certain conditions. The code utilises external data provided by the European Activation File, which provides cross-section and decay rate data for a wide range of isotopes [50]. Unfortunately, there is a lack of cross-section data for certain neutron energies that contribute in a non-negligible manner to a fusion environment. Interpolating to fill the gaps would not be appropriate as the data is non-smooth and subject to resonance peaks, so the solution is imperfect.

Transmutation products will always be problematic, but they are a fact of working with fusion-relevant materials. Fortunately, there are ways in which inclusions may be modelled via DD (section 1.4.4.2). DD may also be used to model dislocation movement and interaction within heterogeneous media (section 1.4.4) as is the case for oxide-dispersion-strengthened (ODS) steels; and transmutation alloys of Tungsten divertors, where Osmium, Rhenium and Tantalum clusters which prove highly problematic for its temperature conductivity and structural integrity [51–57].

1.3 Parallel computing

Processing chips are made up of millions or even billions of transistors acting as switches in logic gates. Each time a logic gate fires, the capacitors inside them charge and discharge at the chip's frequency otherwise known as clock speed [58]. Consider the power of any electronic component,

$$P(t) = I(t)V(t)\,, \tag{1.2}$$

where I is current and V is voltage, both as functions of time, t. The current, I, of a capacitor and the definition of power, P, as functions of time, t, as well as the definition of capacitance are given by,

$$I(t) = C\frac{\mathrm{d}V(t)}{\mathrm{d}t}\,, \qquad P(t) = \frac{\mathrm{d}E(t)}{\mathrm{d}t}\,, \qquad C = \frac{Q_\mathrm{c}}{V_\mathrm{c}} \tag{1.3}$$

where C is capacitance, E the energy stored in the capacitor, Q_c the charge stored in the capacitor and V_c the voltage of the capacitor. Substituting eq. (1.3) into eq. (1.2) and integrating twice, we obtain the expression for the energy stored in a capacitor,

$$\int_0^{E_c}\int_0^{\infty} \frac{\mathrm{d}E(t)}{\mathrm{d}t}\,\delta t = C\int_0^{V_c}\int_0^{\infty} V(t)\frac{\mathrm{d}V(t)}{\mathrm{d}t}\,\delta t\,, \tag{1.4a}$$

$$E_\mathrm{c} = \frac{CV_\mathrm{c}^2}{2} = \frac{Q_\mathrm{c}V_\mathrm{c}}{2} = \frac{Q_\mathrm{c}^2}{2C}\,, \tag{1.4b}$$

where V_c, E_c, Q_c are the voltage, energy and charge stored in the capacitor. Recalling that in a processing chip, the capacitors are charged and discharged at the chip's clock speed f, we arrive at the expression for power consumption, P_c, of a capacitor charging and discharging at frequency f [59],

$$P_\mathrm{c} = E_\mathrm{c} f \propto CV_\mathrm{c}^2 f\,. \tag{1.5}$$

Computer chips are rather more complicated, but their power consumption (also known as power dissipation) is described by a simple addition of terms [60],

$$P = P_\mathrm{dyn} + P_\mathrm{sc} + P_\mathrm{leak}\,, \tag{1.6}$$

where P_dyn is the dynamic power dissipation given by eq. (1.5). The two other dissipation mechanisms are: *1*) short-circuit, sc, which depends on frequency and occurs when a direct path from transistor to ground is made as a result of multiple transistors conducting simultaneously; and *2*) leakage, leak, which depends on the voltage and is due to micro-currents between doped parts a transistor. Furthermore, higher voltages and frequencies result in higher temperatures, which in turn mean decreased transistor performance and increased capacitance. The overall result is an energy expenditure curve similar to fig. 1.2 [61] for every processor unit.

This set of optimum conditions is the reason behind the multicore and multi-threaded design of modern Central Processing Units (CPUs) and Graphics Processing Units (GPUs). It is also why in recent years there has been such a massive push for parallelism in all computing markets. It is simply not feasible to continually increase clock speeds and voltages because cooling solutions would struggle to remove heat fast enough, and power consumption would skyrocket.

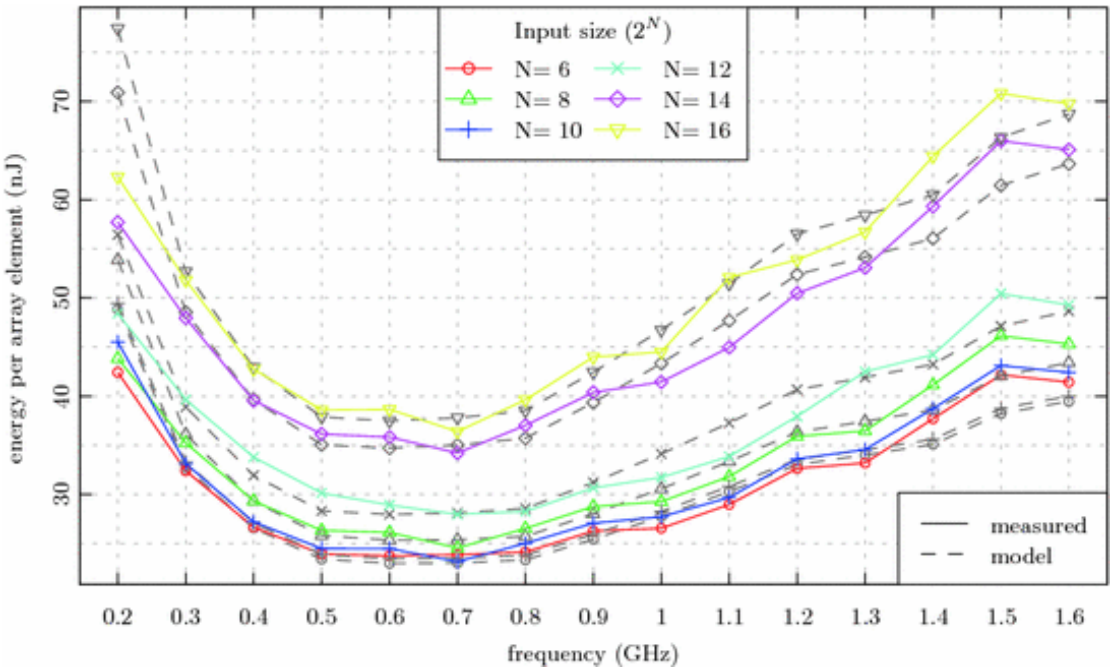


Figure 1.2: Energy required by a Samsung Galaxy S2 CPU at 37 °C to complete the Gold-Rader implementation of the *bit-reverse* algorithm. The dashed lines denote a theoretical model, image and model found in [61].

1.3.1 Computation on graphics processing units

Central Processing Units (CPUs) are designed not only to perform mathematical operations but logical ones that control program flow. They are tailored to perform the wide variety of operations required by an operating system. These include program scheduling (instruction priorisation, load balancing), instantiation (program loading, unloading, loops, recusion instances) & branching (if/case statements, go to's), memory operations (fetching, storing, allocation), input/output (IO), and program monitoring (program counters, recursion counters). Modern CPUs have a degree of parallelism and asyncronicity that allows them to increase their total throughput while keeping their operation within near optimal conditions. They are commonly divided into cores and threads, though certain high-end chips have an additional layer named hyperthreading [63].

Given the limited scope of the first computers, the general purpose of CPUs was enough to cover their needs. However, with the advent of personal computers, the demands on CPUs drastically increased. For industrial users these revolved around data acquisition, filtering and preprocessing [64–66]. On the other hand, the domestic market demanded ever-increasing levels of abstraction and usability in the form of Graphical User Interfaces (GUIs) such as windows, cursors and sound effects which originally acted as a replacement for haptic feedback. Manufacturers identified this need and moved to provide specialised modules for such

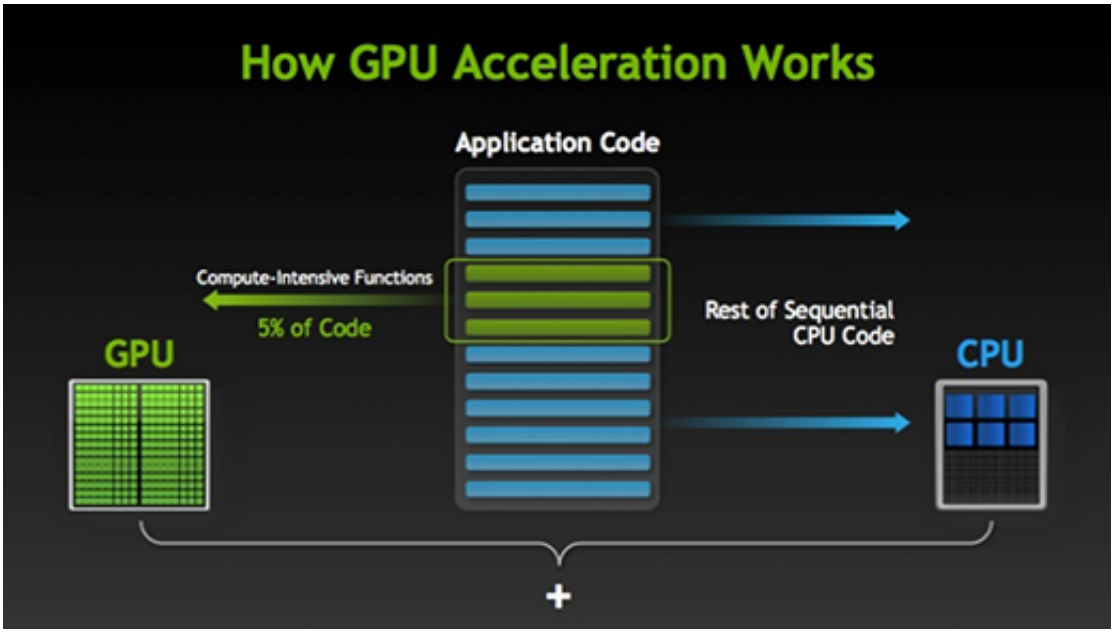


Figure 1.3: CUDA runtime schematic. Repetitive but computationally intensive tasks can be offloaded to the GPU. Both GPU and CPU are completely indipendent from each other and can work on different parts of the code at the same time, care must be taken to ensure proper syncronisation. Image taken from [62].

tasks. This freed CPU resources and improved user experience by accelerating different processes through hardware means [67–70]. These modules include Sound Cards (SCs), Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), Application Specific Integrated Circuits (ASICs), Cryptographic Accelerators, Regular Rxpression (RegEx) Accelerators, among others. These external pieces of specialised hardware are collectively dubbed *hardware accelerators*.

Graphics Processing Units were originally intended to offload the very data-intensive but computationally simple operations needed for 3D gaming and rendering [67–69]. Graphics processing was a prime candidate for hardware acceleration because the operations on each pixel are largely the same across the screen. Due to their original purpose as gaming and rendering accelerators, they were never designed to operate on higher than single precision data. In fact, single precision (32-bit precision) is still good enough to encode 32-bit colour depth (8-bit channel per RGB colour + 8-bit alpha channel), which only the most high-end monitors support [71]. It is also worth noting that because the same operations apply to different pieces of (mostly) independent data, they can all be performed at the same time, often trivially reducing the order of polynomial complexity algorithms. For example, an $\mathcal{O}(N)$ process can be reduced to $\mathcal{O}(1)$ if all N instances fit in a single parallel execution and are independent of one another.

As previously mentioned, GPU parallelism frees up enormous amounts of CPU processing power that can be put to good use running other programs or perform-

ing complementary serial processes, while the GPU works concurrently on its dataset. Parallelism has been used by the scientific community for years, but the focus has mainly been on CPU parallelism [72]. Consequently, its scope was lagerly limited to the use of computational clusters. The two main reasons for this were the fact that GPUs lacked support for higher precision arithmetic, and the very limited to non-existent support for scientific computing in languages such as OpenGL [73].

It wasn't until the development of the OpenCL and OpenACC standards that GPUs caught the attanetion of the scientific community as a viable way of exploiting parallelisation without access to a computing cluster.

OpenCL allows one to work with heteragoenous systems and is similar to C in that it's very low level. It works on a wide range of hardware accelerators and is therefore useful for many scientific and engineering applications, but it's also relatively hard to use. One can make use of libraries written in OpenCL to facilitate development, but it is still a fully fledged C-type language [74].

OpenACC is similar to OpenMP in that they both use pragmas[1] and they both work in shared memory environments—same GPU and same CPU respectively. This is no coincidence as OpenACC was designed as an extension of OpenMP for developing parallel applications on hardware accelerators. Unfortunately, being pragma based, the standard requires significant work by compiler manufacturers, so its adoption has therefore been slow [75]. Furthermore, despite simplifying development and minimising the barrier to entry, the use of pragmas limits the flexibility and adaptability of the framework compared to OpenCL.

Recently however, a third option has become viable. NVidia's Computer Unified Device Architecture (CUDA) framework provides the best aspects of both OpenCL and OpenACC. The tradeoff is that it only works on NVidia GPUs and is a closed source product. However, the accessibility and flexibility of CUDA provides anyone familiar with C/C++ the means to develop a GPU application with little issue. NVidia is also strongly backing scientific research by adding double precision support on their GPUs. They have also worked to provide parallel equivalents of well known serial libraries—such as cuBLAS & cuFFT—for scientific computing. They have additionally developed a wide range of specialist graphics cards tailor-made for scientific purposes. As such, they are the leaders in GPU computing in scientific communities [62].

[1]Pragmas are especially formated comments that specific compilers recognise and turn into special instructions at compile time. They are not part of the programming language's official standard, but rather compiler-specific extensions. Pragmas are usually programmed in C or Assembly and must be implemented by the compiler manufacturer. Nothing prevents application developers from writing the would-be pragma's code directly into their application, but this can prove a lengthy and difficult process. The rigid nature of pragmas, as well as the compiler manufacturer's priorities, limit their scope and usability.

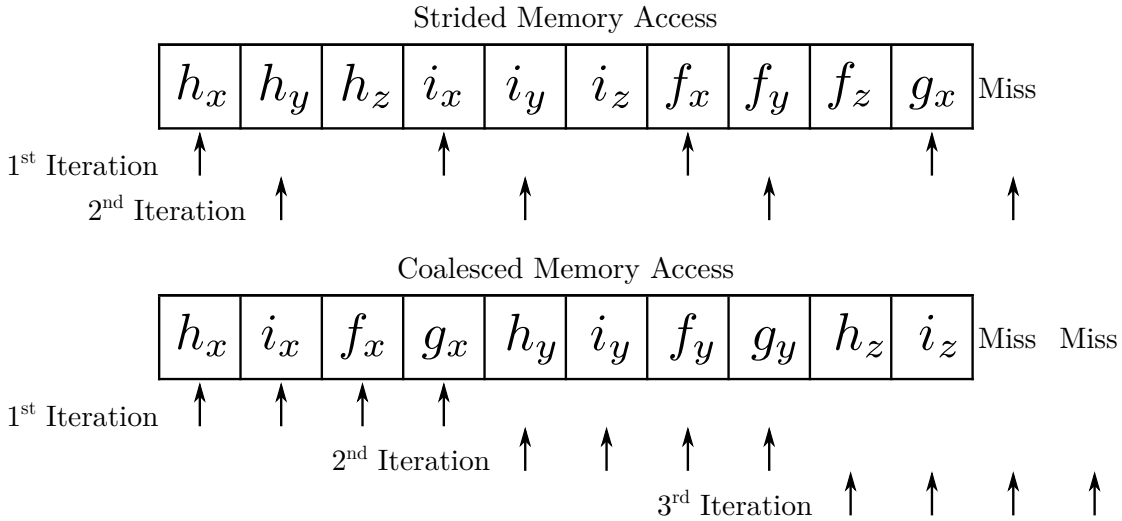


Figure 1.4: Arrows represent fetch requests by single threads in a GPU. Data should be arranged in such a way that all threads in a warp (collections of 32 threads) can simultaneously access contiguous memory locations, reducing cache-misses and improving throughput. This access pattern can be unintuitive when parallelising serial algorithms, but is extremely important, especially when using scientific computing cards, as they are optimised for long computation times and low memory fetch frequency.

1.3.1.1 Hurdles for parallelisation

The difficulty in parallelisation varies tremendously from problem to problem. Problems where data is uncorrelated and independent—such as sampling well-behaved probability distributions—are almost trivially parallelisable. Problems where data is correlated or strongly dependent on its neighbours—such as Dislocation Dynamics—require a more careful approach [76].

The largest hurdle when implementing parallel algorithms is often the efficient use of data. In order to obtain good parallel performance, a lot of thought has to be placed on data access patterns, data read/write conflicts, and memory allocation and transfer [62]. For best performance, all this must be analysed on a case by case basis. If done incorrectly, the performance of a parallel application may be lower than the serial version. One must consider a wide range of parameters to successfully parallelise a problem. Among these are GPU architecture, problem size, computational and memory complexity, code branching, required arithmetic precision, and error tolerances [62, 77].

Efficient parallelisation of many problems requires *coalesced memory access* as shown in fig. 1.4, which means we have to be extremely careful when mapping CPU memory to global GPU (device) memory. The fact that threads work "simultaneously"[2] means that in order to obtain good performance, data which is to

[2]Not quite but essentially simultaneously. See [62] for details.

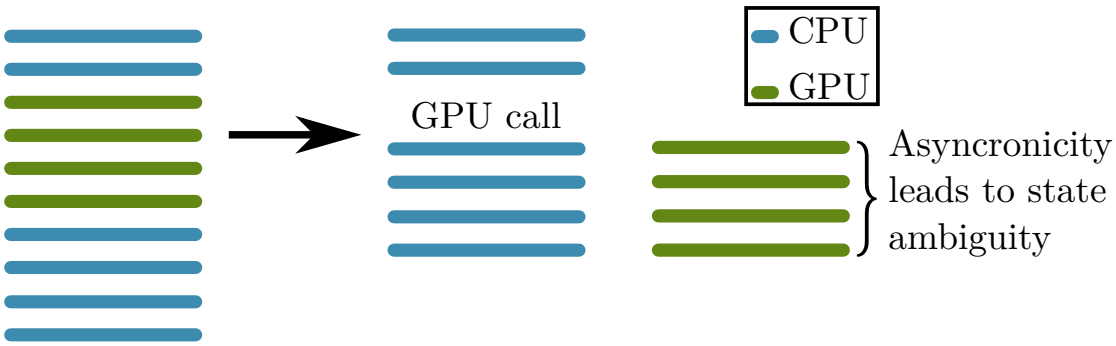


Figure 1.5: GPU and CPU code run independent of each other. This can lead to state ambiguities if both sides need to be aware of one another [62].

be simultaneously loaded into each thread must be contiguous. This maximises cache memory use and therefore reduces slow memory fetch operations.

Special cases, such as having a parallel dislocation line segment to a surface, as discussed in section 1.4.3, must be treated carefully due to the way code branching works in GPUs. There are various ways of doing so: *1*) if the special case is inexpensive, it can be treated within the same GPU function; *2*) if the special case is expensive and always known (such as certain boundary conditions in FEM), it can be placed in its own GPU function that treats it separately; *3*) if the special case is expesive and only found at runtime, it may be asynchronously treated by the CPU or buffered into its own GPU function to be executed at a later time.

One of the advantages of GPU-CPU independence is that both can work concurrently on different aspects of the problem. If both systems have to talk to each other, or there are any race conditions (one process needs to finish before the other can start), then one must tread carefully, ensuring proper syncronisation and data mapping before proceeding (see fig. 1.5).

The reason why code branching is bad for GPU parallelisation is down to the fact that they work like software-customisable vector machines. As in vector processors, collections of threads all carry out the same operation on different pieces of data at the same time. The threads essentially behave as one, operating on different data. This is advantageous to a GPU because it means more energy and space can be used for computation rather than logic and scheduling. However, this eliminates the ability of code to branching within each collection of threads. This means that every branch of an `if` or `switch-case` statement will be executed whether a condition is met or not[3]. Only data storage depends on whether a condition is true or false, as illustrated in fig. 1.6.

[3]If the branch can be resolved at compile time, it is possible for the compiler to prune it as part of compile time optimisation. However, this can be unreliable and depends on how transparent the optimsation is to the compiler, how agressive the optimisation setting is, and how the compiler is implemented. Therefore, relying on compile time optimisation to prune branches is not recommended.

```
if (y == 0){
    x = a+b;
}
else if (y == 1){
    x = a*b;
}
else{
    x = a/b;
}
```

(a) CPU code will only execute if the condition is met. This is called code branching.

```
p = (y == 0);
p: x = a+b;

p = (y == 1);
p: x = a*b;


!p: x = a/b;
```

(b) GPU code executes every line but only stores results if the flag before the colon is true.

Figure 1.6: The NVidia CUDA compiler replaces `if` and `case` statements with logical flags `p`. Every line is executed, but the data is only stored if the flag prior to the colon is true [62]. This means that having rare but computationally expensive special cases will tank parallel performance and must therefore be dealt with separately.

Dislocation Dynamics (DD)—in particular 3D Discrete Dislocation Dynamics (3D DDD)—can greatly benefit from parallelisation, especially when coupling to Finite Element Methods (FEM). It is worth noting that there are potential issues arising from the very computationally expensive functions and special cases that often arise from analytical solutions in 3D DDD. There are also potential issues with data redundancy—which are non-limiting in the short term—that may eventually require a more data-efficient approach as the computational capabilities and data capabilities of GPUs converge. Section 1.4.5 expands on these issues in the context of DD.

1.4 3D dislocation dynamics modelling

The plastic deformation of materials is generally governed by the generation and motion of line defects known as dislocations through the crystal lattice. Microstructural features such as grain boundaries, precipitates and inclusions impede dislocation motion causing strengthening but often limiting ductility [78]. Understanding the behaviour of the dislocation ensemble is highly complicated even when ignoring dislocation-microstructure interactions. However, if we want to truly comprehend their real-world behaviour, we cannot limit ourselves to idealised scenarios.

One of the most often used parametrisations of DD is Discrete Dislocation Dynamics (DDD). Where dislocations are parametrised as a series of nodes linked by straight line segments. This reduces computational requirements and allows

for analytic solutions to be obtained. At present, neither DDD nor Finite Element (FE) models can truly handle all the complexities of real-world alloys [79–82]. For one, DDD relies on assuming a linear-elastic isotropic solid domain with periodic or infinite boundary conditions, while FE relies on assuming continuum properties inside a *finite* domain.

Crystal Plasticity Finite Element Methods (CPFEM) use constitutive equations to calculate dislocation motion and generation on a set of slip systems. These are given as dislocation densities and thus can still be considered "bulk" models because there are no explicit dislocations interacting with one another [83]. CPFEM can handle finite strains and anisotropy but not strain localisation/slip bands, etc.

Furthermore, dislocations often accumulate in the vicinity of microstructural features. The interaction of dislocations with microstructure and other dislocations can lead to work hardening and local stress concentrations that can lead to failure initiation [84, 85]. Hardware accleration, i.e. using Graphics Processing Units (GPUs), has been shown to be very effective in DDD [86], and has the potential to enable the simulation of much larger numbers of dislocations for longer timescales. In order to study these phenomena we must find a way of coupling DDD and FEM into a single multiscale model with the potential to simulate micromechanical tests more accurately than CPFEM. With a model such as this, we may potentially be able to predict and observe emergent phenomena/properties, explain michromechanical behaviours, and even predict and explain experimental results based on underlying dislocation mechanisms[4].

1.4.1 Coupling dislocation dynamics to finite element methods

Coupling dislocation dynamics (DD) to FEM is important to properly simulate micromechanical tests because DD provides us with a more precise set of inputs and greater granularity for solving the FE problem. There are at present two methods with which to do so, the Superposition model and the Discrete continuum model. Discrete dislocation dynamics, where the dislocation is broken into nodes connected by segments is a relatively compuationally- and data-efficient implementation of dislocation dynamics, which has the added advantage of having analytical solutions for straight-line segments. However, as discussed in section 1.4.1.3, explicit discretisation is not the only way to tackle the problem.

[4]Though predicting and replicating experimental results requires the initial conditions of the experiment and simulation be roughly equivalent.

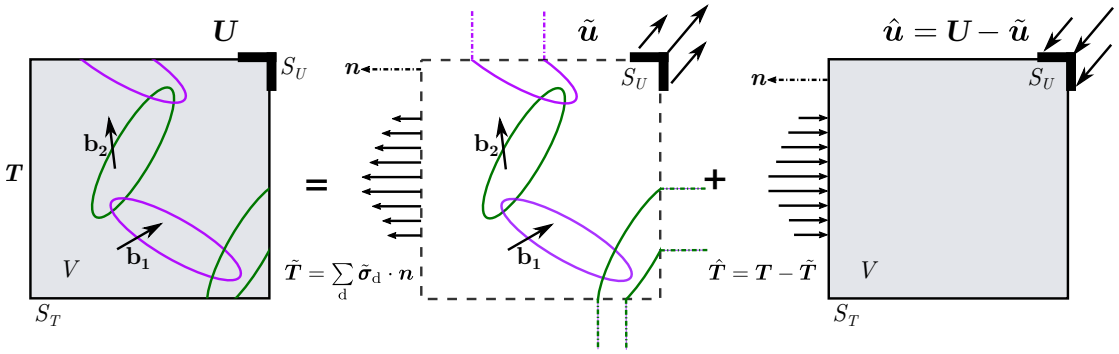


Figure 1.7: The superposition used to couple DDD and FEM. The volume V is bounded by a surface $S = S_T \cup S_U$ and contains a dislocation ensemble and is subjected to tractions $\boldsymbol{T}$ on S_T and $\boldsymbol{U}$ on S_u. First, the traction, $\tilde{\boldsymbol{T}}$, and displacement, $\tilde{\boldsymbol{u}}$, 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 $\hat{\boldsymbol{u}} = \boldsymbol{U} - \tilde{\boldsymbol{u}}$.

1.4.1.1 Superposition model

The Superposition Model (SM) works by decomposing the problem into separate DDD and FE problems (fig. 1.7). It is assumed that a linear-elastic body V bounded by a surface S is subject to traction boundary conditions, $\boldsymbol{T}$, on S_T and displacement boundary conditions, $\boldsymbol{U}$, on S_u. The formulation proposed in [87] to impose traction-displacement boundary conditions on DDD problems in finite domains states that the total displacement and stress fields can be written as a superposition of displacement and stress fields obtained from DDD and FE,

$$\boldsymbol{U} = \tilde{\boldsymbol{u}} + \hat{\boldsymbol{u}}\,, \tag{1.7a}$$

$$\boldsymbol{\sigma} = \tilde{\boldsymbol{\sigma}} + \hat{\boldsymbol{\sigma}}\,. \tag{1.7b}$$

The, $(\tilde{\ })$, fields are those associated with the dislocation in an infinite medium and are obtained by evaluating analytic fields in a DDD simulation. The corrective, $(\hat{\ })$, fields are those which must be superimposed to ensure the boundary conditions are met. This means that the image fields can be obtained by running a FE simulation with the "corrected" displacement and traction fields,

$$\hat{\boldsymbol{u}} = \boldsymbol{U} - \tilde{\boldsymbol{u}}\,, \tag{1.8a}$$

$$\hat{\boldsymbol{\sigma}} \cdot \boldsymbol{n} = \hat{\boldsymbol{T}} = \boldsymbol{T} - \underbrace{\tilde{\boldsymbol{T}}}_{\tilde{\boldsymbol{\sigma}} \cdot \boldsymbol{n}}\,, \tag{1.8b}$$

where $\boldsymbol{n}$ is the outer unit normal vector to S. As a dislocation segment moves closer to the surface, its $(\tilde{\ })$ field diverges and starts causing numerical problems [88]. Section 4.1.1 is a more in-depth discussion on the superposition method and

dislocation-induced surface tractions.

