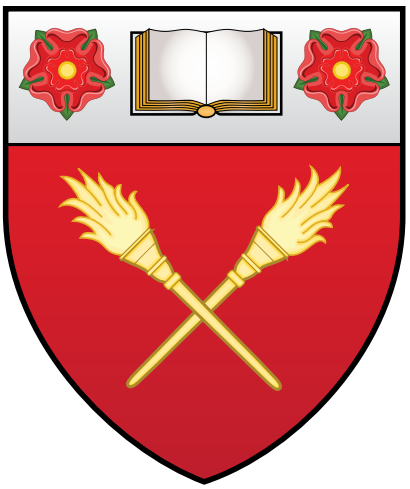
# Dislocation Based Modelling of Fusion Relevant Materials.

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*

Trinity Term 2020

# Dedication

"Ohana significa familia, y tu familia nunca te abandona, ni te olvida."
"Ohana means family, family means nobody gets left behind, or forgotten."
– 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 repagar en mil vidas, pero ten por seguro que eres nuestro ejemplo a seguir. Nuestra madre chingona y chambeadora, solo hay una y como ella no hay ninguna. Sin tí 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 avisé "Sí dad, igualito que mi jefe." entre risas respondí. Como siempre, no platicamos mucho pero esta vez porque le querías avisar 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 a tí en la tuya. Escribo esta dedicatoria antes de acabar porque una promesa es una promesa y esta madre la voy a acabar. No te tendré para pedirte consejos y contarte mis avances y tropiezos. Pero a veces pretendo que me escuchas mientras intento comprender o arreglar algo. Te queremos y te 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 un chingo de falta ojete, te extraño mucho we. Perdón por no hablar tan seguido, pero me duele colgar we. No estoy tan chisqueado como mamá pero siento feo cuando terminamos de platicar. Me gusta mucho ver tus streams porque me recuerda un poco a sentarme a tu lado a verte jugar… o a cuando veíamos Twitch juntos y veíamos a TB missing legal.

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 reir cuando solo sabía llorar. Gracias por hacerme quien soy.

This is for my Ohana by blood and by choice. You believed in me when I did not. You pushed me to be better. Offered me a helping hand when nobody would. Made me laugh when all I knew was sorrow. Thank you for making me who I am.

# Acknowledgements

"You can always judge a man by the quality of his enemies."
– Oscar Wilde

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 awesome 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.

- Ed and Angus
- Fusion CDT
- Harris-Manchester
- CONACYT
- Anatoly Kolomeisky, John F. Stanton
- Isla
- Clan del Rano Sentado
- Goosehood
- Kopse Lane Krump v1.0 v2.0
- Kristmas Karnage: n
- OUPLC: Perhaps the heaviest things that we lift are not our weights, but our feels.
- Damian Rouson
- Bruce Bromage, Haiyang Yu, Fenxian Liu, Daniel Hortelano-Roig, Junnan Jiang

I guess I'll content myself with being judged unfavourably by the world.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Outline

1. Lit review
2. DDD-FEM
3. Analytic tractions
    (a) Traction errors
    (b) Reaction force errors
    (c) Simulations
    (d) Recommendations and Conclusions
4. Parallelising tractions
    (a) Algorithms
    (b) Performance
    (c) Simulations
    (d) Recommendations and Conclusions
5. EasyDD v2.0
    (a) Adaptive integration
    (b) New mobility law
    (c) Surface velocities
    (d) Collisions
    (e) Code redesign
6. Future work
    (a) Julia redesign
    (b) Preliminary comparisons
7. Conclusions

# Chapter 0

# Preface

## 0.1 Notation

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

- Tensors are denoted by sans serif bold italics, $\boldsymbol{T}$.
- Matrices are denoted by serif bold roman, $\mathbf{M}$.
- Vectors are denoted by serif bold italics, $\boldsymbol{V}$.
- Scalars are denoted by serif italics, $S$.
- Host variables are denoted by ${}^{\mathrm{h}}a$.
- Global device variables are denoted by ${}^{\mathrm{d}}a$.
- Thread variables are denoted by ${}^{\mathrm{t}}a$.

## 0.2 Conventions

All angles are given in radians unless stated otherwise. All pseudo-code is C-style and assuming C conventions (row-order, 0-start indexing, memory allocation).

Elements, be they Surface Elements (SEs), Finite Elements (FEs), dislocation line segments or else, are denoted by $e$ and their total as $E$. Similarly nodes are denoted by $n$ and their total $N$. Parallel indices are denoted by idx to distinguish them from regular indices/counters $i$, $j$, $k$. For disambiguation purposes the exponential function will de denoted as $\exp(x)$.

## 0.3 Typesetting

This document was typeset using a custom-made LaTeX document class created by the author, compiled with XeLaTeX, and the bibliography was produced with BibTeX. The custom document class can be found in https://github.com/dcelisgarza/latex_temp

## 0.4 Diagrams

All diagrams are vector graphics drawn using the open-source image editing software InkScape for its ease of use, elegant simplicity, high quality outputs, and "Draw Freely" philosophy.

# Chapter 1

# Introduction

## 1.1 Section 1

### 1.1.1 Subsection 1

# Chapter 2

# Coupling Discrete Dislocation Dynamics to Finite Element Methods

## 2.1 Superposition Scheme

Coupling Discrete Dislocation Dynamics (DDD) to Finite Element Methods (FEMs) [1] is important to properly simulate micromechanical tests because DDD provides us with a more precise set of inputs and greater granularity for solving the FE problem. This can be achieved by using a so-called superposition scheme fig. 2.1 that enables the independent solution of both problems, whilst feeding information from one to the other in a continuous feedback loop.

## 2.2 Extracting Surface Nodes

The FEM coupler arranges the nodes starting on the $xz$-plane where $y = 0, \ldots, n\mathrm{d}y,\ n \in \mathbb{N}$. However in order to couple DDD to FEM we only require the surface nodes where displacements are not calculated. Because we're working with rectangular prisms, we can easily pick out the surface nodes using a search algorithm with a logical mask. MATLAB and Fortran provide vector intrinsics that allow one to do so. Figure 2.2a illustrates only the surface nodes according to our implementation's node arrangement—which is the $xz$-plane going from $y = y_{\min} \rightarrow y = y_{\max}$. However, due to the nature of the analytical solutions in chapter 3, we need all the surface nodes of the rectangular faces for which there are no displacements. This means that edge nodes are shared between 2 adjacent faces and corner nodes between 3 adjacent faces. In order to properly apply the logical mask to find *only* the surface nodes need to know that each FEs' nodes are numbered according to

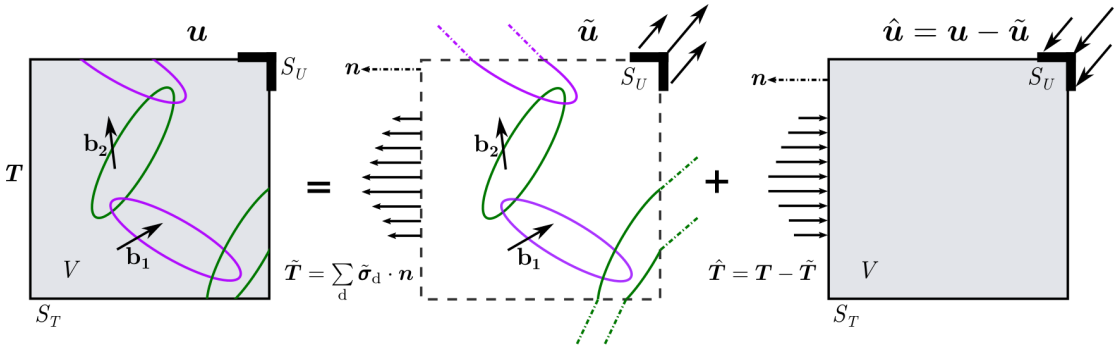
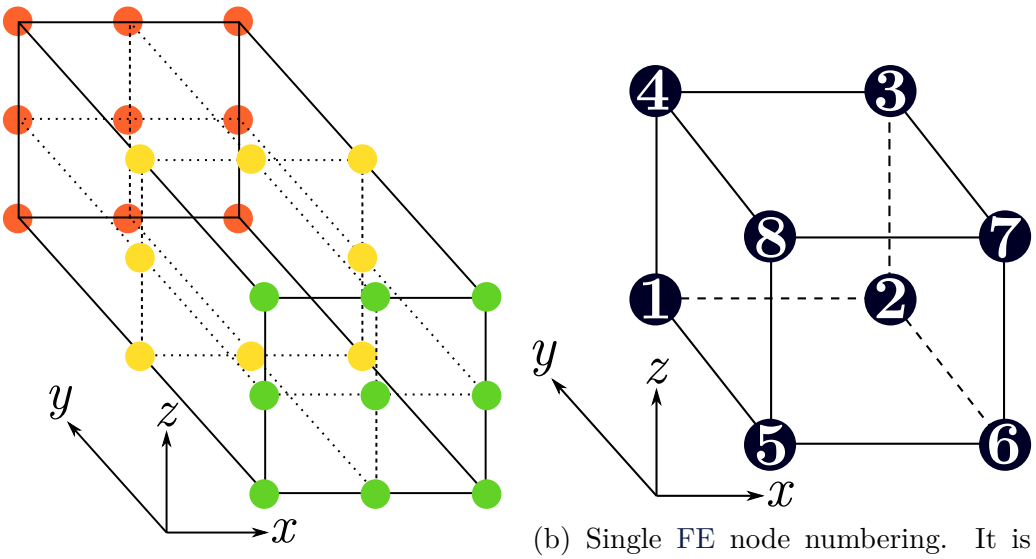



Figure 2.1: The dislocation ensemble in a volume $V$ is bounded by surface $S$. First, the traction field $\sum_{\mathrm{d}} \tilde{\boldsymbol{\sigma}}_{\mathrm{d}}$ due to the dislocation ensemble is evaluated at the surface. Then, a traditional FEM or Boundary Element Method (BEM) calculates the image traction field $\hat{\boldsymbol{\sigma}} \times \boldsymbol{n}$. Which is then fed back to the DDD problem to evolve the dislocation positions and repeat the cycle. Image edited from [1].




(a) Arrangement of the surface nodes of our FE model. FE nodes are arranged in chunks of $\Delta x \Delta z$ nodes in our implementation, but we only want the surface nodes.

(b) Single FE node numbering. It is necessary to know which FE plane corresponds to which node labels. This lets us design the auxiliary matrix that selects nodes according to the planes we want to extract.

Figure 2.2: FE node arrangement for coupling to DDD.

fig. 2.2b.

Using fig. 2.2 one can work out which nodes are of interest to whichever surface is being extracted. The ordering of the nodes in the final array will depend on the definition of the problems in chapters 3 and 4.

