# An Efficient Adaptive PD Formulation for Complex Microstructures

# DISSERTATION

Zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing)
an der Fakultät Bauingenieurwesen
der Bauhaus-Universität Weimar
Deutschland

vorgelegt von

ALI JENABIDEHKORDI
(interner Doktorand)

Mentor: Prof. Dr.-Ing. Timon Rabczuk

Reviewers: Prof. Dr. Erdogan Madenci
Prof. Dr. Esteban Samaniego
Prof. Dr. rer. nat. Klaus Hackl

Weimar, April 2021

# Acknowledgements

I would like to thank my advisor Prof. Dr.-Ing. Timon Rabczuk for all of his support throughout my Ph.D. and for his patience, motivation, and thoughtful suggestions during my study. I would also like to thank the members of my dissertation committee: Professor Erdogan Madenci, Professor Esteban Samaniego, Professor Guido Morgenthal, Professor Klaus Hackl, and Professor Matthias Kraus for generously offering their time, support, and guidance throughout the preparation and review of this document.

I would also like to express my deepest gratitude to my parents, family, and friends who have supported me in challenging moments with their understanding, love, and patience.

Ali Jenabidehkordi
Weimar, Germany
April. 2021

# Authorship Declaration

I, Ali Jenabidehkordi, hereby solemnly declare that the following dissertation is my own work and that no impermissible assistance from others or references other than those specially cited were used in its making. Data and/or concepts taken directly or indirectly from other sources have been properly referenced. Parts of the dissertation which have already appeared in examination papers are unambiguously marked as such.
I affirm that no other individuals were involved in producing the content of this dissertation. Furthermore, no placement or consulting services (promotion consultants or other persons) were paid to assist me in any way. I affirm that no one received direct or indirect pecuniary compensation or payment in kind for work conducted in connection with the content of this dissertation.
This dissertation has not been previously submitted in the same or similar form to any other examination authority in Germany or abroad.
I certify that, to the best of my knowledge, the declaration above is absolutely true and nothing has been concealed.

Ali Jenabidehkordi
Weimar, Germany
Apr. 2020

***To my parents***

***and my family.***

# Abstract

The computational costs of newly developed numerical simulation play a critical role in their acceptance within both academic use and industrial employment. Normally, the refinement of a method in the area of interest reduces the computational cost. This is unfortunately not true for most nonlocal simulation, since refinement typically increases the size of the material point neighborhood. Reducing the discretization size while keeping the neighborhood size will often require extra consideration. Peridynamic (PD) is a newly developed numerical method with nonlocal nature. Its straightforward integral form equation of motion allows simulating dynamic problems without any extra consideration required. The formation of crack and its propagation is known as natural to peridynamic. This means that discontinuity is a result of the simulation and does not demand any post-processing. As with other nonlocal methods, PD is considered an expensive method. The refinement of the nodal spacing while keeping the neighborhood size (i.e., horizon radius) constant, emerges to several nonphysical phenomena.

This research aims to reduce the peridynamic computational and implementation costs. A novel refinement approach is introduced. The proposed approach takes advantage of the PD flexibility in choosing the shape of the horizon by introducing multiple domains (with no intersections) to the nodes of the refinement zone. It will be shown that no ghost forces will be created when changing the horizon sizes in both subdomains. The approach is applied to both bond-based and state-based peridynamic and verified for a simple wave propagation refinement problem illustrating the efficiency of the method. Further development of the method for higher dimensions proves to have a direct relationship with the mesh sensitivity of the PD. A method for solving the mesh sensitivity of the PD is introduced. The application of the method will be examined by solving a crack propagation problem similar to those reported in the literature.

New software architecture is proposed considering both academic and industrial use. The available simulation tools for employing PD will be collected, and their advantages and drawbacks will be addressed. The challenges of implementing any node base nonlocal methods while maximizing the software flexibility to further development and modification

will be discussed and addressed. A software named Relation-Based Simulator (RBS) is developed for examining the proposed architecture. The exceptional capabilities of RBS will be explored by simulating three distinguished models. RBS is available publicly and open to further development. The industrial acceptance of the RBS will be tested by targeting its performance on one Mac and two Linux distributions.

# Contents

# List of Figures

# List of Tables

# Nomenclature

| Symbol | Description |
|---|---|
| $\delta$ | horizon radius |
| $\Delta t$ | time step duration |
| $\nu$ | Poisson's ratio |
| $\rho$ | mass density |
| $\mathbf{b}$ | body force |
| $BB-PD$ | Bond-Based Peridynamics |
| $BC$ | Boundary Condition |
| $BD$ | Boundary Domain |
| $C_v$ | volume correction factor |
| $c$ | wave speed in material |
| $d_n$ | nodal displacement at node $n$ |
| $Dri$ | direction function |
| $E$ | Young's modulus |
| $f_n$ | nodal force at node $n$ |
| $FE$ | Finite-Element |
| $G_0$ | fracture energy per unit area |
| $H_x$ | horizon centered at $x$ |
| $ID$ | Identification Number |
| $m$ | length of subdomain edge |
| $m$ ratio | ratio of horizon radius to the node spacing |
| $\underline{\mathbf{M}}$ | deformed direction vector field |
| $MD$ | Molecular Dynamic |
| $MHPD$ | Multi-Horizon Peridynamics |
| $NSB-PD$ | Non-ordinary State-Based Peridynamics |
| $OOP$ | Object Oriented Programming |
| $OS$ | Operating System |
| $OSB-PD$ | Ordinary State-Based Peridynamics |
| $Pa$ | Pascal |
| $PD$ | Peridynamic |
| $RBS$ | Relation-Based Simulator |
| $S_\alpha^-$ | left side horizon stiffness |
| $S_\alpha^\beta$ | the stiffness of the horizon from $\alpha$ to $\beta$ angles |
| $S_\alpha^+$ | right-side horizon stiffness |
| $s_c$ | critical stretch |
| $SB-PD$ | State-Based Peridynamics |
| $\underline{\mathbf{T}}$ | force vector state field |
| $\underline{\widehat{\mathbf{T}}}$ | peridynamic material |

| | |
|---|---|
| $\mathbf{u}$ | displacement |
| $UI$ | User Interface |
| $UML$ | Unified Modeling Language |
| $V_x$ | volume currection for the subdomain centered at $x$ |
| $V_x$ | volume of the subdomain centered at $x$ |
| $Vp$ | volume fractions of polymer particles |
| $\widehat{V}_x$ | currected volume of the subdomain centered at $x$ |
| $\mathbf{x}$ / $\mathbf{X}$ | position vector |
| $\mathbf{x}'$ / $\mathbf{X}'$ | neighbor's position vector |
| $\underline{\mathbf{Y}}$ | deformation vector state field |

# Chapter 1

# Introduction

## 1.1 Motivation

Today, scientists working in fields of engineering and most industries are utilizing computational methods in the form of either opensource or commercial tools for understanding the fundamentals of the physics. The cost of simulations and their accuracy when predicting physical mechanisms are essential keys for selecting a numerical method to solve a specific problem. For instance, while the complexity of the underlying physics in the atomic scale created interest in the development of atomistic and molecular computational methods, the developed methods are hopeless for simulation of the bulk model due to the high head computation cost. Throughout the history of numerical simulations, cost efficiency and accuracy have inspired researchers to develop a new method or modify an old one. Therefore, as can be expected, each numerical simulation has a native domain of implimentation which promises either more accurate results or lower costs.

Peridynamics (PD) is a nonlocal continuum mechanics theory proposed by Silling [1]. One of the key advantages of PD over methods such as the extended finite element method is that the crack is a natural outcome of the formulation. This is achieved by rewriting the equation of motion and substituting the divergence of the Cauchy stress tensor with an integral expression that contains the so-called bond forces. The first version proposed in 2000 is called bond-based peridynamics (BB-PD). It allows the symmetric interactions between material points. BB-PD has several restrictions. For instance, it allows only material models with a Poisson's ratio of $0.3\overline{3}$ in 2D and $0.25$ in 3D. Furthermore, the incorporation of arbitrary material models is quite cumbersome or impossible, as the physical interpretation of the bond force vector is complicated, especially with the increasing complexity of the constitutive behavior. Therefore, the so-called state-based peridynamics (SB-PD) has been developed [2]. In SB-PD, the bond force is related to continuum mechanics stresses and hence allows for the incor-

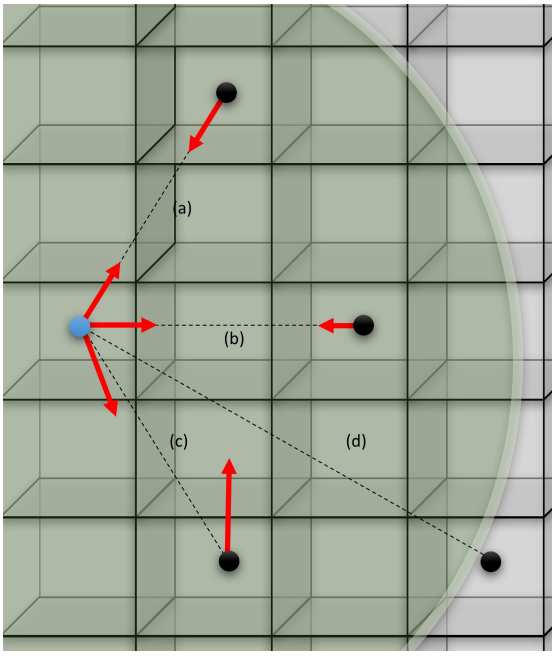



Figure 1.1: PD discretization, horizon presentation, and schematic bound forces for three neighbors in 2D; (a) Bond-Base PD forces; (b) Ordinary State-Based PD forces; (c) Non-ordinary State Base PD forces; (d) nodes outside of the horizon.

poration of more complex material models. Also, the Poisson's ratio limitation has been removed. SB-PD can be further categorized into ordinary state-based and non-ordinary state-based PD. An illustrative figure briefly highlighting the fundamental idea of the different PD formulation is shown in Figure 1.1, where the BB-PD forces always have the same magnitude and lie in the direction of the bound. OSB-PD forces can have different sizes but still must lie in the direction of the bound. NSB-PD forces can have any direction with any magnitude, and finally, nodes outside of the horizon cannot have any forces.

PD benefits from highly accurate and simple implementation where the simulation of discontinuities and time-dependent boundaries or interfaces conditions is concerned. For instance, PD does not require any additional criteria or treatment for simulating discontinuities since they are defined by neglecting the continuum connections (bonds) between points of two sides of the fracture. Moreover, since the PD introduces an equation of motion, the method is dynamic in nature, and no further consideration is required to simulate a dynamic problem. The nonlocality of the PD makes it a perfect match when it comes to the interaction problem, since no surface recognition is required when considering the nodes are connected between two bodies. The separation and contact between multiple bodies is only required to define different simple behavior for bonds involved in the interaction. An example of such simulation is represented

in chapter 4.

Due to several fundamental requirements and limitations, peridynamics cost of computation is considered high when compared to other numerical methods. The PD formulation does not have any limitation on the shape of the subdomain and discretization scheme (see subsection 3.4.2). However, it requires the subdomain volume to have a one to one relation to the volume of the domain. Otherwise, any empty space will be treated as a material vacancy, and any of the overlapping subdomain's volume will be modeled as a jump in material stiffness. This will result in either an unhomogenized/anisotropic behavior of the material or accumulation of the bond forces leading to instability. To avoid such difficulties, most PD implementation utilizes a uniform grid of nodes with cuboidal shapes and constant distance [3].

The PD material points interact with each other within a limited distance creating a spherical space around each node, known as the horizon. The horizon radius is linked with the grid's spacing, where it is a characteristic length-scale of the material and, possibly, the simulated phenomenon [4, 5]. Bobaru *et al.* [6] have demonstrated that the refinement in peridynamic has a close relation to multiscale modeling, since the grid spacing has to be changed with the horizon size. This fact has a significant effect on the refinement of the PD. On the one hand, if the horizon size is maintained while reducing the grid spacing, the neighborhood size, and thus the number of solved equations, will approximately increase by $r^D$. Where $r$ is the initial radius of the horizon in an unrefined area, and $D$ is the dimension of the model. On the other hand, if we reduce the size of the horizon by the same factor as the grid spacing, the ghost forces will result in artificial wave reflection at the refinement bond [6] The introduction of refined areas leads to artificial wave reflections and high sensitivity in the crack path. The dual horizon peridynamics (DH- PD) [7] drastically alleviates this issue by distinguishing between active and reactive bond forces. DH-PD can be applied to BB-PD as well as SB-PD. The proposed algorithms by Bobaru *et al.* [6] and Bobaru and Ha [8], introduces techniques for adaptive refinement for 1D and 2D BBPD.

Although the implementation of the peridynamic in commercial software like ANSYS [9, 10], ABAQUS [11, 12], is reported, the nature of hidden solvers makes them untestable. Thus the reported result of such implementation requires extra caution, especially when developing a new method. The nature of commercial-software hidden implementation forces them to use scripting or, in the best case, provides a library to work with the code, which limits the use of new programming paradigms like object-oriented programing.