A further problem with this method is that modelling elastic inclusions not only requires the calculation of forces induced by dislocations on the inclusion's surface, but also demands the calculation of so-called polarisation stresses due to differences in the inclusion's and matrix elastic properties [87–89].

That said, the relative simplicity of the superposition method has made it a popular choice [90–92] for coupling DDD and FEM because all it requires is the calculation of forces and displacements on the boundaries (see section 1.4.3).

1.4.1.2 Discrete continuum model

The Discrete Continuum Model (DCM) takes an alternative approach to solving the same coupling problem. The DCM only treats short-range dislocation-dislocation interactions analytically while all other interactions are numerically calculated via FEM [93]. It is based on the regularisation of the atomic displacement jump across the slip plane into a plastic strain inclusion according to eigenstrain theory [94]. Like the Superposition Model, the DCM also assumes the simulated volume to be linear-elastic.

The eigenstrain formalism assumes that material defects can be represented as stress-free strain distributions, dubbed *eigenstrains* [94]. For example, a dislocation loop of any shape may be approximately represented by thin, coherent, plate-like inclusions with the same contour as the loop and a characteristic thickness, h, as shown in fig. 1.8 [93]. The eigenstrain tensor, ϵ^{p}, can then be defined as a symmetric dyadic product of the Burgers vector, $\boldsymbol{b}$, and slip plane normal $\boldsymbol{n}$,

$$\boldsymbol{\epsilon}^{\mathrm{p}} \equiv \frac{1}{2h}(\boldsymbol{b} \otimes \boldsymbol{n} + \boldsymbol{n} \otimes \boldsymbol{b})\,. \tag{1.9}$$

Equation (1.9) can be used to calculate approximate elastic fields from the stress-free eigenstrain distribution. The approximation is accurate far from the dislocation core [95]. As we move closer—but still outside the core region—the approximation tends toward the exact discrete dislocation solution as $h \to 0$. From linear elasticity, the total plastic strain is the sum of the individual plastic strains due to each dislocation segment.

The formalism then lets us solve the boundary value problem by finding the stress tensor $\boldsymbol{\sigma}$, elastic strain $\boldsymbol{\epsilon}^{\mathrm{e}}$ and displacement $\boldsymbol{u}$ in mechanical equilibrium with the boundary conditions and plastic strain distribution $\boldsymbol{\epsilon}^{\mathrm{p}}$ from the "inclusions".

It is worth noting that in order to accurately calculate the eigenstrains, there must be sufficient FE nodes inside the thin plate. Therefore, smaller values of h necessitate finer FE meshes. Furthermore, the original DCM experienced numerical blow up as dislocation-dislocation distances approached h [96], but this has

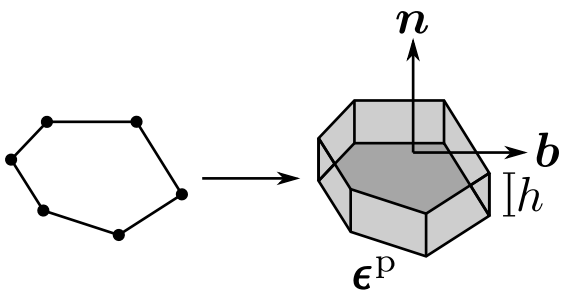


Figure 1.8: The eigenstrain formalism as defined in [94]. The dislocation is being approximated by a thin coherent inclusion of thickness h, whose strain-free stress tensor $\boldsymbol{\sigma}^{\mathrm{p}}$ as defined by eq. (1.9). The vectors $\boldsymbol{n}$ and $\boldsymbol{b}$ are the normal vector to the slip plane and the dislocation line's Burgers vector.

since been addressed. Assuming linear-elasticity, the revised model [93] yields,

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{f} = \boldsymbol{0} \quad \in V \setminus \{A\}\,, \tag{1.10a}$$

$$\boldsymbol{\sigma} = \boldsymbol{E} : \boldsymbol{\epsilon} \quad \in V \setminus \{A\}\,, \tag{1.10b}$$

$$[[\boldsymbol{u}]] \quad \text{across } \{A\}\,, \tag{1.10c}$$

$$\boldsymbol{u} = \boldsymbol{u}_{\mathbf{o}} \quad \in S_U\,, \tag{1.10d}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{T} \quad \in S_T\,. \tag{1.10e}$$

At time t, $\{A\}$ denotes the area swept by the dislocation loops since the start of the simulation. $[[\boldsymbol{u}]]$ denotes displacement jumps tangent to $\{A\}$ due to dislocation glide; its magnitude and direction depend on the Burgers vector $\boldsymbol{b}$. $\boldsymbol{E}$ is the 4^{th}-order elasticity tensor, $\boldsymbol{\sigma}$ the small strain tensor, $\boldsymbol{f}$ are the body forces and, :, is the double dot product defined as, $(\boldsymbol{E} : \boldsymbol{\epsilon})_{ij} = E_{ijkl}\epsilon_{kl}$, between a rank 4 tensor ($\boldsymbol{E}$) and a rank 2 tensor ($\boldsymbol{\epsilon}$). The operator, $\setminus$, is the set difference defined as, $A \setminus B = \{x \in A | x \notin B\}$. Equation (1.10) can then be linearly decomposed three parts which are solved via FEM or DDD and coupled as described in [93].

It is worth noting that the DCM is *substantially* more complicated than the SM. By requiring the finite elements be small enough for the eigenstate formalism to work, it strongly couples DDD to the FE model and software implementation. It shifts the brunt of the computational workload from DDD to FEM. Essentially trading computationally intensive, long-range interactions which scale on the order of $\mathcal{O}(N^2)$, where N is the number of dislocation line segments; for computationally intensive tasks scaling on the order of $\mathcal{O}(M^3)$ where M is the number of finite elements in 3D (assuming linear cubic elements). Furthermore, it cannot be used with BE methods as the eigenstrain formalism demands the use of internal elements rather than simply requiring a surface mesh.

That said, the DCM allows for reductions in the computational complexity of certain parts of the problem which would otherwise have to be done via DDD [93];

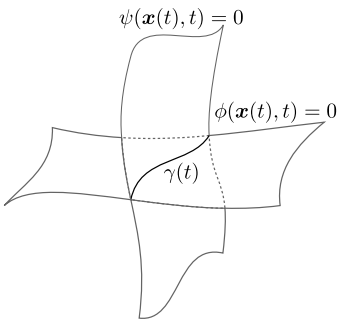


Figure 1.9: Diagram of a continuous dislocation $\gamma(t)$ as the intersection of the zero level of two level set functions $\phi(\boldsymbol{x}(t),t),\ \psi(\boldsymbol{x}(t),t)$ as posed by eq. (1.11). Image edited from [98].

namely long-range dislocation-dislocation or dislocation-surface interactions via an interaction distance cutoff that is only possible with the eigenstrain formulation. Compared to the superposition method, gains in computational efficiency grow as dislocation density increases with respect to the number of finite elements. Since dislocation-dislocation and dislocation-surface interactions are among the most computationally expensive aspects of DDD, the DCM can reduce the overall computational cost of simulations with large enough dislocation densities and fine enough finite element meshes.

1.4.1.3 Level set method

Even though the level set method has not strictly been used to couple DD to FEM, it has been used to model inclusions [97], as described in section 1.4.4.2. This type of dislocation dynamics is fundamentally distinct from DDD because Level Set DD uses arbitrary functions rather than a discretisation approach to represent dislocations lines. The idea is that in 3D, a dislocation $\gamma(t)$ can be represented by the intersection of two zero levels of two level set functions (see fig. 1.9),

$$\phi(x(t),y(t),z(t),t)=0\,, \qquad \psi(x(t),y(t),z(t),t)=0\,, \tag{1.11a}$$

$$\frac{\mathrm{d}\phi}{\mathrm{dt}}=\partial_t\phi+\boldsymbol{v}\cdot\nabla\phi=0\,, \qquad \frac{\mathrm{d}\psi}{\mathrm{dt}}=\partial_t\psi+\boldsymbol{v}\cdot\nabla\psi=0\,, \tag{1.11b}$$

where $\boldsymbol{v}$ is the dislocation velocity. This definition uses the material derivative because it is assumed that the function and its spatial coordinates are all functions of time, t. The material derivative also comes up when deriving the Navier-Stokes equations of fluid dynamics.

The level set method is significantly more computationally expensive than DDD [98]. It must solve two coupled quasi-linear partial differential equations whose character is in general undefined. It therefore requires the use of very ro-

bust numerical solvers such as higher order Total Variation Deminishing (TVD) Runge-Kutta methods for time discretisation (TVD-RK4 or higher), and high order ENO (Essentially Nonoscillatory $\mathcal{O}(N^5)$ or higher) or WENO (Weighted Essentially Non-Oscilatory) interpolation methods for spatial discretisation [99] for the solutions to converge satisfactorily. The method also makes it impossible to obtain many of the useful analytical solutions one can find by discretising dislocations into straight line segments. Another problem is that this method has never been truly coupled to the finite element method. However, if one were to do so there are two naïve ways of going about it: *1*) numerically integrating along the dislocation lines and finding a way to numerically calculate the forces on FE nodes, or *2*) discretising the level set curves and using either the DCM or SM to find forces and displacements. The former would be very computationally expensive. Furthermore, no one has found a way to compute the forces or displacements exerted by generally shaped, non-discretised dislocations, numerical solutions are possible as they are needed to find self-interaction energies and dislocation-dislocation forces [98] but general, closed-form analytical ones are most likely impossible to find. Both approaches defeat the purpose of the level set method, necessitating some form of discretisation whilst keeping the high computational cost. As a result of this, the level set method is not nearly as popular as DDD utilising either the SM or DCM.

1.4.2 Non-singular continuum theory of dislocations

The Peach-Koehler formula describes the fundamental interactions of dislocations [100] according to the force, $\boldsymbol{f}$, that a local stress $\boldsymbol{\sigma}$ exerts on a dislocation line with Burgers vector, $\boldsymbol{b}$, and line direction, $\boldsymbol{\xi}$,

$$\boldsymbol{f} = (\boldsymbol{\sigma} \cdot \boldsymbol{b}) \times \boldsymbol{\xi} \,. \tag{1.12}$$

According to [101], the internal stress field of a dislocation loop in a homogenous, infinite, linear-elastic medium is given by the contour integral around a loop L,

$$\sigma_{ij}^{\infty}(\boldsymbol{x}) = C_{ijkl} \oint_L \epsilon_{lnh} C_{pqmn} \frac{\partial G_{kp}(\boldsymbol{x} - \boldsymbol{x}')}{\partial x_q} b_m \, \mathrm{d}x'_h \,, \tag{1.13}$$

where C_{ijkl} is the elastic stiffness tensor, ϵ_{lnh} the permutation operator, $\boldsymbol{b}$ the Burgers vector, and $G_{kp}(\boldsymbol{x} - \boldsymbol{x}')$ is Green's function of elasticity [101]. $G_{kp}(\boldsymbol{x} - \boldsymbol{x}')$ is defined as the displacement component in the x_k direction at point $\boldsymbol{x}$ in response to a unit point force applied in the x_p direction at point $\boldsymbol{x}'$ [102]. In an isotropic

elastic solid, $G(\boldsymbol{x}-\boldsymbol{x}')$, takes the form,

$$G_{ij}(\boldsymbol{x}-\boldsymbol{x}') = \frac{1}{8\pi\mu}\left[\delta_{ij}\partial_{pp} - \frac{1}{2(1-\nu)}\partial_{ij}\right] R(\boldsymbol{x}-\boldsymbol{x}')\,, \tag{1.14}$$

where μ, ν are the isotropic shear modulus and Poisson's ratio respectively, δ_{ij} is the Kronecker Delta, $\partial_{x_1\dots x_n} \equiv \dfrac{\partial^n}{\partial x_1 \dots \partial x_n}$, and $R = \|\boldsymbol{x}-\boldsymbol{x}'\|$. However, considering that $\boldsymbol{x} \to \boldsymbol{x}' \Rightarrow R \to 0 \Rightarrow \partial_i R \to \infty$, some (or all) components of the stress field diverge. Furthermore, the total elastic energy also diverges,

$$E = \frac{1}{2}\int S_{ijkl}\sigma_{ij}(\boldsymbol{x})\sigma_{kl}(\boldsymbol{x})\ \mathrm{d}^3\boldsymbol{x}\,, \tag{1.15}$$

where $S = C^{-1}$ is the elastic compliance tensor. These divergent properties often prove problematic when numerically computing dislocation forces and energies. Equation (1.15) can also be expressed as a double line integral [103, 104],

$$\begin{aligned} E = &-\frac{\mu}{8\pi}\oiint_L \partial_k\partial_k R(\boldsymbol{x}-\boldsymbol{x}')b_i b'_j\ \mathrm{d}x_i\ \mathrm{d}x'_j \\ &-\frac{\mu}{4\pi(1-\nu)}\oiint_L \partial_i\partial_j R(\boldsymbol{x}-\boldsymbol{x}')b_i b'_j\ \mathrm{d}x_k\ \mathrm{d}x'_k \\ &+\frac{\mu}{4\pi(1-\nu)}\oiint_L \partial_k\partial_k R(\boldsymbol{x}-\boldsymbol{x}')b_i b'_i\ \mathrm{d}x_j\ \mathrm{d}x'_j \\ &-\nu\oiint_L \partial_k\partial_k R(\boldsymbol{x}-\boldsymbol{x}')b_i b'_j\ \mathrm{d}x_j\ \mathrm{d}x'_i\,, \end{aligned} \tag{1.16}$$

which is important when describing how Cai et al. [102] derived their non-singular expression.

The singularity in R is the result of unreasonably and unphysically assuming a dislocation loop's Burgers vector distribution is a Delta function. The assumption was made to allow for closed-form and relatively simple expressions. As noted in [105], other distributions may be used but they either result in significantly more complicated expressions or destroy the analytical nature of the classical formulation.

There have been many attempts at removing this singularity, including finite-strain elasticity [106], non-local and gradient elasticity [107, 108], interaction cut-off radius [100], average stress at two points on opposite sides of the dislocation line [109, 110], and spreading the Burgers vector distribution out over a finite width [105, 111, 112]. Unfortunately all of these approaches failed in one way or another [102]. Depending on the approach, various undesirable qualities may present themselves, including *1*) inconsistencies with other theories; *2*) impractical implementation; *3*) lack of closed-form solutions for non-straight finite disloca-

tions; *4*) lack of self-consistency; and *5*) the possibility for multiple non-degenerate expressions and solutions for the line integral of the dislocation line energy.

Cai et al. [102] took it upon themselves to define and justify a different Burgers vector distribution that maintains the mathematical convenience of the classical formulation that also eliminates such an unphysical assumption. They did so by introducing a Burgers vector density function,

$$\boldsymbol{b} = \int \boldsymbol{g}(\boldsymbol{x}) \, \mathrm{d}^3\boldsymbol{x} \,, \tag{1.17a}$$

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{b}\tilde{w}(\boldsymbol{x}) = \boldsymbol{b}\tilde{w}(r) \,, \tag{1.17b}$$

where $r \equiv \|\boldsymbol{x}\|$. When substituting eq. (1.17a) into eqs. (1.13) and (1.16). The result of multiplying R with components of $\boldsymbol{b}$ results in the following integrals,

$$R(\boldsymbol{x} - \boldsymbol{x}')b_m = \int R(\boldsymbol{x} - \boldsymbol{x}'')g_m(\boldsymbol{x}'' - \boldsymbol{x}') \, \mathrm{d}^3\boldsymbol{x}'' \,, \tag{1.18a}$$

$$R(\boldsymbol{x} - \boldsymbol{x}')b_m b'_n = \iint R(\boldsymbol{x}'' - \boldsymbol{x}''')g_m(\boldsymbol{x} - \boldsymbol{x}'')g_n(\boldsymbol{x}''' - \boldsymbol{x}') \, \mathrm{d}^3\boldsymbol{x}'' \, \mathrm{d}^3\boldsymbol{x}''' \,. \tag{1.18b}$$

Using eqs. (1.17) and (1.18) as guidelines, they defined the following convolutions,

$$w(\boldsymbol{x}) \equiv \tilde{w}(\boldsymbol{x}) * \tilde{w}(\boldsymbol{x}) = \int \tilde{w}(\boldsymbol{x} - \boldsymbol{x}')\tilde{w}(\boldsymbol{x}') \, \mathrm{d}^3\boldsymbol{x}' \,, \tag{1.19a}$$

$$R_a \equiv R(\boldsymbol{x}) * w(\boldsymbol{x}) = \int R(\boldsymbol{x} - \boldsymbol{x}')w(\boldsymbol{x}') \, \mathrm{d}^3\boldsymbol{x}' \,. \tag{1.19b}$$

At which point they assumed there exists an integrable function $w(\boldsymbol{x})$ such that

$$R_a = \sqrt{R(\boldsymbol{x})^2 + a^2} = \sqrt{x^2 + y^2 + z^2 + a^2} \,, \tag{1.20}$$

where a is an arbitrary constant meant to represent the dislocation core radius, whose value may be estimated from atomistic simulations. This is essentially a definition that can be used to replace R in the classical equations and eliminate the singularity by including a free parameter. However, in order to ensure this is mathematically sound, there must indeed exist a function which yields R_a as Cai et al. [102] defined it. This can be done by making use of the following property for the convolution of two suitably differentiable functions, $\partial_i(f * g) = \partial_i f * g = f * \partial_i g$.

We may take the Laplacian twice,

$$\nabla^2[\nabla^2\{R(\boldsymbol{x}) * w(\boldsymbol{x})\}] = \nabla^2[\nabla^2 R_a(\boldsymbol{x})]\,, \tag{1.21a}$$

$$\nabla^2[\nabla^2\{R(\boldsymbol{x})\}] * w(\boldsymbol{x}) = \nabla^2[\nabla^2 R_a(\boldsymbol{x})]\,, \tag{1.21b}$$

$$\nabla^2[\nabla^2\{R(\boldsymbol{x})\}] = \nabla^2\left[\frac{2}{R}\right] = -8\pi\delta^3(\boldsymbol{x})\,, \tag{1.21c}$$

$$\nabla^2[\nabla^2 R_a(\boldsymbol{x})] = \nabla^2\left[\frac{2}{R_a} + \frac{a^2}{R_a^3}\right] = -\frac{15a^4}{R_a^7}\,, \tag{1.21d}$$

$$w(\boldsymbol{x}) = \frac{15a^4}{8\pi R_a^7}\,. \tag{1.21e}$$

Equation (1.21c) is a very brave statement given that,

$$\nabla^2\left[R^{-1}\right] = \nabla^2\left[(x^2+y^2+z^2)^{-1/2}\right] = 0/\,, \tag{1.22}$$

but may be physically justified by pretending $\nabla^2\left[R^{-1}\right]$ arises from assuming a spherical surface of radius 0. Equation (1.21e) is a similarly dubious statement that can be justified by noting that,

$$\lim_{a\to 0} w(\boldsymbol{x}) = \delta^3(\boldsymbol{x})\,. \tag{1.23}$$

Nevertheless, the non-singular formulation by Cai et al. [102] fixes the physically and mathematically problematic assumption that Burgers vectors follow 3D Dirac delta distributions, and proves useful in producing analytical expressions (see section 1.4.3) that are not much more complex than the singular case.

1.4.3 Analytical forces exerted by a dislocation line segment on surface elements

Whether using the SM or DCM, coupling DDD to FEM requires the traction field $\boldsymbol{\sigma}^{\infty}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})$ to be distributed among the set of relevant discrete nodes of a FE or BE model. In the DCM model this applies to dislocations that are sufficiently close to the boundary; while in the SM, it applies to all dislocations. Regardless of the coupling model, the force exerted by a dislocation ensemble on a node n on element e is given by,

$$\boldsymbol{F}^{(n)} = \int_{S_e} \left[\boldsymbol{\sigma}^{\infty}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\right] N_n(\boldsymbol{x})\,\mathrm{d}S_e\,, \tag{1.24}$$

where $\mathrm{d}S_e$ is the infinitesimal surface element with surface area S_e. $N_n(\boldsymbol{x})$ are so-called shape functions (interpolation functions) that distribute the traction field among the surface element's nodes.

The problematic singularity associated with the classical Volterra dislocation is avoided by using the non-singular formulation of Cai et al. [102] discussed in section 1.4.2, which changes eq. (1.14) into eq. (1.25),

$$G_{ij}(\boldsymbol{x}-\boldsymbol{x}') = \frac{1}{8\pi\mu}\left[\delta_{ij}\partial_{pp} - \frac{1}{2(1-\nu)}\partial_{ij}\right] R_a(\boldsymbol{x}-\boldsymbol{x}')\,. \tag{1.25}$$

Using the non-singular definition of $G(\boldsymbol{x}-\boldsymbol{x}')$ in eq. (1.13) we obtain the expression for the stress field of a single straight dislocation line segment bounded by two dislocation nodes at $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$1, 2) [102],

$$\begin{aligned}
\tilde{\boldsymbol{\sigma}}(\boldsymbol{x}) = &-\frac{\mu}{8\pi}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}}\left(\frac{2}{R_a^3}+\frac{3a^2}{R_a^5}\right)\left[(\boldsymbol{R}\times\boldsymbol{b})\otimes\mathrm{d}\boldsymbol{x}' + \mathrm{d}\boldsymbol{x}'\otimes(\boldsymbol{R}\times\boldsymbol{b})\right] \\
&+\frac{\mu}{4\pi(1-\nu)}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}}\left(\frac{1}{R_a^3}+\frac{3a^2}{R_a^5}\right)\left[(\boldsymbol{R}\times\boldsymbol{b})\cdot\mathrm{d}\boldsymbol{x}'\right]\boldsymbol{I_2} \\
&-\frac{\mu}{4\pi(1-\nu)}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}}\frac{1}{R_a^3}\left[(\boldsymbol{b}\times\mathrm{d}\boldsymbol{x}')\otimes\boldsymbol{R}+\boldsymbol{R}\otimes(\boldsymbol{b}\times\mathrm{d}\boldsymbol{x}')\right] \\
&+\frac{\mu}{4\pi(1-\nu)}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}}\frac{3}{R_a^5}\left[(\boldsymbol{R}\times\boldsymbol{b})\cdot\mathrm{d}\boldsymbol{x}'\right]\boldsymbol{R}\otimes\boldsymbol{R},
\end{aligned} \tag{1.26}$$

where,

$$\boldsymbol{R} = \boldsymbol{x}-\boldsymbol{x}' = y\boldsymbol{l} + r\boldsymbol{p} + s\boldsymbol{q} \tag{1.27}$$

$$R_a = \sqrt{\boldsymbol{R}\cdot\boldsymbol{R}+a^2} \tag{1.28}$$

$$\mathrm{d}\boldsymbol{x}' = -\mathrm{d}y\boldsymbol{l}\,. \tag{1.29}$$

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,

$$y = \frac{\boldsymbol{R}\cdot\boldsymbol{n}}{\boldsymbol{l}\cdot\boldsymbol{n}}, \quad r = \frac{\boldsymbol{R}\cdot(\boldsymbol{q}\times\boldsymbol{l})}{\boldsymbol{p}\cdot(\boldsymbol{q}\times\boldsymbol{l})}, \quad s = \frac{\boldsymbol{R}\cdot(\boldsymbol{p}\times\boldsymbol{l})}{\boldsymbol{q}\cdot(\boldsymbol{p}\times\boldsymbol{l})}\,. \tag{1.30}$$

Equation (1.26) can be turned into a series of triple integrals solved via recursion relations and a few seed functions.

Figure 1.10 diagramatically summarises the over 40 triple line integrals needed to find an analytical expression of the nodal forces on linear rectangular elements. Chapter 4 contains more detail regarding the theory, implementation and ad-

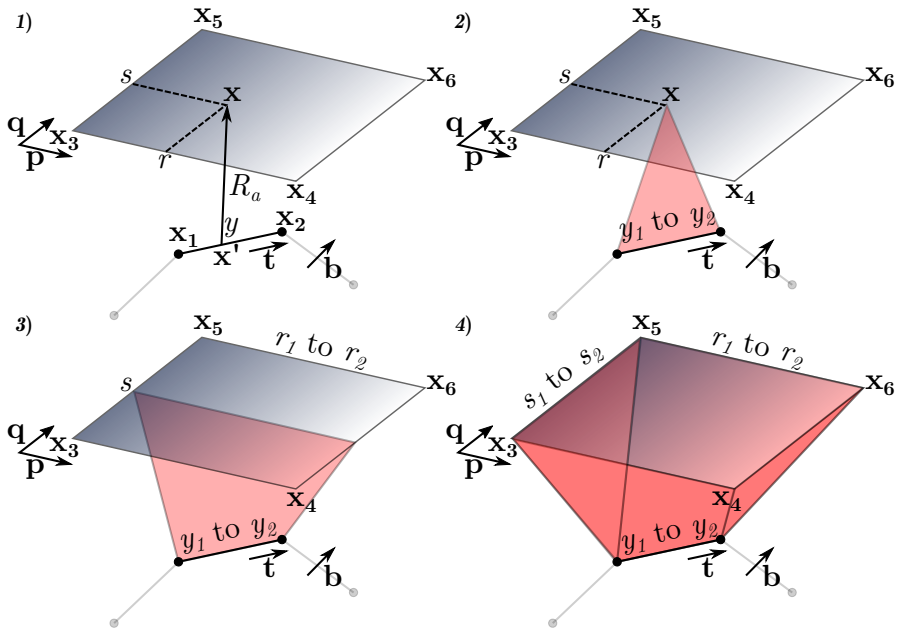


Figure 1.10: Diagram of the parametric line integrals solved by Queyreau et al. [90] to find the forces on linear rectangular surface elements.

vantages of these analytic tractions over the traditional solution of using Gauss quadrature.

1.4.4 Multiphase simulations

Modelling heterogeneity in DDD remains challenging. The core assumption of DDD is that we have a homogenous linear elastic space with invariant properties. However, modelling multiphase systems such as multicrystals and inclusions necessitates a weakening of such assumptions, particularly the space's homogeneity. Multiphase models are also necessary to apply DDD to more realistic scenarios.

Fusion in particular is riddled with examples of highly heterogenous materials. As previously mentioned, neutron bombardment often causes clustering of transmutation products, particularly on divertor and first wall materials; such clusters often substantially change many of their mechanical and thermal properties. The whole point of Oxide-Dispersion Strengthened (ODS) steels [113] is to hamper radiation damage by distributing oxide inclusions within their matrix; if we hope to model such alloys with DDD we must find a way to relax our homogeneity criteria.

Furthermore, many potential candidates for structural and even first wall materials of fusion reactors are designed with complex microstructures in order to resist thermal creep and radiation damage. Understanding dislocation behaviours near the crystal boundaries of these microstructurally complex materials is another niche where DDD simulations can shed light into the mechanisms that give

rise to these materials' desirable properties.

1.4.4.1 Polycrystalline materials

Polycrystalline materials have been studied using DDD, however most of these are in 2D. One of the easiest phenomena to investigate with 2D DDD are grain size effects [114, 115].

It is well known that much of the grain size strengthening effects arise from dislocation-grain boundary (GB) interactions including transmission, reflection, emission and absorption of dislocations [116, 117]; with transmission being the most typically observed.