Node selection remains an expensive operation and minimising array indexing is of the utmost importance for the best perfomance. Selecting nodes in the traditional sense, i.e. with code branching such as `if statements` or `case selection` is unmantainable, verbose and very prone to mistakes. The issue was solved by introducing an auxiliary matrix which defines various parameters that aid node selection and greatly reduces code size, improves readability, and eliminates the need for code branching. The matrix can be constructed utilising fig. 2.2 in order to know which nodes correspond to which FE planes. The $p^{\text{th}}$ column of the matrix[1] corresponds to the $p^{\text{th}}$ plane (according to an arbitrary plane numbering) and is defined as,

$$\boldsymbol{V_p}^{\mathsf{T}} = \begin{bmatrix} L_{1p} & L_{2p} & \cdots & L_{Np} & A_p & C_p \end{bmatrix}, \tag{2.1}$$

where $L_{np}$ is the numeric label for node $n$ as given by fig. 2.2, $A_p$ is the area of the plane, and $C_p$ is the numeric label of the orthogonal coordinate to the plane $C_p = 1,\ 2,\ 3$ for the $x,\ y,\ z$ coordinates respectively. $A_p$ lets us segment our output and transitional arrays so that the only data being modified is that which corresponds to the correct plane and $C_p$ lets us know which coordinate we must use in our selection criteria. Using our particular node labelling scheme (with dimensions $\Delta x,\ \Delta y,\ \Delta z$ respectively in the $x,\ y,\ z$ directions), the matrix is defined as,

$$\mathbf{V} = \begin{bmatrix} 5 & 2 & 6 & 1 & 5 & 4 \\ 1 & 6 & 5 & 2 & 6 & 3 \\ 8 & 3 & 7 & 4 & 1 & 8 \\ 4 & 7 & 8 & 3 & 2 & 7 \\ \Delta y\Delta z & \Delta y\Delta z & \Delta x\Delta z & \Delta x\Delta z & \Delta x\Delta y & \Delta x\Delta y \\ 1 & 1 & 2 & 2 & 3 & 3 \end{bmatrix}. \tag{2.2}$$

The information codified in eq. (2.2) lets us index and process only the necessary columns to extract the surface nodes we're interested in. The advantage of this setup over a naïve implementation is that it can be relatively easily expanded, maintained, and is general enough that it lends itself to a variety of selection criteria. The columns from left to right (1 to 6) represent: face 1 $\equiv \min(x)$, $yz$-

[1] MATLAB uses column-major ordering, so this gives us the best performance for vectorised code.

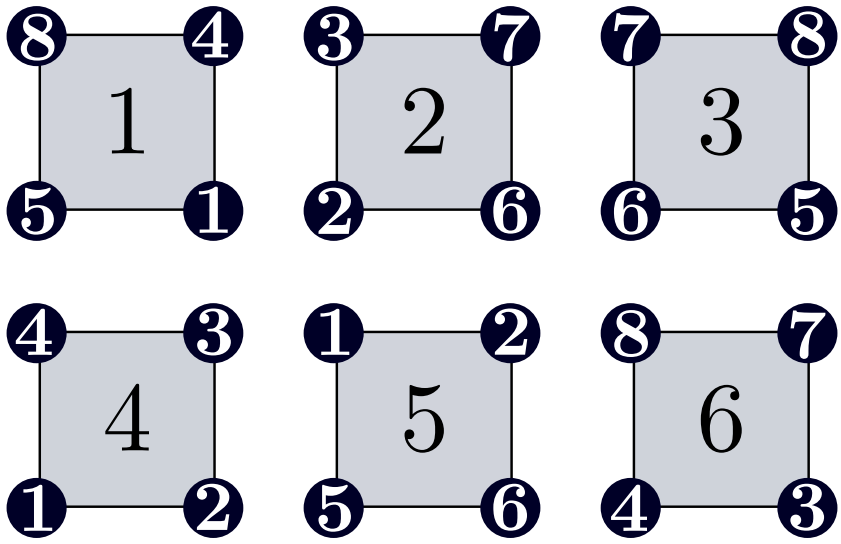



Figure 2.3: Self-consistent, chirality preserving surface planes. Plane and node numbering according to the columns of eq. (2.2) and fig. 2.2.

plane; face $2 \equiv \max(x)$, $yz$-plane; face $3 \equiv \min(y)$, $xz$-plane; face $4 \equiv \max(y)$, $xz$-plane; face $5 \equiv \min(z)$, $xy$-plane; face $6 \equiv \max(z)$, $xy$-plane.

It is important to make sure the nodes are extracted in a self consistent manner because the problem definitions in chapters 3 and 4 assume a specific node ordering. In particular, the labelling chirality must remain the same. This is due to the fact that the problems in chapters 3 and 4 utilise internal coordinates derived from a set of basis vectors, and scalar projections onto them. If the relationship between the vectors is kept, but the chirality is different (opposite) the calculated quantities will have the wrong sign. If however the relationship between the vectors is different to the problem formulation, the program will crash due to division by 0. Self-consistency was achieved by placing an imaginary observer inside the FE model facing the $\min(x)$, $yz$-plane (face 1) and rotating it to view all 6 planes according to fig. 2.3.

## 2.3 Mapping Forces

$\hat{\boldsymbol{F}}$ only cares about the global node numbers, of which there are $M$. The columns of $\mathbf{N_L}$, $\mathbf{F_N}$ represent the 4 nodes of a SE, the rows represent the SE they are part of. The column vector $\boldsymbol{x}$ represents the $x$, $y$, $z$-coordinates of the node-element or global node corresponding to their subscript. The column vector $\boldsymbol{\gamma_t}$ is the list

**Algorithm 2.1** If $\hat{\boldsymbol{F}}$'s columns are arranged the same way as $\boldsymbol{\gamma_t}$.

▷ Loop through the array containing the node labels of the relevant surface nodes.
**for** $i = 0;\ i < \text{length}(\boldsymbol{\gamma_t});\ i++$ **do**
▷ Save the global node label for the current iteration.
$n \leftarrow \boldsymbol{\gamma_t}[i]$
▷ Use the node label to find a vector with the linearised index of all surface nodes whose labels correspond to the node whose forces we want to pass on to the FEM coupler.
$\boldsymbol{L} \leftarrow \text{find}(\mathbf{N_L} == n)$
▷ 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 due to dislocations on surface nodes. 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 $\mathbf{F_n}$.
$\hat{\mathbf{F}}[3 \times \boldsymbol{L} + k] \leftarrow \hat{\mathbf{F}}[3 \times \boldsymbol{L} + k] + \sum \mathbf{F_n}[3 \times \boldsymbol{L} + k]$
**end for**
**end for**

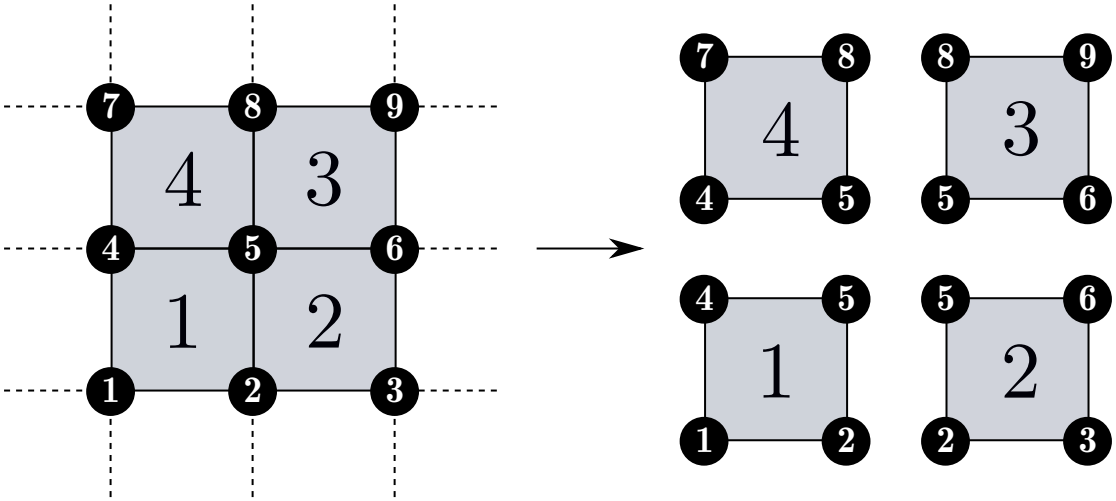



Figure 2.4: FE nodes shared by multiple SEs.

of global nodes for which the forces induced by dislocations must be calculated.

$$\boldsymbol{x_{e,n}}^{\mathsf{T}} \equiv \begin{bmatrix} x_{en} & y_{en} & z_{en} \end{bmatrix}, \quad \hat{\boldsymbol{x}}_{\boldsymbol{n}}^{\mathsf{T}} \equiv \begin{bmatrix} x_n & y_n & z_n \end{bmatrix} \tag{2.3}$$

$$\mathbf{N_L} = \begin{bmatrix} l_{1,1} & l_{1,2} & l_{1,4} & l_{1,5} \\ l_{2,2} & l_{2,3} & l_{2,5} & l_{2,6} \\ l_{3,5} & l_{3,6} & l_{3,8} & l_{3,9} \\ l_{4,4} & l_{4,5} & l_{4,7} & l_{4,8} \end{bmatrix}, \quad \boldsymbol{\gamma} = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_9 \end{bmatrix} \tag{2.4}$$

$$\mathbf{F_e} = \begin{bmatrix} \boldsymbol{x_{1,1}} & \boldsymbol{x_{1,2}} & \boldsymbol{x_{1,3}} & \boldsymbol{x_{1,4}} \\ \boldsymbol{x_{2,1}} & \boldsymbol{x_{2,2}} & \boldsymbol{x_{2,3}} & \boldsymbol{x_{2,4}} \\ \boldsymbol{x_{3,1}} & \boldsymbol{x_{3,2}} & \boldsymbol{x_{3,3}} & \boldsymbol{x_{3,4}} \\ \boldsymbol{x_{4,1}} & \boldsymbol{x_{4,2}} & \boldsymbol{x_{4,3}} & \boldsymbol{x_{4,4}} \end{bmatrix}, \quad \hat{\boldsymbol{F}} = \begin{bmatrix} \hat{\boldsymbol{x}}_{\boldsymbol{1}} \\ \hat{\boldsymbol{x}}_{\boldsymbol{2}} \\ \hat{\boldsymbol{x}}_{\boldsymbol{3}} \\ \hat{\boldsymbol{x}}_{\boldsymbol{4}} \end{bmatrix}. \tag{2.5}$$

# Chapter 3

# Analytical Forces Induced by Dislocations on Linear Rectangular Surface Elements

## 3.1 Forces Exerted by a Dislocation Line Segment on Linear Rectangular Surface Elements

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 Boundary Element (BE) model. This is usually achieved as follows,

$$\boldsymbol{F}^{(m)} = \int_{S^e} \left[\boldsymbol{\sigma}^{\infty}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\right] N_m(\boldsymbol{x}) \, \mathrm{d}S_e \,, \tag{3.1}$$

where $\mathrm{d}S_e$ is the infinitesimal surface element with surface area $S_e$. $N_m(\boldsymbol{x})$ 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. [2]. The stress field of a dislocation in a homogenous infinite linear elastic domain can be calculated as a contour integral along the loop [3],

$$\sigma_{ij}^{\infty}(\boldsymbol{x}) = C_{ijkl} \oint \epsilon_{lnh} C_{pqmn} \frac{\partial G_{kp}(\boldsymbol{x} - \boldsymbol{x}')}{\partial x_q} b_m \mathrm{d}x'_h \,, \tag{3.2}$$

where $C_{ijkl}$ is the elastic stiffness matrix, $\epsilon_{lnh}$ the permutation operator, $\boldsymbol{b}$ the Burgers vector, $\boldsymbol{x}'$ the coordinate that spans the dislocation, and $G_{kp}(\boldsymbol{x} - \boldsymbol{x}')$ is Green's function of elasticity [3]. $G_{kp}(\boldsymbol{x} - \boldsymbol{x}')$ is defined as the displacement component in the $x_k$ direction at point $\boldsymbol{x}$ due to a force applied in the $x_p$ direction

at point $\boldsymbol{x}'$. The traditional singularity comes from taking the Burgers vector distribution as a delta function. Cai et al. [2] proposed an alternative definition of $G_{kp}(\boldsymbol{x}-\boldsymbol{x}')$ which has a wider isotropic spread mainly localised in a radius $a$ around the dislocation core,

$$G_{ij}(\boldsymbol{x}-\boldsymbol{x}') = \frac{1}{8\pi\mu}\left[\delta_{ij}\partial_{pp} - \frac{1}{2(1-\nu)}\partial_{ij}\right] R_a\,, \tag{3.3}$$

where $\mu$, $\nu$ are the isotropic shear modulus and Poisson's ratio respectively, $\delta_{ij}$ is the Kronecker Delta, $\partial_{x_1\dots x_n} \equiv \dfrac{\partial^n}{\partial x_1 \dots \partial x_n}$. $R_a$ is the defined as,

$$\begin{aligned} R_a(\boldsymbol{x}) &= R(\boldsymbol{x}) * w(\boldsymbol{x}) = \int R(\boldsymbol{x}-\boldsymbol{x}')w(\boldsymbol{x}')\mathrm{d}^3\boldsymbol{x}' \\ &= \sqrt{R^2+a^2} \qquad (3.4a) \\ w(\boldsymbol{x}) &= \frac{15a^4}{8\pi(R^2+a^2)^{7/2}}\,, \qquad (3.4b) \end{aligned}$$

where $w(\boldsymbol{x})$ is the isotropic Burgers vector distribution derived in the appendix of [2], $\boldsymbol{x} = (x, y, z)$ and $R(\boldsymbol{x}) = \sqrt{x^2+y^2+z^2}$.