There are various peridynamic codes available online. While the majority are targeting specific benchmarks [13], some are written in an objected-oriented fashion with compatibility for multiprocessing for large problems. For instance, Peridigm is a well-known opensource peridynamic code where the authors have provided vast possibilities [14]. However, the provided code is a monolith, and thus, in order to change or

develop part of this code, one must understand the entirety of the code in depth. Combining two or more opensource projects is also not favorable since when any of the involved project despatch features (e.g., for maintenance reasons) or redesigning the architecture, the developer will often have high compatibility and adaptability costs. The use of Molecular Dynamic (MD) codes like LAMMPS is also another option to implement PD [15]. This approach is a perfect match for multiscale problems if they are utilizing both MD and PD simulations. However, using MD code for PD simulations will require end-users to have prior knowledge both in the MD and the code. The development of new PD methods in MD codes is even less convenient due to the fact that a large part of the code is MD specific and is not needed for PD. The same applies to the most meshfree method open-source code.

Refinement and optimizing the order of complexity of the computer implementation are two ways for reducing peridynamics computation costs. The current work is dedicated to addressing both of the above approaches while introducing a native refinement method that does not require changing the PD constitutive model and a road map for the implementation of any node-based methods (e.g., peridynamics). An example of such implementation is provided by utilizing modern computer paradigms. The strength of the suggested implementation and its result validity is presented via three distinguished simulations.

## 1.2 Peridynamic

Peridynamic (PD) is an adjective created by combining the prefix "peri," which means surrounding, and "dyna," which means force or power. Silling *et al.* first used the term "peridynamic" to describe their newly developed method capable of describing the discontinuities and long-range forces. In their initial work, they introduce the following equation of motion to relate the material points' displacement to each other within a body.

$$\rho\left(\mathbf{x}\right)\ddot{\mathbf{u}}\left(\mathbf{x},t\right)=\int_{H_x}\mathbf{f}\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right),\mathbf{x}'-\mathbf{x}\right)dV_{x'}+\mathbf{b}\left(\mathbf{x},t\right) \tag{1.1}$$

where $x$ and $x'$ are the position vectors of two material points within the same domain. $\rho$ is the density, $\mathbf{b}$ denotes the body force, $\mathbf{u}$ is the displacement field and $H_x$ the so-called horizon, which is comparable to the domain of influence in mesh free methods; superimposed dots denote material time derivatives. $\mathbf{f}$ is a linearly dependent function on $\mathbf{x}'-\mathbf{x}$ stretch (i.e., bond stretch) called the pairwise force function. The equation of motion then proved to satisfy the balance of linear momentum, the balance of angular momentum, and the balance of energy over any random body.

The pairwise force can be driven from a micro potential $\omega$ as

$$\mathbf{f}\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right),\mathbf{x}'-\mathbf{x}\right)=\frac{\partial\omega\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right),\mathbf{x}'-\mathbf{x}\right)}{\partial\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right)\right)}. \tag{1.2}$$

The PD material then is called micro-elastic, and the strain energy density can be calculated as

$$W=\frac{1}{2}\int_{H_x}^{\omega}\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right),\mathbf{x}'-\mathbf{x}\right)dV_{\mathbf{x}'-\mathbf{x}} \tag{1.3}$$

since the pairwise force function provides two forces with equal magnitude and opposite direction at two ends of the bond, hence the 1/2 factor. A linear micro-elastic potential leads to a linear relationship between the bond stretch ($s$) and bond force $\mathbf{f_x}$:

$$\omega\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right),\mathbf{x}'-\mathbf{x}\right)=\frac{1}{2}c(\|\mathbf{x}'-\mathbf{x}\|)s^2\left\|\mathbf{x}'-\mathbf{x}\right\| \tag{1.4}$$

where

$$s=\frac{\|\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right)+\mathbf{x}'-\mathbf{x}\|-\|\mathbf{x}'-\mathbf{x}\|}{\|\mathbf{x}'-\mathbf{x}\|}. \tag{1.5}$$

By subscribing equation 1.4 into equation 1.2 one can compute the pairwise bond forces based on the micro modules function $c(\|\mathbf{x}'-\mathbf{x}\|)$ as

$$\mathbf{f}\left(\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right),\mathbf{x}'-\mathbf{x}\right)=c(\|\mathbf{x}'-\mathbf{x}\|)s\frac{\partial\left\|\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right)+\mathbf{x}'-\mathbf{x}\right\|}{\partial\mathbf{u}\left(\mathbf{x}',t\right)-\mathbf{u}\left(\mathbf{x},t\right)} \tag{1.6}$$

where the micro modules function represents the bond elastic stiffness, and can be computed by equating the classical strain energy density with equation 1.3. For 3D isotropic materials it leads to a contact micro modules function

$$c=\frac{18k}{\pi\delta^4} \tag{1.7}$$

where $k$ is the bulk modulus and $\delta$ is the horizon radius.

Since the pairwise force function provides two forces with equal magnitude and opposite direction at two ends of the bond, it imposes a material constraint on the simulation. Using equation 1.1 as the equation of motion while assuming a circular (in 2D) or spherical (in 3D) horizon will enforce the Poisson's ratio of $0.3\overline{3}$ in 2D and $0.25$ in 3D.

Silling *et al.* [1] addressed the initial material limitation of the peridynamic by introducing vector states. A vector state is similar to a second-order tensor in that they both map vectors into vectors, but they have three major differences:

1. A state is not, in general, a linear function of the initial vector field.

2. A state is not, in general, a continuous function of the initial vector field

3. The real Euclidean space of a vector state has infinite-dimensional, while the real Euclidean space of second-order tensors has nine dimensions.

Using the vector state, Silling *et al.* [1] introduced the force vector state $\underline{\mathbf{T}}$ to the equation of motion as follows

$$\rho\ddot{\mathbf{u}} = \int_{H_x} \{\underline{\mathbf{T}} - \underline{\mathbf{T}}'\}\, dV_{x'} + \mathbf{b} \tag{1.8}$$

where the force vector state $\underline{\mathbf{T}}$ can be related to the deformed direction vector $\underline{\mathbf{M}}$, a unit vector pointing from the deformed position of the neighborhood center towards the deformed position of the neighbor:

$$\underline{\mathbf{T}} = \underline{t}\,\underline{\mathbf{M}} \tag{1.9}$$

$\underline{t}$ denoting the magnitude of $\underline{\mathbf{T}}$ thus it is a scalar. If $\underline{t} \equiv 1$, the PD equation of motion is equivalent to equation 1.1. The PD version is then called bond-based peridynamics (BB-PD) in contrast with state-based peridynamics (SB-PD), which can be further distinguished into ordinary and non-ordinary state-based peridynamics. The differences between these three PD versions can be illustrated best in terms of the bond forces, as shown in Figure 1.1.

In contrast to BB-PD, SB-PD allows for general constitutive models. PD can also be regarded as a nonlocal extension of classical continuum mechanics as it replaces the divergence of the stress by an integral term in the equation of motion. One key application of PD is the material failure, though the method has meanwhile been applied to other fields, see e.g., the contributions in [16–18]. In this section, an overview of the peridynamic, related literature review, and PD implementation approaches are discussed.

### 1.2.1 Discretizations

Since $\underline{\mathbf{T}}$ in equation 1.8 is Riemann integrable we can split the right-hand side

$$\rho\ddot{\mathbf{u}} = \int_{H_x} \underline{\mathbf{T}} dV_{x'} - \int_{H_x} \underline{\mathbf{T}}' dV_{x'} + \mathbf{b} \tag{1.10}$$

where the $\underline{\mathbf{T}}$ and $\underline{\mathbf{T}}'$ are the forces applied to two different material points (collocation points). The discretized form of the above equation is

$$\rho_i\ddot{\mathbf{u}}_i = \sum_{j=1}^{n} \mathbf{T}_i V_j - \sum_{j=1}^{n} \mathbf{T}_j V_j + \mathbf{b}_i \tag{1.11}$$

where $i$ is the index of any point with $n$ neighbors ($j$s).

Similar to classical finite-element (FE) codes, peridynamic codes use unique global IDs to identify the nodes from each other. The node sets IDs then facilitate the separation of the initial and boundary condition nodes from the others. If the program supports parallel computation, local node IDs and a mapping system between global and local IDs is often implemented to keep track of the nodes in each computation unit. The use of FE mesh generator tools for PD is then straightforward: solid hexahedral or tetrahedral centroid will be the PD node location in space, and the volume of the mesh will be the nodal volume of the PD node. Utilizing a FE mesh generator for performing PD discretization has several disadvantages

$1^{st}$, PD requires the boundary condition (BC) to be applied over a volume, while FE mesh generators do not typically provide any functionality to define extra layers of mesh around BC faces.

$2^{nd}$, extra caution should be taken when considering the point ID and the set IDs, since they might not be preserved when converting the FE mesh to PD discretization.

$3^{rd}$, the FE mesh generator might use a highly graded discretization in some regions, which is unsuitable for the PD, since it decreases the computational expense dramatically.

$4^{th}$, some of the FE mesh generators return a volume of zero for the 2D mesh, in contrast to the PD which requires the node's volume to solve the equation of motion.

Some of the distribution methods applicable to peridynamic are discussed by Henke and Shanbhag [19].

### 1.2.2 Horizon

The expanded version of Eq. 1.8 peridynamic equation of motion for any two points (**x** and **x'**) within any continuum media is as follows

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int_{H_x} \left( \underline{\mathbf{T}}[\mathbf{x},t] < \mathbf{x'} - \mathbf{x} > - \underline{\mathbf{T}}[\mathbf{x'},t] < \mathbf{x} - \mathbf{x'} > \right) dV_{\mathbf{x'}} + b(\mathbf{x},t) \quad (1.12)$$

where horizon ($H_x$) is a set of points belonging to the continuum medium and each of its subsets can influence the force field at **x**. $\mathbf{u}(\mathbf{x},t)$, $\mathbf{b}(\mathbf{x},t)$, and $\rho(\mathbf{x})$, are the displacement vector field, body force density field and mass density on **x** at time $t$. The

expanded force vector state field on **x** due to the **x'** $-$ **x** deformation at time $t$ denoted as $\underline{\mathbf{T}}[\mathbf{x}, t] < \mathbf{x'} - \mathbf{x} >$.

The balance of linear momentum is naturally satisfied since the same force vector state filed is in use all over the domain [2], provided that both **x** and **x'** can cover each other within their horizons. In otherwords, the peridynamic equation of motion cannot satisfy the balnce of linear momentum if a single point in the domain cannot be found in one of its neighbor's neighborhood. Thus, the peridynamic horizons are required to have central symmetry. Satisfying the balance of angular momentum in any selected bounded body of the domain by utilizing peridynamic constitutive model gives

$$\int_{H_x} (\ \underline{\mathbf{Y}}[\mathbf{x}, t] < \mathbf{x'} - \mathbf{x} > \times\ \underline{\mathbf{T}}[\mathbf{x}, t] < \mathbf{x'} - \mathbf{x} >)\, dV_{(\mathbf{x'}-\mathbf{x})} \tag{1.13}$$

where $\underline{\mathbf{Y}}[\mathbf{x}, t] < \mathbf{x'} - \mathbf{x} >$ is the deformation vector state field on **x** due to the **x'** $-$ **x** deformation at time $t$. Since the $\underline{\mathbf{Y}}$ is only zero if **x'** $-$ **x** is zero, the constitutive model must satisfy equation 1.13. The ordinary state base peridynamic OSB-PD and bond based peridynamic BB-PD constitutive model can be written as

$$\underline{\mathbf{T}}[\mathbf{x}, t] < \mathbf{x'} - \mathbf{x} >= \begin{cases} 0 & if\ \ \underline{\mathbf{Y}}[\mathbf{x}, t] < \mathbf{x'} - \mathbf{x} >= 0 \\ \underline{t}\ \frac{\underline{\mathbf{Y}}[\mathbf{x},t]\ <\mathbf{x'}-\mathbf{x}>}{|\underline{\mathbf{Y}}[\mathbf{x},t]|\ <\mathbf{x'}-\mathbf{x}>} & otherwise \end{cases} \tag{1.14}$$

Thus, the balance of angular momentum demands a force vector state field ($\underline{\mathbf{T}}$) that does not change with the direction of the deformation vector state field ($\underline{\mathbf{Y}}$). In other words, the forces of the bonds should be independent of the direction of the bonds, but may be dependent on the deformation and length of the bonds. A hypersphere horizon is the simplest horizon shape that can ease the computation of the force vector state field ($\underline{\mathbf{T}}$) and has central symmetry for satisfying the balance of linear momentum condition on the peridynamic equation of motion.

### 1.2.3 Damage

Silling and Askari [3] introduced scalar damage in PD, which ranges from 0 to 1, 0 denoting intact bonds, and 1 denoting that the bond is broken. The horizon damage is then the average of all its bonds. Thus a horizon with damage of one denotes that all the connections between the center of the neighborhood and its neighbors are broken, and the center is completely disconnected from other nodes in the simulation and can freely move inside the simulation box to compute the effectiveness of the bond force on the node force by multiplying the computed bond force to $(1 - bond_{damage})$. Foster *et al.*[20] examined different standard damage in classical mechanics and created a connection to peridynamic bond failure laws, including the formulation of work-based bond failure. Silling and Askari [3] introduced the so-called critical stretch criterion,

where the bond breaks only when it reaches a critical stretch. The critical stretch is defined as

$$s_c = \sqrt{\frac{5G_0}{9k\delta}}. \tag{1.15}$$

### 1.2.4 Neighborhood search

The neighborhood search is required to identify the neighbor material points of each point (i.e., the horizon center) inside each body that PD simulation should take place. The found neighbor material points form the horizon, where the equation of motion will be applied to identify the displacement of the center. Often a proximity search will take place to form a so-called neighbor list. The neighbor list might then be extended to include more neighbors than those composing the horizon, so the further neighborhood search at runtime can be eliminated. In practice, initiating the neighbor list is cumbersome though it seems straightforward in theory. For the sake of simplicity, let us assume only one horizon radius is applied to the entire simulation. The computation of the material point distance is not sufficient if a partial neighbor intersection is considered [8, 21, 22]. Seleson [21] and Seleson and Littlewood [22], suggested dividing the domain into a set of subvolumes centralized at material points similar to what is known as the meshing procedure in the standard finite element method. The proposed algorithm does not, however, use the subvolumes in computation, but rather as a pre-processing tool that will be used to determine which points should interact during processing. This will add more complexity to the neighborhood search since the volume shape of the subvolumes, and the horizon shape, should also be considered during the search. The result of the search should be stored and used throughout the simulation run time, which increases the computation cost even more. The treatment of a variable horizon in PD can be found in the [23] and [8]. Note that, if the Lagrangian approach is used [3], the neighbor lists may not undergo changes during simulation. This allows the creation and storing of the neighborhoods once during pre-processing.