These models depend on the grain boundary energy density, $E_{\mathrm{GB}}(\delta\theta)$, where $\delta\theta$ is the crystallographic misorientation between neighbouring crystals; resolved shear stress, τ, on the incoming dislocation with Burgers vector $\boldsymbol{b_1}$; critical penetration stress for the GB, τ_{GB}; and Burgers vector of the dislocation debris, $\boldsymbol{\Delta b} = \boldsymbol{b_1} - \boldsymbol{b_2}$, left behind when the incoming dislocation, $\boldsymbol{b_1}$, transmits throught the grain boundary to become a dislocation with Burgers vector, $\boldsymbol{b_2}$. The relationship may be approximated by,

$$\tau|\boldsymbol{b_1}|^2 \geq \tau_{\mathrm{pass}}|\boldsymbol{b_1}|^2 = E_{\mathrm{GB}}(\theta)|\boldsymbol{b_1}| + \alpha G|\boldsymbol{\Delta b}|^2\,, \tag{1.31}$$

where α is the material constant and G the shear modulus. The grain boundary energy density was proposed by Hasson and Goux [118] to have the simple form,

$$E_{\mathrm{GB}} = \begin{cases} k\dfrac{\delta\theta}{\theta_1} & 0 \leq \delta\theta < \theta_1\,, \\ k & \theta_1 \leq \delta\theta < \theta_2\,, \\ k\dfrac{\pi/2 - \delta\theta}{\pi/2 - \theta_2} & \theta_2 \leq \delta\theta < \pi/2\,, \end{cases} \tag{1.32}$$

where $k,\ \theta_1,\ \theta_2$, are material specific.

This is of course a gross simplification of the real 3D problem, but this approach was used by Li et al. [114] to investigate the Hall-Petch effect, which correlates grain size with flow stress of a polycrystal,

$$\sigma = \sigma_0 + \kappa\left(\frac{d_0}{d}\right)^n\,, \tag{1.33}$$

where σ_0 is the yield stress, d_0 a reference crystal size, κ the Hall-Petch slope and n the crystal size sensitivity parameter. Li et al. [114] showed that the model reproduces the Hall-Petch effect quite successfully. As a consequence, they showed that even what might appear as an overly simplistic approach can describe a

complex emergent phenomenon such as the Hall-Petch effect.

Aside from the Hall-Petch effect, the model has also been utilised by Fan et al. [115] to study the thickness effects of three different types of polycrystalline thin films: *1*) no surface treatment, *2*) surface passivation layer, and *3*) surface grain refinement zones. In their study, Frank-Reed sources were seeded across the domain, and the superposition principle was utilised to calculate displacement, strain and stresses in the thin films. Their results qualitatively reproduced experimental observations from expected dislocation patterns, stress distributions, and the eventual disappearance of the size effect as the films got thicker.

Though two-dimensional models might seem useless, they have their place, particularly when thin films are concerned. However, they may also be of use when investigating the effects of dislocations on the mechanical behaviour of superconducting tape—whose applications range from medical imaging to fusion energy production. The exotic composition and crystallography of many superconductors (perovskites) would make 3D models very complex and computationally expensive, so 2D models may offer viable alternatives. On top of this, a superconducting tape's operating environment would often have it under strains which might not strictly lie in-plane, but given a small enough segment of tape, strains orthogonal to its plane may be neglected. Thus making 2D models an acceptable first attempt at tacking the problem, at least until developments in 3D models make more realistic studies of such complex systems possible.

The 3D case is substantially more complicated than the 2D case. A study by Fan et al. [119], looked at the role of twinning on the hardening response of polycrystalline Mg using 3D DDD. The article is a perfect example of why 3D DDD is so much more complex than 2D DDD. Admitedly, it uses a material with a HCP crystal structure, whose mobility laws are substantially more complex than those of FCC or BCC materials.

In 3D DDD one must account for dislocation-twin boundary (TB) interactions. These may be obtained via geometric considerations, MD, and experimental evidence [120–123]. We must have information regarding how dislocations are transmitted through a TB such as *1*) whether a dislocation leaves a residual dislocation on the TB, *2*) the possible pre-TB and post-TB slip planes a dislocation can be transmitted to, *3*) which loading conditions are conducive to which transmission behaviour and *4*) which types of dislocation can be transmitted or reflected and in what ways [119]. All this obviously depends on the twinning plane, so in order to study more realistic scenarios one must have all the necessary knowledge to properly account for all dislocation-TB interactions. Furthermore, it is possible that certain dislocations may leave behind twinning dislocations that end up as TB steps, whose movement can lead to TB migration and twin growth [124, 125].

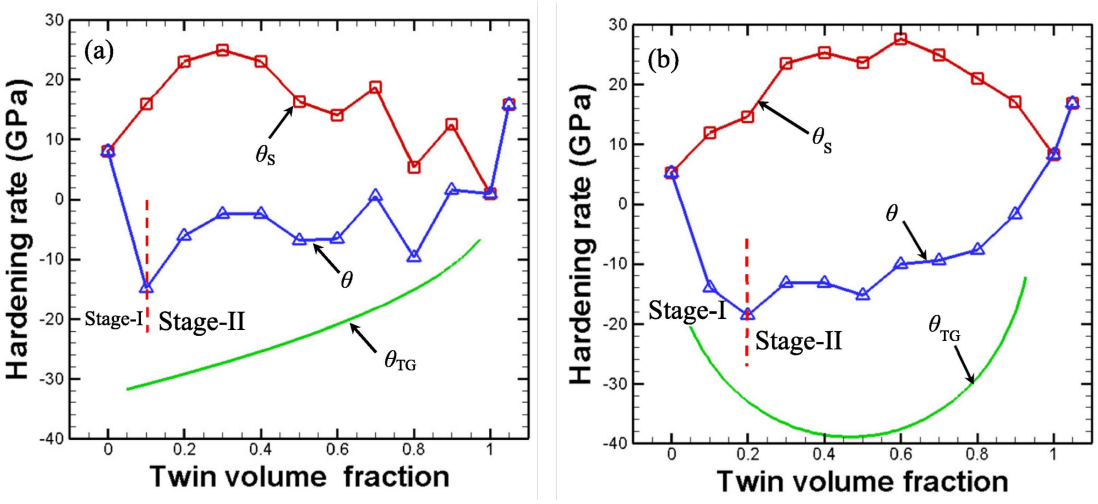


Figure 1.11: Simulated hardening rate as a function of twin volume fraction for (a) yz compressive loading and (b) tensile loading. Image edited from [119].

Fan et al. [119] modelled four scenarios in order to deconvolve the effects that twins and grain boundaries have on the mechanical behaviour of a cube-shaped crystal. They modelled *1*) a twinned polycrystal with two TBs and GBs on the edges of the cube, *2*) a single crystal with no TBs or GBs, *3*) a polycrystal with no TBs but GBs, *4*) a twinned crystal with TBs but not GBs. It is worth noting that the effects of twin growth in plasticity were not accounted for in their DDD simulation and instead had to be numerically computed via the hardening rate θ,

$$\theta = \frac{\mathrm{d}\sigma}{\mathrm{d}\epsilon} = \frac{\mathrm{d}}{\mathrm{d}\epsilon}\left[E\left(\epsilon - \epsilon^{\mathrm{p}}_{\mathrm{slip}} - \epsilon^{\mathrm{p}}_{\mathrm{twin}}\right)\right] = \theta_{\mathrm{s}} + \theta_{\mathrm{TG}}\,, \tag{1.34a}$$

$$\theta_{\mathrm{TG}} = -\frac{\mathrm{d}}{\mathrm{d}\epsilon}\left[E\epsilon^{\mathrm{p}}_{\mathrm{twin}}\right] = -E\bar{m}\gamma_{\mathrm{twin}}\frac{\mathrm{d}}{\mathrm{d}\epsilon}\left[f_{\mathrm{twin}}\right]\,, \tag{1.34b}$$

where θ_{TG} is the hardening rate due to twin growth, $\bar{m}$ the mean Schmid factor of the participating twins γ_{twin} the characteristic twinning shear and $\frac{\mathrm{d}}{\mathrm{d}\epsilon}\left[f_{\mathrm{twin}}\right]$ is the rate of change of the twin volume fraction with respect to the applied strain. These parameters must be empirically obtained.

With their model, Fan et al. [119] managed to qualitatively reproduce experimental observations, including the concave shape of strain-stress curves that arises from the competing hardening effect of TBs restricting dislocation motion (fig. 1.11), and the softening produced by twin growth as obtained from eq. (1.34b).

Twin boundaries in 3D are analogous to grain boundaries in 2D, given that they are coherent, narrow boundaries rather than messy, often incoherent grain boundaries. Consequently, grain boundaries are much harder to treat in 3D; instead, the community focuses on multiphase models where the details of dislocation transmission between phases are ignored or simplified as expanded upon in

section 1.4.4.2.

1.4.4.2 Inclusions

Dislocation-permeable inclusions have been simulated via the SM [126]. As mentioned in section 1.4.1.1, there is a need to calculate polarisation stresses due to differences in the elasticity of both phases. Using the same notation as section 1.4.1.1 this can be done by breaking up the image stress into,

$$\hat{\boldsymbol{\sigma}} = {}^{\mathrm{m}}\boldsymbol{C}\hat{\boldsymbol{\epsilon}} \quad \in V_{\mathrm{m}} \tag{1.35a}$$

$$\hat{\boldsymbol{\sigma}} = {}^{\mathrm{p}}\boldsymbol{C}\hat{\boldsymbol{\epsilon}} + \left[{}^{\mathrm{p}}\boldsymbol{C} - {}^{\mathrm{m}}\boldsymbol{C}\right]\tilde{\boldsymbol{\epsilon}} \quad \in V_{\mathrm{p}}\,, \tag{1.35b}$$

where m, p denote whether a variable belongs to the matrix or precipitate respectively and $\boldsymbol{C}$ is the elastic stiffness tensor.

Yashiro et al. [126] not only modelled a cuboid inclusion, but bimetallic interfaces in 3D. They found that under the right conditions, dislocations can pass from one phase to the other. When a dislocation passes into a different phase, it leaves an antiphase boundary on the slip plane. Therefore, in order for a dislocation to move from one phase to another, it needs excess energy equal to the antiphase boundary (APB) energy of the area it sweeps within the precipitate. This means that as a node passes from one phase to another, it feels a repulsive force F_b. The next dislocation moving along the same APB will then feel an attractive force, F_b to dissolve the APB. Once both dislocations move into the new matrix, they form a superdislocation bound by the APB energy [127]. However, as the dislocation length increases with respect to the precipitate's volume, the production of Orowan loops around it becomes more energetically favourable than moving into the new phase [126, 127]. Both of these behaviours have been observed when using the superposition method.

The eigenstrain method [128] as described in section 1.4.1.2 has been shown to be a viable solution to this problem. One can compute the stresses and displacements produced by the dislocation ensemble on the inclusions' surfaces via DCM or SM. The SM method however needs to calculate polarisation stresses [88], an operation that significantly increases the number of calculations necessary to solve the problem.

The DCM method has been used to simulate multiphase materials where the simulated dislocations reproduced experimentally observed behaviours such as the zig-zag patterns and dislocation forests produced by dislocations around precipitates in nickel superalloys [96, 129]. Reproducing such behaviours is not as trivial as simply adding inclusions to the simulation domain. In order to produce accurate results, one requires knowledge of the coherency stress between the different

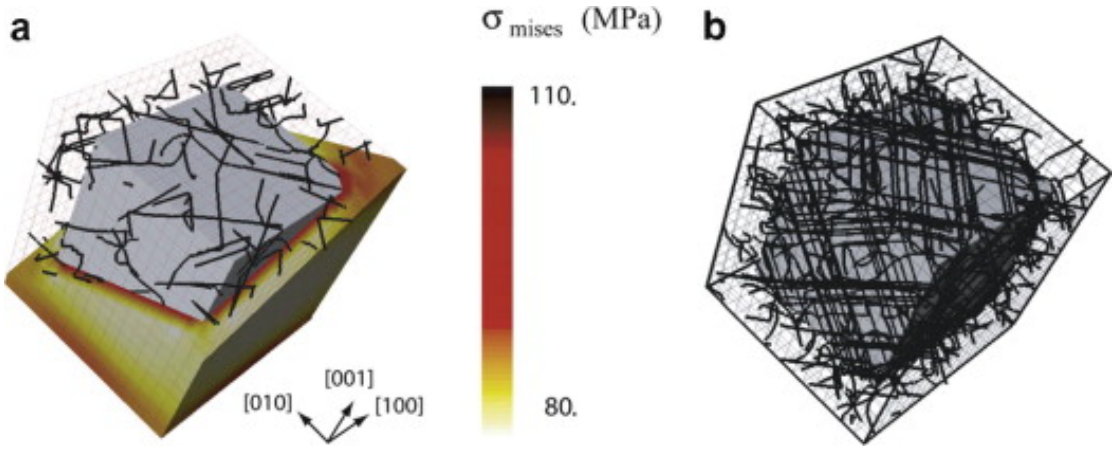


Figure 1.12: (a) Shows the initial dislocation configuration after relaxation in the presence of the calculated von Mises coherency stresses $\boldsymbol{\sigma}_{\text{mises}}$. (b) Shows the dislocation forresting around the inclusion as a result of 0.2% plastic strain for the [001] case. Image obtained from [129].

phases [129]. Such stresses are often due to lattice mismatches and differences in the thermal expansion coefficients. These coherency stresses localise the dislocations around the inclusions and are responsible for the observed localisation and aggregation of dislocations around precipitates as seen in fig. 1.12 [129].

The level set method has also been successfully used to model inclusions. In contrast to the aforementioned methods, they are assumed to be regions in space where dislocations experience a repelling force, instead of a region in space with distinct characteristics to the matrix. The force profile can be very simply defined to depend on the radial distance from the dislocation to a point in the "inclusion's" volume [97]. It can also be modulated depending on the nature of the inclusion by making the force a radial function, thus avoiding many of the numerical issues facing DDD-FEM couplings (see sections 1.4.1 and 1.4.3). It is also possible to make the inclusions impermeable or semi-permeable depending on the magnitude and steepness of the force gradient, as presented in [97]. Arbitrarily shaped inclusions are also possible if one were to also use angular components in the force function, through the use of spherical harmonics and even piecewise-parametric fourier series. This is a simple and versatile method that can be applied to DDD as well.

Though mathematically appealing, such an approach is far from representing the reality of dislocation-inclusion interactions. More realistic scenarios require the use of more commonplace DDD-FEM coupling methods. Implementing arbitrarily-shaped particles is not impossible in DDD—particularly if they can be discretised. Aside from the significantly lower computational requirements of the time evolution and dislocation-dislocation interactions compared to the level set method, one of the biggest advantages of using the SM and DCM methods is that

everything, from tractions and displacements on the inclusions, to the coherence stresses on the matrix, can be natively computed from experimentally obtained values. Besides, the method for treating inclusions as functions is still viable. On the other hand, the level set method is so far limited to this approach. Furthermore, if the inclusion is a different crystal or a twin of the same material, as in [119], it is possible to use the DCM or SM to study the transmission of dislocations from one phase to the other; something that is not yet possible with the level set method.

1.4.5 Parallelising discrete dislocation dynamics

It has been shown that DDD lends itself well to parallelisation. Ferroni et al. [86] investigated some of the more computationally intensive parts of DDD in DD-Lab [130], a freely available DD code for Matlab. The algorithms whose parallel performance was studied in [86] were 1. remote (segment-segment) forces, 2. surface tractions and 3. image stresses. The remote force computation has $\mathcal{O}(N^2)$ computational complexity, where N is the number of dislocation segments. The quadratic scaling is due to the r^{-1} dependence of the dislocation stress fields, so long range interactions cannot be ignored. Both the surface traction and image stress calculations have $\mathcal{O}(kN)$ computational where k is the total number of surface nodes and N the number of dislocation line segments.

As is usually the case in parallel computing, there is often more than one way to parallelise a problem. The optimal strategy depends on the problem itself. It was no different in [86] as shown in fig. 1.13. Ferroni et al. [86] realised that the segment-segment computation can be done via $\mathcal{O}(N^2)$ parallelism, where single pairs of dislocation segments are sent to a single thread, and then globally reduced. However, this approach is $\mathcal{O}(N^2)$ in memory and can lead to inefficient data reuse. So they opted instead for $\mathcal{O}(N)$ parallelisation and serialisation. In this scheme each thread is assigned a unique segment, and each thread computes the forces between its assigned segment and all the others while serially adding the force contributions. The memory access pattern they found best was that of a serial procedure, the reason is that in-thread calculations are serialised so having coalesced memory access patterns for threads would cause problems when serially accessing data for the calculation.

However, there is another approach Ferroni et al. [86] did not consider. It combines both of the aforementioned strategies, $\mathcal{O}(N^2)$ memory and $\mathcal{O}(N)$ parallelisation and serialisation. Such an approach requires two copies of the input data (one per access pattern). One pattern would allow each thread to obtain its uniquely assigned segment in a coalesced manner. The second would be the se-

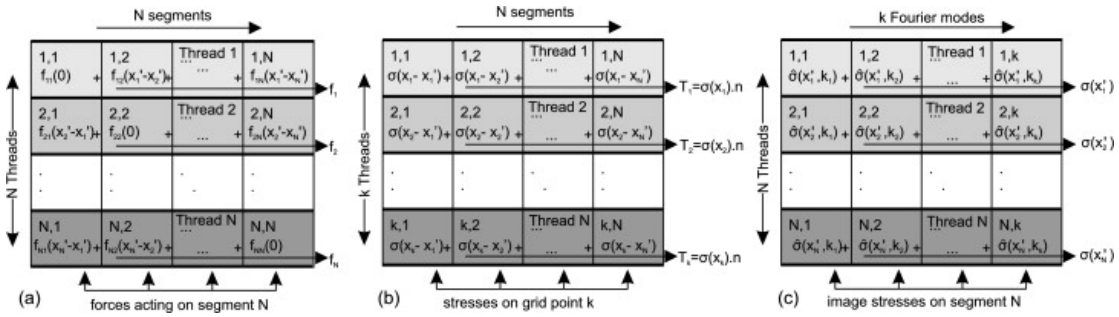


Figure 1.13: Parallelisation strategies for: (a) the N^2 segment interactions; (b) surface traction at k grid points and; (c) and image stresses on N segments. Image taken from [86].

rial access pattern already present. Thus obtaining the best possible performance at the cost of doubling the required memory. This is a better strategy provided there is enough device memory[5], if that is not so, the problem must be split into separate GPU calls.

New NVidia architectures have made varying degrees of nested parallelisation possible, NVidia calls this "dynamic parallism". When applied to the computationally optimal solution, the $\mathcal{O}(N^2)$ memory requirement is relaxed to $\mathcal{O}(N)$ (with the same parallel-specific data structure) and $\mathcal{O}(N^2)$ computational parallelism.

The parallisation strategies for the other two problems, (b) and (c) in fig. 1.13, involved similar decompositions as the remote force computation. For surface tractions Ferroni et al. [86] parallelised over k surface nodes with sequential calculation and addition of the stress from each dislocation line segment. For the image stresses, they chose the reverse strategy since it allows for the serialised addition of the image stresses on dislocation line segments.

Their findings showed very promising results for parallelising DDD simulations. Compared to serial implementations in C, their findings showed that depending on the specific problem and parameters used. The parallel implementation reduced computational time by a factor of ~ 110 at best and ~ 3 at worst, for sufficiently large problems. This is of course, provided one uses a card designed for high performance computing, as these are tailored towards `double` precision arithmetic and computationally intensive procedures. However, gaming graphics cards are increasingly capable of double precision arithmetic and though not as drastic,

[5]Assuming the line direction is calculated inside the program, each dislocation line segment represents 3, 3-element `double` precision vectors (8-bytes per entry), totalling 72 bytes in global memory per dislocation line segment (two nodes + Burgers vector per segment). There must also be enough global memory to output the forces, which represent 1, 3-element `double` precision vector per dislocation node, totalling 24 bytes per node. Assuming a single dislocation loop with N nodes (therefore N segments), this brings the total global memory requirement to $96N$ bytes. Using two access patterns for the dislocation line segments and Burgers vector means an extra $72N$ bytes, giving a grand total of $168N$ bytes. Assuming the material parameters are stored in constant memory.

speed gains are still substantial for sufficiently large problems.

1.5 Project outline and new science

If anything should be clear from the present work, is that materials science is far from solved. Even within the small niche of dislocation dynamics modelling, there are a huge number of open problems and an equally large number of solutions; each with its particular sets of advantages, disadvantages, ideosyncrasies and challenges.

The rapid advancement in computer technology; pushed by the ever-increasing complexity of the problems that occupy humankind; but simultaneously restricted by the physical constraints of classical computers, have opened up new avenues for research. Research that blurs the boundaries between fields and produces something new. Undoubtedly materials science sprang up this way, when metallurgists decided to take elements from the fields of chemistry, physics and mathematics to create a new discipline.

With increasing frequency, one finds people across vastly different fields who can understand each other via a common language. The language of scientific computing, where fields intertwine and produce new research methods; where techniques developed accross disciplines are taken, improved and adapted to work on the individual researcher's problem of interest.

This project applies such techniques to accelerate and improve the ease with which outstanding problems in materials science can be tackled; making use of more accurate models, hardware acceleration and the techniques of high performance computing to enable more accurate, larger, more complex simulations than before—striving for increasingly faithful and insightful recreations of reality with a user-friendly code on a desktop PC.

Much of the work presented herein has already proven useful in the simulation of complex simulations such as nano-indentation [131] and several other research projects within the Tarleton group. Here we showcase the new capabilities, findings and methods; improvements, fixes and redesign of the codebase; as well as a showcase of relatively simple, yet formerly inaccessible simulations resulting from this work. Finally, we present an ambitious proposal for future work resulting from the insights gained during the project.

Chapter 2

EasyDD v2.0

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

As mentioned in section 1.5, 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 unreasonably 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 Euler-trapezoid predictor-corrector solver described in algorithm 10.2 and equations (10.42, 10.45 and 10.46) in [130, p. 214–216], reproduced here by 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. The algorithm contains two free parameters, $\Delta t_{\max}$ and ϵ, which respectively 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 original, supposed implementation of this is found in commit `65907b0` under the name `int_trapezoid.m`.

However, this was not the case. There were a number of problems with the implementation. Firstly, it only used $\Delta t_{\max}$ only on the first timestep. It also only increased the timestep once every timestep. Ideally, an adaptive time algorithm should increase the time step to the maximum that is still under the error criteria. The way the old version of the integrator worked meant that the time step would take multiple steps to ramp back up after decreasing. Take for example, a case where the time step had to decrease once for two consecutive steps, $\Delta t = \Delta t_0/4$ would have taken $\lceil \log_1 .2(4) \rceil = 8$ subsequent time steps to surpass what it was originally, Δt_0. In general, N time step halvings, would require $\log_b(2^N)$ steps get back to the original value, using b as the time-step increase factor. Since only one increase in time step was allowed per simulation step, simulations were taking many more timesteps than necessary.

An improved algorithm is described in algorithm 2.2. This algorithm maximises the time step given absolute and relative error criteria, ($\epsilon_{\max}$, $\upsilon_{\max}$), absolute maximal and minimal timesteps, ($\Delta t_{\max}$, $\Delta t_{\min}$) and number of iterations, $\mathrm{iter}_{\max}$. It also uses a modulated timestep increase by how far under the absolute accuracy criteria we are. It also fixes a bug that made the error calculation only account for the largest magnitude of a single component of the 3D vectors, rather than the largest norm. The new implementation can be found in the most current version of `int_trapezoid.m`.

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 if $\Delta t < \Delta t_{\min}$ **then**
    $\Delta t \leftarrow \Delta t/2$
    iter $\leftarrow$ iter $+ 1$
  else if $\Delta t == \Delta t_{\min}$ **then**
    iter $\leftarrow$ iter$_{\max} + 1$
  else
    $\Delta t \leftarrow \Delta t/2$
  end if
  if iter $>$ iter$_{\max}$ or $\Delta t == \Delta t_{\max}$ **then**
    Convergent $\leftarrow$ true
  end if
end while
$t \leftarrow t + \Delta t$
Proceed with the rest of the simulation.

Algorithm 2.2 is more computationally expensive per iteration, but can increase the timestep more rapidly, optimally and is more robust than algorithm 2.1. The original algorithm also wasn't correctly implemented. Together, these changes let the new 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 different loading conditions, as simulations were now more capable of effectively adjusting the timestep to more efficiently match the error tolerances. However, as discussed in chapter 5, there is still a danger of overloading simulations. Fixing the integrator was the first major bottleneck we fixed, allowing us to identify and resolve downstream issues that were previously unkown.

2.1.2 Matrix conditioning

A common problem with discrete dislocation dynamics is the tendancy for some networks to have a large degree of variation in the speed of dislocation nodes [130, 132, 133]. This is a consequence of the fact that nodes are discretised representations of dislocation lines. 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-slips, accounting for dislocation character and slip system. Moreover, dislocation motion can be assumed to follow a linear, anisotropic, viscous drag model [130],

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

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 can experience very different resitances to moving along its allowed movement vectors.

Different mobility laws can have different parametrisations for their permitted movement vectors. A problem that often plagues traditional mobility laws[1] is the movement of a node along a line segment itself. Figure 2.1 shows how a node moving along a line segment does not change its topology, and should therefore have no effect on the energy of the network. Consequently, nodal movement along

[1]Bruce Bromage, of [134] 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, $\mathbf{B}$ in eq. (2.8), can still be singular however, so this law also includes the scale-averaged Levenberg-Marquardt regularisation discussed in this chapter.

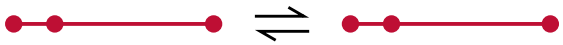


Figure 2.1: Node movement along dislocation line should have no drag contribution.

the line direction must not contribute to the total drag, so the line drag component of the stress tensor, $\mathcal{B}$, should tend to zero. On the other end of the spectrum we have climb, which is orders of magnitude less favourable than glide, so that component of the drag tensor should be much larger than the rest. However, these requirements can make the drag matrix, $\mathbf{B}$ in eq. (2.8), singular. Keeping the matrix invertible while simultaneously meeting the physical requirements is a careful balancing act. Moreover, the low drag along the line direction often causes nodal speeds along these vectors to be much higher than other directions, often limiting the integration time step as a result.

In the overdamped regime, $\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t}(\boldsymbol{x}) \to \mathbf{0}$, known driving forces (Peach-Köhler force + remote forces + self-force) and the unkown drag force sum to give,

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

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. (2.6) requires global knowledge of the network. Fortunately, 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 an approximate expression for the velocity of a single node,

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

where i is the node in question, j are the nodes it shares segments with, and $\boldsymbol{f}_i$ is the local driving 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 in the drag coefficients used to construct $\mathcal{B}$—and therefore $\mathbf{B}$—means the condition number of $\mathbf{B}$ has the potential to be very large. Given that $\mathbf{B}$ 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, and min, max are the minumum and maximum eigenvalues.

The condition number of a matrix codifies how much collinearity exists in its basis. The larger the condition number, the greater its basis' 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 [135].

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 or subroutines is a *very* bad practice in software engineering.

The "buginess" of a function $\mathcal{X}$ with a set of $\mathbb{X}$ bugs whose phase space is defined by $\chi(x)$, can be mathematically expressed as, $\forall x \in \mathbb{X}\ \chi(x)$. That is, for all inputs, x, some process, $\chi(x)$, produces a set of bugs $\mathbb{X}$. In other words, how buggy a function is, depends on the severity and frequency with which the process, $\chi(x)$, produces bugs, $\mathbb{X}$, for different inputs.