For two dislocation nodes (1, 2) connected by straight line segments eq. (3.2) becomes,

$$\begin{aligned} \boldsymbol{\sigma}^{(12)}(\boldsymbol{x}) = &- \frac{\mu}{8\pi}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}} \left(\frac{2}{R_a^3} + \frac{3a^2}{R_a^5}\right)\left[(\boldsymbol{R}\times\boldsymbol{b})\otimes\mathrm{d}\boldsymbol{x}' + \mathrm{d}\boldsymbol{x}'\otimes(\boldsymbol{R}\times\boldsymbol{b})\right] \\ &+ \frac{\mu}{4\pi(1-\nu)}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}} \left(\frac{1}{R_a^3} + \frac{3a^2}{R_a^5}\right)\left[(\boldsymbol{R}\times\boldsymbol{b})\cdot\mathrm{d}\boldsymbol{x}'\right]\mathbf{I}_2 \\ &- \frac{\mu}{4\pi(1-\nu)}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}} \frac{1}{R_a^3}\left[(\boldsymbol{b}\times\mathrm{d}\boldsymbol{x}')\otimes\boldsymbol{R} + \boldsymbol{R}\otimes(\boldsymbol{b}\times\mathrm{d}\boldsymbol{x}')\right] \\ &+ \frac{\mu}{4\pi(1-\nu)}\int_{\boldsymbol{x_1}}^{\boldsymbol{x_2}} \frac{3}{R_a^5}\left[(\boldsymbol{R}\times\boldsymbol{b})\cdot\mathrm{d}\boldsymbol{x}'\right]\boldsymbol{R}\otimes\boldsymbol{R} \end{aligned}\,. \tag{3.5}$$

### 3.1.1 Resolving Singularities when Dislocation Line Segments are Parallel to Surface Elements

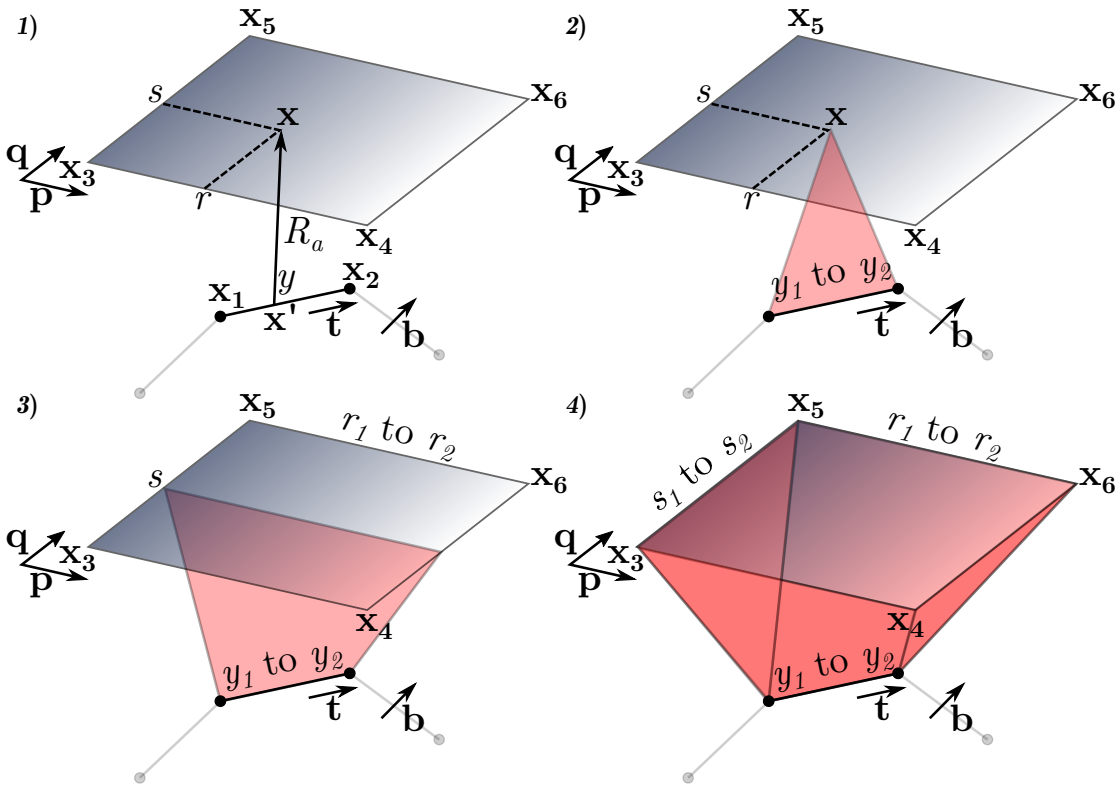



Figure 3.1: Diagram of the line integral method used to find analytical expressions for the forces exerted by dislocations on linear rectangular SEs [1]. *1)* For any given point $\boldsymbol{x}$ on the SE and any given point $\boldsymbol{x}'$ on the dislocation line segment, define distance $R_a$. *2)* Integrate from $x_1 \to x_2$ along line direction $\boldsymbol{t}$. *3)* Integrate from $r_1 \to r_2$ along vector $\boldsymbol{p}$. *4)* Integrate from $s_1 \to s_2$ along vector $\boldsymbol{q}$.

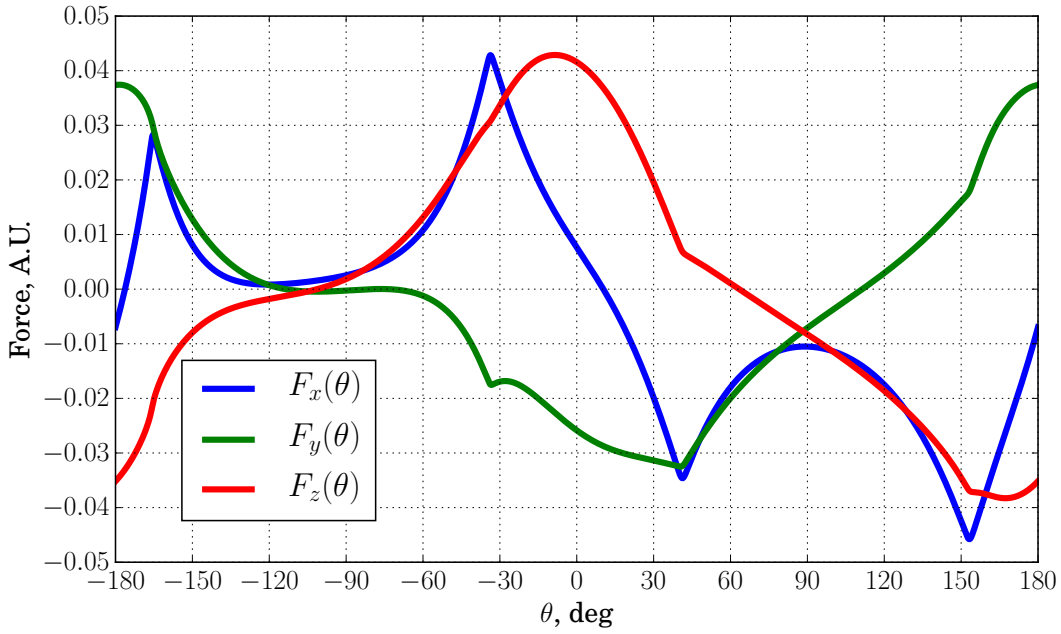



Figure 3.2: Effects of rotating a single dislocation line segment on the forces exerted by it on a linear rectangular SE. The specific values of this function are not known *a priori*, all that is known is that it must be periodic ($T = 2\pi$) and have finite maximum and minimum values. The singularity is avoided by perturbing the angle $\theta = 0 \rightarrow \theta = \pm\epsilon\,, \epsilon \gtrsim 0$.

# Chapter 4

# Analytical Forces Induced by Dislocations on Quadratic Triangular Surface Elements

## 4.1 Forces Exerted by a Dislocation Line Segment on Quadratic Triangular Surface Elements

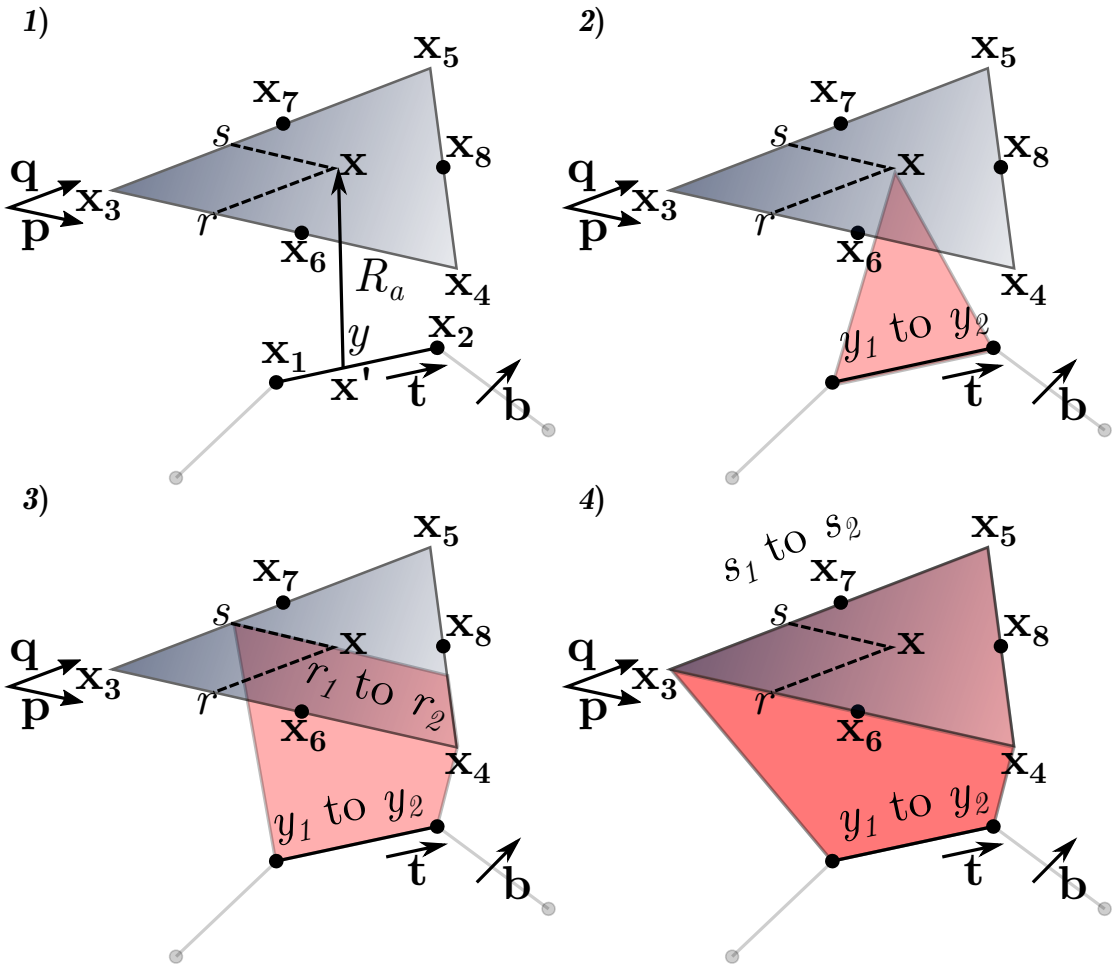



Figure 4.1: Diagram of the line integral method used to find analytical expressions for the forces exerted by dislocations on quadratic triangular SEs.