### 1.2.5 Contacts

Contact modeling is an important feature for all of the numerical simulations. The short-range force approach models by Silling and Askari [3] allows simulating of multi-body contact interactions, fragmentation, penetration, and impact. The short-range force approach is recovered from the molecular dynamics techniques [3]. At each time step, a spring is assumed between the points that are close to each other. The spring applies pairwise repulsive forces with an inverse relation to the spring length. This approach alows disconnected bodies to become full contact or vice versa. Other approaches can be found in the literature [24], where a conventional (local) contact

algorithm is applied to the PD. The short-range force method has difficulties detecting the contact surfaces, since it represents surfaces as a collection of nodes. This becomes particularly important when a nonuniform hexahedral or tetrahedral mesh is used for discretization. Although some solutions can be found in the literature [25], the unphysical interpenetration of the contact models is often possible with extra attention.

Contact detection and enforcement algorithms are requisites of any contact model. The contact detection defines which nodes are in a range to become part of the contact face, which may or may not be deferent from the neighborhood search algorithms. The enforcement algorithm then applies the forces to the contact nodes.

The high cost of contact modeling usually governs the whole simulation computational expenses. Often computer programs limit the range of possible contact interactions by including specific nodes into the contact, which reduces the contact modeling computational cost. The other problem is that the contact model often reduces the maximum stable time step of explicit time integration schemes. Typically, the contact model results in artificial material stiffening in regions where contact is occurring. This can cause instability if the critical time step changes suddenly between two sequential time steps.

To date, PD contact modeling is a pen area of research. A nonlocal contact model is more likely to be successful since it follows the nature of the PD. The importance of a nonlocal contact model is better understood when comparing the boundary condition in PD with the contact. In principle, if we apply the boundary condition locally to PD simulation, the stiffness be raised artificially, which causes artificial wave reflection and hardening close to the BC. As explained above, the local short-range force contact method has the same problem. If a contact model can produce the force density over a volumetric region, we can apply a nonlocal contact model along with PD simulation. To the best knowledge of the author, this has not been addressed in the literature yet, which can be due to the lack of currently available computer programs allowing nonlocal contact modeling.

### 1.2.6 Time integration

Like any other numerical model, time integration drives the PD simulation. The time integration scheme computes the nodes displacement from computed nodal PD forces by the constitutive model. The displacement then will cause extra bond forces in the next time step. In principle, the PD time integration schemes are not any different from those available for classical local computational methods [26, 27]. Due to the dynamic nature, PD is often employed for transient simulations, which makes the explicit time integration schemes a suitable match. The explicit time steps are conditionally stable, which is resulting in a limited maximum time step duration. One can use the Courant-

Friedrichs-Lewy approach to determine the maximum critical time step duration.

$$\Delta t_{critical} = \frac{h}{c}, \qquad c = \sqrt{\frac{k}{\rho}} \tag{1.16}$$

where c denotes the wave speed in material, and h is the minimum length scale associated with the discretization. Equation 1.16 will give a very conservative critical time step duration if the h selected to be equal to the minimum node spacing. It is proven that the h can be set to the smallest horizon radius to improve the computation costs [28].

In some cases, the implicit time integration schemes may be employed to compute the preloading as a quasi-static process.

## 1.3 Organization of Dissertation

The key concerns on the peridynamic computational costs, literature review on PD refinement methods, currently available tools for simulating PD, and its implementation difficulties are reported in the above sections. A novel refinement method will be introduced in the next chapter; its application in a one-dimensional simulation will be examined. The difficulties rasing from implementing the new refinement method in a higher dimension will be explained in detail, and a solution will be proposed and be tested using numerical simulation of the crack propagation in a plate.

In chapter 3 a new architecture for a PD implementation will be introduced, which is capable of addressing some of the aforementioned implementation difficulties of the PD in this chapter, and those originating from the new refinement method. The new architecture allows the researchers to expand the PD beyond its initial formulation, a developed code called Relation-Based Simulator is developed by the author to demonstrate this fact. The proposed architecture's new features will be practiced by simulating three models in chapter 4. Finally, chapter 5 will summarize the works that have been presented in this dissertation and will include some recommendations for future works.

# Chapter 2

# Peridynamics Refinement

As explained in subsection 1.2.2 in the conventional PD, a fixed horizon size over the whole domain is often applied in order to eliminate unexpected behavior, which causes high computational cost. The refinement approaches presented in [29] allow for variable horizon sizes but introduce so-called ghost forces, which lead to artificial wave reflection between domains of different horizon sizes. Other contributions dealing with improving the computational efficiency of PD have been proposed, for instance, by Pasetto *et al.* [30] or by Lindsay *et al.* [31], and Ren *et al.* [7, 32]. However, to the best of the author's knowledge, none of the proposed refinement methods in the literature offers a native solution. A native refinement method would not require alternation of the PD constitutive model for implementation. For instance, the FEM refinement is native since it does not require the deriving of a new set of equations in the refined area. Refining the PD grid, while keeping the horizon radius constant, is considered native. Still, this approach is not applicable since the high cost of PD will grow exponentially by the power of the simulation dimension (see section 1.1).

In this chapter, a simple alternative idea of the native refinement approach while dealing with multiple horizon sizes is described. The implementation in 1D, its advantages, and disadvantages are reported. The difficulties arising from extending such refinement methods to 2D and 3D are discussed, and solutions provided.

## 2.1 The Multi-Horizon Peridynamics

As illustrated in Figure 2.1, in a condition where a subdomain at a position $X$ has a horizon radius of $R$ and its neighbor at $X'$ has a horizon radius of $r$, where $R > r$ and $r < |X - X'| < R$. The $X$ will include the $X'$ in its neighborhood but will not be included in the $X'$ neighborhood. Thus the $X$ will have a force toward $X'$ (i.e., ghost force) without $X'$ having any force toward $X$, causing an imbalance in the bond

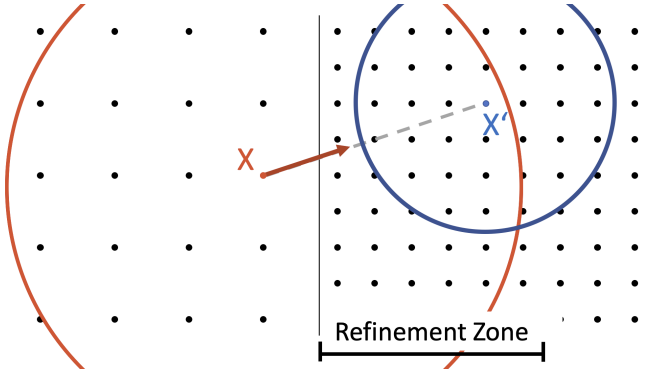



Figure 2.1: The appearance of the ghost force at the refinement bound.

between $X$ and $X'$.

The PD equations of motion theoretically introduce no limitation on the integration over its horizon [1] (see also subsection 1.2.2). The PD domain can be refined by utilizing the 'standard' formulation, which requires a fixed spherical horizon with a constant radius. The refinement cost will increase by the neighborhood size, which is exponentially growing by the refinement factor (the difference between refined and course subdomains dimension). Employing different horizon radiuses will lead to a defection in PD formulation, causing artificial results or even instability of the simulation, namely the formation of ghost forces.

For the sake of simplicity, let us assume refinement occurs along a straight line, as illustrated in Figures 2.2 and 2.3. The proposed approach suggest enforcing the symmetry of the interaction nodes having different horizon sizes by forcing all the nodes to include themselves to all of their neighbor nodes' neighborhood (horizon). For instance, $x'$ in Figure 2.2 will include all volumes of the subdomains in which the $x'$ is included in their neighborhood (e.g., $x$ in Figure 2.2). Thus, the nodes in the refined domain may have more than one horizon. Note that, the interaction between the domains is a one to one relation, thus multi-horizons of the refined nodes cannot coincide.

Let us call the part of the multi-horizon that has the same radius as those of refined nodes as the natural horizon ($H_n$), and the rest of the multi-horizon as the interaction horizon ($H_i$). Figure 2.2 illustrates the horizons of two nodes on the refinement zone, $x$, and $x'$, while Figure 2.3 presents the natural horizon and interaction horizon of the refined node. Outside of the refinement zone, multi-horizons do not affect the PD equation of motion which can be written for each point of domain positioned at $x$ and a node $x'$ inside its horizon $H$ in the global form of

$$\rho\ddot{\mathbf{u}} = \int_H \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} + \mathbf{b} \tag{2.1}$$

where $\rho$, $\mathbf{b}$, and $\mathbf{u}$ are the density, the body force, and the displacement of the center

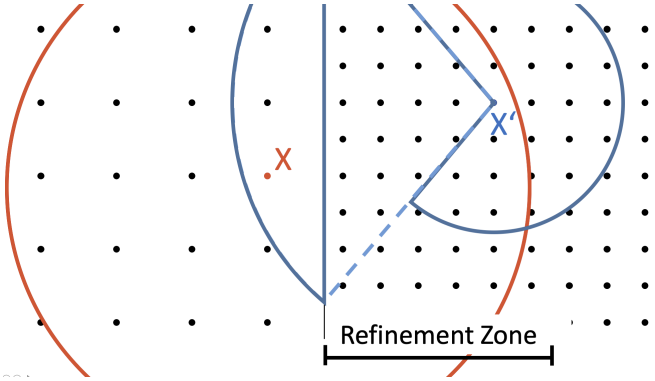



Figure 2.2: The proposed horizons for nodes inside the refinement zone.

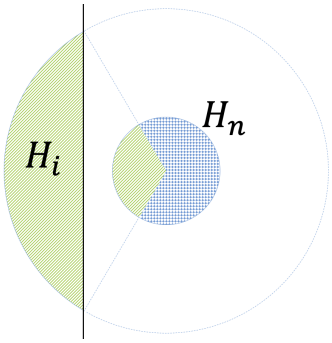



Figure 2.3: Multi-horizon of a node inside refined domain ($H_i$: interaction horizon, $H_n$: natural horizon).

of $x$'s horizon respectively. Note that equation 2.1 is valid for all PD-types, i.e. bond-based (BB-PD), ordinary state-based (OSB-PD), and non-ordinary state-based (NSB-PD) peridynamics. Let us consider the force vector state $\underline{\mathbf{T}}$. Its relation to the deformed direction vector state $\underline{\mathbf{M}}$ is given as

$$\underline{\mathbf{T}} = \underline{t}\,\underline{\mathbf{M}} \tag{2.2}$$

where $\underline{t}$ can be written as a scalar function over the domain for OSB-PD, and in the special case where $\underline{t} \equiv 1$ Eq. 2.2 presents the BB-PD, and any other form of function for $\underline{t}$ gives NSB-PD.

The Eq. 2.1 within the refinement zone can be written as

$$\rho\ddot{\mathbf{u}} = \int_{H_n} \{\underline{\mathbf{T}} - \underline{\mathbf{T}}'\}\, dV_{x'} + \int_{H_i} \left\{\underline{\mathbf{T}}_{\mathbf{i}} - \underline{\mathbf{T}}'_{\mathbf{i}}\right\} dV_{x'} + \mathbf{b}, \tag{2.3}$$

where $\underline{\mathbf{T}}_{\mathbf{i}}$s are the force vector states for the interaction horizon, and $\underline{\mathbf{T}}_{\mathbf{i}}$s must be defined in a way that Eq. 2.1 and Eq 2.3 have one to one equivalent integral functions

on their right-hand side. In other words, the multi-horizon formulation must provide the same acceleration, as if the whole domain was refined. This requires

$$\int_{H} \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} = \int_{H_n} \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} + \int_{H_i} \left\{\underline{\mathbf{T_i}} - \underline{\mathbf{T}'_\mathbf{i}}\right\} dV_{x'}$$

$$\int_{H-H_n} \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} = \int_{H_i} \left\{\underline{\mathbf{T_i}} - \underline{\mathbf{T}'_\mathbf{i}}\right\} dV_{x'} \tag{2.4}$$

where $H - H_n$ is the difference between the 'standard' horizon and the natural horizon, which is a subset of the interaction horizon since it still has no intersections with the natural horizon. Rewriting Eq. 2.2 for $\underline{\mathbf{T_i}}$ and utilizing the deformed direction vector state $\underline{\mathbf{M}}$, we obtain

$$\underline{\mathbf{T_i}} = \underline{t_i}\, \underline{\mathbf{M}},$$

thus,

$$\frac{\underline{\mathbf{T_i}}}{\underline{t_i}} = \frac{\underline{\mathbf{T}}}{\underline{t}}$$

$$\underline{\mathbf{T_i}} = \frac{\underline{t_i}}{\underline{t}}\, \underline{\mathbf{T}} = \underline{\alpha}\underline{\mathbf{T}}. \tag{2.5}$$

Substituting 2.5 into 2.4

$$\int_{H-H_n} \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} = \int_{H_i} \{\underline{\alpha}\underline{\mathbf{T}} - \underline{\alpha}\underline{\mathbf{T}'}\}\, dV_{x'} \tag{2.6}$$

Since $\underline{\alpha}$ is constant for both BB-PD and OSB-PD, we can rewrite Eq. 2.6 as following

$$\int_{H-H_n} \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} = \alpha \int_{H_i} \{\underline{\mathbf{T}} - \underline{\mathbf{T}'}\}\, dV_{x'} \tag{2.7}$$

where the only unknown parament is $\alpha$, that can be easily computed having the interaction horizon and its equivalent horizon in case of refining the whole domain $(H - H_n)$. Note that $\alpha$ remains time-independent and therefore, it needs to be computed only at the initial configuration. It is also worth mentioning that NSP-PD can also be implemented as a multi-horizon method if Eq. 2.6 is satisfied. If the refined node is located within a finite number of refinement zones, Eq. 2.7 can be utilized for each of the interaction horizons individually.