Bugs that evade tests and make it into production code are often either low impact with broad phase spaces, or high impact with narrow phase spaces. The way by which both types of bugs commonly make it into production code differs. Usually, low impact, broad phase space bugs make it into production due to a lack of proper unit testing on "obvious" or "trivial" cases. High impact, narrow phase space bugs are usually down to rare corner cases such as catastrophic cancellation, loss of precision, division by zero, memory over or underflows, memory leaks, silent errors/crashes due to poor API (Application Programming Interface) design, etc. Both types of bugs can be avoided with a combination of proper software design, and testing protocols. Unfortunately, mitigation strategies go out the window when functions are strongly coupled. We want functions to be independent variables in our process. Just like in probability, phase spaces are multiplicative, and we want their products to be linearly independent. By having strongly coupled

functions we make them non-linear, as one function depends on the other to solve its problems.

The way our old 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, an example of a mobility function with this problem can be found under commit `65907b0` under the name `mobbcc1.m`. Every mobility law in that commit has this issue. There are various problems with algorithm 2.3.

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

```
Compute eigenvalue matrix, D ← P^{-1}BP
κ(B) ← |λ_max|/|λ_min|
if κ(B) < κ_min then
                          ▷ Normalise D and f to the maximum eigenvalue.
    D ← D/λ_max
    f ← f/λ_max
                          ▷ Invert D by inverting each non-maximal eigenvalue.
    for all (λ_k ∈ D) ≠ λ_max do
            ▷ λ_crit → 0 is some arbitrary threshold for the minimum value of λ_k.
        if λ_k > λ_crit then
            λ_k ← 1/λ_k
        else
                                          ▷ The inverse of 0 is not 0.
            λ_k ← 0
        end if
    end for
                                          ▷ D is already inverted.
    v = PDP^{-1}f
else
    v = B^{-1}f
end if
```

Assuming that an ill-conditioned matrix can be fixed by diagonalising and normalising. This is simply not true, it can help with numerical stability in some algorithms for the solution of linear systems but may not be enough. Perturbing the smallest eigenvalue in a direction such that κ improves is a good way to renormalise a system 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 mostly intact, i.e. we want to maintain the relative sizes of the eigenvalues with respect to one another. If other eigenvalues are close in magnitude to the smallest one, the dynamics of the system can very easily change, even a very small perturbation may have undesireable consequences. 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 robust enough.

The assumption that the case where $\lambda_k < \lambda_{\text{crit}}$ for non-maximal eigenvalues is so rare that 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 a consequence of the assumption of locality in eq. (2.8). Moreover, when every non-maximal eigenvalue is smaller than λ_{crit}, the error is much greater. For every eigenvalue under the threshold of $\lambda_{\text{crit}} = 10^{-k}$, the upper bound of the error is k digits in the corresponding mobility component.

Finally, assuming the integrator will not allow obviously wrong solutions because it will fail the error check and reduce the time step is not something to hang our hats on. Erroneous velocities only get caught within the integrator, not when nodal mobility is calculated outside the time integration, which happens in every topological operation where a new node is spawned. Spawning a new node with extremely incorrect velocities can prove to be quite a serious problem, especially if this happens in a region with a high dislocation density.

The regularisation of ill-conditioned systems is an active field of research [136–138]. However, they require some knowledge of the matrix structure. Regularlisation is most often applied to high-rank matrices because poorly conditioned, low-rank matrices are often very sensitive to perturbations. In this case, $\mathbf{B}$ is positive definite 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 Levenberg-Marquardt coefficient—to a diagonal element of the matrix. This is roughly equivalent to perturbing the smallest eigenvalue as previously mentioned. If this factor is added the whole diagonal, the inversion can be sufficiently dampened, whilst keeping the main contribution from each basis vector roughly equal in relation to each other. This amounts to adding a small multiple of the identity matrix, $\mathbf{I}$, 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 Levenberg-Marquardt coefficient is,

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

where ϵ is machine precision and $|A|$ denotes the absolute values of matrix $\mathbf{A}$. 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 this renormalisation is a source of error. Adding noise to a system—even if small and along the diagonal—moves 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 mentioned 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 $|\lambda_{\max}|/|\lambda_{\min}| > 1/\epsilon$ **then**
  $\mathbf{B}_+ \leftarrow \mathbf{B} + \sqrt{\epsilon}\max(|\mathbf{B}|)\mathbf{I}$
  $\mathbf{B}_- \leftarrow \mathbf{B} - \sqrt{\epsilon}\max(|\mathbf{B}|)\mathbf{I}$
  $\boldsymbol{v}_+ \leftarrow \mathbf{B}_+^{-1}\boldsymbol{f}$
  $\boldsymbol{v}_- \leftarrow \mathbf{B}_-^{-1}\boldsymbol{f}$
  $\boldsymbol{v} \leftarrow (\boldsymbol{v}_- + \boldsymbol{v}_+)/2$
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 velocities to be very 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 ongoing 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 [139–141]. With modelling thrust into the spotlight, and the public release of the Imperial College modelling code, which was used to inform so much policy [141], `https://github.com/mrc-ide/covid-sim`, criticism was levied at the lack of care taken to ensure the correctness of the code [142].

In experimental research, a lot of care is take to ensure the 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 [143–145]. In the modern age, software is ubiquitous in the obtention, analysis and communication of scientific research. Anyone hoping to do science will interface with software at some point. This software should therefore be as high quality as possible, *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 ongoing. The amount of initial technical debt since its inception as the infinite-domain discrete dislocation dynamics code, `DDLab` [130], 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.

As this is actively developed research software, 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 its organisation. However, as a codebase grows, it becomes more important for code be properly organised and source controlled. This makes it easier to browse, understand, and provides a crucial 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 is the very first thing to set up when starting a software development project. Academia is *very* far behind in this regard. Source control software can perform a wide variety of tasks but among the most important are: *1*) provide a history of restore points in case things break; *2*) keep a history of changes, who and when they made them; *3*) lets developers and users make changes independently and request their changes be added to the main code as patches or new versions; *4*) detect conflicts between versions to be merged; and *5*) a robust versioning system integrated to the history of changes.

With regard 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. Source control can be used in local and remote services via private LAN (Local Area Network), as well as webhosting services such as Gitlab, Bitbucket, or GitHub as in our case. Webhosted services are a popular choice because they simplify and enhance the user and developer experience. For example, the visibility of code can be easily set to 1. private, where only those explicitly given access have read and write access; or 2. public, where anyone can view the code and propose changes which can then be reviewed and either accepted or rejected by those with write access. They also simplify the use of addons and webhooks that can perform myriad tasks like, 1. automated remote testing on different platforms; 2. code coverage reports (what lines and how many times each of them was hit during automated testing); 3. automated documentation generation; 4. automated merge checks; and even 5. automated encryption/installation key generation.

LAN-hosted version control software can make use of everything webhosted implementations can[2]. Indeed, LAN-hosting is used for things pertaining to de-

[2]Although man quality of life improvements such as GUIs (Graphical User Interfaces) and

fense and national security, but the use of webhosted services is an easy choice for academia, industry, and private users. In fact, the digital version of this document is found in https://github.com/dcelisgarza/DPhil_Thesis, among many other repositories from recreational and past work. Moreover, whenever this document references a specific file, the digital version directly links to the relevant information (including relevant lines).

Publically available, webhosted source control is a boon for open, reproducible science. It can be used to provide a snapshot of the specific version of the code and data used to generate the results of a manuscript when submitting to a journal. Such platforms can be used to generate funding and copyright, and even enable independendant publishing through the ISBN registry and crowd-sourced peer-review. Through the use of tools such as Jupyter, one can write interactive publications, dubbed "notebooks". These notebooks give the power of peer-review back to the community, as well as being great tools for science communication, teaching, and even interactive techincal talks.

In summary, source control is the solution to multiple problems that inhibit scientific progress. On one hand, it provides a safety blanket with many bells and whistles that routinely saves researchers and companies time, money, effort, and heartache. On the other, it democratises scientific research by sidestepping the politics, costs, and ransoming of *publically* funded research that is, at its core, for the benefit of humanity.

2.2.1.2 Folder structure

An ongoing 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 a large quality of life improvement for 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 "modularisation".

code editor integration might be unaccessible.

2.2.2.1 Encapsulation

The act of keeping variables neatly stored in a way that fits a purpose is called encapsulation. By encapsulating variables, we greatly improve the usability of software by increasing readability, reducing the risk of human error by reducing the number of things to keep track of, and—if done properly—can even improve performance. How we go about encapsulating code is tightly bound to the design philosophy and programming paradigm of a software product, and the capabilities of the language it is written in. Common programming paradigms are outlined below.

- 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 paradigm 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 can be similar to OOP in some ways, 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 (that makes use of functions) 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 and as a 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 "spagghetti" code[3]. Unfortunately, education in programming or software engineering in science is sorely lacking. As

[3]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 many other compiler optimisations.

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 very useful where safety and fault-tolerance is a concern. It sees a lot of use in telecommunications, social media, cryptography, finance [146, 147], but usually as part of a solution. It is however, impractical for most applications.

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. This often leads to software being written in a procedural manner.

However, we can take a data driven approach and leverage `struct` to simplify the process. These are more of an ordered dictionary (Hash table), than a true object, but they are what we have available. Converting the code to use them is an ongoing process, but they have so far allowed us to make use of generic functions for different mobility functions, loading conditions, and various auxiliary variables. All of which have a 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 already 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. `MATLAB` has an old method that enables generic programming 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)
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, easier to follow, can be returned as part of a `struct` (thus providing a poor-man's way of making classes), and can be nested. Generic functions were already in use for dislocation mobility, but there are now functions for calculating numeric and analytic tractions (see chapter 4), loading conditions (multi-stage loading, constant loading, cyclic loading), 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 choose whether to use it by changing an input file rather than directly altering the source code. If instead we were to directly modify the source code, we would need to account for new inputs, outputs and perhaps create new branching behaviour. Doing so every time new functionality is added will lead to an explosion of complexity that spaggettifies code, making it difficult to work with and innefficient to run.

2.2.3 Optimisation

> We *should* forget about small efficiencies, say about 97% of the time: pre-mature optimisation is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
>
> — *Donald E. Knuth [148, p. 268]*

2.2.3.1 Spurious memory allocation

`MATLAB` is infamous for the way it will happily grow an array dynamically 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 quite a few low-hanging fruit that have been picked.

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.

More memory allocation also means more garbage collection (GC) [149] i.e. automatically freeing memory, based on a set of criteria. The cost of GC can be significantly higher than that of memory allocation. Timing the equivalent of allocating a $20 \times 20 \times 20$ mesh in `MATLAB 2020b` takes on the order of 70 µ sec on x86 CPU architectures, which is in-line with what it costs in `C`. However, deallocating memory takes about 5× longer. In a function that otherwise takes a few m sec, adding almost 0.5 m sec because 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. Taking advantage of this sparsity significantly reduces memory footprint and computational expense. The code already used these special data structures, along with Cholesky factorisation for faster solves 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_T, as a result of the forces acting upon them,

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

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 than previously, as well as letting us build meshes over two orders of magnitude faster, mostly thanks 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 compiling the C and CUDA code for the first time. 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 bring others into the fold. 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

Throughout this chapter we will refer to various subroutines by what they perform.

1. Time evolution: solve the differential equation governing dislocation motion.
2. Separation: separate high energy, highly connected nodes into lower energy nodes with fewer connections.
3. Collision detection: detect collisions between segments, i.e. which segments collide, where they collide, what nodes are involved.
4. Collision resolution: resolve collisions between segments, i.e. where they collide, what nodes to merge and where to do so.
5. Remeshing: ensuring the network has the appropriate resolution. Broken into 3 main parts:
   (a) Virtual: nodes that have exited the finite volume need to be tracked so the slip step and displacements can be properly accounted for. Virtual-surface segments, and virtual-virual segments are considered external and do not interact with internal segments in any way.
   (b) Surface: nodes that exit the finite volume need a node on the surface to keep track of where they connect to the internal network. Surface-internal nodes are considered internal, so they interact with other internal segments.
   (c) Internal: nodes that are internal to the finite volume.

3.1 Introduction

In 2D, dislocations are parametrised as points on a surface. Therefore, aside from annihilation, interactions with grain boundaries [116, 117], 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.

3.2 Non-commutativity of topological changes

For a given iteration, the time-evolution and topological changes used to be performed in the following order:

1. time-evolution,
2. separate highly connected nodes,
3. detect and resolve collisions, and
4. remesh network.

This ordering was problematic. Fundamentally, it means the time evolution erroneously accounts for complex structures that should have dissipated into lower energy ones as per the quasi-static principle the model is based on. In other words, detecting and resolving collisions, generates highly connected structures that are high energy and thus produce large local stress gradients. The assumption that a node and its neighbours experience similar conditions breaks down near these highly connected nodes. Moreover, the assmumption that the network is at thermodynamic equilibrium at every time step breaks down when there are highly connected nodes that would otherwise be broken up by separation. Both of these lead to eq. (2.8) not holding, which is the fundamental assumption behind our mobility laws.

Moreover, the order in which topological changes were carried out was also very sub-optimal from a practical aspect. Colliding after separating would sometimes

result in nodes being separated only to immediately collide again. Remeshing only at the end meant the input network for the topological operations could have contained segments whose lengths were outside the bounds defined by the input file. Meaning the network may not have been properly discretised, leading to unecessarily complex structures that may not have had the required resolution. The solution was to simply remesh the network after its time evolution and before topological functions.

The other problem with this ordering was the combinatorial complexity of the separation subroutine. The consequence of the old ordering is that as simulations advanced, collisions tend to become more common. A higher collision rate means more highly connected nodes. The role of separation is to break these high energy, highly connected nodes into lower energy nodes with fewer connections. However, splitting nodes is a combinatorial problem, so the gross computational complexity is functionally $\mathcal{O}(N!)$, which is pretty much the worst possible scaling barring special cases such as tetration. Section 3.4 contains a more detailed explanation as well as resolution of the problem.

Aside from the order of operations, a very large number of changes, fixes and improvements needed to be made to practically all topological operations. Here we detail only the ones this project had a major hand in fixing, namely collision detection and resolution, separation, and part of the surface remeshing. The rest were led by Bruce Bromage as part of his project, and the one preventing the formation of superdislocations by Haiyang Yu.

The end result of all this work is the new order of operations:

1. time-evolution,
2. remesh,
3. collision detection and resolution + separation loop,
4. mobilise highly connected immobile nodes,
5. separate highly connected nodes,
6. remesh.

This ensures the time evolution and topological functions are all fed a network that is as clean and well-resolved as the input parameters say it should be. Although this incurs a computational cost, the benefits of having a cleaner network for the other functions to work with vastly outweigh the increased cost of calling the cheap remeshing subroutine more times.

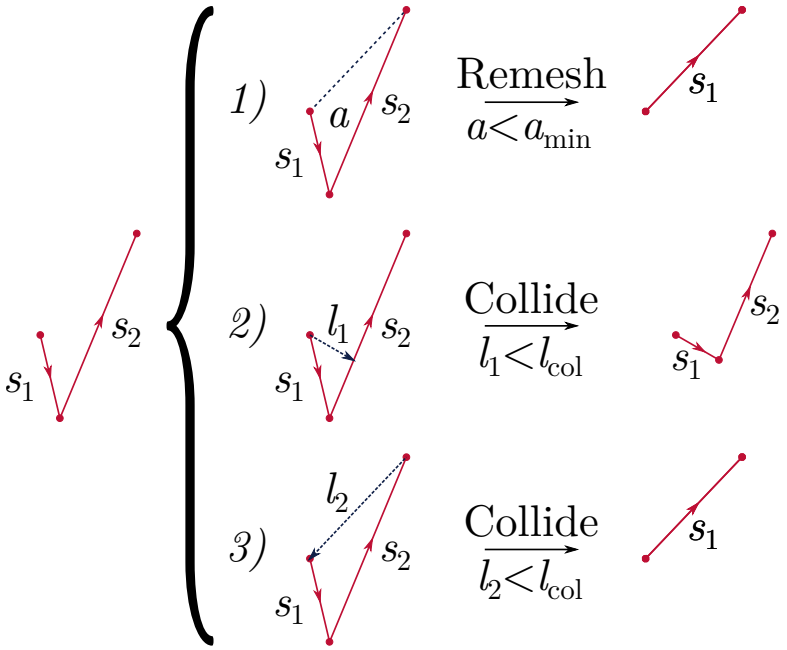


Figure 3.1: A single dislocation hinge can lead to three final topologies (two of which are degenerate), the conditions for which can be simultaneously met in any combination. Where a is the enclosed area, l_1 and l_2 are possible minimum lengths between both segments.

3.3 Collision

3.3.1 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, which we call *hinges*, as shown in fig. 3.1. A single dislocation hinge can lead to three final topologies (two of which are degenerate). 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 the conditions can be met, and the order in which they are checked could 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 prior to many of the improvements made to the codebase, but with the increased ability to model a larger number of dislocations to a sufficiently advanced point where this was having a significant impact, particularly in Haiyang Yu's nanoindentation simulations, which would either go wrong by creating unphysical junctions and/or massively slow down after reaching a certain point.

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 the other type of collision, which we call *two-line* collision. This meant detecting and resolving all possible hinge collisions before detecting and resolving any two-line collision. The rationale behind this is that segments in a hinge are already part of a single dislocation line. Because they are connected to each other at a single hinge node, they are already interacting via line tension and remote interaction; whereas two unconnected segments approaching one another only interact via remote interaction. The other, more practical advantage of resolving hinges first is that hinge collisions decrease the total segment length of the involved segments, therefore reducing their collision radii and the probability of them colliding with other segments via two-line collisions.

The second was much more subtle. Figure 3.1 has two options for colliding a hinge, as seen in fig. 3.1. Both resulting in different topologies. Which topology we ended up with was down to which segment, s_1 or s_2, the code came across first. If s_1 was first, then the answer would be 2), otherwise it would be 3). At the very least, the answer should be the same for a given network topology, regardless of how it is represented. So we decided to ensure the distance calculated was always the minimum out of both scenarios. Which means the distance calculated should be the minumum distance between the non-hinge node of the short segment, to the long segment. There is an argument to be made that the solution should be the other way round, as that would ensure the final structure has the minimum total length, as well as one less node. However, this could potentially leave many hinges around that would have otherwise been resolved.

3.3.2 Collision distance

In the same vein as section 3.3.1, two-line collisions were being improperly prioritised. They occurred arbitrarily as the code came across them. Again yielding

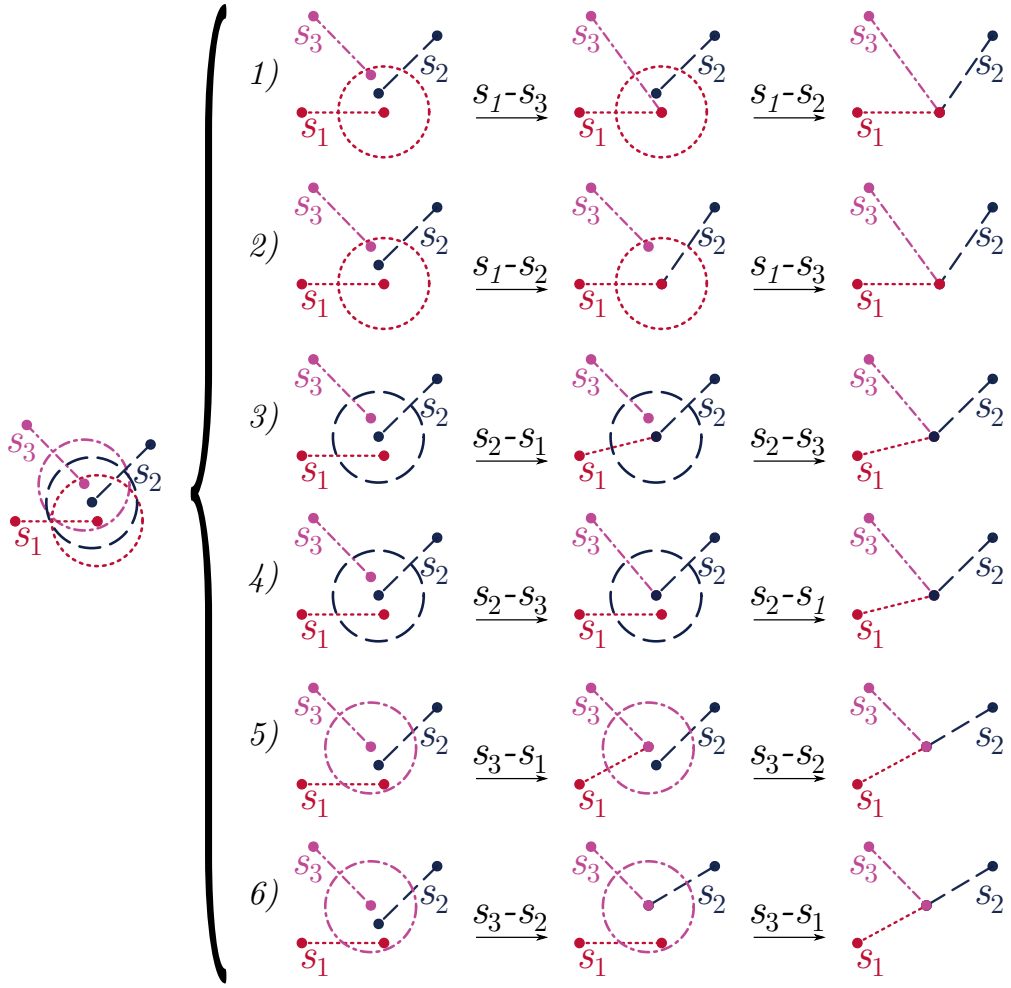


Figure 3.2: Multiple dislocation segments inside the same collision radius. Resolving all collisions at once leads to 3 different structures, 6 total but half are degenerate. Circles represent collision radii for the segments with corresponding line style and colour.

different answers for different representations of the same network. Only this time it had to do with the collision radius, as shown in fig. 3.2.

The actual algorithm is more sophisticated because it accounts for whole segments, as well as their relative velocities. It returns the points along the segments that minimise the distance between segments. These points are then used to decide which nodes and how they should be merged.

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 was 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; and perhaps the true minimal distance was not even between s_1 and another segment, but between s_2 and s_3. We had to apply the same fix to hinge collisions. Admittedly, it matters less in that case as they tend to be rarer, plus their size is limited anyway, but

still ensures a more correct solution.

The original algorithm had $\mathcal{O}(N^2)$, $\Theta(N^2/4)$ and $\Omega(N)$ for its upper, average, and lower bounds respectively for fixed N, where N is the number of segments. The new one is always $\mathcal{O}(N^2)$. However, $N^2/4 \to N^2$ as $N \to \infty$, so as simulations grow larger, the performance hit gets lower. Moreover, the algorithm is quite cheap compared to other processes, and the improved accuracy is very much worth the small cost.

Better collision detection tends to result in faster simulations because the resulting structures are guaranteed to either produce the shortest segments—which can be cleaned up by remeshing—and therefore more accurate junctions. Referring back to fig. 3.2, it is evident that the structure formed by colliding s_1 and s_3 is higher energy than that formed by colliding s_1 and s_2. In this case, if we fully resolve the two possible collisions for every combination of segments, we arrive at three different scenarios (6 total but half are degenerate). However, this still leaves room for producing different answers given different representations of the network. If, for example the most appropriate collision is s_2-s_3 (cases 4 and 6 in fig. 3.2), the order in which it is performed can yield two slightly different structures after resolving the remaining collisions. However, this is preferrable over finding other, less favourable collisions that may lead to unecessarily complex structures.

We must also bear in mind that fig. 3.2 shows a minimal example for a case where three collision radii overlap, and the resulting structures are not dissimilar to one another. In fact, if the final junction formed is glissile, they should evolve similarly. However, as the code was now able to handle and evolve larger numbers of dislocations further than before, it became apparent that the collision algorithms needed to be improved. After all, as the number of candidate collisions increased, so did the chance for the code to pick the wrong one and for that to have a deleterious effect on the code.

3.4 Separation

One of the larger problems with high 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. The separation criteria is the power dissipation of the split configuration, compared to the power dissipation of the unseparated structure. The power dissipation is

given by

$$W_m = \sum_i^N \boldsymbol{v}_i \cdot \boldsymbol{f}_i \,, \tag{3.1}$$

where W_m is the power dissipation of splitting mode m, $\boldsymbol{v}_i$ is nodal velocity, and $\boldsymbol{f}_i$ the nodal force of node i participating in the splitting mode. Locality is assumed, so the computation of the pre-separation—reference—power dissipation involves only the node to be separated ($N = 1$); whereas the computation of the post-separation power dissipation involves the two nodes the node splits into ($N = 2$). The splitting mode chosen by the separation subroutine is that which dissipates the largest amount of power. If the pre-separation node has a higher power dissipation than any separation mode, it remains unseparated. In practice, the reference power dissipation is multiplied by a factor 1.05 to prevent nodes from flickering between being separated and colliding over multiple timesteps until the nodes can move far enough apart that they no longer collide.

Separation is a very poorly scaling operation. For a single multi-connected node we have,

$$M = \left(\sum_{j=2}^{\lfloor C/2 \rfloor} \frac{C!}{(C-j)!j!} \right) . \tag{3.2}$$

Where C is the number of connections a node has, j the number of connections in a splitting mode[1], and M is the total number of splitting modes. Mirror symmetries where $C = 2j$ half the number of possible splitting configurations in those cases. This type of scaling leads to a combinatorial explotion in computational complexity at higher connectivities. At best, this puts the lower bound at approximately Sterling's factorial formula $N! \approx \Omega(N \log N)$ for large N. Furthermore, 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 has the largest energy dissipation—provided it is higher than the reference configuration—is picked as the final configuration. Since locality is assumed, only the forces and mobilities of the nodes and segments directly involved in the split are accounted for, however this still involves an $\mathcal{O}(2N)$ call to the remote force calculation per splitting mode. Since the procedure must be performed for all nodes exceeding the maximum allowed

[1]This is the number of connections taken by one node after the split, the other would retain $C - j$ connections.

connectivity, it is extremely important that individual nodal connectivities as well as, the total number of highly connected nodes, remain as low as possible.

The solution may appear to be as simple as calling separation after resolving a single collision. Then looping through all collisions in a given timestep. However, this can cause collisions to be undone, leading to an infinite loop. Solving this was a long, multi-stage process. However, the solution is rather simple and elegant. We simply applied the power dissipation criteria to the collision-separation loop as described by algorithm 3.1.

Algorithm 3.1 Collision-separation algorithm with power dissipation.

1. Set up `s1Skip` and `s2Skip`, which are empty buffers for skipping pairs of corresponding segments in the collision detection.
2. Detect collision, skips corresponding segment pairs in `s1Skip` and `s2Skip`. Among the return values are the colliding segments `s1` and `s2`.
3. If no collision was detected, exit collision-separation loop and carry on with the simulation.
4. If a collision was detected, resolve collision. Among the return values are the pre-collision power dissipation and the node merged into, `mergednodeid`.
5. Separate multiply connected nodes, among the return values is the maximum power dissipation post-separation for node `mergednodeid`. If node `mergednodeid` is not separated, this value is zero. Otherwise, it is equal to the maximum power dissipation; be it from the unseparated `mergednodeid`, or the nodes it separated into.
6. If the power dissipation post-separation is larger than the power dissipation pre-collision, then clear `s1Skip` and `s2Skip`, update the network and auxiliary matrices, and return to 2. Else, add `s1` and `s2` to `s1Skip` and `s2Skip` respectively, don't update the network and auxiliary matrices, and return to 2.