# Chapter 5

# Parallelisation of the Analytical Forces Induced by Dislocations on Surface Elements

## 5.1 Parallelisation on Graphics Processing Units

Efficient parallelisation requires coalesced memory access, which means we have to be extremely careful when mapping CPU memory to device memory. The fact that threads work "simultaneously"[1] means that in order to obtain good performance, data which is to be "simultaneously" loaded into each thread must be contiguous. This maximises cache memory use and therefore reduces slow memory fetch operations to global or shared memory.

The parallelisation was done only over the SEs in order to avoid the undesirable and inefficient GPU branching that would occur under other schemes.

The most natural form of parallelisation is to have blocks of $4n$ threads where $n \leq 8 \in \mathbb{N}$. This also fits nicely into the 32 thread per warp paradigm.

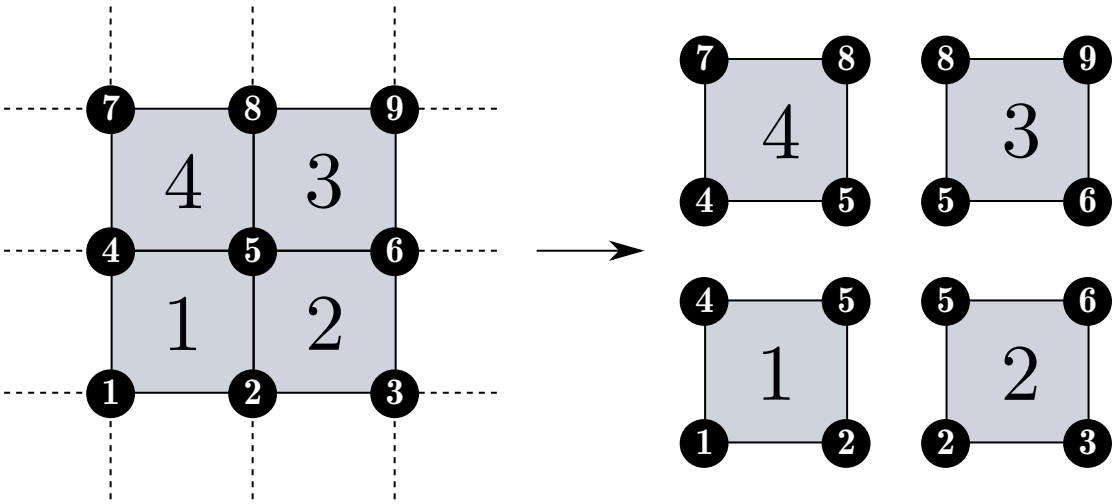



Figure 5.1: Each linear rectangular SE is mapped to one thread.

**Algorithm 5.1** Elements in host $\mapsto$ device.

**function** ELEMENT_HOST_DEVICE_MAP($^{\mathrm{h}}\boldsymbol{X}[N][3 \times E]$)
    ▷ Input, temporary, and output indices set to zero.
    $i, j, k \leftarrow 0$
        ▷ $^{\mathrm{d}}\boldsymbol{X}$ accomodates all 3 coordinates of all $N$ nodes in all $E$ elements.
    $^{\mathrm{d}}\boldsymbol{X} \leftarrow \mathrm{malloc}(3 \times E \times N)$
    **for** $(n = 0; n < N; n++)$ **do** ▷ Loop over nodes in element.
        ▷ Set output index to point at the $n^{\mathrm{th}}$ node of the first coordinate of the first element.
        $k \leftarrow j$
        **for** $(c = 0; c < 3; c++)$ **do** ▷ Loop over coordinates.
            ▷ Set input index to point at the $c^{\mathrm{th}}$ coordinate of the first element of the $n^{\mathrm{th}}$ node.
            $i \leftarrow c$
            **for** $(e = 0; e < E; e++)$ **do** ▷ Loop over elements.
                $^{\mathrm{d}}\boldsymbol{X}[k + e] \leftarrow^{\mathrm{h}} \boldsymbol{X}[n][i]$
                    ▷ Advance output index to point at the $n^{\mathrm{th}}$ node of the $c^{\mathrm{th}}$ coordinate of the first element.
                $i \leftarrow i + 3$
            **end for**
                ▷ Advance output index to point at the $n^{\mathrm{th}}$ node of the $c^{\mathrm{th}}$ coordinate of the first element.
            $k \leftarrow k + E \times N$
        **end for**
        ▷ Advance temporary index to point at the $(n + 1)^{\mathrm{th}}$ node of the first coordinate of the first element.
        $j \leftarrow j + E$
    **end for**
    **return** $^{\mathrm{d}}\boldsymbol{X}$
**end function**

**Algorithm 5.2** Elements in device $\mapsto$ thread.

**GPU function** ELEMENT_DEVICE_THREAD_MAP(${}^{\mathrm{d}}\boldsymbol{X}[3 \times E \times N]$, ${}^{\mathrm{t}}\boldsymbol{X}[N][3]$)
    ▷ Thread, input indices.
    idx, $i \leftarrow \text{threadIdx}.x + \text{blockIdx}.x \times \text{blockDim}.x$
    **for** $(c = 0;\ c < 3;\ c++)$ **do** ▷ Loop over elements.
        **for** $(n = 0;\ n < N;\ n++)$ **do** ▷ Loop over nodes in element.
            ${}^{\mathrm{t}}\boldsymbol{X}[n][c] \leftarrow {}^{\mathrm{d}}\boldsymbol{X}[i + n \times E]$
        **end for**
        $i \leftarrow i + E \times N$ ▷ Advance input index to the $(n+1)^{\text{th}}$ node of element $i$.
    **end for**
    **return** ${}^{\mathrm{t}}\boldsymbol{X}[n][c]$
**end GPU function**

**Algorithm 5.3** Force in thread $\overset{+}{\mapsto}$ device.

**GPU function** ADD_FORCE_THREAD_DEVICE(${}^{\mathrm{t}}\boldsymbol{F_n}[N][3]$, ${}^{\mathrm{t}}\boldsymbol{F_e}[3]$, ${}^{\mathrm{d}}\boldsymbol{F_n}[3 \times E \times N]$,
${}^{\mathrm{d}}\boldsymbol{F_e}[3 \times E]$, idx)
    ▷ Nodal force.
    ▷ Set output index to whatever parallelisation index is given to the function. In this case the same index we use for the main parallelisation.
    $i \leftarrow \text{idx}$
    **for** $(n = 0;\ n < N;\ n++)$ **do** ▷ Loop over nodes.
        **for** $(c = 0;\ c < 3;\ c++)$ **do** ▷ Loop over coordinates.
            ▷ Ensure nodal forces are correctly added and mapped from local thread memory to global device memory.
            atomicAdd(${}^{\mathrm{d}}\boldsymbol{F_n}[i + c \times E \times N]$, ${}^{\mathrm{t}}\boldsymbol{F_n}[n][c]$)
        **end for**
        ▷ Displace output index to point at the first coordinate of the $(n+1)^{\text{th}}$ node of the $\text{idx}^{\text{th}}$ SE.
        $i \leftarrow i + E$
    **end for**
    ▷ Total force.
    $i \leftarrow \text{idx}$ ▷ Reset output index.
    **for** $(c = 0;\ c < 3;\ c++)$ **do** ▷ Loop over coordinates.
        atomicAdd(${}^{\mathrm{d}}\boldsymbol{F_e}[i]$, ${}^{\mathrm{t}}\boldsymbol{F_e}[c]$)
        ▷ Advance the output index to point at the $(i+1)^{\text{th}}$ coordinate of the $\text{idx}^{\text{th}}$ SE.
        $i \leftarrow i + E$
    **end for**
    **return** ${}^{\mathrm{d}}\boldsymbol{F_n}$, ${}^{\mathrm{d}}\boldsymbol{F_e}$
**end GPU function**

**Algorithm 5.4** Force (parallelise over dislocations) in thread $\overset{+}{\mapsto}$ device.

1: **GPU function** DLN_ADD_FORCE_THREAD_DEVICE($^{\mathrm{t}}\boldsymbol{F_n}[N][3]$, $^{\mathrm{t}}\boldsymbol{F_e}[3]$, $^{\mathrm{d}}\boldsymbol{F_n}[3 \times E \times N]$, $^{\mathrm{d}}\boldsymbol{F_e}[3 \times E]$, $k$)
2: ▷ Nodal force.
3: ▷ Set output index to correspond to whatever SE we're on. By parallelising over dislocation lines, we must loop through SEs in the main code, this is the value of $k$ we provide. Multiply by 3 and $N$ because each SE has 3 coordinates and $N$ nodes.
4: $i \leftarrow 3 \times N \times k$
5: $j \leftarrow 0$ ▷ Auxiliary index.
6: **for** $(n = 0;\ n < N;\ n++)$ **do** ▷ Loop over nodes.
7: **for** $(c = 0;\ c < 3;\ c++)$ **do** ▷ Loop over coordinates.
8: ▷ Ensure nodal forces are correctly added and mapped from local thread memory to global device memory.
9: atomicAdd($^{\mathrm{d}}\boldsymbol{F_n}[i + j + c]$, $^{\mathrm{t}}\boldsymbol{F_n}[n][c]$)
10: **end for**
11: ▷ Displace auxiliary index to point at the first coordinate of the $(n+1)^{\text{th}}$ node of the $k^{\text{th}}$ SE.
12: $j \leftarrow j + 3$
13: **end for**
14: ▷ Total force.
15: ▷ Set auxiliary index to point at the first coordinate of the $k^{\text{th}}$ SE.
16: $j \leftarrow 3 \times k$
17: **for** $(c = 0;\ c < 3;\ c++)$ **do** ▷ Loop over coordinates.
18: atomicAdd($^{\mathrm{d}}\boldsymbol{F_e}[j + c]$, $^{\mathrm{t}}\boldsymbol{F_e}[c]$)
19: **end for**
20: **return** $^{\mathrm{d}}\boldsymbol{F_n}$, $^{\mathrm{d}}\boldsymbol{F_e}$
21: **end GPU function**