## 2.2 Absence of Ghost Forces

Ghost forces occur due to violation of Newton's law, which is the case for unsymmetric interactions of particles, which commonly occur for nodes with different horizon sizes.

This is also possible for un-symmetric or dissimilar shapes of the horizons. For models with only spherical horizon shapes for all sub-domains, only differences in the horizon radius between two sub-domains can cause ghost forces in the refinement zone. Multi-horizons guarantee the existence of the nodes inside the domain with larger horizon radiuses in the nodes of the refinement zone, which ensures the absence of ghost forces.

## 2.3 Numerical Examples

The computation of $\alpha$ can be cumbersome for complex refinement zones, especially in 2D and 3D as the intersections of the associated volumes of the neighboring nodes may demand complex geometry computations. For the sake of simplicity, we present a simple 1D example to demonstrate the performance of the multi-horizon approach. Consider a 1D bar of length 100 mm and two fixed ends applied to a velocity wave excitation at the midpoint of the bar, as illustrated in Figure 2.4. The two ends of the bar have a node distance of 0.1mm. The distance between nodes and the horizon radius at the midpoint of the bar is four-times bigger than the ends. The velocity wave has an exponential equation of

$$v = e^{-0.03(x-0.05)^2} m/s \tag{2.8}$$

To study the artificial wave reflection added by refinement zone, the response of the bar is recorded whenever the displacement of the mid node arrives at its peak. Figure 2.5 presents the first eight returned velocity waves. The artificial wave reflection due to the refinement zone is relatively small. As illustrated in Figure 2.6, the maximum error of $15\%$ is observed after the eight wave reflections.

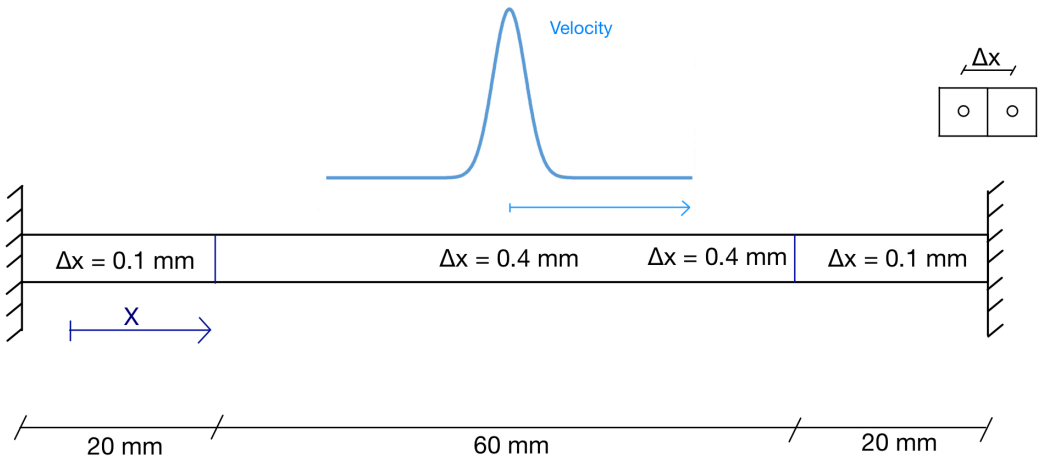



Figure 2.4: Initial configuration of numerical example

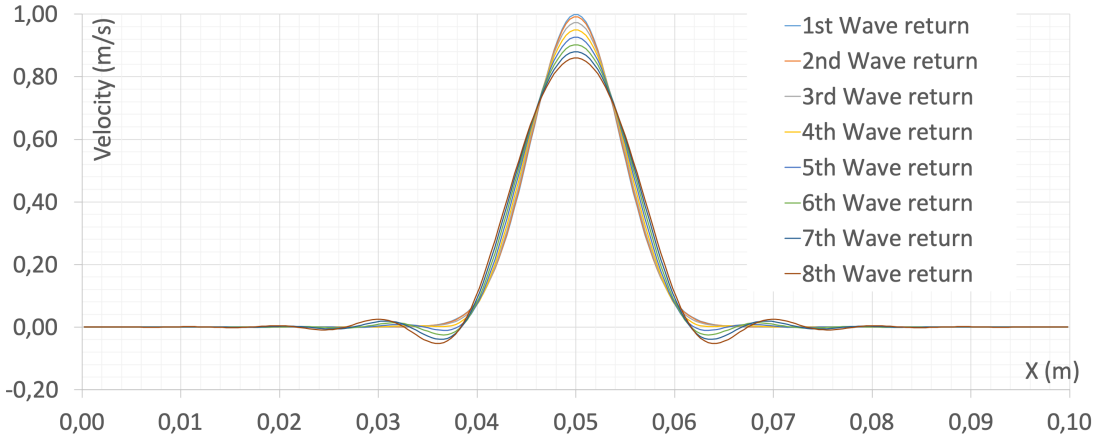



Figure 2.5: The first eight returned velocity waves for the bar in Figure 2.4

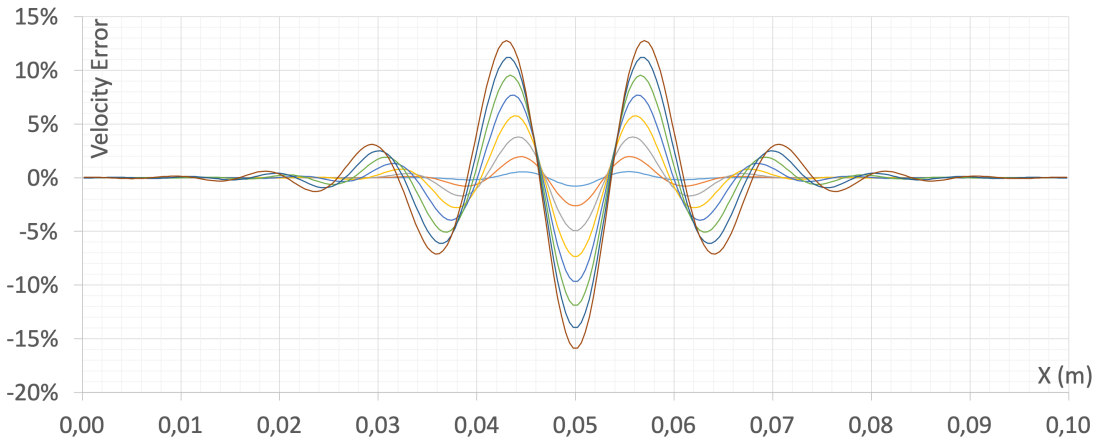



Figure 2.6: The error of velocity waves in Figure 2.5 compare to the non-refined bar (Colors are the same as Figure 2.5)

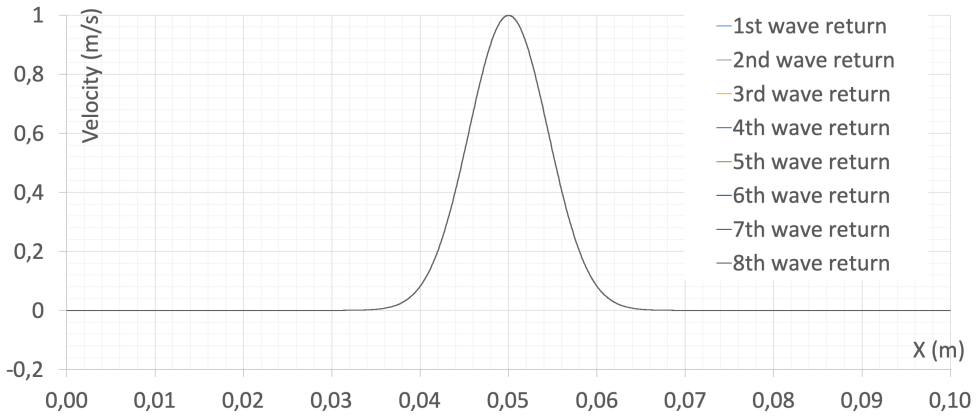



Figure 2.7: The first eight returned velocity waves for a bar similar to the bar Figure 2.4 with half of the node distance

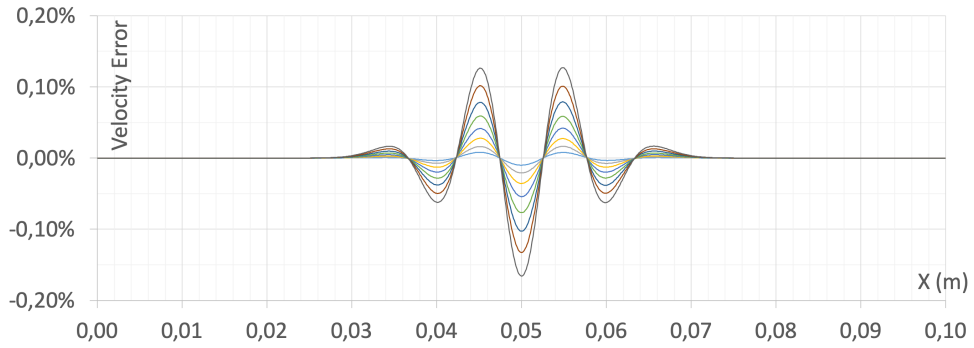



Figure 2.8: The velocity error of waves in Figure 2.7 compare to the non-refined bar (Colors are the same as Figure 2.7)

Let us consider now a 0.2mm node distance in the middle and 0.05mm node distance at the two ends of the bar. The velocity and its error after the eight wave reflections can be found in Figures 2.7 and 2.8, respectively.

Finally, we test a pulse excitation on the same bar (see Figure 2.4) where the node distances are 0.05 and 0.2mm at the end and middle, respectively. Figure 2.9 illustrates the first eight wave reflections. Although the error is about $40\%$, the simulation still remains stable. Note that this error is expected as PD is not well suited for capturing such sharp wave shapes.

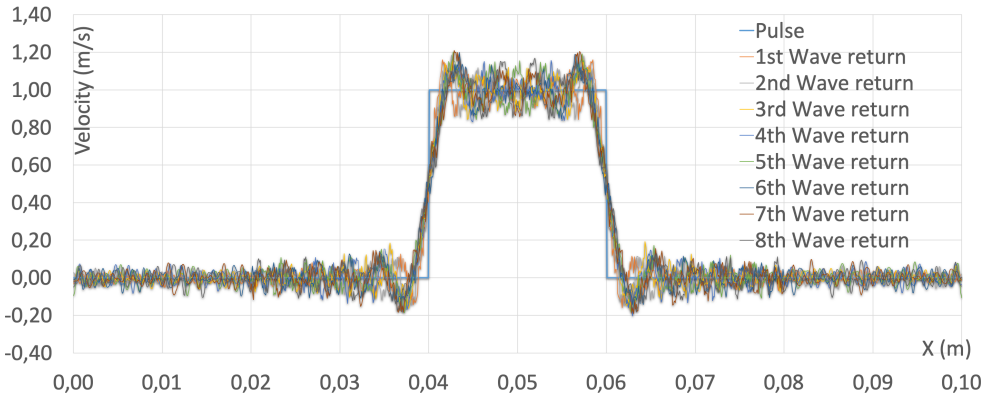



Figure 2.9: The first eight returned velocity waves for a bar similar to the bar Figure 2.4 with a pulse excitation

## 2.4 Computer Implementation

Implementing the concept of multi-horizons into an existing PD-code requires three changes:

- At the initial configuration, the nodes in the refinement zone have to be determined.
- The parameter $\alpha$(s) for each node located inside the refinement zone should be determined. Note that neighbor nodes may share their representative volume with more than one horizon.
- After computing the bond forces, the forces of the interaction horizons have to be multiplied with $\alpha$.

Implementing the first two changes usually requires the modification of the data structure in which the horizons and their neighbors are stored.