This can lead to collisions being checked more than once, but in keeping with the quasi-static assumption, ensures the network is always in a first-order minimum energy configuration.[2]

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

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

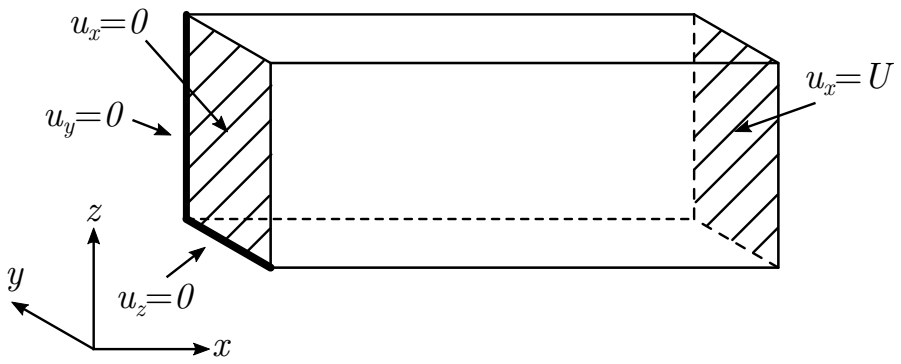


Figure 3.3: Displacement boundary conditions for dislocation plasticity modelling of single crystal, micro-tensile tests.

because nodes may need separating despite there being no collisions during the timestep.

It is worth noting that the power dissipation criterion has not been used in chapter 5 as this change was completed some time after the simulations started running. Instead, they use an out-dated method that simply checks whether the pre-collision and post-separation networks have the same number of nodes and segments. If they do, then the network and auxiliary matrices are not updated and the colliding segments get added to `s1Skip` and `s2Skip`, otherwise the buffers are emptied and the collision-separation is allowed to happen. This can miss some collisions, so after the collision-separation loop, we collide the rest of the nodes and exit. The outer call to separation deals with these as it sees fit.

This is an imperfect solution as a node may be separated into a different configuration than it was initially. It also can create highly connected nodes, but nowhere near as many or as highly connected as before. However, this was the method available to us at the time and was still a huge improvement over the previous one.

3.5 Surface remeshing

Most of the improvements to remeshing subroutines were done by Bruce Bromage, including the surface remeshing. His work on dislocation displacements necessitated the surface and virtual remeshing to be accurate. Among these requirements is the need to prevent dislocations from exiting out of surfaces with fixed displacements. If we take fig. 3.3, which describes the displacement boundary conditions for chapter 5, we see two surfaces upon which displacements are specified. Having dislocations exit those surfaces creates a slip step on them. This leads to very high stresses because the FEM essentially has to push the slip step back in order to make it conform to the displacement boundary conditions as in fig. 3.4. This is

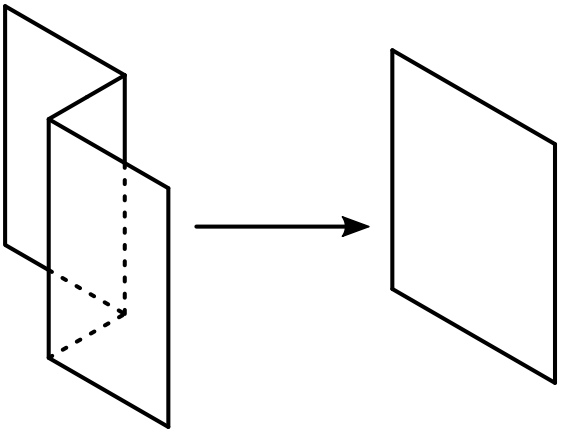


Figure 3.4: Slip step created by a dislocation exiting the volume has to be moved back to enforce displacement boundary conditions.

unphysical, as in reality the dislocations simply move into the bulk of the material, which is not modelled. In the bulk, they stop experiencing the Peach-Köhler force, because it is not subject to the forces or displacements of the experiment, so they pile up as the only forces they experience are those from themselves and other dislocations.

This was originally hard-coded for cantilever bending, but a general solution can be found with eq. (3.3),

$$D = \hat{\boldsymbol{n}} \cdot (\boldsymbol{x_o} - \boldsymbol{x_n}) \,, \tag{3.3}$$

where D is the signed distance from the point, $\boldsymbol{x_o}$, to a plane with unit normal, $\hat{\boldsymbol{n}}$, which passes through the point, $\boldsymbol{x_n}$. D is positive if point $\boldsymbol{x_o}$ is on the side $\hat{\boldsymbol{n}}$ points towards and negative otherwise. We can identify nodes that have exited from these surfaces by using their surface normal, a point on them, and whether D is positive. They can then be moved back to the surface and pinned.

This is an imperfect solution, but works quite well and removes these nodes from the mobility calculation. Perhaps a better solution would be to flag those nodes such that Peach-Köler forces, tractions, displacements, and surface remeshing criteria no longer apply to them. However, this will have to be implemented and tested as part of future improvements to the code.

3.6 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 scaling constants of the algorithms we use; $\mathcal{O}(N^2)$ for seg-seg forces and collisions; $\mathcal{O}(C!)$ for a single separation; $\mathcal{O}(MN)$ for tractions. Where N is the number of segments, C the number of connections, and M the number of surface elements. Simply put, we disproportionately benefit from having fewer items these expensive functions operate over.

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 can be and due to the high cost of the remote forces, but 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 [131, 150–152]. 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) [153]. One of the simples and most popular ways of doing so is the superposition method [154–156]. 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

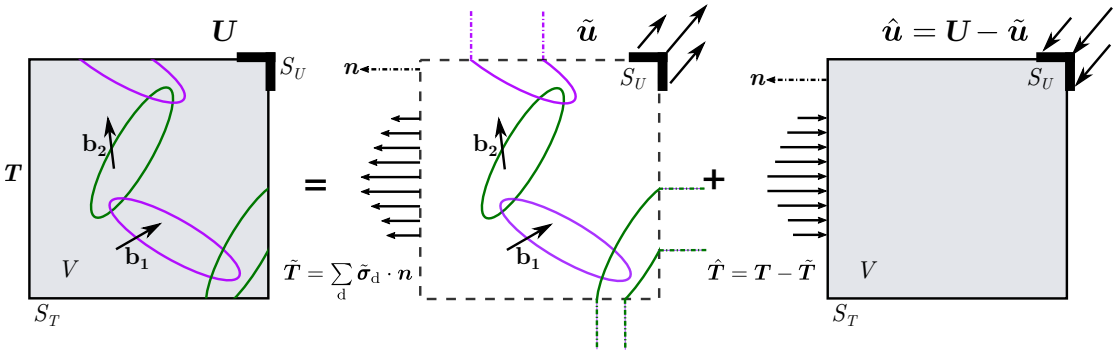


Figure 4.1: The superposition used to couple DDD and FEM. The volume V is bounded by a surface $S = S_T \cup S_U$ and contains a dislocation ensemble and is subjected to tractions $\boldsymbol{T}$ on S_T and $\boldsymbol{U}$ on S_u. First, the traction, $\tilde{\boldsymbol{T}}$, and displacement, $\tilde{\boldsymbol{u}}$, 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 $\hat{\boldsymbol{u}} = \boldsymbol{U} - \tilde{\boldsymbol{u}}$.

[102]. Formally, the dislocation field satisfies,

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

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

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

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,

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

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

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

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 [157]. This can be partially solved by using a non-singular formulation for $\tilde{\boldsymbol{\sigma}}$, such as that proposed by Cai et al. [102]. 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 [154, 157, 158].

A modified superposition scheme [159] can overcome this by dividing the problem into separate DDD-FEM problems coupled through an 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 [134] 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. [90] to use the non-singular formulation [102] 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 [160, 161].

We avoid the strict pole in $\tilde{\boldsymbol{\sigma}}$ by using the non-singular formulation described in [102], 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.

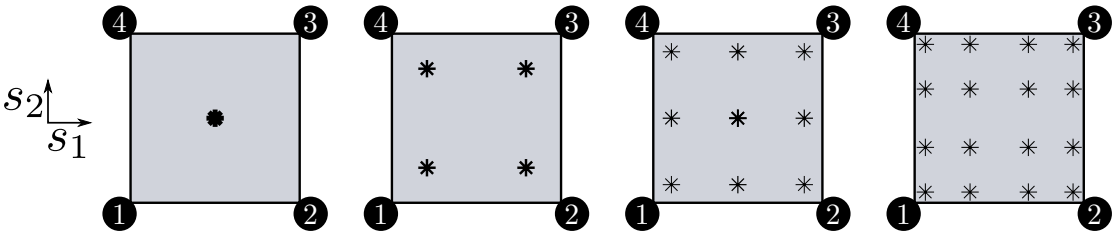


Figure 4.2: Examples of 2D Gauss-Legendre quadrature of the parent element with $Q = 1, 2, 3, 4$. The point size represents the weight w of the integration point. The parent elements are centred at the origin and $s_1, s_2 \in [-1, 1]$. We use an anticlockwise node numbering scheme.

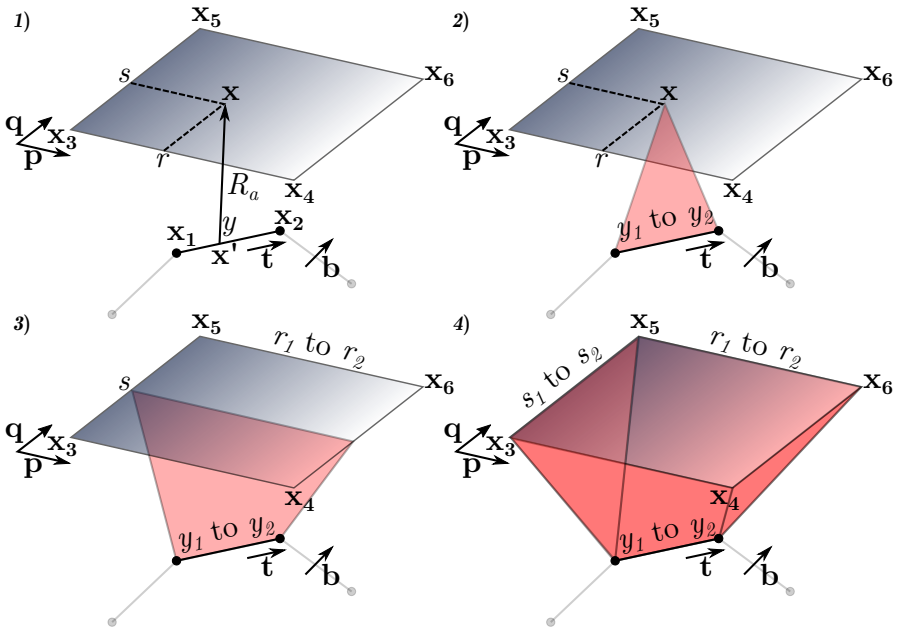


Figure 4.3: Diagram of the parametric line integrals solved by Queyreau et al. [90] to find the forces on linear rectangular surface elements.

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 [102], the dyadic form of the stress tensor produced by a straight, finite dislocation segment bounded by nodes at $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ [90] is,

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

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. [90] 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 [90].

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 [162], and screw dislocations [163, 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 [162] 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 [162] 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,

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

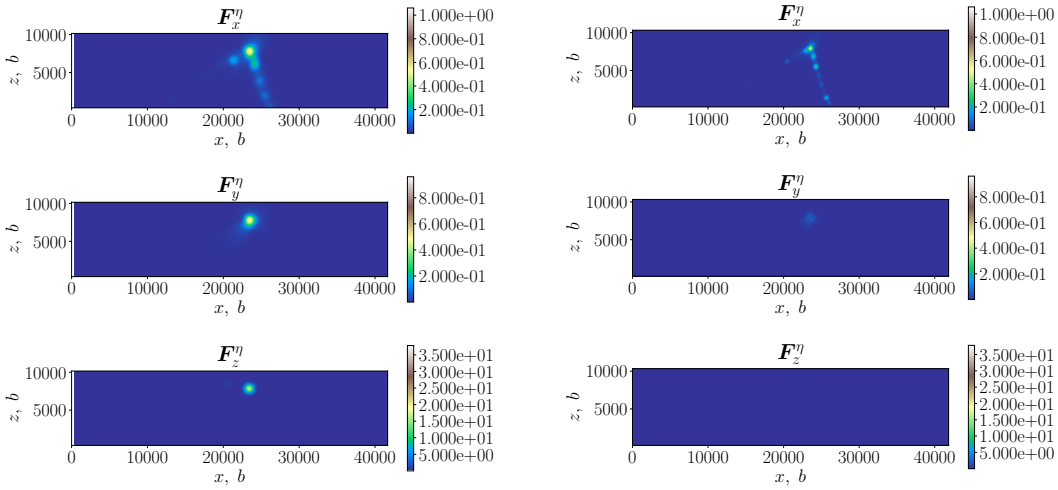
$$\sigma_{yy} = D\left\{(x-a)\frac{(x-a)^2+3(y-c)^2}{[(x-a)^2+(y-c)^2]^2}-(x+a)\frac{(x+a)^2+3(y-c)^2}{[(x+a)^2+(y-c)^2]^2}\right. \tag{4.22b}$$
$$\left.-2a\frac{(x-a)(x+a)^3-6x(x+a)(y-c)^2+(y-c)^4}{[(x+a)^2+(y-c)^2]^3}\right\},$$

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

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. [102], 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.

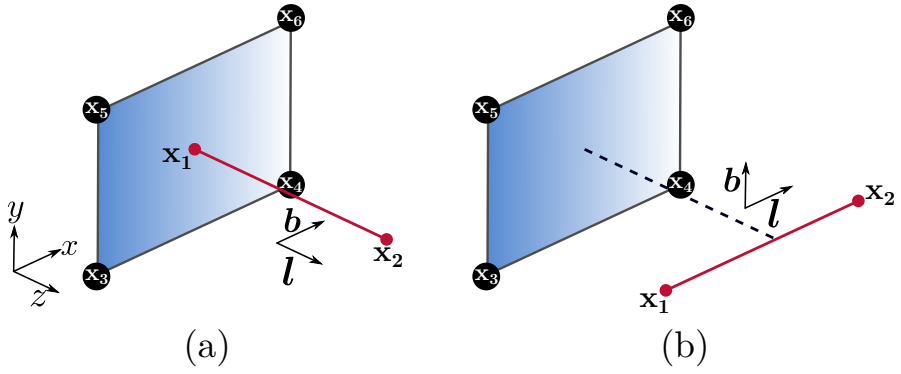


Figure 4.5: Simple test cases for an edge segment and surface element perpendicular (a) and parallel (b). The perpendicular dislocation is centered at the midpoint of the surface element, node $\boldsymbol{x}_1$ is separated by a perpendicular distance $\boldsymbol{x}_1{}^z$ to prevent the dislocation 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.

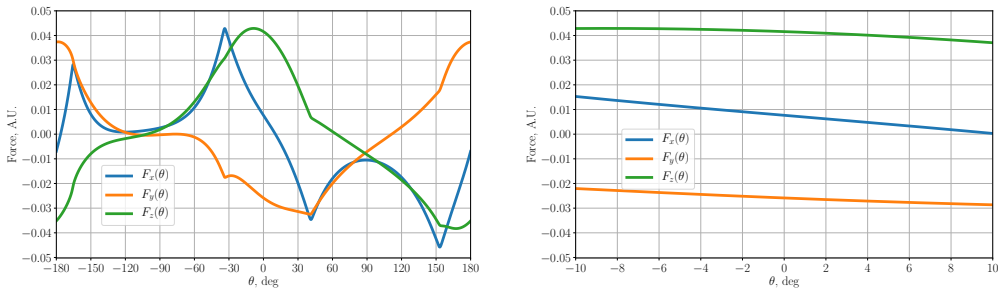
Queyreau et al. [90] identified that for a given number of quadrature points, the error is dependent on the dislocation character but always increases rapidly as the distance between the segment and element surface decrease (see figs. 4.9 and 4.10).

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. [90] 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 [90], 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,

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

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 $\boldsymbol{x_1}$, $\boldsymbol{x_2}$ 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 domain, this is dependent 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

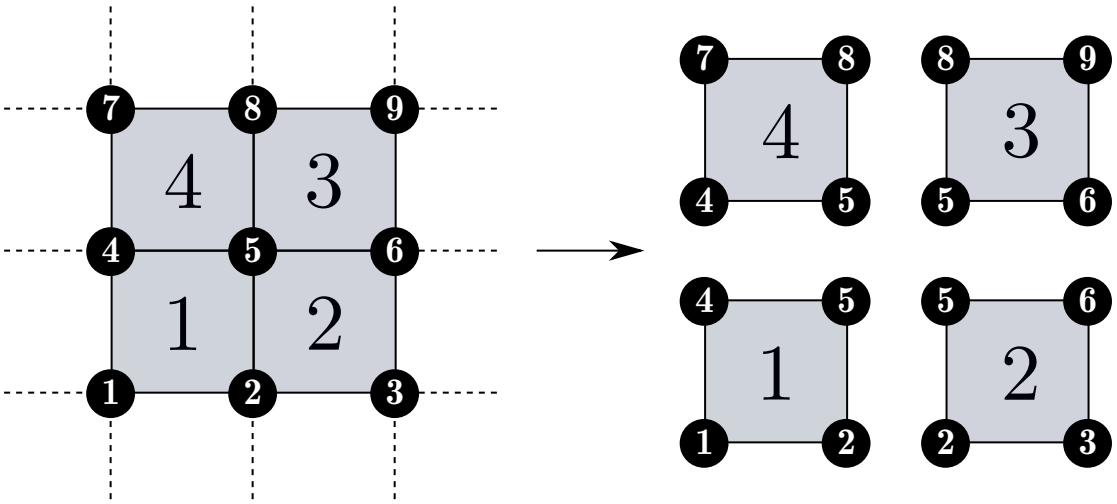


Figure 4.7: FE nodes are shared by either 4 element faces or 3 if it is a corner node. The total force on a given node is the summation of the force contributions from each element it belongs to.

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.

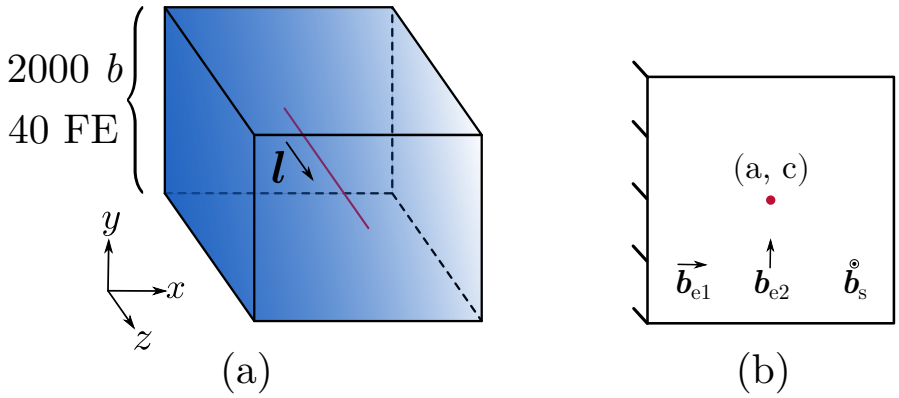


Figure 4.8: Dislocation parallel to a surface described by the line $x = 0$, where the $(x,\ y)$ dislocation coordinates are $(a,\ c)$ and the dislocation line going in the positive z-direction. (a) describes the box we used for our comparison, a $40{\times}40{\times}40$ element cubic box with side lengths 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 [162], $\boldsymbol{b}_{\text{e1}} \equiv \boldsymbol{b} = [1\,0\,0]$, $\boldsymbol{b}_{\text{e2}} \equiv \boldsymbol{b} = [0\,1\,0]$ and the screw Burgers vector $\boldsymbol{b}_{\text{s}} \equiv \boldsymbol{b} = \boldsymbol{l} = [0\,0\,1]$ in [163, 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 [162]; and $\boldsymbol{b}_{\text{s}} \equiv \boldsymbol{b} = \boldsymbol{l} = [0\,0\,1]$, as defined in [163, 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 [102], 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 [134], 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.

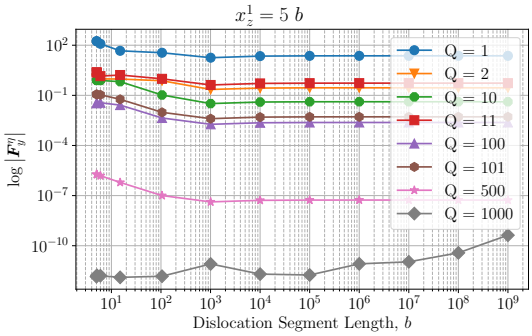
Parameter	Value
Crystal Structure	BCC
$\boldsymbol{b}$	$[1\,1\,1]$
$\boldsymbol{n}$	$[1\,\bar{1}\,0]$
$\boldsymbol{l}$	$[1\,1\,\bar{2}]$
b	$\sqrt{3}/2$
a	5
Grid Size	(20, 20, 20)
Domain Size	(2000, 2000, 2000)
Lattice size	3.18×10^{-4} μm
$(x_0,\, y_0)$	(62.5, 1000)
Min segment length	50
Max segment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

same rotation technique as [131]. 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.

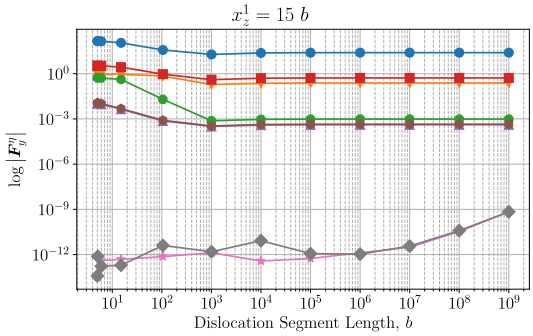
4.1.4 Results and discussion

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.

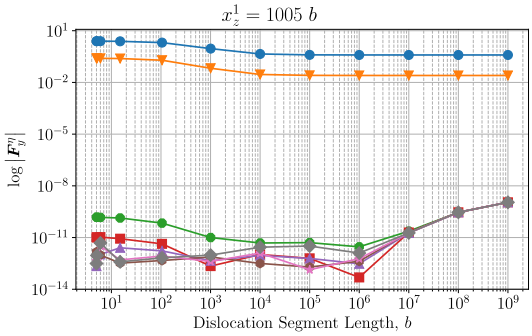
The limitations become even more evident when the dislocation line segment is parallel to a surface element. In fig. 4.10, we observe the relative errors are quite large when close to the surface. At distances larger than 10^4, loss of significance causes the relative errors to converge at ~ 1 which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



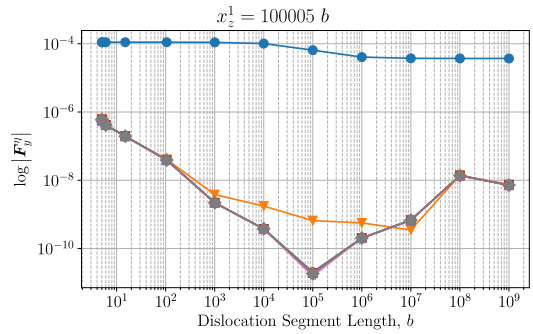
(a) Segment starts a single dislocation core radius away.



(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 $\boldsymbol{x_1}$. $\boldsymbol{b} = [1\,0\,0]$, 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.

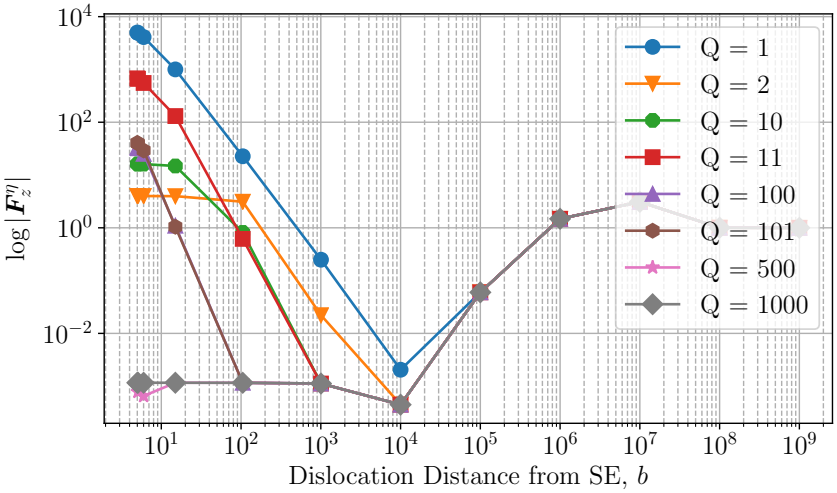


Figure 4.10: Log-log plot of the relative error as a function of distance from the surface element for a parallel edge dislocation (fig. 4.5b). Burgers vector, $\boldsymbol{b} = [0\,1\,0]$, line direction, $\boldsymbol{l} = [1\,0\,0]$, dislocation core and surface element parameters are the same as fig. 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

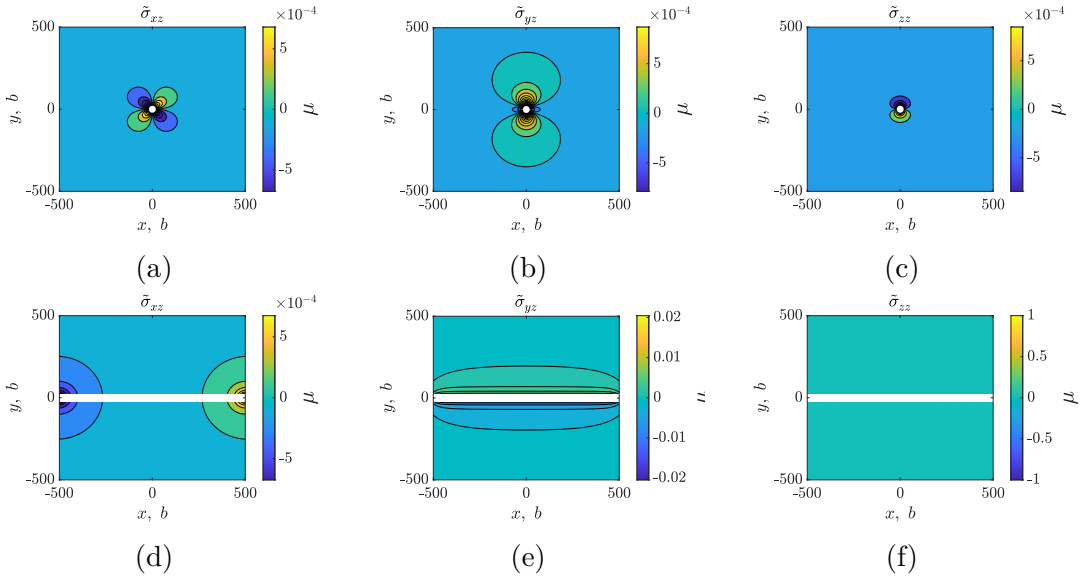


Figure 4.11: (a) to (c) show the real stress fields from a dislocation in the same configuration that yields the plots found in fig. 4.9 as shown in fig. 4.5(a), where the closest node is a dislocation core radius away from the surface, $a = 5$. (e) to (f) does the same for the configuration that yields 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 so many quadrature points are 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 numerous dislocations are close to the surface, such as nanoindentation simulations.