**Algorithm 5.5** Nodal force in device $\mapsto$ host.

**function** FX_DEVICE_HOST_MAP($^{\mathrm{d}}\boldsymbol{F_n}[3 \times E \times N]$, $^{\mathrm{h}}\boldsymbol{F_n}[N][3 \times E]$)
  $i,\ j \leftarrow 0$ ▷ Set input and output indices to zero.
  **for** $n = 0;\ n < N;\ n++$ **do** ▷ Loop over nodes.
    $j \leftarrow 0$ ▷ Reset output index to point at the first element.
    **for** $e = 0;\ e < E;\ e++$ **do** ▷ Loop over elements.
      **for** $c = 0;\ c < 3;\ c++$ **do** ▷ Loop over coordinates.
        $^{\mathrm{h}}\boldsymbol{F_n}[n][j + c] =^{\mathrm{d}} \boldsymbol{F_n}[i + e + c \times E \times N]$
      **end for**
      ▷ Advance output index to point at the first coordinate of the $(e+1)^{\text{th}}$ element.
      $j \leftarrow j + 3$
    **end for**
    ▷ Advance input index to point at the first coordinate of the $(n+1)^{\text{th}}$ node of the first element.
    $i \leftarrow i + E$
  **end for**
  **return** $^{\mathrm{h}}\boldsymbol{F_n}$
**end function**

### 5.1.1 Data Mapping

#### 5.1.1.1 Elements: Host $\mapsto$ Device

#### 5.1.1.2 Elements: Device $\mapsto$ Thread

#### 5.1.1.3 Force: Thread $\overset{+}{\mapsto}$ Device

#### 5.1.1.4 Force (Parallelise over Dislocations): Thread $\overset{+}{\mapsto}$ Device

#### 5.1.1.5 Nodal Force: Device $\mapsto$ Host

#### 5.1.1.6 Total Force: Device $\mapsto$ Host

$$\boldsymbol{X}_{en} \coloneqq [x_{en},\ y_{en},\ z_{en}] \tag{5.1a}$$

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

$$\boldsymbol{X}_n \coloneqq [x_{1n},\ y_{1n},\ z_{1n}, \ldots, x_{En},\ y_{En},\ z_{En}] \tag{5.1b}$$

$$\boldsymbol{X}_{1\to N} \mapsto \boldsymbol{X}^{\mathrm{SE}}$$

$$\boldsymbol{X}^{\mathrm{SE}} \coloneqq \begin{bmatrix} x_{11},\ \ldots,\ x_{E1},\ x_{12},\ \ldots,\ x_{E2},\ \ldots,\ x_{1N},\ \ldots,\ x_{EN}, \\ y_{11},\ \ldots,\ y_{E1},\ y_{12},\ \ldots,\ y_{E2},\ \ldots,\ y_{1N},\ \ldots,\ y_{EN}, \\ z_{11},\ \ldots,\ z_{E1},\ z_{12},\ \ldots,\ z_{E2},\ \ldots,\ z_{1N},\ \ldots,\ z_{EN} \end{bmatrix} \tag{5.1c}$$

where $e$ is the SE label, $n$ the node label, $E$ the number of SEs in the scope. $N$

[1] Not quite but essentially simultaneously.

**Algorithm 5.6** Total force in device $\mapsto$ host.

**function** FTOT_DEVICE_HOST_MAP(${}^{\mathrm{d}}\boldsymbol{F_e}[3 \times E]$, ${}^{\mathrm{h}}\boldsymbol{F_e}[3 \times E]$)
    $i \leftarrow 0$ ▷ Set output index to zero.
    **for** $e = 0$; $e < E$; $e++$ **do** ▷ Loop over elements.
        **for** $c = 0$; $c < 3$; $c++$ **do** ▷ Loop over coordinates.
            ${}^{\mathrm{h}}\boldsymbol{F_e}[i + c] = {}^{\mathrm{d}}\boldsymbol{F_e}[e + c \times E]$
        **end for**
            ▷ Advance the output index to point at the first coordinate of the $(e + 1)^{\mathrm{th}}$ surface element.
        $i \leftarrow i + 3$
    **end for**
    **return** ${}^{\mathrm{h}}\boldsymbol{F_e}$
**end function**

the number of nodes in a SE.

Data-mapping according to **??** and relabelling the nodes so they go from $0 \rightarrow N - 1$, the data from fig. 5.1 would be arranged in device memory like eq. (5.2),

$$\boldsymbol{X}^{\mathrm{SE}} = \Big[\underbrace{h_x,\, i_x,\, f_x,\, g_x}_{\mathbf{x_0}},\, \underbrace{a_x,\, b_x,\, i_x,\, h_x}_{\mathbf{x_1}},\, \underbrace{i_x,\, d_x,\, e_x,\, f_x}_{\mathbf{x_2}},\, \underbrace{b_x,\, c_x,\, d_x,\, i_x}_{\mathbf{x_3}},\; \ldots\ y\text{-coord}\ \ldots,\; \ldots\ z\text{-coord}\ \ldots\Big] \tag{5.2}$$

In the GPU, each thread will cater to one SE at a time. This means that each thread will have to extract the relevant data from the 1D array with length $3 \times E \times N$ into four 1D arrays of length 3. The purpose of CNE mapping is to provide the warp with coalesced memory access. This is achieved via **??**.

Since **??** is performed in a CUDA GPU, threads in a single block execute sequentially from,

$$\text{threadIdx}.x = 0 \rightarrow \text{threadIdx}.x = \text{blockDim}.x - 1\,, \tag{5.3}$$

while threads in different blocks execute in parallel. Coalesced memory access is ensured by having each block load a cache line whose entries are contiguously accessed by the threads in the block. Using the same notation as **??**, full cache line utilisation (optimal cache use) is achieved if cache lines can accomodate $l$ entries

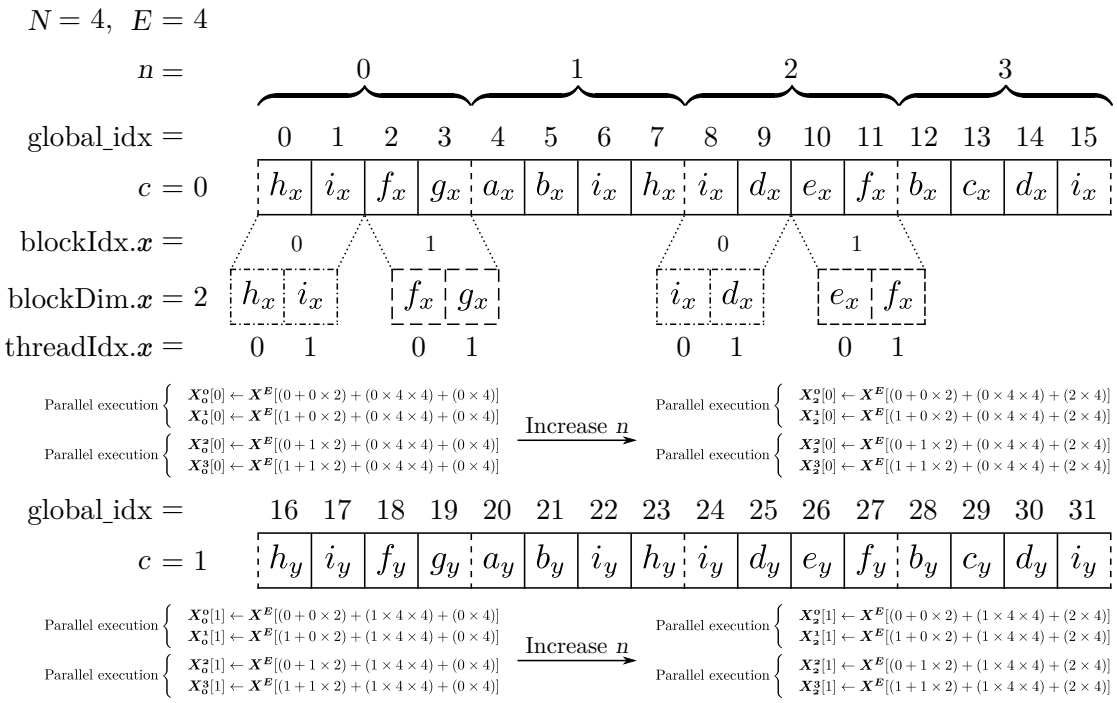



Figure 5.2: Minimum working complete example of the backward coodinate-node-element-map. The explicit calculations of global indices correspond to subsituting the values for "idx" and "idxi" as in **??**. $\boldsymbol{X}_{\boldsymbol{a}}^{\boldsymbol{b}}$ denotes a 1D array of length three containing 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.

given by eq. (5.4),

$$l = \begin{cases} a \times N \times E & , a > 0 \in \mathbb{N} \\ & \text{or} \\ \dfrac{1}{2^a} \times N \times E & , a \geq 0 \in \mathbb{N},\ N \times E \equiv 0 \pmod{2^a}. \end{cases} \tag{5.4}$$

Figure 5.2 shows an example of **??** up to the $y$-coordinate.

#### 5.1.1.7 Node Coordinate Element Map

The node-coordinate-element (NCE) data mapping in eq. (5.5) is carried out by algorithm 5.7, each thread looks after a given SE.

**Algorithm 5.7** NCE data mapping.

**for all** n nodes ∈ surface element **do**
  **for all** c coordinates ∈ $[x, y, z]$ **do**
    **for all** e surface elements ∈ surface mesh section **do**
      list.append(data of the $n^{\text{th}}$ node with $c^{\text{th}}$ coordinate of the $e^{\text{th}}$ SE)
    **end for**
  **end for**
**end for**

$$\boldsymbol{X}_{en} \coloneqq [x_{en},\, y_{en},\, z_{en}] \tag{5.5a}$$

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

$$\boldsymbol{X}_n \coloneqq [x_{1n}, \ldots,\, x_{En},\, y_{1n}, \ldots,\, y_{En},\, z_{1n}, \ldots,\, z_{En}] \tag{5.5b}$$

$$\boldsymbol{X}_{1\to N} \mapsto \boldsymbol{X}^{\text{SE}}$$

$$\boldsymbol{X}^{\text{SE}} \coloneqq \begin{matrix} [x_{11}, \ldots,\, x_{E1},\, y_{11},\, \ldots,\, y_{E1},\, z_{11}, \ldots,\, z_{E1} \\ \ldots, \\ x_{1M}, \ldots,\, x_{EN},\, y_{1N}, \ldots,\, y_{EN},\, z_{1N}, \ldots,\, z_{EN}] \end{matrix} \tag{5.5c}$$

where $e$ is the surface **e**lement, $n$ the **n**ode, $E$ the total number of SEs in scope, $N$ the total number of nodes in each SE.

Data-mapping according to algorithm 5.7 and relabelling the nodes so they go from $0 \to N-1$, the data from fig. 5.1 would be arranged in device memory like eq. (5.6),

$$\boldsymbol{X}^{\text{SE}} = \begin{matrix} [\underbrace{h_x,\, i_x,\, f_x,\, g_x,\, h_y,\, i_y,\, f_y,\, g_y,\, h_z,\, i_z,\, f_z,\, g_z}_{\mathbf{x_0}}\,, \\ \ldots\ \mathbf{x_1},\, xyz\text{-coords}\ \ldots, \\ \ldots\ \mathbf{x_2},\, xyz\text{-coords}\ \ldots, \\ \ldots\ \mathbf{x_3},\, xyz\text{-coords}\ \ldots] \end{matrix} \tag{5.6}$$

#### 5.1.1.8 Resolving Data Write Conflicts

Data write conflicts can be a problem in parallel applications where global data is changed by multiple threads within the same clock tick. This can be avoided with atomic operations.