## 2.5 Mesh Sensitivity Affects on Refinement

Recently, several researchers have observed PD sensitivity to its meshing [19, 33, 34], Dipasquale *et al.*[34] reported the sensitivity of the fracture path to the meshing direction as an effect of the lower energy released by the crack path parallel to the meshing. Nevertheless, the nature of this sensitivity is not discussed in detail. The PD mesh sensitivity has frequently been reported as the effect of the ratio between horizon size to the subdomain sizes (known as $m$ ratio), which are the result of variational studies that have been done comparing the sensitivity to the $m$ ratio. The proposed relation,

however, can be questioned since the variational research does not promise any connection between two phenomena. Hence, there is a possibility that they are caused by another factor rather than each other. This section clarifies the understanding of horizon smoothness, which explains the PD mesh sensitivity and its connection to the refinement.

The general form of peridynamic constitutive model can be written as

$$\underline{\mathbf{T}} = \underline{\widehat{\mathbf{T}}}(\underline{\mathbf{Y}}, \Lambda)) \tag{2.9}$$

where $\underline{\widehat{\mathbf{T}}}$ is called peridynamic material and is a bounded, Reiman-integrable function on the horizon. $\Lambda$ is representing all other parameters where the current deformation vector state cannot describe but must be used to equalize the PD material with the continuum material behavior. PD Material is called simple if $\Lambda \equiv 0$. Since $\underline{\widehat{\mathbf{T}}}$ is a bijective function [2], the continuum domain point set and PD domain point set required to be equivalent. If PD material is unable to present even one point of it equivalent continuum model, the PD simulation does not present its equivalent continuum model. Therefore, the PD simulation domain cannot possess any cavitation or duplication of material points. Thus, regardless of the discretization method, the PD domain should exactly match the continuum problem geometry. A random discretization demands comprehensive overlapping and cavitation search over the PD domain.

The discretized form of the equation of motion (equation 1.8) is:

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \sum_{E} \int_{E} (\ \underline{\mathbf{T}}[\mathbf{x},t] < \mathbf{x'} - \mathbf{x} > - \ \underline{\mathbf{T}}[\mathbf{x'},t] < \mathbf{x} - \mathbf{x'} >)\, dV_{\mathbf{x'}} + b(\mathbf{x},t) \tag{2.10}$$

where $E$s are the discretized subdomains which sharing volume with the horizon. since $\underline{\mathbf{T}}$ linearly depends on $\underline{\mathbf{Y}}$ (equation 1.14) in BB-PD and OSB-PD, performing the integration over the subdomain gives:

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \sum_{E} (\ \underline{\mathbf{T}}[\mathbf{x},t] < \mathbf{x}_n - \mathbf{x} > - \ \underline{\mathbf{T}}[\mathbf{x}_n,t] < \mathbf{x} - \mathbf{x}_n >)\, \widehat{V}_{\mathbf{x}_n} + b(\mathbf{x},t) \tag{2.11}$$

where, $\mathbf{x}_n$ are the integration points of the subdomains, $V_{\mathbf{x}_n}$ is the volume of the subdomain that contains $\mathbf{x}_n$, and $\widehat{V}_{\mathbf{x}_n}$ is the shared volume of $V_{\mathbf{x}_n}$ with the horizon of $\mathbf{x}$.

The exact computation of $\widehat{V}_{\mathbf{x}_n}$ has a high-cost, especially in 3D. Therefore, using a weighting function for computing the $\widehat{V}_{\mathbf{x}_n}$ from $_{\mathbf{x}_n}$ is practically acceptable for PD implementation. The instance of weighting function at the $\mathbf{x}_n$ called volume correction and noted by $C_V$. Re-writing equation 2.11 using volume correction concept we have

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \sum_{E} (\ \underline{\mathbf{T}}[\mathbf{x},t] < \mathbf{x}_n - \mathbf{x} > - \ \underline{\mathbf{T}}[\mathbf{x}_n,t] < \mathbf{x} - \mathbf{x}_n >)\, C_V V_{\mathbf{x}_n} + b(\mathbf{x},t). \tag{2.12}$$

Similar to the continuous form of the peridynamic equation of motion, the satisfaction of the balance of angular momentum leads to a set of $x_n$s with centroid symmetry. Thus an ununiform discretization (either soft change of subdomain size in one dimension or sharp change as refinement) cannot satisfy the balance of angular momentum and needs extra care [7]. Due to the limitation mentioned above, the numerical and theoretical horizon differ in shape (Figure 2.10a). Similarly, distribution of the volume correction differs between numerical and theoretical horizon, compare Figures 2.10b and 2.10c.

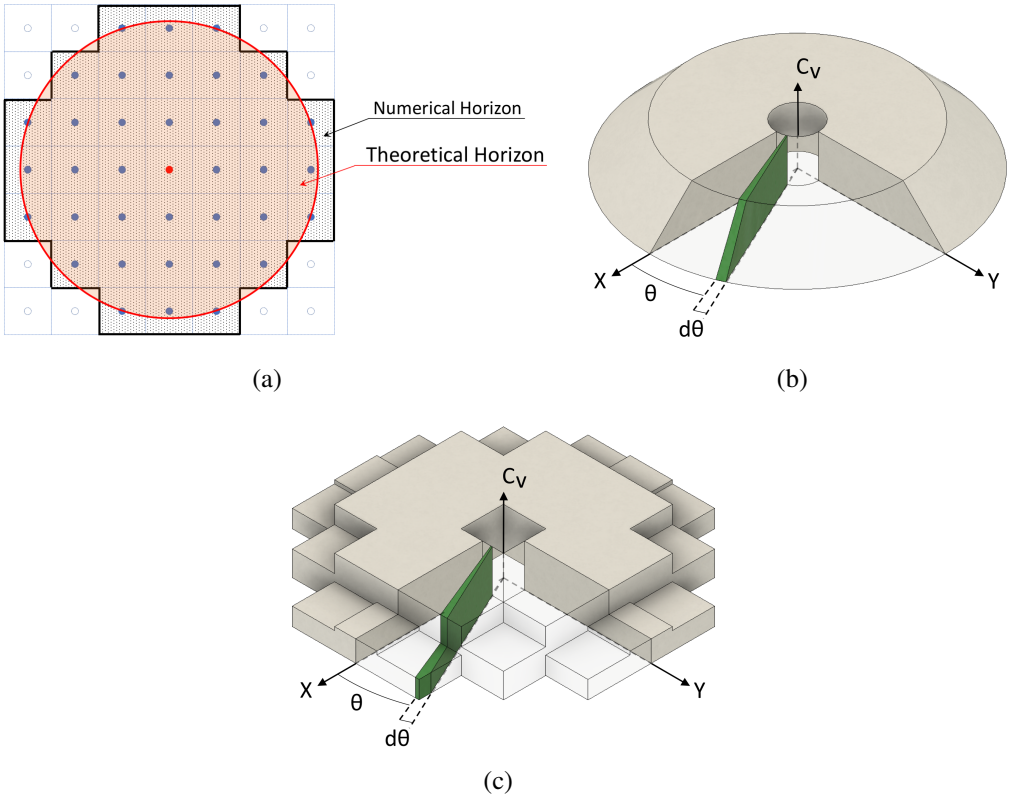



Figure 2.10: (a) Numerical and theoretical horizon shape differences on 2D domain (b) Theoretical horizon volume correction (c) Numerical volume correction

Thus the boundary of the numerical horizon is a pixelized version of its theoretical counterpart (spherical horizon) generated by the background of the uniform mesh of the domain. Since $\|\underline{\mathbf{T}}\|$ is not dependent on the direction of the bond, the only factor that can create a dependency of force value on the right-hand side of equation 2.12 to the direction of the bond is volume correction ($C_V$).

The bond forces highly depend on the boundary condition of the domain, but the resistance of the horizon to deformation (the accumulated bond stiffness of the hori-

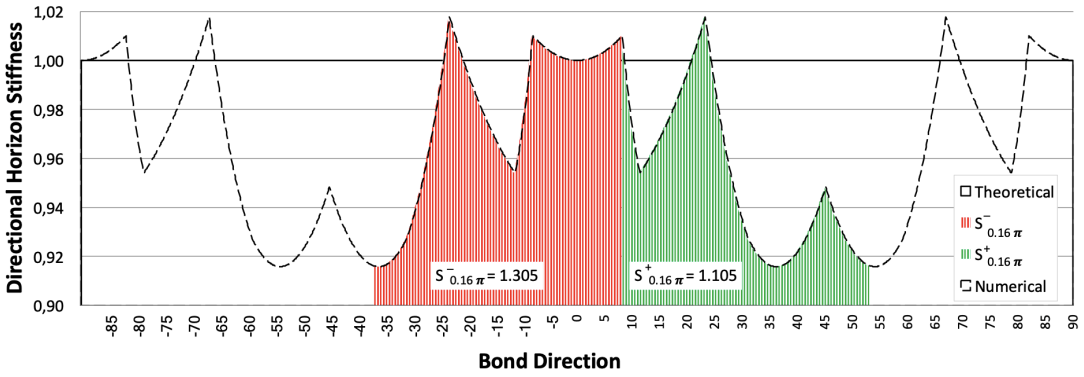



Figure 2.11: Change of the horizon stiffness respect to direction.

zon) only depends on the constitutive model and is more efficient for identifying the numerical horizon's smoothness effects. The illustrated volume correction functions in Figures 2.10b and 2.10c are used for determining the horizon stiffnesses dependency to direction. A very fine mesh ($\Delta\theta = 0.001°$) applied to both numerical and theoretical horizon. The right-hand side of equations 2.12 and 2.10 were integrated and divided to the deformation of the slice caused by a uniform constant deformation in all directions. The result of the computation is a dimensionless parameter called "Directional Horizon Stiffness" which depends only on the meshing direction. This parameter is illustrated in Figure 2.11, and it can efficiently present the bond force dependency of numerical simulations to the mesh direction. The area below the "Directional Horizon Stiffness" diagram (Figure 2.11) is the density of the stiffness of the horizon and it can be calculated between any two angles. For instance, the area between any two random bond angles, $\alpha$ and $\beta$ is noted by $S_\alpha^\beta$ which represents the stiffness of the horizon between $\alpha$ and $\beta$. If $\beta = \frac{\pi}{4}$ the density of horizon's stiffness is called right-side horizon stiffness and noted as $S_\alpha^+$. Similarly left side horizon stiffness is indicated as $S_\alpha^-$.

The nature of the PD allows only force waves to propagate through the domain. Other imposed forms of waves on boundary domains (e.g., velocity wave) will be integrated by time integration scheme to find the displacements, and then the forces will be defined by the PD, which will propagate through the domain. (see equation 1.8). The dynamic force wave then travels through the horizons by producing bond forces and strains. If a wave force inters a horizon at a specific angle of the $\beta$, it will experience different horizon stiffness on the left and right side if and only if the $S_\beta^+ \neq S_\beta^-$. This condition is valid for any angle on the theoretical horizon since its stiffness is not dependent on the direction of the meshing, but it is only true for $\theta = \mp k\frac{\pi}{4}$ on the numerical horizon where $k$ is an integer. For instance, in Figure 2.11, the $S^+_{\arctan(0.5/3.5)=0.142\pi}$ and $S^-_{\arctan(0.5/3.5)=0.142\pi}$ are calculated to be $1.105$ and $1.305$ respectively. If a wave enters the horizon with the angle of $0.142\pi$, it will receive less stiffness in its right

side and will distribute faster to the left side. Then it will enter to a stiffer volume of the $S^{-}_{0.642\pi}$ and softer volume of $S^{+}_{-0.358\pi}$ and will return to the same angle of $0.142\pi$. Similarly, in the other half of the horizon, the wave will distribute to the right and then left but finally continues its pass with the same angle at which it entered the horizon. Therefore, mesh sensitivity of the numerical horizon will not affect the numerical result since we are not interested in the bonds' forces but rather their accumulation, which governs the displacement of the nodes. However, the difference between the left and right directional horizon stiffnesses becomes essential if the numerical horizon does not have symmetry around the perpendicular face to the force wave. Such scenarios can be found in any sharp change of material property, either as the interaction of two domains (with or without refinement) or fracture (mostly at the tip of the crack path).

It is worth mentioning that the angles of the bonds are also discontinuous. The distribution of the bonds effectiveness and their contribution to the angular horizon stiffness illustrated in Figure 2.12 for $m = 3$ and due to the symmetry of the horizon only for span between 0 to +45 degrees.

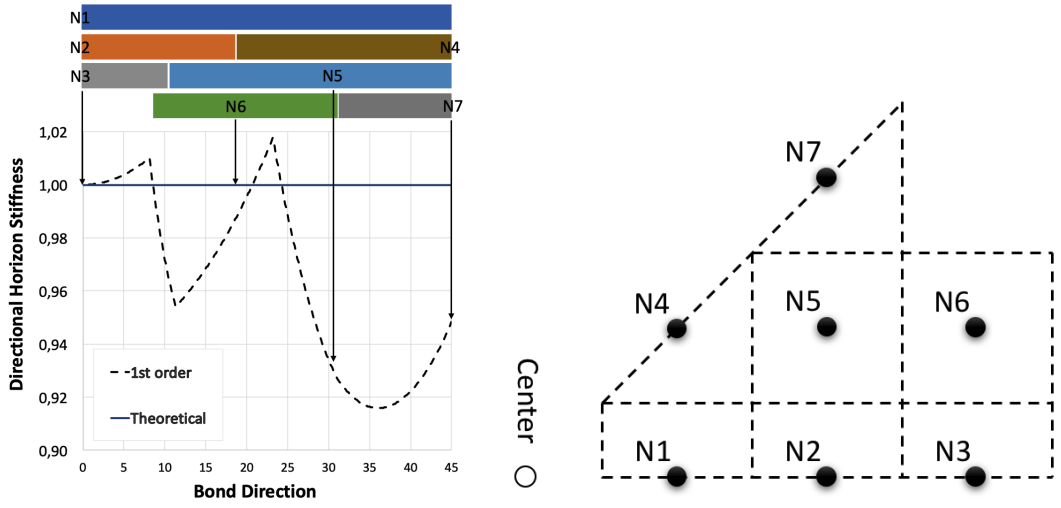
## 2.6 Smoothed Horizon

Since the pixelization of the horizon leads to mesh sensitivity along the domain interaction lines and crack paths, reducing the difference between the numerical and theoretical horizon should reduce the PD mesh sensitivity. This can be achieved by different approaches, namely, horizon enlargement, horizon refining, and adaptive horizon refining.