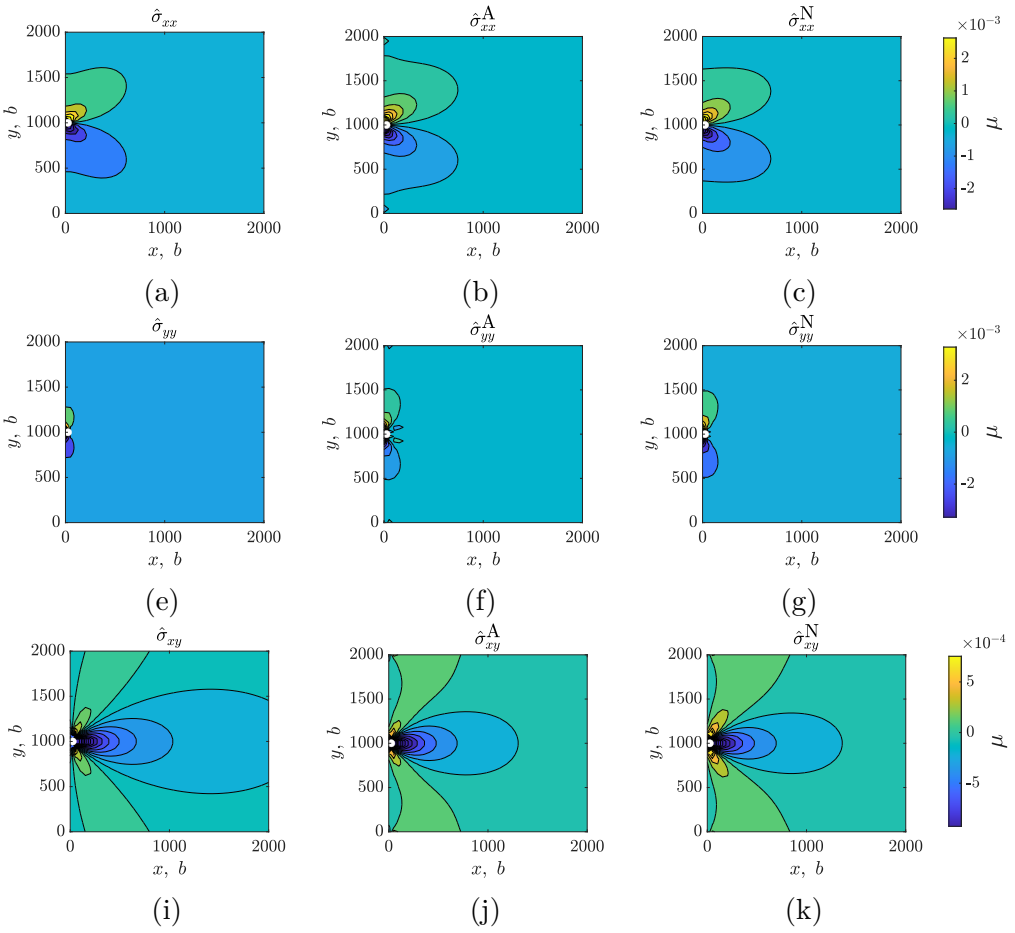


Figure 4.12: Image stresses for an edge dislocation with $\boldsymbol{l} = [001]$, $\boldsymbol{b} = [100]$, where $a = 5$, with coordinates (26, 1000), i.e. the centre of the first FE from the surface at $x = 0$, and in the centre of the simulation box along the y-direction. (a), (e) and (i) are the 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.

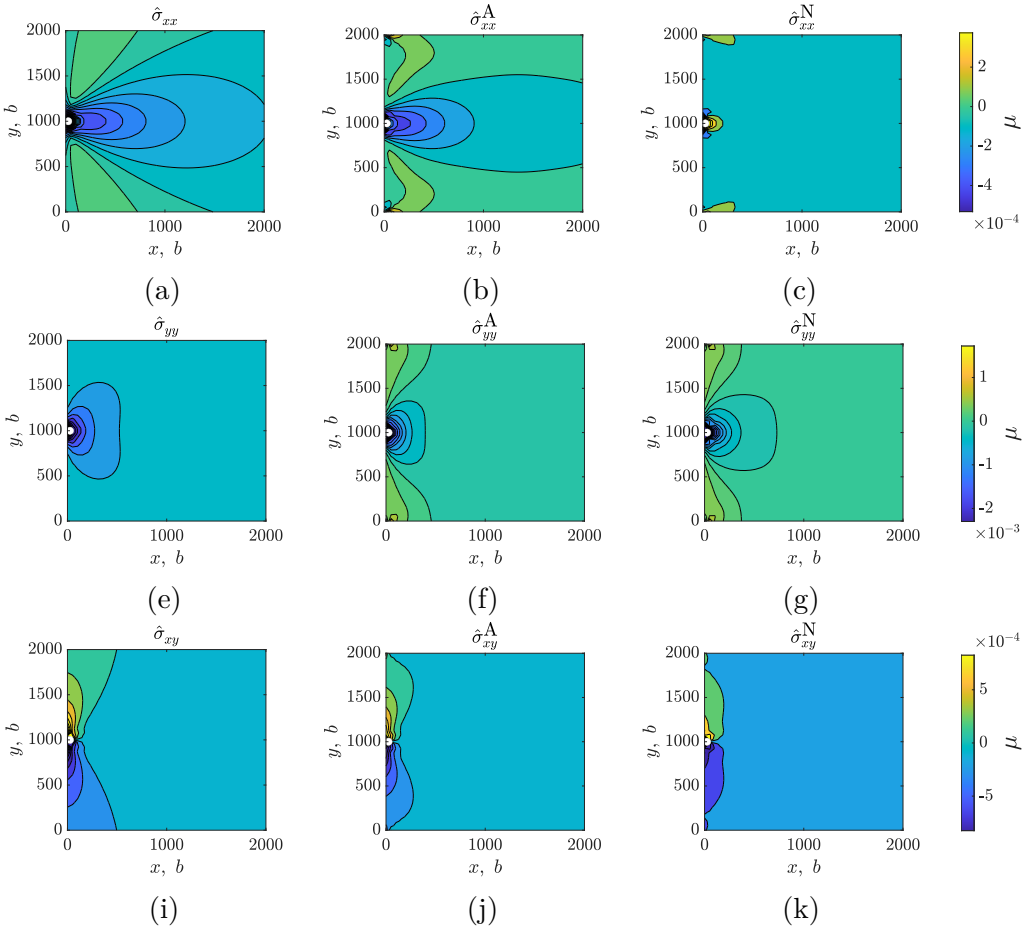


Figure 4.13: Image stresses for an edge dislocation with $\boldsymbol{l} = [0\,0\,1]$, $\boldsymbol{b} = [0\,1\,0]$, where $a = 5$, with coordinates (26, 1000), i.e. the centre of the first FE from the surface at $x = 0$, and in the centre of the simulation box from top to bottom. (a), (e) and (i) are 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.

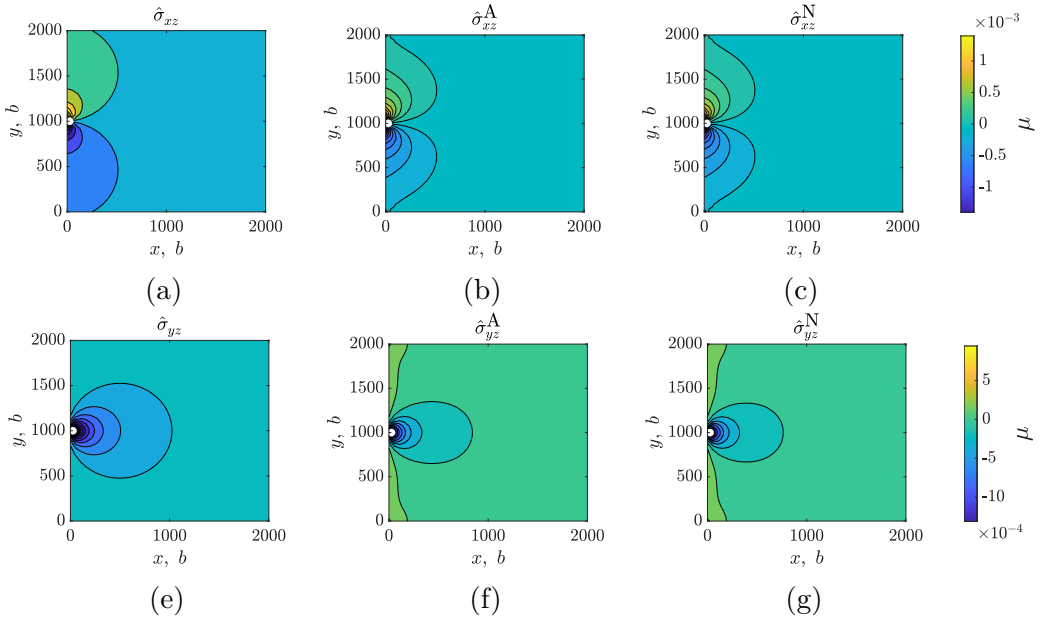


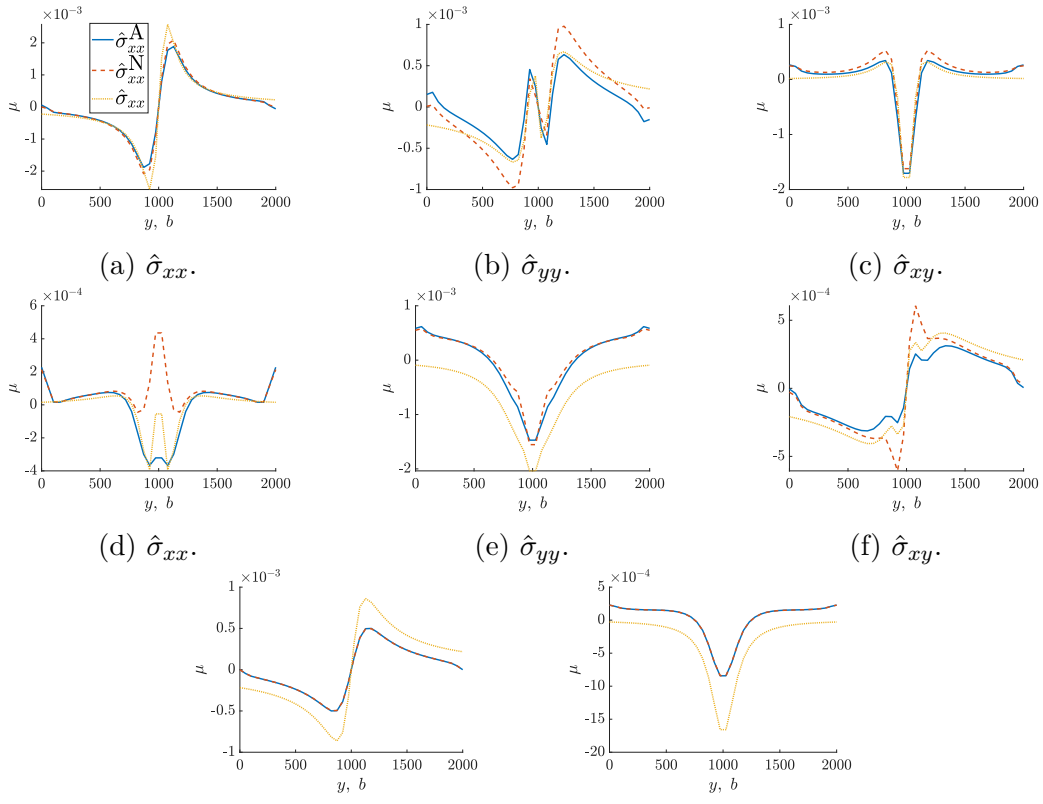
Figure 4.14: Image stresses for a screw dislocation with $\boldsymbol{l} = [0\,0\,1]$, $\boldsymbol{b} = [0\,0\,1]$, where $a = 5$, with coordinates (26, 1000), i.e. the centre of the first FE from the surface at $x = 0$, and in the centre of the simulation box from top to bottom. (a) and (e) are the 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 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

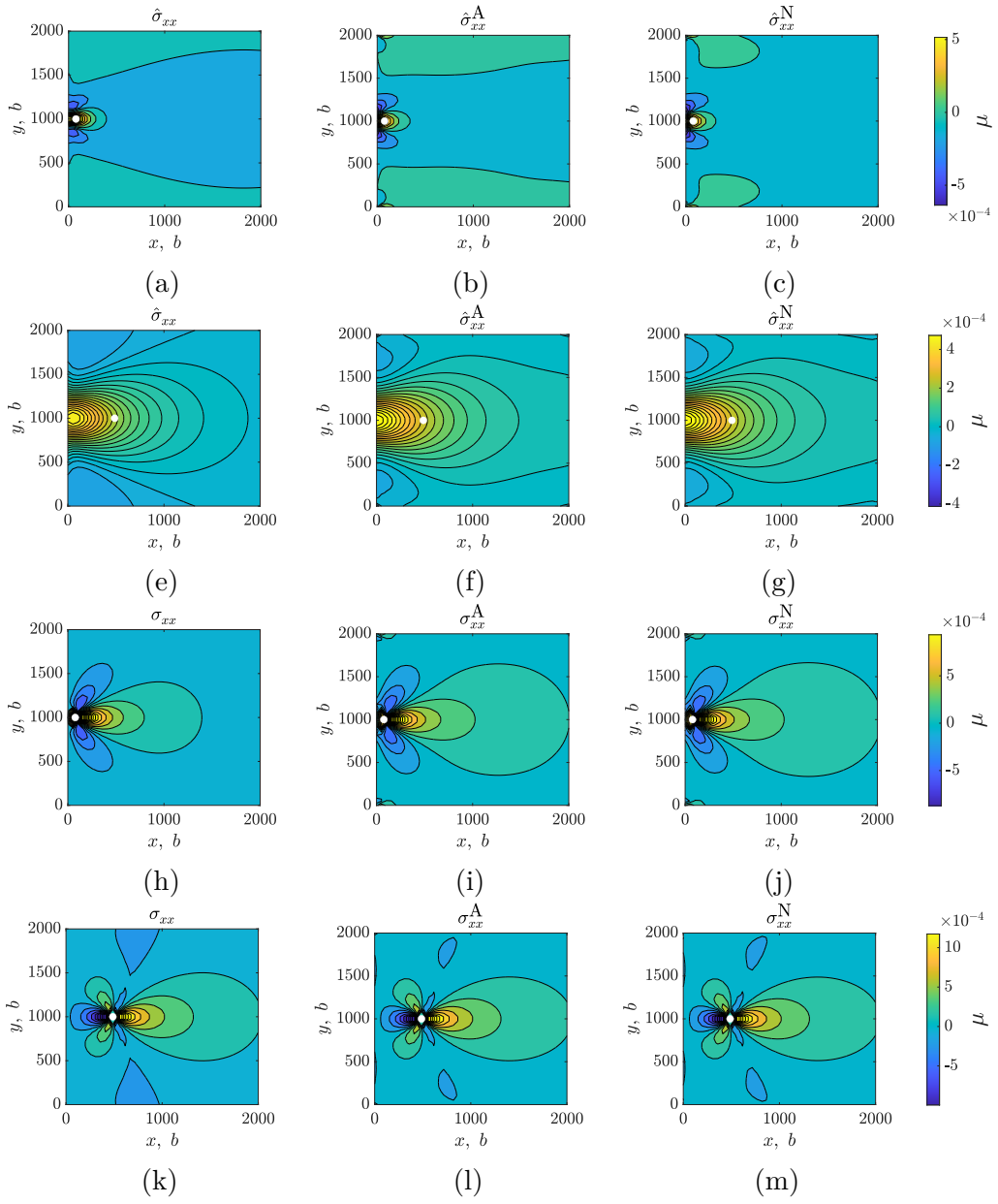
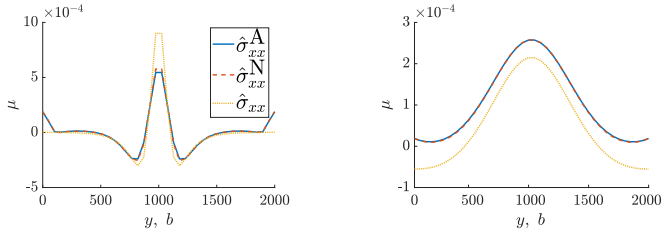
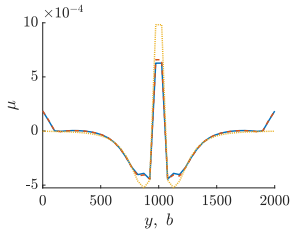


Figure 4.16: Stresses for an edge dislocation with $\boldsymbol{l} = [0\,0\,1]$, $\boldsymbol{b} = [0\,1\,0]$. Subfigures (a) to (g) show image stresses; (h) to (m) show total stresses. In subfigures (a) to (c) and (h) to (j) the dislocation is found at (77, 1000); in (e) to (g) and (k) to (m) the dislocation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

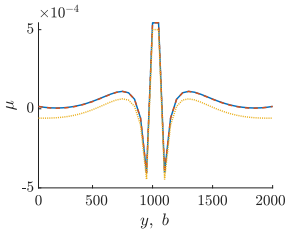


(a) $\hat{\sigma}_{xx}$.

(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 $x = 513$.

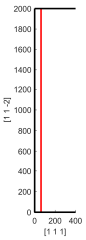
even the infinite-domain solutions such as in fig. 4.15d. Again, the general shape of the line plots is the same but the tendancy for numerical tractions to spike under specific circumstances is evident in almost every case.

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 $\boldsymbol{b} = \boldsymbol{b}_{\mathrm{e}2}$ 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 $x = 103, 513$.

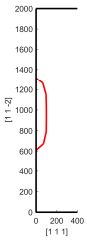
Notice that the image stresses in figs. 4.16a to 4.16c are all very close to fig. 4.13c, which is the stress field calculated via numerical tractions for a dislocation that is slightly closer to the surface. Essentially, the numerical tractions over or underestimated a set of forces something that become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

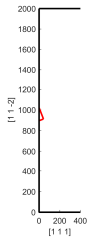
Lastly, to show how much numeric tractions can affect a result, fig. 4.18 shows a few snapshots of the simulation described in section 4.1.3. The figures shown are not at equivalent times because the simulations using numeric tractions ran indefenitely. The line tension equilibrated with the image forces 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



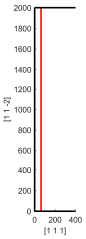
(a) Analytic tractions $t = 0$.



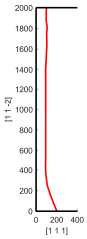
(b) Analytic tractions mid-simulation.



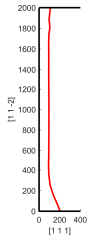
(c) Analytic tractions end.



(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 tendency 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 [90]), 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 [134] 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 could possibly be undertaken without many negatives. Furthermore, [90] 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 simulations. 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 [67–69], 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 [73, 164, 165]. 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 [166].

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 [62].

- 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.

NVidia GPUs have highy specialised architecture shown in fig. 4.19 which was kindly made available via the "Creative Commons Attribution 4.0 International" licence by Hernández et al.. The unit of processing is called a thread. Warps are physical collections of 32 threads that act as a single unit under a single scheduler, i.e. all threads in a warp execute the same instruction on different data. This type of computing model is known as SIMD (Single Instruction Multiple Data). As an aside, CPUs can also perform SIMD operations within vector modules. In contrast to threads in CPUs, threads in GPUs are not independent, they are always part of a warp. Thread blocks, simply known as blocks, are collections of threads assigned to a Streaming Multiprocessor (SM). All threads in a multiprocessor share common resources like shared memory and a register file. This means the total memory used by all threads in a block must not exceed the maximum amount of memory per block. If this happens, the programme may fail to execute or crash. Blocks are then aggregated into grids, which are equally but non-deterministically distributed across all SMs. More information can be found in NVidia's programming guide [62].

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 many algorithms. Doing so poorly can result in decreased

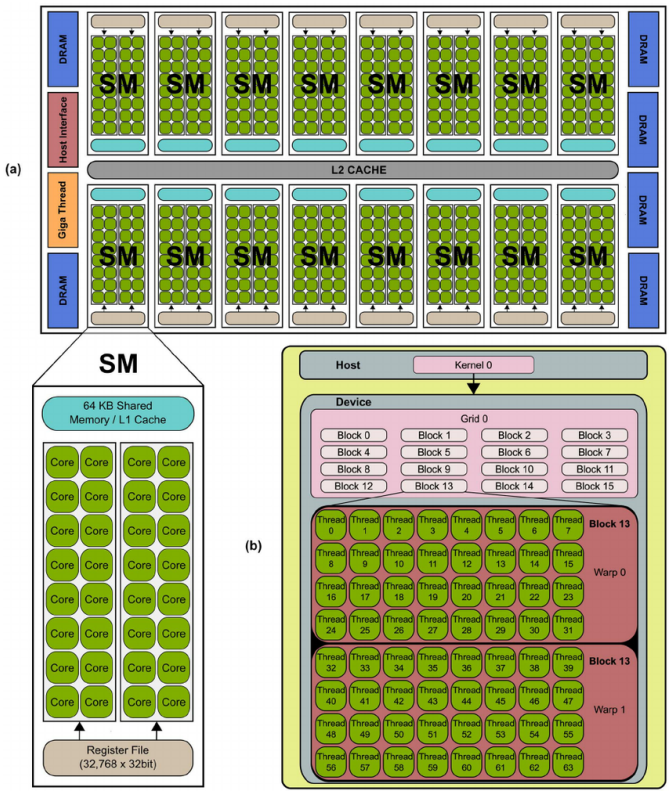


Figure 4.19: (a) The typical NVidia GPU is made up of a set of Streaming MultiProcessors (SM). Each SM controls a number of CUDA cores, also known as Streaming Processors (SP). (b) Programmers can control GPU resources through this abstracted model. Image kindly provided through the "Creative Commons Attribution 4.0 International" licence by [1]

performance, particularly when using high performace/scientific computing cards. Maximising resource use is also important. In particular, for CUDA applications each block should aim to contain as many, fully-occuppied warps (all 32 threads working on relevant data) as possible without exceeding the available memory in a block. Furthermore, memory access patterns are fundamentally different for GPUs, given how threads are aggregated into warps. 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 new data arrives. As processors have got faster, memory speeds have not kept up. As a result, minimising the ratio of cache misses to hits has become an important aspect of optimising software applications [167, 168]. 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 [169–171]. 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 in-

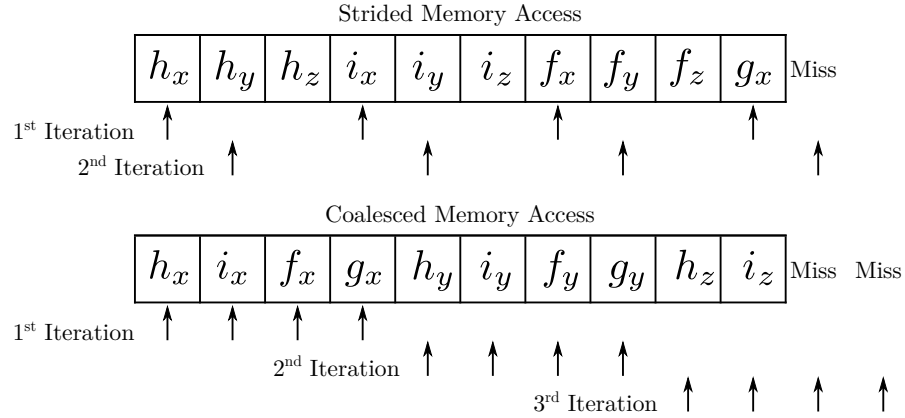


Figure 4.20: Memory access patterns. Arrows are threads in a warp simultaneously requesting access to memory from the cache. All processors work simultaneously, so they can only proceed until every one of them has the necessary data.

struction/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 4.20 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. 4.20, it's easy to see how this is suboptimal. To start with, a new cache line is needed 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 is as follows,

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

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

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

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

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

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. 4.21, parallelising over the surface elements with node

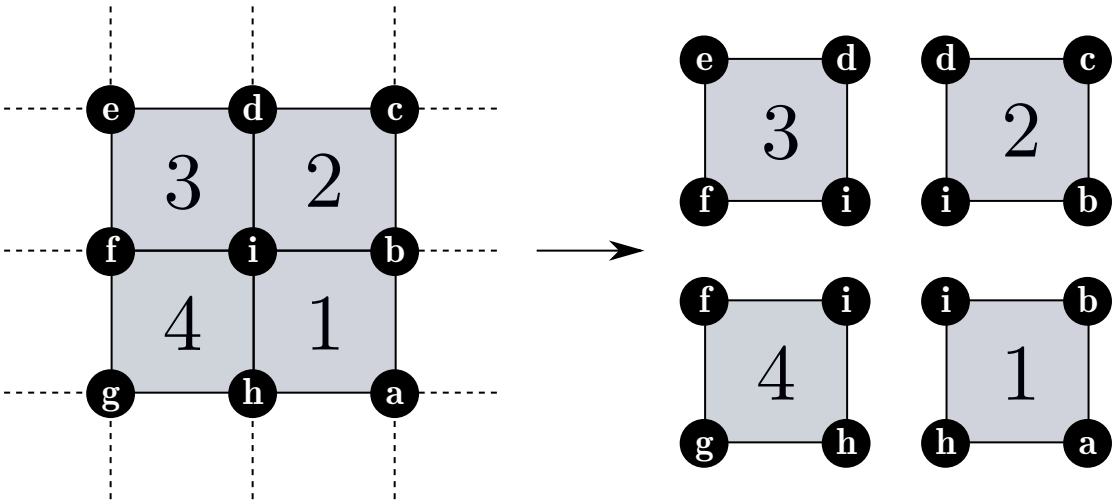


Figure 4.21: Each linear rectangular surface element is mapped to one thread.

labels corresponding to fig. 4.3, $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $\boldsymbol{x_3}$ and $\boldsymbol{x_4}$ yields,

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

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,

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

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 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 4.22 has an example of how this works for a block of two threads and a set-up as described by fig. 4.21 and mapping by eq. (4.28).

There are $6{*}3{*}8{+}6{*}8 = 192$ bytes of information in each unit problem: 6 nodes with 3 coordinates each, all 8 bytes each (double precision), plus 6 pointers of 8 bytes each[2]. We must consider this when computing the number of threads because GPUs have a finite amount of shared memory per block. If we go above this threshold the GPU will error, which under normal circumstances is fine because

[2]Therea are 6 arrays where the are stored, each array needs a pointer, each pointer is 8 bytes in x64 architecture

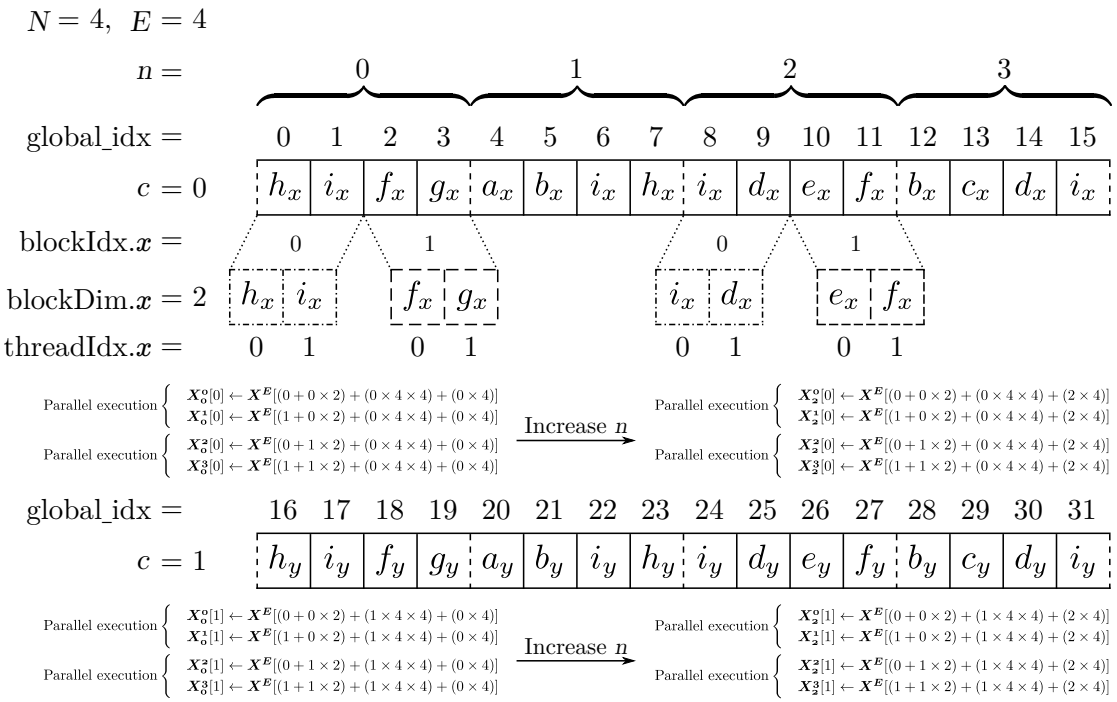


Figure 4.22: Minimum working complete example of a thread block obtaining data from global memory, from parallelising over the four surface elements in fig. 4.21 and mapped by eq. (4.28). $\boldsymbol{X}_{\boldsymbol{a}}^{\boldsymbol{b}}$ denotes a 1D array of length three containing the xyz-coordinates 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.

the error causes a crash, but when `MATLAB` interfaces with GPUs, the error does not get reported back and the programme keeps running without having calculated anything.