### 5.1.2 Resolving Parallel Dislocation Line Segments to Surface Elements

Dealing with the special case when the dislocation line segment $\boldsymbol{t}$ is parallel to the SE is relatively trivial when in a serial program. By the mean value theorem, we

can slightly perturb $\boldsymbol{t}$ by rotating it by a small angle around the midpoint of $\boldsymbol{t}$ with respect to the axis of rotation defined by $\boldsymbol{t} \times \boldsymbol{n}$. In contrast to serial code, program branches can have a serious effect on parallel performance due to warp divergence. Due to the complexity of the force calculation and relative rarity of the edge case, there is no branching behaviour in the parallel code until *after* the calculation is performed. Where algorithm 5.8 is performed.

**Algorithm 5.8** Resolving cases when $\boldsymbol{t} \parallel \boldsymbol{n}$ on GPUs.

**for all** surface elements *and* line segments **do**
    ...
    **if** $\boldsymbol{t}_{\text{thread}} \parallel \boldsymbol{n}_{\text{thread}}$ **then**
        $\boldsymbol{x}_{\text{buffer}} \leftarrow \boldsymbol{x}_{\text{thread}}$
        $\boldsymbol{t}_{\text{buffer}} \leftarrow \boldsymbol{t}_{\text{thread}}$
    **else**
        $\boldsymbol{F}_{\text{total}} += \boldsymbol{F}_{\text{thread}}$
    **end if**
**end for**
return to serial code
**if** $\boldsymbol{x}_{\text{buffer}} \neq$ empty **then**
    **for all** surface elements *and* line segments **do**
        perform calculation for $\boldsymbol{t} \parallel \boldsymbol{n}$
    **end for**
**end if**

#### 5.1.2.1 Implementation

The node mapping would be the same as **??**. The parallelisation would be on individual packets of a SE with a slightly rotated dislocation line segment, as in a single iteration of the serial code where the dislocation line segment is rotated. The forces would be averaged with atomic operations and `__syncthreads()` before they are added to the total nodal forces.

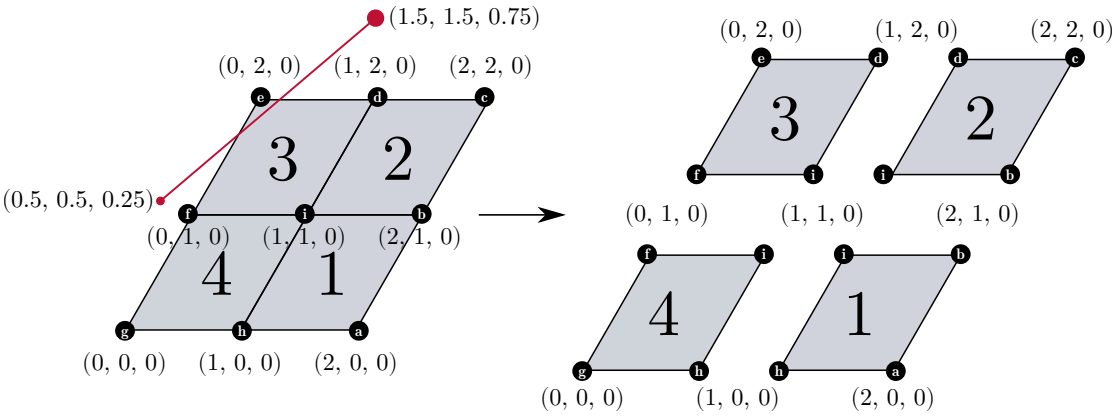



Figure 5.3: Test case for CUDA development. Dislocation line not shown in node separation as the parallelisation happens across surface elements. Not to perspective or scale.

### 5.1.3 Test Case

According to **??** and the node labelling scheme of fig. 3.1 the data mapping would look like:

$$\boldsymbol{X}^{\mathrm{SE}} = \left[\begin{array}{c} \overbrace{\underbrace{1,1,0,0}_{\mathbf{x_3}},\underbrace{2,2,1,1}_{\mathbf{x_4}},\underbrace{1,1,0,0}_{\mathbf{x_5}},\underbrace{2,2,1,1}_{\mathbf{x_6}}}^{x\text{-coord}}, \\ \overbrace{\underbrace{0,1,1,0}_{\mathbf{x_3}},\underbrace{0,1,1,0}_{\mathbf{x_4}},\underbrace{1,2,2,1}_{\mathbf{x_5}},\underbrace{1,2,2,1}_{\mathbf{x_6}}}^{y\text{-coord}}, \\ \overbrace{\underbrace{0,0,0,0}_{\mathbf{x_3}},\underbrace{0,0,0,0}_{\mathbf{x_4}},\underbrace{0,0,0,0}_{\mathbf{x_5}},\underbrace{0,0,0,0}_{\mathbf{x_6}}}^{z\text{-coord}} \end{array}\right] \tag{5.7}$$

In general the number of indices in the array would be $3 \times e \times n$ where $e$ is the number of surface elements, $n$ the number of nodes per element and 3 the number of spatial dimensions.

## 5.2 Thread Block Size Optimisation

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#execution-configuration-optimizations

# Bibliography

[1] 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.

[2] 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.

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

# Appendix A

# Best Practices

This chapter defines the best practices that will make improve the development and testing pipeline by defining rules and standards that facilitate collaboration.

## A.1 Filesystem

All files must be within `/project_name`.

1. All files must be within `/project_name`,
   `~/` = `project_name/`,
   `~/~/` = `project_name/src/` or `project_name/dev/`.
2. Release code must be within `~/src/`.
3. Development code must be within `~/dev/`.
4. Test data must be within `~/tests/`.
5. Documentation must be within `~/~/doc/`.
6. Examples must be within `~/~/exmp/`.
7. External libraries must be within `~/lib/`.
8. Generated images must be within `~/~/images/`.
9. Old versions recordkeeping must be within `~/prv/va.b.c/`.

## A.2 Versioning

1. Use a version control system like GitHub or PasteBin.

2. There must be a master branch that is only changed when the code is stable and bug free.

3. Development branches should be exploited as seen fit without making things overly convoluted.

4. Commits must be as bug free and regular as possible. When to commit is left to the developer's discresion.

5. Commit messages should be as descriptive as possible.

6. Versions should be specified as `va.b.c` where `a`, `b`, `c` = integers. The three levels are `a` = release version (usable, bug free code), `b` = beta version (code that is undergoing testing), `c` = alpha version (code that is under active development).

## A.3 Documentation

### A.3.1 Commenting

Every codefile must be appropriately commented by meeting the following guidelines.

1. The start of each codefile must have a heading detailing the creator, date of creation and edit history (date and name of editor).

2. Below the heading there must be a general explanation of the code. It must state any procedures, structures, objects and how they are to be utilised. Any backward or forward dependencies must be stated.

3. Below the description and edit history any relevant literature must be mentioned (dois are preferred). Must be as detailed as possible, include equation numbers/ranges if necessary.

4. The start of every procedure has an explanation of its purpose, inputs, outputs and inputs-outputs.

5. Particularly complicated code blocks must have an in-depth explanation of what it does. Comment each line if necessary.

6. Corrections or additions must be explicitly bounded by comments at the start and end of the change. Both bounding comments must have the author's name and the date. Below the starting comment, there should be an explanation of the change. Any punctual comments can be made as normal.

### A.3.2 README

Every codefile must have an associated README `.tex` document that documents the codefile's contents. It must meet the following guidelines as appropriate.

1. The name must be that of the file it documents (minus the extension of course).
2. Description and overall explanation of the codefile's purpose.
3. Overall flow chart or pseudo code describing the file's purpose.
4. Document the codefile's procedures. This means describing and explaining their corresponding inputs, outputs, inputs-outputs, forwards and backwards dependencies, and flow charts or pseudo codes.
5. Unit test designs and results for each procedure. If appropriate also include those of integral tests.

The codefile may also be appended at the end of documentation if desired (the `minted` package is highly recommended).

## A.4 Modularisation

1. Code repetition must be kept to a *strict* minimum. Any piece of code that will be reused must be modularised.
2. Procedures must be as self-sufficient as possible *without* repeating code. If repeating code is necessary, replace it with a procedure that is to be repeatedly called instead. Minimising repeated code $\ggg$ procedure self-sufficiency.

## A.5 Coding Style

All names must meet the following guidelines.

1. Indent appropriately. Four space tabs are a good compromise between code necking and readability.
2. Minimise the use of nested code blocks, use intrisics, libraries or create procedures instead.
3. Break up lines that are uncomfortably long, typically anything over 80–100 characters.

4. Names should be appropriately descriptive and human readable.

5. All code and names must be systematic and logical.

6. the use of upper cases should be reserved for parameters (`const` variables in C).

7. Delimit words with "`_`" *not* case changes.

8. Long and descriptive $\ggg$ short and cryptic.

### A.5.1 Filenames

1. If old versions are to be kept, the old *stable* versions of file must have the date of last modification appended *suffixed* after the file extension in the `_yyyymmdd` format. For example, if the stable version of the release code `hello_world_parallel.c` was last modified on April 25, 2017 it should be archived as
   `hello_world_parallel.c_20170425`.