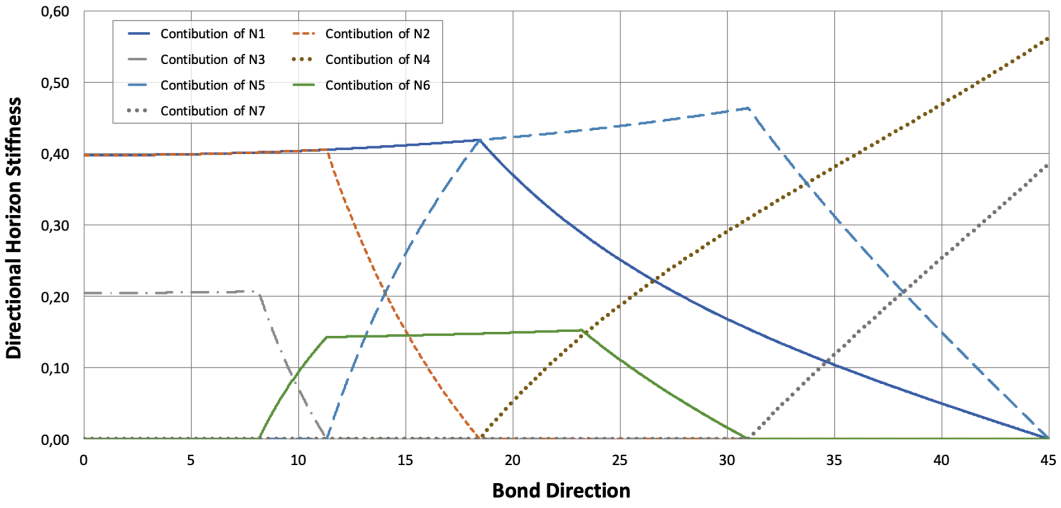
### 2.6.1 Horizon Enlargement

Increasing the ratio of the horizon radius to the subdomains edge length ($m$) will reduce the pixelized effect of the meshing. Figure 2.13 illustrates the horizon stiffness of different $m$ ratios. The horizon stiffness is less sensitive to a finer pixelized horizon. Although this seems to be the most effective way to reduce the numerical horizon effects on PD, a very high computation cost can be expected. The complexity of the bond force computation is $O(n(2m)^D)$, where $n$ is the number of PD nodes, and $D$ is the problem dimension. Increasing the $m$ by a factor of two gives two times higher bond force computation cost for 1D problems, four times higher for 2D problems, and eight times higher for 3D problems in each time step. Considering the minimum of the same amount, higher storage cost, this option is not as efficient as it might seem at first glance. However, it benefits from a very comfortable implementation. Figure 2.13 illustrates the effect of the horizon enlargement to the horizon stiffness. The comparison and effectiveness of horizon enlargement are reported by Dereso et.al. [34].

## 2.6 Smoothed Horizon




(a) (b)




(c)

Figure 2.12: (a) The bonds direction of the nodes and the effective direction of each node. (b) The positioning of the nodes for $m = 3$ from $0$ to $\frac{\pi}{4}$ (c) The contribution of each node in the horizon stiffness.

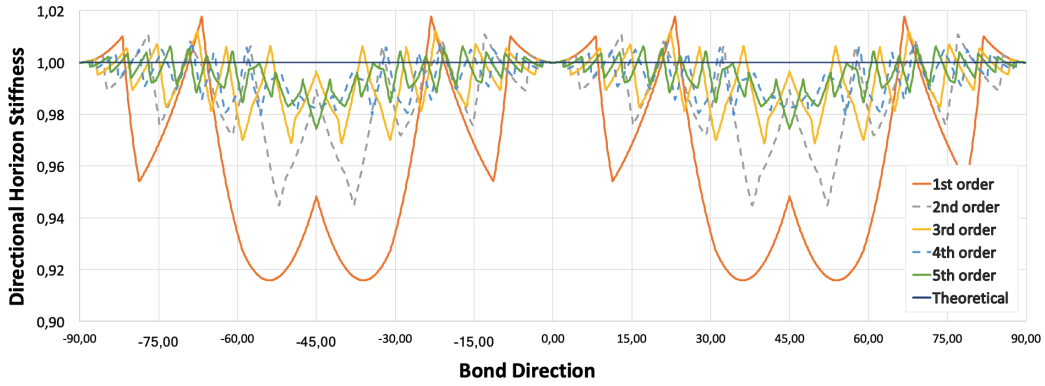



Figure 2.13: Horizon Stiffness change with bond direction for various ratio of the horizon radius to the subdomains edge length ($m$)

### 2.6.2 Horizon Refining

The volume correction of the border neighborhoods ( $C_V \neq 1$ ) is the bases of the horizon stiffness dependency on mesh direction. The refinement of the volume at the border neighborhoods creates a softer change at the horizon stiffness, which leads to smoother horizon stiffness. The suggested refinement occurs only at the horizon level and does not require h-refinement in subdomain meshing. Figures 2.14a and 2.14b illustrate refined horizons of order 2 and 3 in 2D, respectively. The added bonds will take place in force computation rather than the bond itself. The original bond force then is the sum of the added bonds. The same analogy shall be used in damage, the total damage of the bonds is equal to the damage of the original bond. The deformation of the added bonds will be calculated by the displacement interpolation of the relatively added refined-subdomain from the original node displacement.

The constitutive model will not change by horizon refinement, and only the volume correction will be affected. The horizon refinement highly smoothes the stiffness of the numerical horizon, as it can be seen in Figures 2.14c and 2.14d.The comparison between Figure 2.13 and Figure 2.15, shows more effective smoothness caused by horizon refinement than enlargement. Although the horizon refinement reduces the stiffness of the horizon, $\frac{1}{4}\pi$, horizon refinement promises lower costs than enlarging the horizon size. Considering the complexity of $O(n(b_i+b_r d_r))$ where $b_i$ is the number of inner bonds, $b_r$ represents the number of refined bonds at the border of the horizon, and the $d_r$ is the refinement order. The second-order refinement provides us with at least the $\frac{2}{3}$ cost of horizon size enlargement and third-order refinement almost half of the cost in 3D problems.

## 2.6 Smoothed Horizon

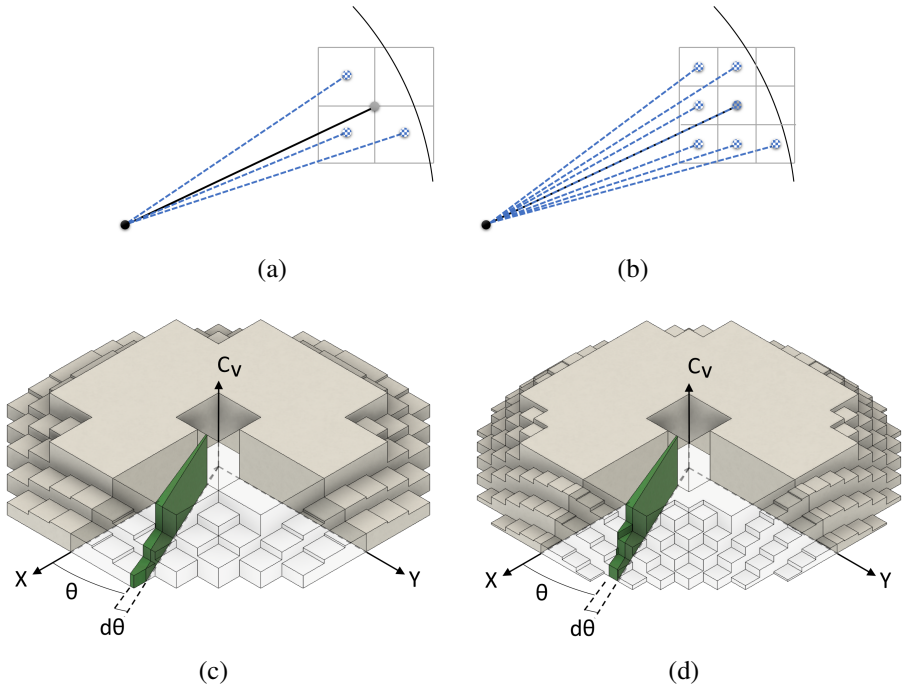



Figure 2.14: 2D schematic illustration of the (a) second and (b) third order horizon refinement bond and the volume correction distribution over the refined horizon of order two (c) and three (d).

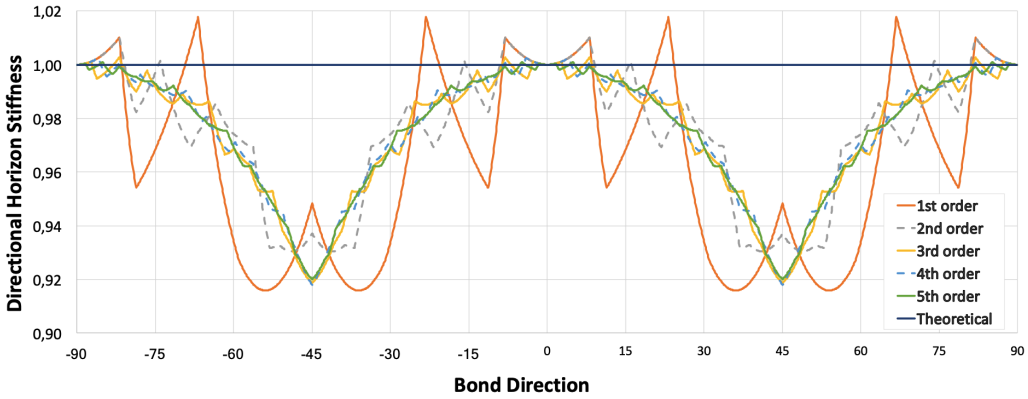
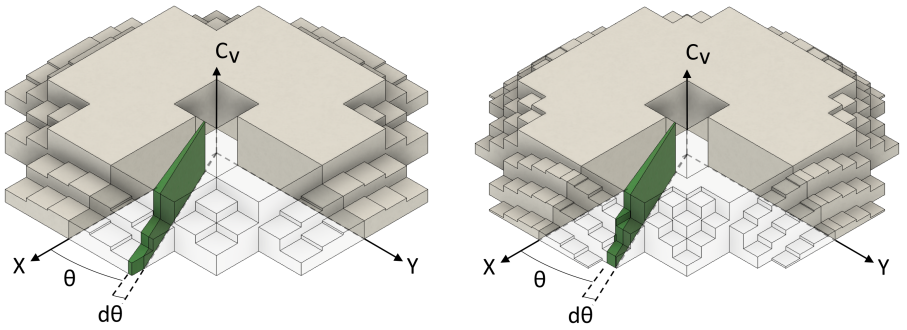



Figure 2.15: Horizon Stiffness change with bond direction for various horizon refinement orders




(a) (b)

Figure 2.16: The volume correction distribution for (a) second and (b) third order adaptive horizon refinement without neighborhood search.

### 2.6.3 Adaptive Horizon Refining

Horizon refinement is relatively faster than horizon size enlargement, but still requires high cost on the PD. Alternatively, the horizon refinement can be adaptively used whenever the horizon overlaps with two or more domains or experiences any damage. Adaptive horizon refinement does not require any remeshing, nor any change in the formulation of the PD, therefore, it enjoys easier implementation, and lower cost in comparison to any other mentioned smoothness method above. Some of the refined subdomains may not be found in the initial neighborhood search. There are different approaches to address this problem. First, by initiating neighborhood search around the newly added nodes to the refinement area at the beginning of each time step. This option has low memory cost, but it comes with high computation cost since the neighborhood search is one of the most expensive processes of the post-processing of PD. Second, reducing the cost of neighborhood search in each time step by storing all the found neighbors in a cubic neighborhood, and applying $C_V = 0$ to their bonds at the initial neighborhood search. The moderate neighborhood search can be expected since the newly added nodes cannot have a new neighbor outside their cubic neighborhood. Third, neglecting the refined subdomains outside of the current horizon. Neglecting the neighborhood search changes the shape and distribution of the volume correction over the horizon. A comparison between Figures 2.16a and 2.16b with their similar order in Figures 2.14c and 2.14d shows that the higher the order of horizon refinement, the larger the difference between the distribution of the volume correction over the horizon with and without neighborhood search. However, the purpose of horizon smoothing is not to find the closest distribution to the theoretical horizon rather smoothing the horizon in a way that the left and right side stiffness stays similar in all of the angles. Figure 2.17 illustrates the horizon stiffness change with the bond direction for the adaptive refinement with no neighborhood search. The minimal difference

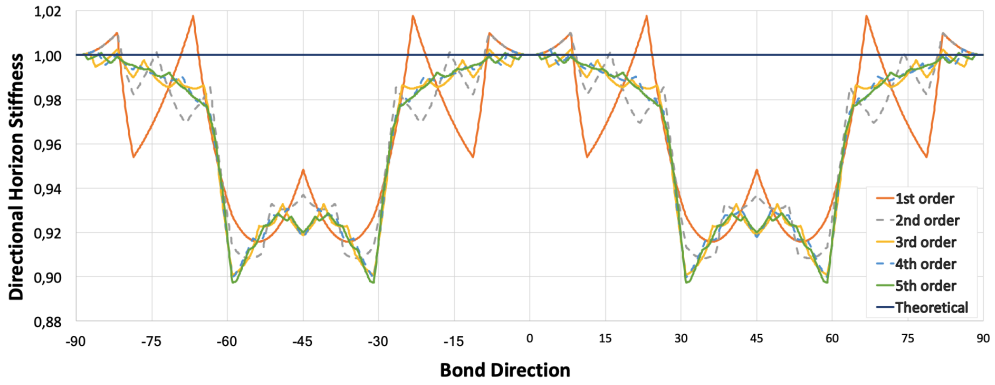



Figure 2.17: Horizon Stiffness change with bond direction for various horizon refinement orders without neighborhood search.

of second horizon refinement shows that the second-order adaptive horizon refinement without neighborhood search can safely be used.