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 effectively as possible,

$$b = 192 \tag{4.33}$$

$$T_{\max} = \min(T_{\text{block}}, floor(M_{\text{block}}/b)) \tag{4.34}$$

$$T = 32\,\lceil (n \mod T_{\max})/32 \rceil \tag{4.35}$$

$$G = \lceil (n + T - 1)/T \rceil \ . \tag{4.36}$$

Where b is the total number of bytes per unit problem; $T_{\max}$ is the maximum number of threads the computation can use without exceeding the shared memory per block; T_{block} is the maximum number of threads a block can have if they do not exceed the shared memory; M_{block} is the shared memory per block; T is the number of threads per block that exactly fills as many warps as possible, without exceeding the shared memory per block; n is the number of unit problems; G is the number of blocks that distributes the computation most evenly. This ensures any GPU that runs the computation will always perform optimally for a given number of unit problems. The heuristic was also added to the GPU parallelisation of the remote force calculation, where the unit problem size is the same.

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). GPUs cannot branch, so expensive special cases can be devastating for GPU parallelisation. 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. Par-

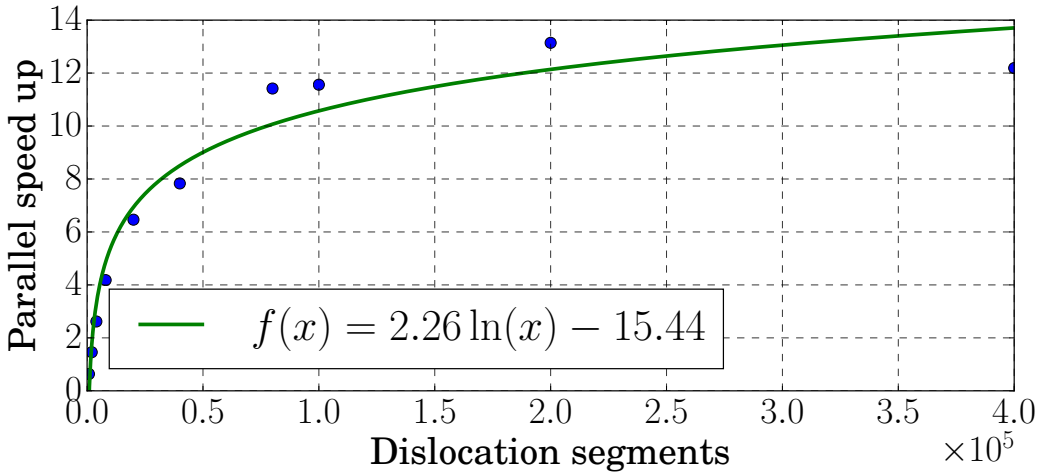


Figure 4.23: Parallel speedup on an NVidia GTX 750 and an Intel Core i5-8500 @ 3.0 GHz.

allelising over special cases is not worth the time because they are not common enough, so the best solution is to 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. Unfortunately, this is still less than ideal because the CPU has to still go over the whole network identifying special cases, which means the calculation can end up being CPU-bound if the GPU finishes before the CPU does. A potential solution would be to scrap this separation and simply perform the special case in the GPU as well. However, this means at least $(1+2r)\times$ more GPU computations, where r is the number of rotations of the parallel segment in each direction[3]. At the time of implementation, this was deemed unacceptable but may become viable as GPUs improve.

4.2.3 Results and discussion

One of the greatest disapointments of this project can be seen in fig. 4.23. 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 helps, 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 do not have the capacity to run a simulation with

[3]Without branching, the GPU would have to perform the standard and perturbed computations for every single unit ragardless of whether it is a special case or not. The rotations are performed in the clockwise and anticlockwise directions, hence $2r$.

10^5 segments. The $(O)(N^2)$ collision and remote force algorithms and $(O)(N!)$ separation prevent us from going much past 10^4 segments. Moreover, fig. 4.23 was obtained using only non-parallel segments, the code checking for parallel ones in the CPU was not even compiled for these tests. Therefore, the speed-up is purely from the GPU. As previously stated, checking for the special case means the CPU can lag behind the GPU in some scenarios, reducing the true gains by a noticeable margin.

That said, when running simulations on a Windows 10 workstation with an NVidia Quadro GP100 and an Intel Core i9-7900X CPU 3.30 GHz, the break-even point is ~ 500 dislocation segments. The Intel i9 series are known for high single-threaded performance, so they reduce the relative effectiveness of parallelisation. Machines with the same graphics card and whose single-threaded CPU perforamnce is worse, will see greater benefits the GPU parallelisation. Moreover, as simulations grow in number of segments, so do the advantages of using the GPU. For simulations such as those presented in chapter 5, the GPU gives an ~ 8–$15\times$ speed-up (only for the traction calculation, not the whole simulation)—this is for ~ 5–20 thousand dislocation segments, fewer than those and the gains aren't very significant. However, with such a low break even point it's recommended that simulations on computers with high performance GPUs use this parallelisation once there are more than ~ 500 segments.

Though it is not the $> 100\times$ speed-up we were hoping, an order of magnitude in the analytic traction calculation during real simulations is very worthwhile. This is especially true compared to the remote force parallelisation, which uses memory less efficiently. In fact, in the same simulations on the same computer, its break-even point is ~ 5–$10\times$ higher, and the acceleration is not as significant, at ~ 3–$4\times$ over the serial implementation at the exact same points in the simulations where tractions give ~ 8–$15\times$. This difference is down to the fact that the traction calculation uses a contiguous memory access pattern, as compared to the remote force function which uses contiguous memory access for one loop over all segments, but not the other [86].

4.3 Conclusions

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, 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. Fortunately, the parallelised code is close to optimal, so its performance will improve as the rest of the software and hardware catch up.

As it stands, it is worthwhile for the Tarleton group to use this parallelisation in most simulations (once they get past 500 segments). However, other users may see greater or lesser benefit from it. This is typical of hardware acceleration, where the relative performances of the CPU and GPU come into play. Even components such as the motherboard and RAM play a role, so the use of GPU acceleration cannot be blanket recommended. Each PC is different, therefore anyone hoping to use GPU acceleration must perform their own tests to identify when they should leverage it.

Chapter 5

Simulations

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.

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

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

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

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

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.

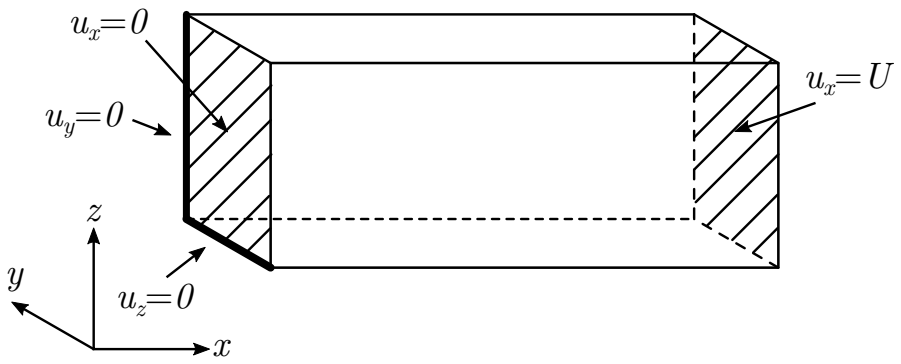


Figure 5.1: Displacement boundary conditions for dislocation plasticity modelling of single crystal, micro-tensile tests.

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 [130], 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 [131] 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 [172], necessitates spontaneous generation of dislocations as a network evolves. Including stochasticity in the form of Langevin dynamics [173] 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` [130]. 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` [174]. A paradigm-shifting, open-source JIT (just in time) compiled[1] 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

[1] Although as of `v1.6`, it might be more appropriate to say call `Julia` a JAOT (Just Ahead Of Time) compiled language thanks to its massively improved precompilation capabilities.

autoparallelise functions on different architectures 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, such as symbolic and automatic differentiation.

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. It also has unicode support and sports all the interactivity and useability of 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 single-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, from left to right they link to 1. the documentation; 2. automated, remote testing; 3. two test coverage reports i.e. which lines of code have been hit during testing and how many times; and 4. an interactive Jupyter notebook on Binder, which runs on a browser and has no external dependancies.

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. It has been a didactic, after hours project to escape the anxieties of the pandemic and ease the frustrations with some fundamental design aspects of EasyDD (many carried over from DDLab). But it has also been an attempt to understand how EasyDD can be improved without all the clutter, risk, and technical debt (some solutions were even trialed in DDD.jl and backported to EasyDD). It is therefore not feature-complete. Nevertheless, it has turned into a promising weekend project for which we can provide a sample of current capabilities.

6.1 DDD.jl: 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 it is clunky, unelegant and opaque. Here we solve both issues in a single stroke and offer greater flexibility.

6.1.1 Parameter definition

Setting up 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` ecosystem. In some cases the loaded data has to be used to manually create the structures, but if `JLD2` is used, it works like `MATLAB`'s `save()` and `load()`. JLD2 is advantageous because it saves structure information so loaded variables will be of the correct type, however it is specific to `Julia` and is still under active development.

2. Use the built-in serialiser. This works like `MATLAB`'s built-in `save()` and `load()` functions. This is another great option that is in-built to the language, but is also `Julia`-specific and successfully reading files is only guaranteed if they were generated with the same version of `Julia` being used.

3. Use the specially defined functions that load `JSON`[2] 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. It is also quite advantageous for sharing data because they have high compressibility ratios and anyone can open a JSON file to have a look at the data. We have internal functions that validate the data and initialise the structures from these files. However, any changes made to the data structures must be accounted for in the IO functions.

[2] `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.

4. Create the structures manually. This can be done by directly using the structure constructors or by using the keyword constructors. When manually creating the variables, using the keyword functions is preferred as they perform validations, calculate derived values, and provide sensible defaults.

Here we show how easy it is to manually set up a simulation. We have to start by defining our parameters.

Almost all of our keyword constructor functions provide default values in case the user does not provide them. They also perform validations on the data to ensure it is sensible, for example the maximum length of a dislocation line must be larger than the minimum length. For demonstration purposes, we provide some values and let the code figure out the rest from what we gave it. This means function signatures can be even shorter than what is shown here.

```
# Dislocation parameters
dlnParams = DislocationParameters(;
    mobility = mobBCC(),
    dragCoeffs = (edge = 1, screw = 1e-1, climb = 1e9, line = 1e-5),
    coreRad = 0.015 * sqrt(2),
    minSegLen = 0.15 * sqrt(2),
    maxSegLen = 1.5 * sqrt(2),
    coreEnergy = 1 / (4 *  ) * log(0.015 * sqrt(2) / 3.5e-5),
    coreRadMag = 3.5e-4
)
# Material parameters
matParams = MaterialParameters(;
    crystalStruct = BCC(),   = 1,  Mag = 80e3,   = 0.25
)
# FEM parameters for a regular cuboid mesh with linear elements
# with the purpose of modelling a cantilever loading experiment
femParamsC = FEMParameters(;
    type = DispatchRegularCuboidMesh(),
    order = LinearElement(),
    model = CantileverLoad(),
    dx = 23.0, # Length in x
    dy = 17.0, # Length in y
    dz = 13.0, # Length in z
    mx = 7, # Elements in x
    my = 5, # Elements in y
    mz = 3  # Elements in z
```

```
)
# Slip system information for two BCC slip systems
slipSystems = SlipSystem(;
    crystalStruct = BCC(),
    slipPlane = Float64[-1 1; 1 -1; 0 0],
    bVec = Float64[1 1; 1 1; 1 -1]
)
# Integration parameters
intParams = IntegrationParameters(;
    method = AdaptiveEulerTrapezoid(),
    abstol = dlnParams.collisionDist / 2,
    reltol = dlnParams.collisionDist / 2
)
# Integration time variables
intTime = IntegrationTime(; dt = 0.0, time = 0.0);
```

All together, these represent about 60 constants that control all aspects of a simulation. But we're only just getting started.

Julia is a fully compiled language, where files can be run from a terminal like any other compiled language. But because it is JIT-compiled, we can ask it questions about our variables, types, functions, etc.[3]

```
julia>?femParams
search: femParamsC femParamsP FEMParameters loadFEMParameters

  No documentation found.

  femParamsC is of type FEMParameters{DispatchRegularCuboidMesh,
  LinearElement, CantileverLoad, Float64, Float64, Float64,
  Int64, Int64, Int64}.

  Summary
  ========
```

[3]We represent interactive REPL (Read Eval Print Loop, i.e. Julia's "terminal"), use by omitting line numbers. Moreover, the LaTeX package, minted, does not accurately represent REPLs so the syntax highlighting will be off in these cases. Regardless, Julia uses Markdown natively, so it produces documentation and answers such queries in Markdown-formatted text. Documentation is canonically written as Markdown docstrings (strings of text in Markdown format) within the source, which allows one to browse it interactively. The standard-library documentation package can then be set up to generate HMTL pages and hooked up to a remote testing service which builds and pushes the generated documentation onto the git webhosting service of choice. If using GitHub, one can use "GitHub Tasks" to directly build and host documentation there. Other modern languages have similar capabilities.

```
struct FEMParameters{DispatchRegularCuboidMesh, LinearElement,
CantileverLoad, Float64, Float64, Float64, Int64, Int64,
Int64} <: Any

Fields
========
type  :: DispatchRegularCuboidMesh
order :: LinearElement
model :: CantileverLoad
dx    :: Float64
dy    :: Float64
dz    :: Float64
mx    :: Int64
my    :: Int64
mz    :: Int64
```

6.1.2 Generation of FE mesh and boundary conditions

With the parameters created, we are now free to build our FE mesh and generate our dislocations. Building and plotting a mesh according to our parameters is simple enough.

```
1 meshC = buildMesh(matParams, femParamsC)
2 plotFEDomain(meshC; camera = (10, -2))
```

The mesh is built using the canonical numbering system defined in [175], and builds the node sets accordingly (shown in fig. 6.1).

`Julia` has a parametric type are called, `NamedTuple`. Which are key-value pairs that can be accessed via indices (as one would a tuple), their keys (as one would a dictionary), or dot syntax (as one would a structure).

```
julia> d = (a = 1, b = 2 + im, c = false)
(a = 1, b = 2.0 + 1.0im, false)
julia> d[1]
1
julia> d[b]
2.0 + 1.0im
julia> d.c
false
```

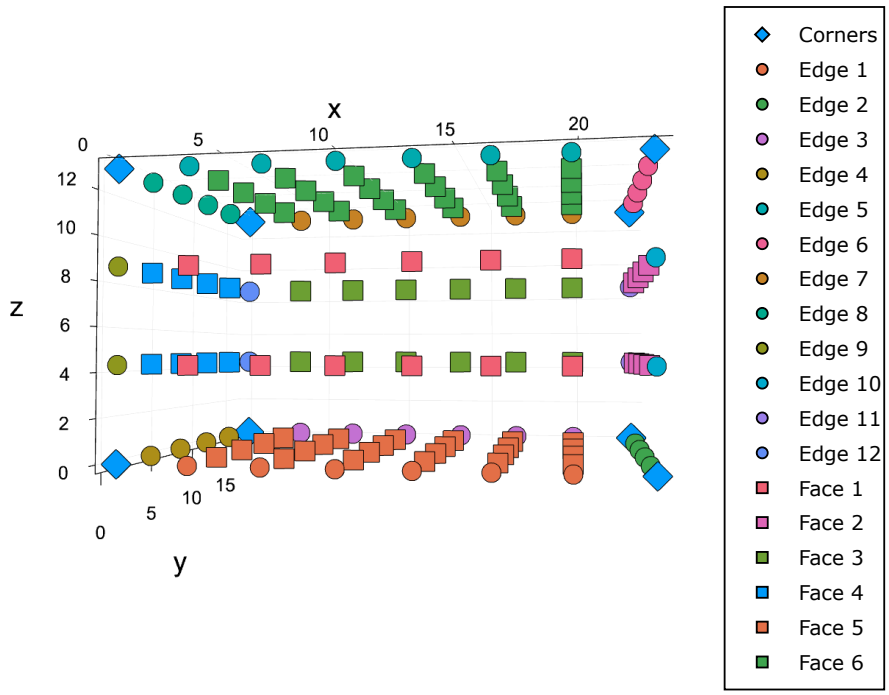


Figure 6.1: Canonically defined finite element node sets.

They act like on-the fly structures and are very convenient when making composable and generic code. The variable `dlnParams`, keeps the drag coefficients in a `NamedTuple`, which can be accessed through the `dragCoeffs` field. This makes swapping mobility functions a breeze without modifying existing source code. `NamedTuples` are also used in the FE mesh data structure, where they represent surface node sets.

```
julia> keys(meshC.surfNode)
(:x0y0z0, :x1y0z0, :x1y1z0, :x0y1z0, :x0y0z1, :x1y0z1,
 :x1y1z1, :x0y1z1, :x_y0z0, :y_x1z0, :x_y1z0, :y_x0z0,
 :x_y0z1, :y_x1z1, :x_y1z1, :y_x0z1, :z_x0y0, :z_x1y0,
 :z_x1y1, :z_x0y1, :xz_y0, :yz_x1, :xz_y1, :yz_x0,
 :xy_z0, :xy_z1)
```

The keys say exactly what they represent, and are canonically ordered [175]. Corners come first (`:x0y0z0`), then edges (`:x_y0z0`), and finally faces (`:xz_y0`), making the creation of different boundary conditions a very easy, self-documenting, literal process.

We can easily generate the boundary conditions for our model, as well as the sparse vectors that will hold the loading information.

```
# Generate boundary conditions and sparse vectors
# that will hold loading information
```

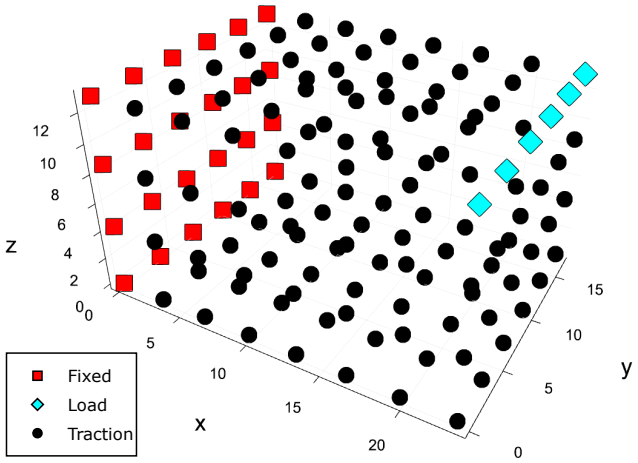


Figure 6.2: DDD.jl canonical cantilever loading boundary node sets.

```
boundaryC, forceDispC = Boundaries(femParamsC, meshC)
# Plot boundary conditions
figCBC = plotBoundaries(boundaryC, meshC)
```

This uses our canonical definition of the boundary conditions (fig. 6.2), but users can modify them through keyword arguments. Through the magic of multiple dispatch, we can also create boundary conditions for pillar loading fig. 6.3. Here we went with a different mesh, but we could just as easily have used the same one as before, as long as the `model` parameter given to the `FEMParameters` is `PillarLoad()`.

```
# Pillar loading
femParamsP = FEMParameters(;
    type = DispatchRegularCuboidMesh(),
    order = LinearElement(),
    model = PillarLoad(),
    dx = 17.0,
    dy = 13.0,
    dz = 23.0,
    mx = 5,
    my = 3,
    mz = 7,
)
meshP = buildMesh(matParams, femParamsP);
boundaryP, forceDispP = Boundaries(femParamsP, meshP);
```

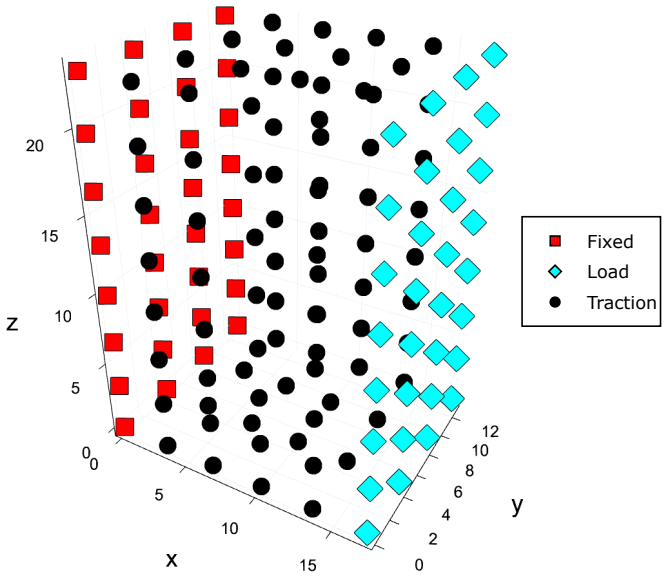


Figure 6.3: DDD.jl canonical pillar loading boundary node sets.

6.1.3 Generating dislocations

Generating dislocation sources is also quite easy. We can use some information from the simulation parameters to generate two types of dislocations, 1. prismatic loop with 8 sides, 8 nodes and 8 segments with $\boldsymbol{b} = [1\,1\,1]$ and $n = [\bar{1}\,1\,0]$, 2. and a shear loop with 5 sides, 10 nodes and 10 segments with $b = [1\,1\,\bar{1}]$ and $n = [1\,\bar{1}\,0]$. Both structures represent 5 loops with segment lengths equal to `segLen`, which will generate random uniformly distributed loops in $x = [0, dx]$, $y = [0, dy]$, $z = [0, dz]$.

```
# Julia supports multiple assignments in a single line
# Assign the domain size and length scale for the dislocation
# segments.
dx, dy, dz = femParamsC.dx, femParamsC.dy, femParamsC.dz
segLen = (dlnParams.minSegLen + dlnParams.maxSegLen) / 2

prismOct = DislocationLoop(;
    loopType = loopPrism(), # Prismatic loop
    numSides = 8,    # 8 sides
    nodeSide = 1,    # 1 node per side (corners)
    numLoops = 5,    # Will generate 5 loops in the network
    segLen = segLen * ones(8), # Each side is of equal length
    slipSystemIdx = 1,  # Slipsystem 1
    slipSystem = slipSystems,    # Available slip systems
```

```
    label = nodeTypeDln.(ones(Int, 8)), # Node type
    buffer = 0, # Spacing for the distribution, may change in the future
    range = [0 dx; 0 dy; 0 dz], # Range on which to distribute
    dist = Rand(),  # Random uniform distribution in space
)
shearPent = DislocationLoop(;
    loopType = loopShear(), # Shear loop
    numSides = 5,
    nodeSide = 2,   # Two nodes per side (corners and mid-segment)
    numLoops = 5,
    segLen = segLen * ones(10), # 10 segments per loop this time
    slipSystemIdx = 2,
    slipSystem = slipSystems,
    label = nodeTypeDln.(ones(Int, 10)),
    buffer = 0,
    range = [0 dx; 0 dy; 0 dz],
    dist = Rand(),
)
```

When the loops are generated, only one of each is made, and they are centred about the origin (fig. 6.4). Their purpose is to serve as templates for later use. How the loops are distributed is down to the `dist` field, there are currently a zero-distribution (loops centered at the origin), random uniform, and random normal distributions. New ones can be quickly and easily added by subtyping the distribution type and making a function that dispatches on it.[4] The loop generation source code does not have to be modified for this to happen.

A nice feature of strongly typed languages is the intrinsic safety they offer via types. One of the ways in which this can be enforced is through the use of enumerated types for categorial variables. In 3D discrete dislocation dynamics we can have different types of nodes with different behaviours. We can safeguard our model against regressions (bugs introduced through further development) and reduce the memory requirements through the use of such types. To find out what they are and represent we can query the documentation for dislocation node type, `nodeTypeDln`.

```
julia>?nodeTypeDln
search: nodeTypeDln nodeTypeFE
```

[4]This can also be achieved with `eval()`, but it is unsafe because `eval()` can execute whatever code it is given.

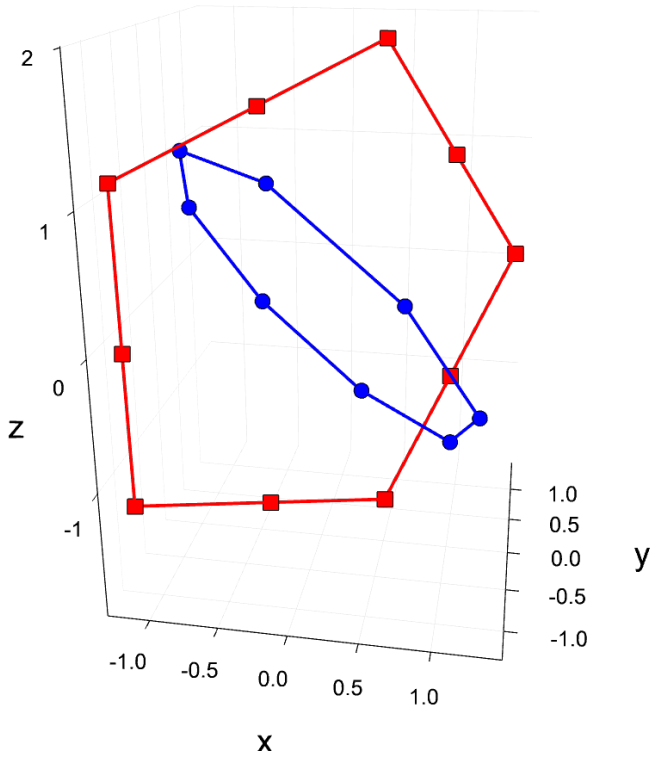


Figure 6.4: Template loops for network generation.

```
@enum nodeTypeDln begin
    noneDln = 0     # Undefined node, value at initialisation
    intMobDln = 1   # Internal mobile node
    intFixDln = 2   # Internal fixed node
    srfMobDln = 3   # Mobile surface node
    srfFixDln = 4   # Fixed surface node
    extDln = 5      # External node
    tmpDln = 6      # Temporary flag, used during topological
                    # operations
end
```

Node labels cannot have values other than these, unless the type is expanded, meaning the change has to be deliberate and thought out.