### A.5.2 Variables, Structures and Objects

1. Only counters and indices can be single letter variables.

2. Structure and object *definitions* are *suffixed* with `_s` and `_o` respectively.

3. Inputs, outputs and input-outputs to procedures must be *prefixed* with `i_`, `o_` and `io_` respectively.

### A.5.3 Procedures

1. Functions and subroutines must be *prefixed* with `f_` and `s_` respectively.

# Appendix B

# Coupling Discrete Dislocation Dynamics to Finite Element Methods

```
function node_plane = extract_node_plane(fem_nodes  ,
↪  fem_node_cnct,...
                                         node_labels, dim, filter
                                          ↪  ,...
                                         node_plane)

    ↪  %%=================================================================%%

    ↪  %-----------------------------------------------------------------%
    % Written by famed MATLAB hater and fan of compiled languages,
    % Daniel Celis Garza, in 13/11/17.
    %
    % This code has been optimised as much as I can and has been
    ↪  thoroughly
    % tested as of 15/11/17.

    ↪  %-----------------------------------------------------------------%
    %
    % Extracts a single node plane according to given filtering
    ↪  criteria
    % and puts it into a 2D array arranged in xyz-coordinates like
    ↪  so.
    %
```

```
16      %     Node 1    Node 2    Node 3 ... Node N
17      %
18      %      x_11      x_12      x_13       x_1N   |
19      %      y_11      y_12      y_13       y_1N   | Element 1
20      %      z_11      z_12      z_13  ...  z_1N   |__________
21      %      x_21      x_22      x_23       x_2N   |
22      %      y_21      y_22      y_23       y_2N   | Element 2
23      %      z_21      z_22      z_23       z_2N   |__________
24      %               .                              .
25      %                         .                    .
26      %                                  .           .
27      %                                              __________
28      %      x_E1      x_E2      x_E3       x_EN   |
29      %      y_E1      y_E2      y_E3  ...  y_EN   | Element E
30      %      z_E1      z_E2      z_E3       z_EN   |
31      %
32
        ↪  %=========================================================================
33      % Inputs
34
        ↪  %=========================================================================
35      %
36      % fem_nodes := dimension(:, 3). Assumed shape 2D array with 3
        ↪  columns.
37      % Describes the coordinates of the nodes of the ft_inite
        ↪  element model.
38      %
39      % fem_node_cnct := dimension(:, 8). Assumed shape 2D array
        ↪  with 8
40      % columns. Describes the node connectivity of the finite
        ↪  elements in
41      % the model.
42      %
43      % node_labels := dimension(:). The labels of the nodes that
        ↪  are to be
44      % extracted from fem_nodes.
45      %
46      % dim := three times the area of the plane to be extracted (3
47      % dimensions).
```

```
48      %
49      % filter := the selection criteria used to filter which nodes
        ↪  we want
50      % to obtain.
51      %
52
        ↪  %=========================================================================%
53      % Dummy variables
54
        ↪  %=========================================================================%
55      %
56      % n_nodes := number of nodes per planar element.
57      %
58      % tmp_nodes := dimension(n_nodes, 3). Contains the
        ↪  xyz-coordinates of
59      % all nodes with the labels specified by node_label.
60      %
61      % mask := same dimension as tmp_nodes. Logical mask whose
        ↪  entries are 1
62      % only when a given condition is met. It is used to index
        ↪  tmp_nodes to
63      % find only the nodes that correspond to a given plane.
64      %
65
        ↪  %=========================================================================%
66      % Input/output variables
67
        ↪  %=========================================================================%
68      %
69      % node_plane := dimension(dim/3, n_nodes). It is the slice of
        ↪  the array
70      % that holds the nodes of the plane. Only pass the array slice
71      % that corresponds to the current plane of nodes to be
        ↪  extracted,
72      % a la Fortran intent(in out).
73      %
74
        ↪  %%=======================================================================%%
75
```

```
76      %% Error checking
77      n_nodes     = size(node_labels, 2);
78      if(n_nodes ~= size(node_plane , 2))
79          error('extract_node_plane:: n_nodes = %d, size(node_plane,
            ↪  2) = %d; they must be the same i.e. the number of nodes
            ↪  per planar element', n_nodes, size(node_plane,2))
80      end %if
81
82      %% Finding plane nodes
83      % Find the all the nodes labelled as stated in node_labels
84      % (corresponding to a plane family)
85      tmp_nodes = fem_nodes(fem_node_cnct(:, node_labels), 1:3);
86      % Find an array of entries that meet the filter criteria in
        ↪  tmp_nodes
87      mask = tmp_nodes(:, filter(1)) == filter(2);
88      % Reshape the array to a column array containing only the
        ↪  nodes which
89      % meet the filter criteria.
90      tmp_nodes = reshape(tmp_nodes(mask, :)', n_nodes * dim, 1);
91
92      %% Passing data to the slice of the array that contains the
        ↪  nodes for every plane
93      step = 0;
94      for i = 1: n_nodes
95          node_plane(1: dim, i) = tmp_nodes(1 + step: dim + step);
96          step = step + dim_3;
97      end %for
98
99      %% Cleanup
100     clear dim_3; clear idxf; clear tmp_nodes; clear mask; clear
        ↪  step;
101 end %function
```

```
1   function [node_plane_lbl, node_plane] = extract_node_planes(
    ↪  ...
2                                           fem_nodes     ,
                                            ↪  fem_node_cnct,...
3                                           surf_node_util,
                                            ↪  fem_planes    ,...
```

```
                                                n_elem        , n_nodes)

    %%=====================================================================%%

    %----------------------------------------------------------------------%
% Written by famed MATLAB hater and fan of compiled languages,
% Daniel Celis Garza, in 13/11/17.
%
% This code has been optimised as much as I can and has been thoroughly
% tested as of 15/11/17.

    %----------------------------------------------------------------------%
%
% _lbl := variables suffixed with it are the global label equivalent
%   of those without.
%
% Extracts a single node plane according to given filtering criteria
% and puts it into a 2D array arranged in xyz-coordinates like so.
%
%     Node 1    Node 2    Node 3 ... Node N
%
%      x_11      x_12      x_13       x_1N   |
%      y_11      y_12      y_13       y_1N   | Element 1
%      z_11      z_12      z_13  ...  z_1N   |___________
%      x_21      x_22      x_23       x_2N   |
%      y_21      y_22      y_23       y_2N   | Element 2
%      z_21      z_22      z_23       z_2N   |___________
%             .                                    .
%                    .                             .
%                           .                      .
%                                                   ___________
%      x_E1      x_E2      x_E3       x_EN   |
%      y_E1      y_E2      y_E3  ...  y_EN   | Element E
%      z_E1      z_E2      z_E3       z_EN   |
%
```

```
36
    %=========================================================================
37  % Inputs
38
    %=========================================================================
39  %
40  % fem_nodes := dimension(:, 3). Assumed shape 2D array with 3 columns.
41  %   Describes the coordinates of the nodes of the ft_inite element model.
42  %
43  % fem_node_cnct := dimension(:, 8). Assumed shape 2D array with 8
44  %   columns. Describes the node connectivity of the finite elements in
45  %   the model.
46  %
47  % surf_node_util := dimension(n_nodes+2, 6). Each column is laid out as
48  %   follows:
49  %       [node_label_initial; ... ; node_label_final;
50  %       plane_area; coordinate_number].
51  %   Each row corresponds to one plane.
52  %
53  % fem_planes := dimension(:). Contains the planes whose nodes are to be
54  %   extracted.
55  %
56  % n_elem := total number of elements to be extracted.
57  %
58  % n_nodes := number of nodes per planar element.
59  %
60
    %=========================================================================
61  % Dummy variables
62
    %=========================================================================
63  %
```

```
64      % plane_idx := the index of the fem plane to be extracted
        ↪  during the
65      %   current iteration.
66      %
67      % dim := plane_area, there are this many surface elements per
        ↪  plane.
68      %
69      % dim_3 := 3*plane_area, there are three coordinates per plane
70      %   node, plane_area is the total number of nodes in a plane
        ↪  and is found
71      %   in surf_node_utils.
72      %
73      % tmp_nodes := dimension(n_nodes, 3). Contains the
        ↪  xyz-coordinates of
74      %   all nodes with the labels specified by node_label.
75      %
76      % mask := same dimension as tmp_nodes. Logical mask whose
        ↪  entries are 1
77      %   only when a given condition is met. It is used to index
        ↪  tmp_nodes to
78      %   find only the nodes that correspond to a given plane.
79      %
80      % coord := orthogonal coordinate to the plane (for node
        ↪  filtering
81      %   purposes).
82      %
83      % idxi, idxf := initial and final index that slice the
        ↪  coordinate
84      %   output array into sections for each extracted plane.
85      %
86      % idxl, idxm := initial and final index that slice the label
87      %   output array into sections for each extracted plane.
88      %
89      % filter := value for filtering data.
90      %
91      % step := step that ensures proper array indexing for each
        ↪  section of
92      %   the linear array that contains the nodes of a given plane.
93      %
```

```
94      % n_nodes_p1 := n_nodes + 1
95      %
96      % n_nodes_p2 := n_nodes + 2
97      %
98
        ↪  %=========================================================================
99      % Output variables
100
        ↪  %=========================================================================
101     %
102     % node_plane := dimension(3*n_elem, n_nodes). It is the array
        ↪  that
103     %   holds the planes' nodes' Cartesian coordinates.
104     %
105
        ↪  %%=======================================================================%
106
107     %% Allocating node plane coordinates and labels
108     node_plane     = zeros(3 * n_elem, n_nodes);
109     node_plane_lbl = zeros(    n_elem, n_nodes);
110
111     %% Looping through the planes to be extracted
112     idxl = 1;
113     idxi = 1;
114     n_nodes_p_1 = n_nodes + 1;
115     n_nodes_p_2 = n_nodes + 2;
116     for i = 1: size(fem_planes, 1)
117         %% Assigning control variables for the iteration
118         plane_idx = fem_planes(i);
119         dim   = surf_node_util(n_nodes_p_1, plane_idx);
120         dim_3 = 3 * dim;
121         coord = surf_node_util(n_nodes_p_2, plane_idx);
122         % Set final index of the output array slice containing the
            ↪  nodes
123         % for this iteration's plane
124         idxm = idxl + dim   - 1;
125         idxf = idxi + dim_3 - 1;
126         %% Finding global node labels and node coordinates of the
            ↪  plane
```

```
127         % Extracting all nodes whose labels correspond to the
            ↪ plane family
128         % to be extracted this iteration
129         tmp_nodes_lbl = fem_node_cnct(:, surf_node_util(1:n_nodes,
            ↪ plane_idx));
130         tmp_nodes     = fem_nodes(tmp_nodes_lbl, 1:3);
131         % If the plane index is odd it is a face where its
            ↪ orthogonal
132         % coordinate is at a minimum.
133         if(mod(plane_idx, 2) ~= 0)
134             filter = min(tmp_nodes(:, coord));
135         % Otherwise find the face whose orthogonal coordinate is
            ↪ at a maximum
136         else
137             filter = max(tmp_nodes(:, coord));
138         end %if
139         % Find an array of entries that meet the filter criteria
            ↪ in tmp_nodes
140         mask = tmp_nodes(:, coord) == filter;
141         % Reshape the array to a column array containing only the
            ↪ nodes which
142         % meet the filter criteria.
143         tmp_nodes = reshape(tmp_nodes(mask,:)', n_nodes * dim_3,
            ↪ 1);
144         % Reshape the array with global node labels so it can be
            ↪ filtered
145         % using the same mask.
146         tmp_nodes_lbl_size = size(tmp_nodes_lbl, 1);
147         tmp_nodes_lbl      = reshape(tmp_nodes_lbl ,...
148                                      tmp_nodes_lbl_size * n_nodes,
                                         ↪ 1);
149         tmp_nodes_lbl = tmp_nodes_lbl(mask);
150
151         %% Passing data to the slice of the array that contains
            ↪ the nodes labels and coords for every plane
152         % Pass the node labels of this iteration to the global
            ↪ array.
153         % Set the slice of the tmp_nodes array to be indexed
154         step     = 0;
```

```
155         step_lbl = 0;
156         for j = 1: n_nodes
157             % Grab the array slice that corresponds to node j in
                ↪ the
158             % tmp_nodes array and send it off to the corresponding
                ↪ column
159             % of the slice of the output array containing all
                ↪ extracted nodes
160             node_plane(idxi: idxf, j) = tmp_nodes(...
161                                                 1 + step: dim_3 +
                                                    ↪ step...
162                                               );
163             node_plane_lbl(idxl: idxm, j) = tmp_nodes_lbl(...
164                                                 1 + step_lbl: dim +
                                                    ↪ step_lbl...
165                                               );
166             % Advance the step to move onto the array slice that
                ↪ contains
167             % node j+1 in tmp_nodes
168             step     = step     + dim_3;
169             step_lbl = step_lbl + dim  ;
170         end %for
171         % Set the initial index to shift the output array slice
            ↪ onto the
172         % next plane to be extracted.
173         idxi = idxf + 1;
174         idxl = idxm + 1;
175         clear tmp_nodes; clear tmp_nodes_lbl; clear mask;
176     end %for
177
178     %% Cleanup
179     clear idxl; clear idxi; clear plane_idx; clear dim; clear
        ↪ dim_3;
180     clear coord; clear idxm; clear idxf; clear filter; clear
        ↪ tmp_nodes_lbl_size;
181     clear step; clear step_lbl; clear n_nodes_p1; clear n_nodes_p2;
182 end %function
```

```
function [f_dln, f_dln_se] = analytic_traction( ...
                    se_node_coord, dln_node_coord, burgers    , n_nodes,...
                    n_nodes_t    , n_se          , n_dln      , idxf   ,...
                    idxi         , f_dln_node    , f_dln_se   , f_dln  ,...
                    mu, nu, a    , use_gpu       , n_threads  , para_scheme)

    %%=====================================================================%%

    %---------------------------------------------------------------------%
    % Written by famed MATLAB hater and fan of compiled languages,
    % Daniel Celis Garza, in 12/11/17--16/11/17.
    % Refactored into independent subroutines in 11/12/2017--15/12/2017.

    %---------------------------------------------------------------------%
    % Calculates analytical forces to the surfaces of a rectangular
    % cantilever whose nodes are labelled as follows:
    %
    % 4. ----- .3
    %  |\       |\
    %  | \      | \
    % 1. -\---- .2 \
    %   \  \     \  \
    %    \ 8. ----\- .7
    %     \ |      \ |
    %      \|       \|
    %      5. ----- .6
    % ^z
    % |  y
    % | /
    % |/
    % |------>x
    %
```

```
30      % Everything has been arranged to conform with the analytical
        ↪  solutions
31      % for forces exerted on linear rectangular surface elements
        ↪  published by:
32      % S. Queyreau, J. Marian, B.D. Wirth, A. Arsenlis, MSMSE,
        ↪  22(3):035004, (2014).
33      %
34
        ↪  %=========================================================================
35      % Inputs
36
        ↪  %=========================================================================
37      %
38      % se_node_coord := dimension(n_se*n_nodes, 3). 2D array
39      %   with the cartesian coordinates of the finite element nodes
        ↪  subject
40      %   to traction boundary conditions. The analytical solution
        ↪  requires
41      %   they be arranged element by element in a specific order.
42      %
43      % dln_node_coord := dimension(n_dln*2, 3). 2D array with
44      %   the cartesian coordinates of the dislocation line nodes.
        ↪  The
45      %   analytical solution requires they be arranged line segment
        ↪  by
46      %   line segment.
47      %
48      % burgers := dimension(n_dln, 3). 2D array with the
49      %   individual dislocation line segments' Burgers vector. The
50      %   analytical solution needs the Burgers vector for each line
        ↪  segment.
51      %
52      % n_nodes := number of nodes per surface element.
53      %
54      % n_nodes_t := total number of nodes with traction boundary
        ↪  conditions.
55      %   It is needed to map the analytical solutions' forces to
        ↪  the force
```

```
56      %   array used to account for the forces exerted by the
        ↪  dislocation
57      %   ensemble on the finite element nodes. This does not
        ↪  necessarily
58      %   contain a factor of n_nodes less entries than
        ↪  se_node_coord because
59      %   not all nodes in an element may have traction boundary
        ↪  conditions,
60      %   but still need to be included in the calculation due to
        ↪  the
61      %   problem's formulation.
62      %
63      % n_se := number of surface elements
64      %
65      % n_dln := number of dislocation line segments
66      %
67      % idxf := dimension(:). 1D array containing the indices of the
        ↪  force
68      %   array used to account for the forces exerted by the
        ↪  dislocation
69      %   ensemble on the finite element nodes. It is needed to map
        ↪  the
70      %   analytical solutions' forces to the right index. It is
        ↪  usually
71      %   equal to gamma*3 because only the nodes with traction
        ↪  boundary
72      %   conditions are needed.
73      %
74      % idxi := dimension(n_nodes_t*n_nodes). 1D assumed shape array
        ↪
75      %   containing the global label indices of the nodes whose
        ↪  tractions
76      %   were calculated by the analytical solution.
77      %
78      % mu := shear modulus of material.
79      %
80      % nu := poisson's ratio.
81      %
```

```
82      % a := dislocation core size parameter (non-singular
        ↪  dislocations).
83      %
84
        ↪  %-------------------------------------------------------------------
85      %
86      % Flags:
87      %
88      % use_gpu := Flag to use the Graphics Processing Unit (GPU) in
        ↪  the
89      %   calculation. If use_gpu == 1, the code is run by a
        ↪  CUDA-enabled
90      %   NVidia GPU. The code will crash if there is none, or if
        ↪  there is no
91      %   compiled MEX CUDA file. If use_gpu == 0, a serial C code
        ↪  will be
92      %   executed. The code will crash if there is no compiled MEX
        ↪  C file.
93      %   If use_gpu is any other number, a MATLAB version will be
        ↪  used. This
94      %   is much slower than either of the others.
95      %
96
        ↪  %-------------------------------------------------------------------
97      %
98      % Optional parameters:
99      %
100     % n_threads := when use_gpu ==1, it is the number of threads
        ↪  per block
101     %   to be used in the parallelisation. This may require some
102     %   experimentation to optimise. Defaults to 512. Does nothing
        ↪  if a
103     %   GPU isn't used.
104     %
105     % para_scheme := when use_gpu is enabled, it dicates the
106     %   parallelisation scheme used. If para_scheme == 1
        ↪  parallelise over
107     %   dislocation line segments. If para_scheme == 0 parallelise
        ↪  over
```

```
108     %   surface elements. Other parallelisation schemes can be
        ↪  added in the
109     %   CUDA C code.
110     %
111
        ↪  %=====================================================================%
112     % Dummy variables
113
        ↪  %=====================================================================%
114     %
115     % f_dln_node := dimension(3*n_se, 4). 3D force on each
116     %   surface element node.
117     %
118     % f_dln_se := dimension(3*n_se, 1). Total 3D force on each
        ↪  surface
119     %   element (sum over all nodes of an element). This isn't
        ↪  used to
120     %   couple the forces to the finite element forces but can be
        ↪  used to
121     %   provide the evolution of the forces exerted by the
        ↪  dislocations on
122     %   the surface elements.
123     %
124     % tmp := dimension(n_nodes). 1D array containing the at-most 4
125     %   instances where a surface element node appears (a single
        ↪  node can
126     %   be shared by up to 4 surface elements).
127     %
128     % tmp2 := dimension(:). Assumed shape 1D array containing the
        ↪  at-most 4
129     %   instances where a surface element node appears (a single
        ↪  node can
130     %   be shared by up to 4 surface elements) but without any
        ↪  zero-valued
131     %   entries because 0 indices crash MATLAB.
132     %
133     % k := used to traverse idxi to the next block of 4 entries
        ↪  which
134     %   contain data relevant to the node in the
```

```
135     %
136
        ↪ %=========================================================================
137     % Inputs/Outputs
138
        ↪ %=========================================================================
139     %
140     % f_dln := dimension(n_nodes_t*3). 1D array containing the
        ↪ forces
141     %   exerted by the dislocations on the surface nodes. It is
        ↪ used to
142     %   correct the finite element forces.
143     %
144
        ↪ %%========================================================================%
145
146     %% Analytical force calculation.
147     % Parallel CUDA C calculation.
148     if use_gpu == 1
149         % Provide a default number of threads in case none is
            ↪ given.
150         if ~exist('n_threads', 'var')
151             n_threads = 512;
152         end %if
153         % Provide a default parallelisaion scheme in case none is
            ↪ given.
154         if ~exist('para_scheme', 'var')
155             % Parallelise over dislocations.
156             para_scheme = 1;
157         end %if
158         [f_dln_node(:, 1), f_dln_node(:, 2),...
159          f_dln_node(:, 3), f_dln_node(:, 4),...
160          f_dln_se] = nodal_surface_force_linear_rectangle_mex_cuda(
              ↪ ...
161                                 dln_node_coord(:, 1),
                                    ↪ dln_node_coord(:, 2)   ,...
162                                 se_node_coord (:, 1), se_node_coord
                                    ↪ (:, 2)   ,...
```

```
163                                 se_node_coord (:, 3), se_node_coord
                                    ↪  (:, 4)   ,...
164                                 burgers(:), mu, nu, a, n_se, n_dln,
                                    ↪  n_threads,...
165                                 para_scheme);
166         % Serial force calculation in C.
167         elseif use_gpu == 0
168             [f_dln_node(:, 1), f_dln_node(:, 2),...
169              f_dln_node(:, 3), f_dln_node(:, 4),...
170              f_dln_se] = nodal_surface_force_linear_rectangle_mex(
                 ↪  ...
171                                 dln_node_coord(:, 1),
                                    ↪  dln_node_coord(:, 2),...
172                                 se_node_coord (:, 1), se_node_coord
                                    ↪  (:, 2),...
173                                 se_node_coord (:, 3), se_node_coord
                                    ↪  (:, 4),...
174                                 burgers(:), mu, nu, a, n_se,
                                    ↪  n_dln);
175         % Matlab version.
176         else
177             [f_dln_node(:, 1), f_dln_node(:, 2),...
178              f_dln_node(:, 3), f_dln_node(:, 4),...
179              f_dln_se] = NodalSurfForceLinearRectangle2(
                 ↪  ...
180                                 dln_node_coord(:, 1),
                                    ↪  dln_node_coord(:, 2),...
181                                 se_node_coord (:, 1), se_node_coord
                                    ↪  (:, 2),...
182                                 se_node_coord (:, 3), se_node_coord
                                    ↪  (:, 4),...
183                                 burgers(:), mu, nu, a, n_se,
                                    ↪  n_dln);
184         end %if
185
186         %% Map analytical nodal forces into a useful form for the
            ↪  force superposition scheme.
187         % Loop through the number of nodes.
188         k = 0;
```

```
189     tmp = zeros(n_nodes, 1);
190     for i = 1: n_nodes_t
191         % Populate tmp array with the indices corresponding to
            ↪ nodes of
192         % the surface relevant surface element.
193         tmp  = idxi(1 + k: k + n_nodes);
194         % Obtain only the indices which are non-zero. Zero indices
            ↪ mean
195         % those nodes are not under traction boundary conditions.
196         tmp2 = tmp(idxi(1+k: k + n_nodes, 1) ~= 0);
197         for j = 2:-1:0
198             % The index is displaced by -2, -1, 0, corresponding
                ↪ to the
199             % indices of the x, y, z coordinate respectively.
200             % We add the force contributions from all surface
                ↪ elements a node
201             % is part of. This gives us the total x,y,z forces on
                ↪ each node.
202             f_dln(idxf(i) - j) = sum(f_dln_node(tmp2 - j));
203         end %for
204         % Step to the block in idxi where the next surface node's
            ↪ indices
205         % are found.
206         k = k + 4;
207     end %for
208     clear tmp; clear tmp2; clear k;
209 end % function
```