Table 2.1: The horizon smoothness simulation parameters

| | Young's modulus | Poisson's ratio | mass density |
|---|---|---|---|
| BB-PD | $E = 73.4GPa$ | $\nu = 0.33$ | $\rho = 2440\frac{kg}{m^3}$ |

## 2.7 Reduced Sensitivity of Smoothed Horizon

The mesh sensitivity only occurs when we find a line passing through a horizon center, which can introduce different horizon stiffnesses in each half of the horizon. (see section 2.5). This situation will occur in the interaction faces of domains with different PD material, or at the tip of the crack since damage material stiffness creates a non-symmetrical horizon stiffness. Since the fracture simulations have only one related parameter (e.g., crack direction) to horizon smoothness, they are preferred to wave propagation simulation in a refinement bound between two domains. Although the fracturing process is more straightforward in PD, fast-wave propagation can cause a complex fracturing scheme. Thus inquiring the PD mesh sensitivity is more convenient on a fracture problem with slow-wave propagation. A PD code developed to present the effect of the smoothed horizon using second order adaptive horizon refining, Figure 2.18, illustrates the geometry of a plate with an initial crack and the geometry of the boundary domains (see chapter 3) and the direction of applied forces on the boundary domain nodes. As expressed by Dipasquale *et al.*[34], the choice of PD (i.e., BB-PD and OSB-PD) has no effect on the crack path direction. Thus for the sake of the comparability, the same parameters (see table 2.1) are selected for the BB-PD simulation.

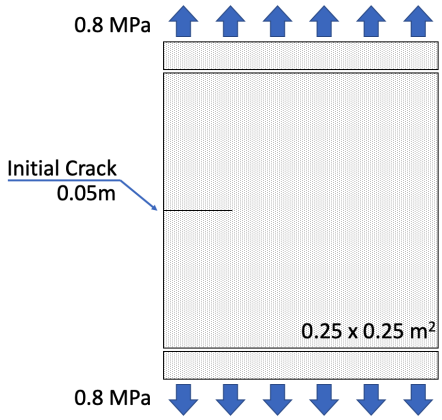



Figure 2.18: The schematic illustration of the initial configuration for the horizon smoothness simulation

The load has an intensity of $0.8$ MPa where applied to the nodes of the upper and lower boundary domain nodes as illustrated in Figure 2.18. Figure 2.19 illustrates the result of a simulation with a meshing angle of 10 degrees to the initial crack direction, where it was reported to have the highest effect on the crack propagation. Smoothing the horizon by utilizing the second-order adaptive horizon refining reduces the mesh sensitivity of the PD to the mesh direction. The same should be expected applying the horizon smoothness to the refinement bound between a course and fine domain.

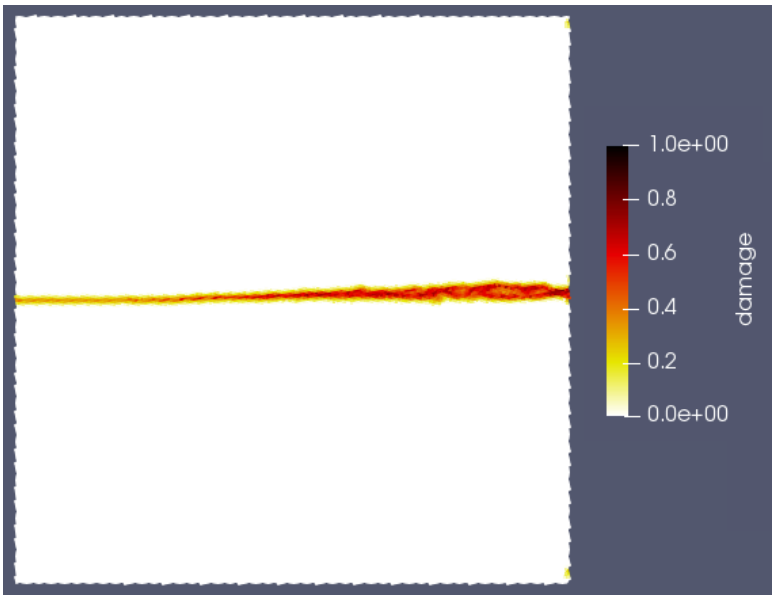



Figure 2.19: Second order adaptive smoothed BB-PD fracture.

# Chapter 3

# Software Architecture and Peridynamic Computer Implementation

The first peridynamic computer implementation was called EMU code. It was developed by Sandia National Laboratories using Fortran 90 [35]. The subsequently released codes by Sandia National Laboratories capable of simulating peridynamic models are SIERRA/SM [36], LAMMPS [15], and Peridigm [14]. The EMU and SIERRA/SM are not publicly accessible. Thus, their implementation details are not available to discuss. The LAMMPS is a dynamic molecular simulator that was initially developed using Fortran77, which was later updated to Fortran 90 and currently available in C++. Although the LAMMPS is capable of simulating different meshfree methods like smooth particle hydrodynamics, [37] and Peridyanmic [15], due to the complexity of its architecture, modifying its source code for applying new methods requires a deep understanding of the provided library. In other words, the LAMMPS is a very useful software for simulating meshfree methods, but further development of its source code is a cumbersome task and needs specialized knowledge. The Peridigm is a C++ code explicitly developed to simulate peridynamic multi-physics problems. It is capable of simulating large-scale parallel domains with a contact in both BB-PD and SB-PD. It is also able to use implicit and explicit time integrations. The Peridigm supports pre- and post-processing tools like Paraview visualization tools. Like LAMMPS, the Peridigm is a monolith that limits its user from the further extension of the code on specific topics. Figure 3.1 illustrates the Peridigm architecture and extendable areas reported by the Peridigm developer team. Thus developing out-of-box constitutive modes, contact face search, neighborhood search, etc., are not achievable in a short time [38]. For instance, the implementation of multi-horizon refinement (see chapter 2) requires the researcher to clone the code and modify the data structure.

Other implementation of PD using commercial software can be found in the litera-

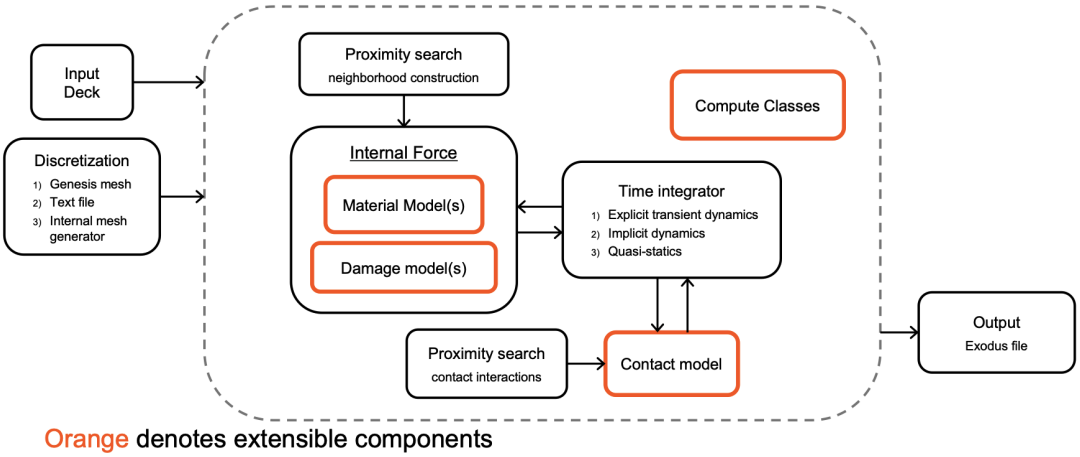



Figure 3.1: The Peridigm architecture and its extendable areas reported by the Peridigm developer team.

ture [9–12]. Due to two main problems associated with such implementation, they are not recommended. First, the limited interaction with the solver prevents the researcher from modifying the solver. Second, the solver's hidden nature can generate unexpected, untestable, artificial results which the user cannot prevent or, in some cases, even recognize.

In this chapter, a software architecture for node-based methods (e.g., peridynamic) is proposed. The proposed architecture aims to reduce the cost of method implementation for computational scientists while providing them with a robust and secure system to use. It also allows collaboration between computational engineering researchers by following the characters of microkernel and agile architectures. Thanks to microkernel initials, researchers can enjoy the code's high performance while securely implementing PD's advancement. The security provided by microkernel architecture allows the researchers of other fields (e.g., mechanical engineers) to utilize the provided code by computational engineering researchers or combine those methods to fit their needs. An implementation of the proposed architecture called Relation-Based Simulator (RBS) is developed incrementally by following the agile development approach during the last three years and can be found on the GitHub website [1]. Note that the proposed architecture is not limited to the PD, and one should be able to implement any other numerical simulation following the provided material in this chapter.

[1] https://github.com/alijenabi/RelationBasedSoftware

## 3.1 Modern Programming Paradigms

Programming paradigms are a set of concepts that help programmers achieve their goals with less effort to maintain, develop, and modify their code at a lower cost. The paradigms are not mandatory to use, and some of the languages are only built around one paradigm. For instance, the Prolog language is built around logic programming paradigms and Haskell around functional programming paradigms.

In this section, the programming paradigms that the proposed architecture uses are explained. The advantage of such paradigms for developing a node base program is described. The practical applications of such advantages are explained.

## 3.2 Procedural Programming

A computer code that is written as a long series of operations to execute is called procedural. Procedural programming is often the choice of computational researchers for presenting the proposed method's implementation because their audiences can simply follow the series of operation execution to understand how to implement the new method [13]. In procedural programming, code can be organized into named functions or sub-routines to make the code modular and maintainable. Although it is easy to create simple programs in terms of sequential steps, those programs suffer from a lack of readability, often have code multiplication leading to high maintenance cost, and most importantly, the code tends to become very complex while growing in size.

### 3.2.1 Object Oriented Programming

Object-Oriented Programming (OOP) splits code into several self-contained objects. Each object contains its logic, data, and behavior related to other objects. A procedural approach and an object-oriented approach are the same although they organize the code in a different matter. Code written in an object-oriented manner is reusable; it is easier to maintain, modify, and develop in an agile manner. Object-Oriented Programming reduces code complexity by introducing four new features: abstraction, polymorphism, inheritance, and encapsulation. In terms of developing a code able to simulate any node-based computational method (e.g., PD), abstraction allows the developer to create an abstract of their object and instantiate as many as they need. Data security is the most significant advantage of encapsulation in OOP, where we can limit access to the information (e.g., initial node position) and ensure that it will not be altered unintentionally during the runtime of the program. Inheritance allows us to develop a series of base classes (also known as parent classes) that are necessary for the code to run, and then another researcher will be able to create subclasses (also known as child classes) of those base classes and add more functionality or even override the base

class behavior on the new class. For instance, in our proposed architecture (Relation-Based Architecture, RBA) a base class called Node manages the storage of vector and constants related to that Node; the PD Node is a subclass of the Node class that only provides functionality related to peridynamic (see subsection 3.4.1). Polymorphism allows us to introduce interfaces where we define only the possible properties of each object, but the object that inherits the interface is responsible for implementing the procedure itself. For instance, Relations in RBA have all the same functionality. Thus, if a researcher is required to introduce a new constitutive model to the program, they can create their own class and inherit the relation interface. In this way, the researcher can benefit from the facilities provided by RBA without knowing how they work in detail. Polymorphism also allows for the rewriting the base class properties. For instance, when developing a new peridynamic refinement method, one can inherit the Peridynamic class and override the required functions.

### 3.2.2 Functional programming

Functional programming is a paradigm for minimizing testing effort by enforcing the program states to be a set of stand-alone functions. Thus if all the functions are working as expected, the program will run flawlessly. Functional programming minimizes data mutations, replaces loops with recursion, and prefers expressions over statements, so that the functions are pure and side-effect free. Following the functional programming paradigm, the outputs of each function depend only on its inputs, and they are not allowed to change an out-of-scope state of the program. Although many languages are not built to support this paradigm, one can implement a set of guidelines and rules to enjoy the benefits of functional programming in any language. Initially, C++ did not support any functional programming feature, but the C++11 standard library provides the means to employ both functional and object-oriented programming. RBA uses functional programming in several spots where a default behavior is defined, but additional change is assumed to be required by future works. For instance, the neighborhood search parameters are defined to support a hypersphere neighborhood shape, but it is possible for the researcher to create a custom function for employing new behavior for the neighborhood search. The theoretical points are explained in subsection 3.4.2.