There are multiple ways of generating dislocation networks, but the preferred way is to use the loops already generated. Bespoke networks can be generated manually and developers can leverage multiple dispatch to do create new methods for the `DislocationNetwork` function. Since we have loops that we've already generated, we can create the network from them. The function automatically allocates $N \log_2 N$ places for N total nodes and segments in the network, which leaves room for expansion without having to dynamically allocate memory every

time a node is created. The topological operations also use this heuristic when the network needs to be expanded beyond its currently allocated memory.[5]

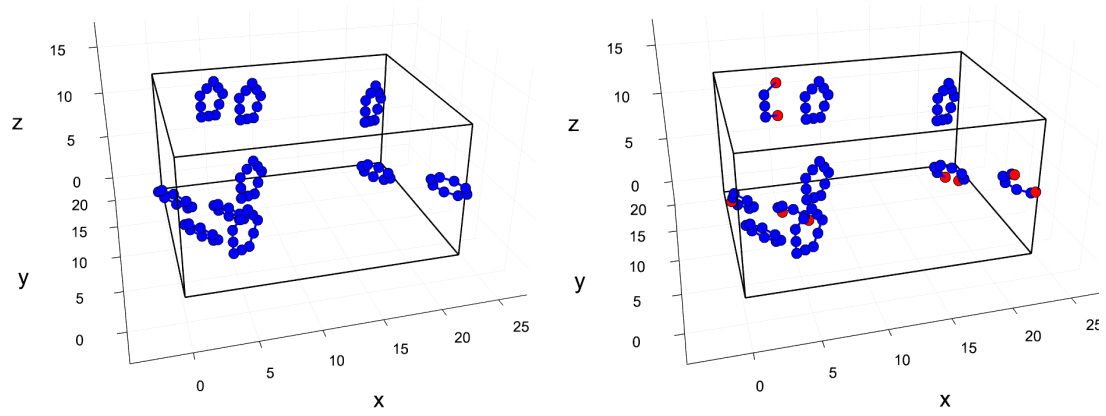
```
network = DislocationNetwork((prismOct, shearPent))
# Plot network pre-remeshing.
networkFEFig = plotNodes(
    meshC,
    network,
    m = 2,
    l = 3,
    linecolor = :blue,
    markershape = :circle,
    markercolor = :blue,
    legend = false,
    camera = camera = (-15, 25)
)
```

6.1.4 Current capabilities

Figure 6.5 shows the network pre- and post- surface and internal remeshing. The surface remeshing algorithm uses nodal velocities to find where the dislocation would have intersected the volume if it were moving with constant velocity. It also incorporates the correction described in section 3.5.

```
# Compute total force on segments =
# Peach-Koehler force (zero because there is no û)
# + Self force
# + Remote force
calcSegForce!(dlnParams, matParams, meshC, forceDispC, network)
# Compute nodal mobilities
dlnMobility!(dlnParams, matParams, network)
# Remesh the surface
network = remeshSurfaceNetwork!(meshC, boundaryC, network)
# Coarsen the internal network
network = coarsenNetwork!(dlnParams, matParams, meshC,
                forceDispC, network)
# Refine the internal network
network = refineNetwork!(dlnParams, matParams, meshC,
```

[5] It does not yet remove excess allocated memory, but will do in the future.



(a) Pre remeshed network. (b) Post remeshing network.

Figure 6.5: Network pre and post remeshing. The boundary conditions require the yz plane at $x = 0$ to be impenetrable to stop a slip step from forming in the surface with fixed displacement boundary conditions as explained in section 3.5.

```
                forceDispC, network)
# Plot remeshed network
remFEFig = plotNodes(
    meshC,
    network,
    m = 2,
    l = 3,
    linecolor = :blue,
    markershape = :circle,
    markercolor = :blue,
    legend = false,
    camera = (-15, 25)
)
# Find all surface nodes (both mobile and fixed)
idx = findall(x -> x   (3, 4), network.label)
# Plot surface nodes on the same figure as red circles.
scatter!(network.coord[1, idx], network.coord[2, idx],
    network.coord[3, idx], m = 2, markercolor = :red)
```

The code can also now compute dislocation induced displacements by Bromage and Tarleton [134], fig. 6.6 plots them scaled 20 times.

```
# Dislocation displacements.
calc_uTilde!(forceDispC, meshC, boundaryC, matParams, network)
# Copy structure to another variable.
deformedMesh = deepcopy(meshC)
```

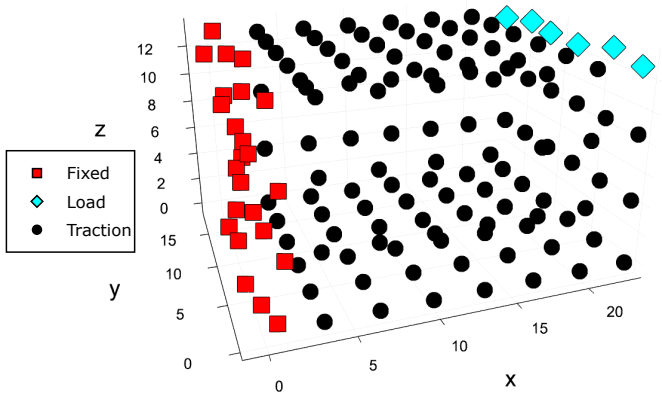


Figure 6.6: DDD.jl calculates dislocation induced displacements.

```
# Add 20 times the displacements to the dislocation
# displacement boundaries.
deformedMesh.coord[reshape(boundaryC.uDofsDln, :, 3)'] +=
    20 * forceDispC.uTilde[reshape(boundaryC.uDofsDln, :, 3)']
plotBoundaries(boundaryC, deformedMesh; camera = (-20, 20))
```

We're also able to calculate field point stresses (for numeric tractions), detect and resolve collisions, and calculate lazy numeric tractions for a single gauss quadrature point. However, resolving collisions and numeric tractions have not been tested yet.

6.1.5 Performance remarks

`Julia` was built as a language for scientific computing. Its primary goal is to be high performance, generic, and easy to use. It should come as no surprise that DDD.jl is substantially faster than the `MATLAB` portions of EasyDD. It is however, somewhat surprising that even C-accelerated portions of EasyDD perform better in DDD.jl and not by an insignificant amount.

1. The $\mathcal{O}(N^2)$ remote force computation is 10% faster than its equivalent `C` function. DDD.jl also has a CPU parallelisation. It was almost trivial to implement, which is definitely not the case for `C`. For large numbers of dislocation segments (> 500), it offers linear scaling with number of threads on an AMD Ryzen 7 1700 3.0 GHz.

2. The self-forces and Peach-Köhler forces as a result of the dislocations being inside a cuboid mesh are both $10\times$ faster than their `MATLAB` counterparts.

3. DDD.jl's mesh refinement uses a better heuristic than dynamically sizing arrays every time a new node is added to the network, so it is about as slow

as MATLAB's when it needs to allocate memory. When it does not need to allocate memory, it is $> 100\times$ faster. Mesh coarsening is $3\times$ faster.

4. The calculation of field point stresses for numeric tractions is 30% faster than its C counterpart.

5. Building the FE mesh is $> 10\times$ faster.

6. The outdated mobility law is $> 30\times$ faster and in all the tests has not required dampening the matrix inversion.

7. The dislocation displacement calculation is $> 10\times$ faster.

8. The cholesky factorisation of the stiffness matrix for free degrees of freedom is $20\times$ faster and uses a substantially smaller amount of memory.

Collision detection has not yet been compared to its EasyDD counterpart (which is written in C), and neither have the untested procedures. However, numeric tractions should be substantially faster as the field point stress calculation is faster in DDD.jl, and the DDD.jl version allocates no memory. If the trend holds, the rest of the functions should also see large increases in speed.

The memory footprint is also *much* smaller. 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 is ~ 3300 with many duplicates due to mutating and non-mutating forms of functions[6]. Some refactoring of common code should put it at ~ 1800 lines, which is about the same number of lines in the remote force calculation written in C.

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 until things break. The barrier to entry and potential for error are quite high. We use the code all the time, and have worked to greatly improve it. Yet we miss things regularly only to catch them after a simulation does something unexpected hours later. Even when everything is in order, seeing whether a simulation is well-callibrated can take hours. Often, multiple iterations of a simulation need to be run to find an appropriate callibration which reproduces an experiment—i.e.

[6]Mutating functions don't allocate memory so tend to be faster.

tuning the source size, number of sources, slip systems, loading rate, Poisson ratio, etc. Having to wait for hours or days before being able to see whether a simulation is properly callibrated is quite unproductive.

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, it seems worthwhile to continue this as a research software engineering project post-DPhil. We think it's 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] Moisés Hernández, Ginés D Guerrero, José M Cecilia, José M García, Alberto Inuggi, Saad Jbabdi, Timothy EJ Behrens, and Stamatios N Sotiropoulos. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus. *PloS one*, 8(4):e61892, 2013.

[2] 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.

[3] John Chapman. *Fragmentation in archaeology: people, places and broken objects in the prehistory of South-Eastern Europe*. Routledge, 2013.

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

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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 systems. *Review of Scientific Instruments*, 79(10):10F304, 2008.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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.

[42] 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.

[43] V Barabash, G Federici, J Linke, and CH Wu. Material/plasma surface interaction issues following neutron damage. *Journal of Nuclear Materials*, 313:42–51, 2003.

[44] Luke L Hsiung, Michael J Fluss, Scott J Tumey, B William Choi, Yves Serruys, Francois Willaime, and Akihiko Kimura. Formation mechanism and the role of nanoparticles in fe-cr ods steels developed for radiation tolerance. *Physical Review B*, 82(18):184103, 2010.

[45] Xunxiang Hu, Takaaki Koyanagi, Makoto Fukuda, NAP Kiran Kumar, Lance L Snead, Brian D Wirth, and Yutai Katoh. Irradiation hardening of pure tungsten exposed to neutron irradiation. *Journal of Nuclear Materials*, 480:235–243, 2016.

[46] 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.

[47] DEJ Armstrong, X Yi, EA Marquis, and SG Roberts. Hardening of self ion implanted tungsten and tungsten 5-wt% rhenium. *Journal of Nuclear Materials*, 432(1):428–436, 2013.

[48] MR Gilbert and J-Ch Sublet. Neutron-induced transmutation effects in w and w-alloys in a fusion environment. *Nuclear Fusion*, 51(4):043005, 2011.

[49] Robin A Forrest et al. *FISPACT-2007: User manual*. EURATOM/UKAEA fusion association, 2007.

[50] Robin A Forrest, Michael G Sowerby, Bryan H Patrick, and David AJ Endacott. The data library ukact1 and the inventory code fispact. In *Nuclear data for science and technology*, 1988.

[51] Makoto Fukuda, Kiyohiro Yabuuchi, Shuhei Nogami, Akira Hasegawa, and Teruya Tanaka. Microstructural development of tungsten and tungsten–rhenium alloys due to neutron irradiation in hfir. *Journal of Nuclear Materials*, 455(1):460–463, 2014.

[52] Akira Hasegawa, Takashi Tanno, Shuhei Nogami, and Manabu Satou. Property change mechanism in tungsten under neutron irradiation in various reactors. *Journal of Nuclear Materials*, 417(1):491–494, 2011.

[53] Akira Hasegawa, Makoto Fukuda, Kiyohiro Yabuuchi, and Shuhei Nogami. Neutron irradiation effects on the microstructural development of tungsten and tungsten alloys. *Journal of Nuclear Materials*, 471:175–183, 2016.

[54] M Klimenkov, U Jäntsch, M Rieth, HC Schneider, DEJ Armstrong, J Gibson, and SG Roberts. Effect of neutron irradiation on the microstructure of tungsten. *Nuclear Materials and Energy*, 2016.

[55] Xiaoou Yi, Michael L Jenkins, Marquis A Kirk, Zhongfu Zhou, and Steven G Roberts. In-situ tem studies of 150 kev w+ ion irradiated w and w-alloys: Damage production and microstructural evolution. *Acta Materialia*, 112: 105–120, 2016.

[56] Alan Xu, Christian Beck, David EJ Armstrong, Krishna Rajan, George DW Smith, Paul AJ Bagot, and Steve G Roberts. Ion-irradiation-induced clustering in w–re and w–re–os alloys: A comparative study using atom probe tomography and nanoindentation measurements. *Acta Materialia*, 87:121–127, 2015.

[57] Alan Xu, David EJ Armstrong, Christian Beck, Michael P Moody, George DW Smith, Paul AJ Bagot, and Steve G Roberts. Ion-irradiation induced clustering in w-re-ta, w-re and w-ta alloys: An atom probe tomography and nanoindentation study. *Acta Materialia*, 124:71–78, 2017.

[58] Robert N Noyce. Microelectronics. *Scientific American*, 237(3):62–69, 1977.

[59] Dermot Roddy. *Introduction to microelectronics*. Elsevier, 2013.

[60] Kihwan Choi, Wonbok Lee, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 29–34. IEEE Computer Society, 2004.

[61] Karel De Vogeleer, Gerard Memmi, Pierre Jouvelot, and Fabien Coelho. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *International Conference on Parallel Processing and Applied Mathematics*, pages 793–803. Springer, 2013.

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

[63] John Goodacre and Andrew N Sloss. Parallelism and the arm instruction set architecture. *Computer*, 38(7):42–50, 2005.

[64] Stephen M Trimberger. *Field-programmable gate array technology*. Springer Science & Business Media, 2012.

[65] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques 2000*, pages 71–82. Springer, 2000.

[66] John A Flanders, David C Ready, Steven Van Seters, Leonard Schwartz, and William D Townsend. Hardware filtering method and apparatus, March 21 2000. US Patent 6,041,058.

[67] 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.

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

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

[70] Thomas J Bensky and SE Frey. Computer sound card assisted measurements of the acoustic doppler effect for accelerated and unaccelerated sound sources. *American Journal of Physics*, 69(12):1231–1236, 2001.

[71] Daniel E Evanicky. Enhanced viewing experience of a display through localised dynamic control of background lighting level, April 9 2013. US Patent 8,416,149.

[72] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

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

[74] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[75] Ruymán Reyes, Iván López, Juan J Fumero, and Francisco de Sande. A preliminary evaluation of openacc implementations. *The Journal of Supercomputing*, 65(3):1063–1075, 2013.

[76] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

[77] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[78] YJ Lee and LB Freund. Fracture initiation due to asymmetric impact loading of an edge cracked plate. *Journal of Applied Mechanics*, 57(1):104–111, 1990.

[79] Stefan Sandfeld, Thomas Hochrainer, Peter Gumbsch, and Michael Zaiser. Numerical implementation of a 3d continuum theory of dislocation dynamics and application to micro-bending. *Philosophical Magazine*, 90(27-28):3697–3728, 2010.

[80] Vasily Bulatov, Wei Cai, Jeff Fier, Masato Hiratani, Gregg Hommes, Tim Pierce, Meijie Tang, Moono Rhee, Kim Yates, and Tom Arsenlis. Scalable line dynamics in paradis. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 19. IEEE Computer Society, 2004.

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

[82] Robert Gracie, Giulio Ventura, and Ted Belytschko. A new fast finite element method for dislocations based on interior discontinuities. *International Journal for Numerical Methods in Engineering*, 69(2):423–441, 2007.

[83] Franz Roters, Philip Eisenlohr, Luc Hantcherli, Denny Dharmawan Tjahjanto, Thomas R Bieler, and Dierk Raabe. Overview of constitutive laws, kinematics, homogenization and multiscale methods in crystal plasticity finite-element modeling: Theory, experiments, applications. *Acta Materialia*, 58(4):1152–1211, 2010.

[84] El Arzt. Size effects in materials due to microstructural and dimensional constraints: a comparative review. *Acta materialia*, 46(16):5611–5626, 1998.

[85] R Madec, B Devincre, L Kubin, T Hoc, and D Rodney. The role of collinear interaction in dislocation-induced hardening. *Science*, 301(5641):1879–1882, 2003.

[86] Francesco Ferroni, Edmund Tarleton, and Steven Fitzgerald. Gpu accelerated dislocation dynamics. *Journal of Computational Physics*, 272:619–628, 2014.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] O Jamond, R Gatti, A Roos, and B Devincre. Consistent formulation for the discrete-continuous model: Improving complex dislocation dynamics simulations. *International Journal of Plasticity*, 80:19–37, 2016.

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

[95] G Saada and XL Shi. Description of dislocation cores. *Czechoslovak Journal of Physics*, 45(11):979–989, 1995.

[96] A Vattré, B Devincre, F Feyel, R Gatti, S Groh, O Jamond, and A Roos. Modelling crystal plasticity by 3d dislocation dynamics and the finite element method: the discrete-continuous model revisited. *Journal of the Mechanics and Physics of Solids*, 63:491–505, 2014.

[97] Yang Xiang and DJ Srolovitz. Dislocation climb effects on particle bypass mechanisms. *Philosophical magazine*, 86(25-26):3937–3957, 2006.

[98] Yang Xiang, Li-Tien Cheng, David J Srolovitz, and E Weinan. A level set method for dislocation dynamics. *Acta Materialia*, 51(18):5499–5518, 2003.

[99] Siu Sin Quek, Yang Xiang, and David J Srolovitz. Loss of interface coherency around a misfitting spherical inclusion. *Acta Materialia*, 59(14):5398–5410, 2011.

[100] John P Hirth and Jens Lothe. *Theory of dislocations*. John Wiley\& Sons, 1982.

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

[102] 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.

[103] R De Wit. Some relations for straight dislocations. *physica status solidi (b)*, 20(2):567–573, 1967.

[104] R De Wit. The self-energy of dislocation configurations made up of straight segments. *physica status solidi (b)*, 20(2):575–580, 1967.

[105] Vladimir L Indenbom and Jens Lothe. *Elastic strain fields and dislocation mobility*. Elsevier, 2012.

[106] Bernard Fedelich. The glide force on a dislocation in finite elasticity. *Journal of the Mechanics and Physics of Solids*, 52(1):215–247, 2004.

[107] M Yu Gutkin and EC Aifantis. Screw dislocation in gradient elasticity. *Scripta Materialia*, 35(11):1353–1358, 1996.

[108] M Yu Gutkin and EC Aifantis. Edge dislocation in gradient elasticity. *Scripta Materialia*, 36(1):129–135, 1997.

[109] LM Brown. The self-stress of dislocations and the shape of extended nodes. *Philosophical Magazine*, 10(105):441–466, 1964.

[110] SD Gavazza and DM Barnett. The self-force on a planar dislocation loop in an anisotropic linear-elastic medium. *Journal of the Mechanics and Physics of Solids*, 24(4):171–185, 1976.

[111] R Peierls. The size of a dislocation. *Proceedings of the Physical Society*, 52 (1):34, 1940.

[112] FRN Nabarro. Dislocations in a simple cubic lattice. *Proceedings of the Physical Society*, 59(2):256, 1947.

[113] K Gururaj and C Robertson. Plastic deformation in ods ferritic alloys: A 3d dislocation dynamics investigation. *Energy Procedia*, 7:279–285, 2011.

[114] Zhenhuan Li, Chuantao Hou, Minsheng Huang, and Chaojun Ouyang. Strengthening mechanism in micro-polycrystals with penetrable grain boundaries by discrete dislocation dynamics simulation and hall–petch effect. *Computational Materials Science*, 46(4):1124–1134, 2009.

[115] Haidong Fan, Zhenhuan Li, Minsheng Huang, and Xiong Zhang. Thickness effects in polycrystalline thin films: Surface constraint versus interior constraint. *International Journal of Solids and Structures*, 48(11-12):1754–1766, 2011.

[116] 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.

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

[118] GC Hasson and C Goux. Interfacial energies of tilt boundaries in aluminium. experimental and theoretical determination. *Scripta metallurgica*, 5(10): 889–894, 1971.

[119] Haidong Fan, Sylvie Aubry, Athanasios Arsenlis, and Jaafar A El-Awady. The role of twinning deformation on the hardening response of polycrystalline magnesium from discrete dislocation dynamics simulations. *Acta Materialia*, 92:126–139, 2015.

[120] S Lay and G Nouet. Interaction of slip dislocations with the (01 1 2) twin interface in zinc. *Philosophical Magazine A*, 70(6):1027–1044, 1994.

[121] JI Dickson and C Robin. The incorporation of slip dislocations in {1102} twins in zirconium. *Materials Science and Engineering*, 11(5):299–302, 1973.

[122] DI Tomsett and M Bevis. The incorporation of basal slip dislocations in {10 1 2} twins in zinc crystals. *Philosophical Magazine*, 19(157):129–140, 1969.

[123] MH Yoo and CT Wei. Slip modes of hexagonal-close-packed metals. *Journal of Applied Physics*, 38(11):4317–4322, 1967.

[124] A Serra and DJ Bacon. A new model for {10 1 2} twin growth in hcp metals. *Philosophical Magazine A*, 73(2):333–343, 1996.

[125] Jindong Wang, Gen He, Jinwen Qin, Lidong Li, and Xuefeng Guo. Preparation of silicon nanowires by in situ doping and their electrical properties. *Colloids and Surfaces A: Physicochemical and Engineering Aspects*, 450: 156–160, 2014.

[126] Kisaragi Yashiro, F Kurose, Y Nakashima, K Kubo, Yoshihiro Tomita, and HM Zbib. Discrete dislocation dynamics simulation of cutting of γ precipitate and interfacial dislocation network in ni-based superalloys. *International Journal of Plasticity*, 22(4):713–723, 2006.

[127] SI Rao S, TA Parthasarathy, DM Dimiduk, and PM Hazzledine. Discrete dislocation simulations of precipitation hardening in superalloys. *Philosophical Magazine*, 84(30):3195–3215, 2004.

[128] Hesam Askari, Hussein M Zbib, and Xin Sun. Multiscale modeling of inclusions and precipitation hardening in metal matrix composites: Application to advanced high-strength steels. *Journal of Nanomechanics and Micromechanics*, 3(2):24–33, 2012.

[129] A Vattré, B Devincre, and A Roos. Orientation dependence of plastic deformation in nickel-based single crystal superalloys: Discrete–continuous model simulations. *Acta Materialia*, 58(6):1938–1951, 2010.

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

[131] 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.

[132] 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.

[133] 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.

[134] 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.

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

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

[137] 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.

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

[139] Coronavirus (covid-19): modelling the epidemic, 2021. URL https://www.gov.scot/collections/coronavirus-covid-19-modelling-the-epidemic/.

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

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

[142] 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.

[143] 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.

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

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

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

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

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

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

[150] 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.

[151] 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.

[152] 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.

[153] 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.

[154] 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.

[155] 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.

[156] 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.

[157] 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.

[158] 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.

[159] 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.

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

[161] 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.

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

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

[164] 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.

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

[166] 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.

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

[168] 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.

[169] 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.

[170] 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.

[171] 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.

[172] 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.

[173] 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.

[174] 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.

[175] Timothy J Tautges. Canonical numbering systems for finite-element codes. *International Journal for Numerical Methods in Biomedical Engineering*, 26 (12):1559–1572, 2010.

[176] Fortran Standards Committee. Contributing to OpenCoarrays, 2021. URL https://wg5-fortran.org/.

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

[178] 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 Programming Models*, 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 [176] 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 [176]. 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;

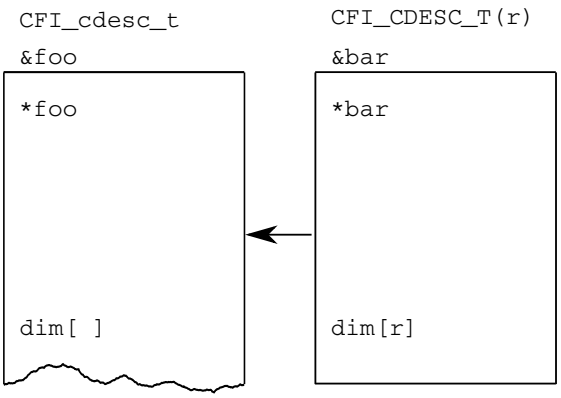


Figure A.1: C descriptor with rank `r` being cast into one with unspecified rank. C looses track of exactly how large the object is, but given the value of the rank element `r`, the size of a single element of `dim` and the information contained within the `dim` member variables, the object can be fully 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 [176] 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.

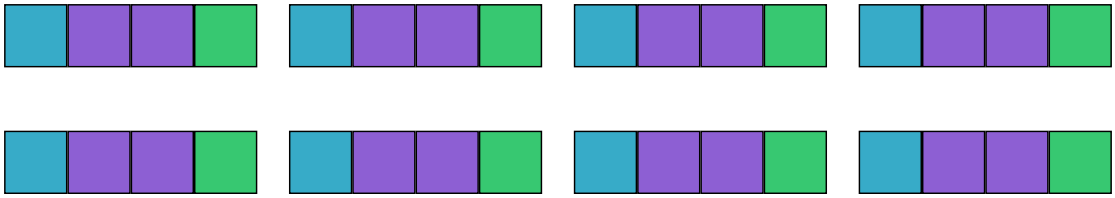
These functions were originally defined by the standard as if they would be used on independent C descriptors. However, when using `CFI_section` and `CFI_select_part` on the same descriptor, the element length and dimensional information can end up being updated in nontrivial ways (see fig. A.2). These 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.

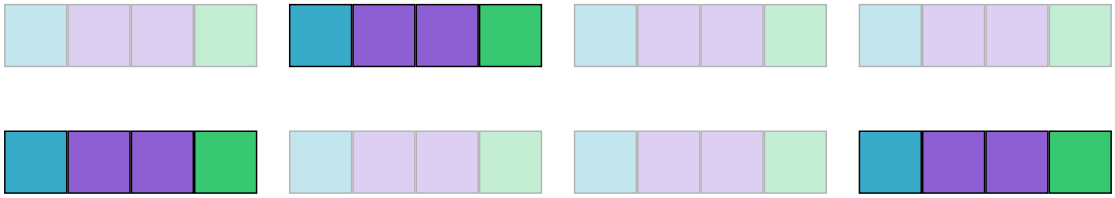
A.4 Results and conclusions

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 [177] and is merely a case of "unrolling" the structure with the variable length array into explicit structures for specific ranks.

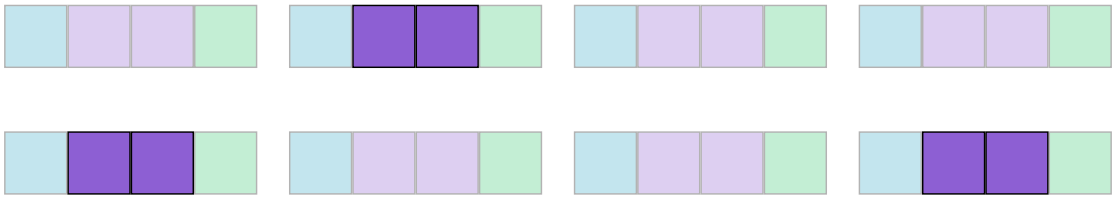
The code can be found in the Github repository /sourceryinstitute/ISO_Fortran_binding. The descriptors are now also being implemented by the `GCC gfortran` developers. They plan on porting the non-standard descriptors and associated programs to use the standard descriptors and



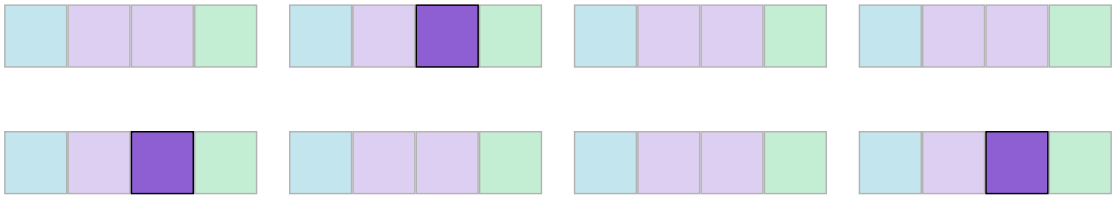
(a) 1D array of length 8 of a derived type containing a scalar, 2 element array, and scalar. The array is contiguous.



(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 [178] 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.