# Appendix C

# Implementation of Analytical Forces Induced by Dislocations on Linear Rectangular Surface Elements

## C.1 Serial C Code MEX File

## C.2 Parallel CUDA C Code MEX File

# Appendix D

# Implementation of Analytical Forces Induced by Dislocations on Quadratic Triangular Surface Elements

## D.1 Serial C Code MEX File

## D.2 Parallel CUDA C Code MEX File

# Appendix E

# Talks

## E.1 Durham July 12–14 2017

Bridging the gap between the microscopic and macroscopic world is an on-going challenge for science and technology. If we hope to understand complex emergent phenomena we need to study systems that blur the line between micro and macro. In materials science, one such area is the study of extended defects called dislocations; whose nucleation and movement mediate the permanent deformation of materials. These large, high energy defects present very complex and long ranged interactions with each other, crystal boundaries, impurities, free surfaces, and themselves. As such, their dynamics are difficult to study experimentally. So we make justified assumptions and simplifications and create models that let us study them in detail. However if our models are to prove useful in real applications, they must be continually refined and improved by weakening assumptions and removing simplifications. Unfortunately, with increased refinement comes increased computational cost and new challenges. Therefore, finding faster alternatives that do not sacrifice accuracy is of the utmost importance—better yet if the alternatives are more accurate or exact. With the advent of increasingly accessible graphics processor units (GPUs) typically used in video gaming, the power of parallel processing is no longer exclusive to researchers with access to supercomputers. In this talk we will discuss the GPU implementation of exact solutions for the forces dislocations exert on the surfaces of materials.

# Index