### 3.2.3 Microkernel (Plugin) Architecture

The microkernel architecture is initially employed on operating systems (OS) by placing the system's essential capabilities (e.g., memory and the file system) to a single stand-alone executable called the kernel. Other stand-alone execution units can be plugged into the kernel to provide extra functionality. These small executables are

often called plugins. The ”plugging in” in today's operating system world relates to allowing the new plugin to call the kernel functions and members directly, which creates an efficient communication path between the plugin and the kernel. In the older version of Unix, the OS had to be recompiled in order to add or remove a plugin from it. The most complicated and challenging task in microkernels architecture is the messaging; in early OSs, a special part of memory was allocated for the plugin's messaging. Even pointers of the kernel functions were available for plugins to call. This approach was not successful since a single plugin could cause insecurity in the system. The implementation of such a complex architecture for RBA is burdensome and overkill. However, RBA plugins like implementation of relations offers the following benefits when employing microkernel architecture:

- The plugins are independent; thus, when two researchers are developing different parts of the code or different methods simultaneously, it is impossible to affect each others' work.

- The isolation between plugins makes it impossible for one plugin to make direct calls to methods within another one. Thus, researchers are limited and must stay in the framework in which they are developing. This leads to a more testable and reliable system. It also ensures that individual mistakes will not affect the core functionality of the system.

- The plugins are relatively small in size. Thus maintenance, debugging, and modification of them will be inexpensive.

It worth mentioning that employing the microkernel architecture also has a downside. If the kernel needs to be changed, all the plugins must be adopted to the new kernel. Moreover, the flaws in the kernel can cause a system failure. Since the RBA's kernel (named Analyses) is not as complicated as an OS might be, this downside is not relevant.

### 3.2.4 RBS Architecture

The main theme of the RBA is OOP, as the nodes, neighborhoods (i.e., horizons), mathematical vectors, points, and many more concepts naturally occur within the OOP concept of Object. Functional programming is often used to minimize the researchers' effort when introducing or replacing a mathematical function in the simulation process in the RBA. For instance, the bond force-stretch function of bond-based peridynamic can be altered by writing a suitable lambda and passing it to the bond-based peridynamic relation constructor. The microkernel architecture (i.e., plugging architecture) concepts as explained above are utilized for structuring Relations and their dependencies. For instance, the time-integration schemes are specific to the methods

(e.g., *PDVelocityVerletAlgorithm*) and not shareable across the constitutive models. The simplicity of following simulation steps by procedural programming is adopted by RBA when implementing simulations. Along with a familiar naming system to for the computational concepts namespaces, objects, and functions, The RBA values the researcher's aim to provide a simple presentation of their work to their audience. Thus, a researcher with limited or no prior C++ knowledge can understand the steps taken on the program's main function. See the following code, which defines a *BrittleFracture* only on a part of the model's body (i.e., *platePart*).

```
const auto shouldFracture =
[plateDimension](const Part::NeighborhoodPtr& horizon, const Part::NeighborhoodPtr&)
    {
  // Get the position vector of horizon's center.
  const auto& posVec = horizon->centre()->initialPosition().value<space::Point<3> >()
      .positionVector();
  // Check if horizon's center is located inside fracturable area.
  return - plateDimension / 4 < posVec[1] && posVec[1] < plateDimension / 4;
};

relations::peridynamic::BondBased::BrittleFracture(
  criticalStretch, materialConstant, gridSpacing, horizonRadius, platePart, true /*
      override forces */, shouldFracture);
```

## 3.3 Agile Development For Scientific Purposes

Agile development is mostly addressed for implementing online businesses as stated in the the agile manifesto [1], "Business people and developers work together daily throughout the project." Agile development is based on a cyclical process known as the inspect and adapt loop, where the developer modifies some part of the code and releases it to get the end-user feedback. This incremental thinking set is not far from computational engineering science, with a small difference: the researchers are seeking other researchers' interest in their work and the applicability of their proposed method rather than the end-user feedback. The RBS architecture follows the agile development principle, so that other researchers' incremental work can improve upon the implemented code.

[1] https://agilemanifesto.org

## 3.4 Data Structure

In this section, the proposed architecture and its implementation are explained in detail. The specifics of peridynamic on each topic are first explained, then the RBS architecture (i.e., RBA) approach and its implementation are presented. Note that the same procedure can be done to introduce any other node-based method to RBS.

### 3.4.1 Co-location Approach and Nodes

The discretization of the space and time introduced by Silling and Askari [3] divides the PD problem's physical system into a finite number of subdomains in which the co-location approach is employed. The co-location approach is relevant to the one point Gauss quadrature scheme where all the physical parameters are constant over the subdomain. The co-location approach requires the subdomains to cover the entirety of the domain, otherwise, the non-covered area of the domain will be handled as material cavitation by PD. Thus a random generation of the subdomains similar to most mesh-free methods cannot be accepted and requires extra attention to fulfill the co-location approach condition.

The implementation of any node-based method approach requires the nodes to carry the relative subdomain information such as position, displacement, velocity, dilatation, etc. The co-location approach does not limit the storage of data at Node in any respect. RBA suggests using a global Node that may or may not own its parameter by itself. For instance, it is recommended that the node's initial position be owned by other quantities that can secure the covering of the entirety of the co-location approach's domain condition. Still, the force is recommended to be stored inside the Node to ensure that the node forces remain independent. The recommended UML diagram of such implementation is illustrated in Figure 3.2, and its implementation details can be found under */source/configuration/Node.h* on the RBS.

The Node is a base class for all of the nodes. Each numerical method (e.g., PD) that desires to have a specialized subclass of the Node must implement its own definition of the Node (e.g., PDNode) inside the relation that is required to use the new node definition to follow the microkernel architecture. Thus the RBS includes the Node inside rbs::configuration namespace while the PDNode is defined inside rbs::relations::peridynamic namespace.

### 3.4.2 Bonds and Neighborhood

Peridynamic equation of motion, Eq.3.1, describes the relationship between the space (i.e., acceleration $\ddot{\mathbf{u}}$ of co-location node located at $\mathbf{X}$), time ($t$), and the inner force of each subdomain. The inner forces are composed of body force density field ($\mathbf{b}$) applied to the subdomain and so-called force vector state field $\underline{\mathbf{T}}$ caused by the displacement

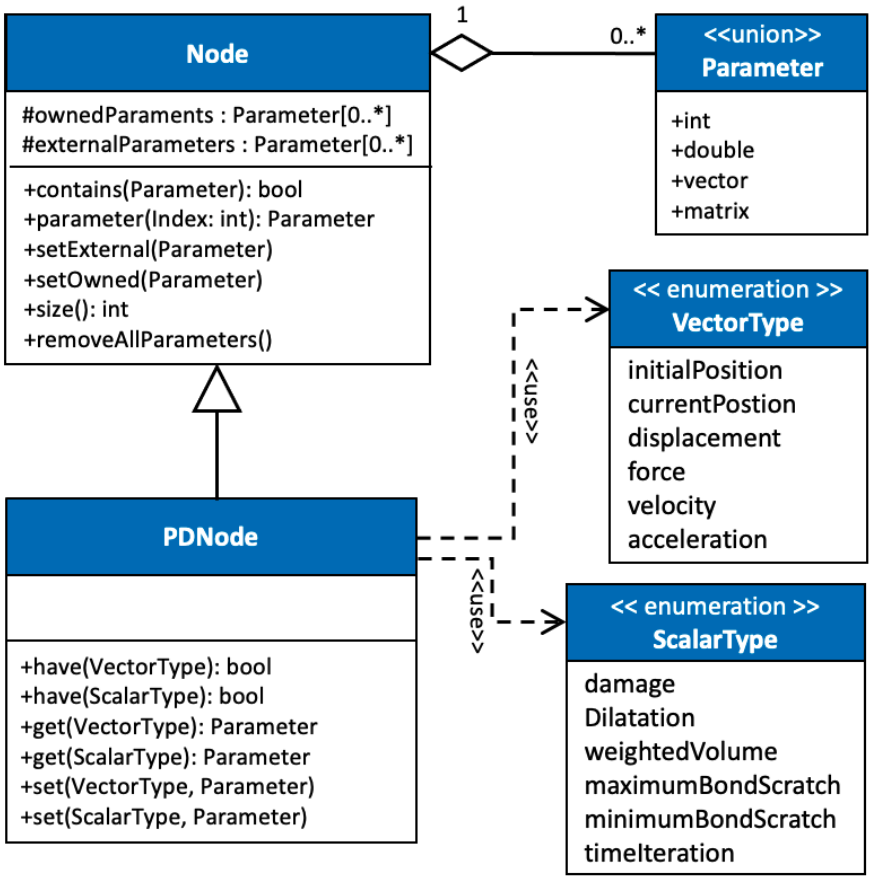



Figure 3.2: The Node UML diagram. The parameter stands for any data type to store node properties or information

(strain) of the surrounding material ($\mathbf{X}'$) within a boundary, which is well-known as the horizon of the subdomain, $\mathcal{H}_X$. Note that the PD equation of motion does not imply any shape requirements on the horizon or the subdomain.

$$\rho(\mathbf{X})\ddot{\mathbf{u}}(\mathbf{X},t) = \int_{\mathcal{H}_X} \{\underline{\mathbf{T}}\,[\mathbf{X},t]\,\langle \mathbf{X}'-\mathbf{X}\rangle - \underline{\mathbf{T}}\,[\mathbf{X}',t]\,\langle \mathbf{X}-\mathbf{X}'\rangle\}\,dV_{X'} + \mathbf{b}(\mathbf{X},t) \quad (3.1)$$

Considering the balance of linear momentum for any bonded body $\beta$, we obtain

$$\int_\beta \rho(\mathbf{X})\ddot{\mathbf{u}}(\mathbf{X},t)dV_{X'} = \int_\beta \mathbf{b}(\mathbf{X},t) \qquad \forall t \geq 0 \qquad (3.2)$$

where it holds regardless of the choice of $\underline{\mathbf{T}}$ [2], Therefore, satisfying the linear momentum does not require the subdomains to include each other on their horizon. Thus as long as the equation of motion and balance of linear momentum are satisfied, the PD equation of motion allows one end bonds.

The balance of angular momentum is required to satisfy

$$\int_{\beta} \{\mathbf{y}(\mathbf{X},t) \times \rho(\mathbf{X})\ddot{\mathbf{u}}(\mathbf{X},t)\}\, dV_{X'} = 0 \qquad \forall t \geq 0 \tag{3.3}$$

where

$$\mathbf{y}(\mathbf{X},t) = \mathbf{X} + \mathbf{u}(\mathbf{X},t) \qquad \forall t \geq 0 \tag{3.4}$$

and $\mathbf{u}(\mathbf{X},t)$ stands for the displacement after time $t$ of the co-location node located initially at $\mathbf{X}$.

The global form of the PD constitutive model as described in Eq 3.8 does not limit the $\underline{\mathbf{T}}$ to be only depended on the displacement of its surrounding nodes, Silling *et al.*[2] however, introduced the SB-PD constitutive model in order to depend only on the deformation vector state field

$$\underline{\mathbf{T}}\left[\mathbf{X},t\right]\langle \mathbf{X}' - \mathbf{X}\rangle = \underline{\widehat{T}}\left(\underline{\mathbf{Y}}\left[\mathbf{X},t\right]\langle \mathbf{X}' - \mathbf{X}\rangle\right) \tag{3.5}$$

where $\underline{\widehat{T}}$ is a bounded Riemann-integrable operator over the horizon and the deformation vector state field defined as

$$\underline{\mathbf{Y}}\left[\mathbf{X},t\right]\langle \mathbf{X}' - \mathbf{X}\rangle = \mathbf{y}(\mathbf{X}',t) - \mathbf{y}(\mathbf{X},t) \tag{3.6}$$

While $\mathbf{y}$ introduced by Eq. 3.4, Eq. 3.6 assumes that no two nodes could be located inside the same subdomain and there exist no overlapping subdomains inside the domain. The state-based constitutive model is then introduced by Silling *et al.*[2] in the form of

$$\underline{\mathbf{T}}\left[\mathbf{X},t\right]\langle \mathbf{X}' - \mathbf{X}\rangle = \mathbf{t}\left[\mathbf{X},t\right] M\left(\underline{\mathbf{Y}}\left[\mathbf{X},t\right]\langle \mathbf{X}' - \mathbf{X}\rangle\right), \tag{3.7}$$

$$M\left(\underline{\mathbf{Y}}\left[\mathbf{X},t\right]\langle \mathbf{X}' - \mathbf{X}\rangle\right) = \left(Dir\underline{\mathbf{Y}}\left[\mathbf{X},t\right]\right)\langle \mathbf{X}' - \mathbf{X}\rangle, \tag{3.8}$$

where if $t$ is scalar force state field, the PD method is called ordinary; Otherwise, it is called nonordinary. $M$ is the deformed direction vector state, and the $Dir$ refers to direction of state $\underline{\mathbf{Y}}$ defined as

$$\left(Dir\underline{\mathbf{Y}}\left[\mathbf{X},t\right]\right)\langle \mathbf{X}' - \mathbf{X}\rangle = \begin{cases} 0 & \text{if} \quad |\underline{\mathbf{Y}}| = 0 \\ \frac{\underline{\mathbf{Y}}[\mathbf{X},t]\langle \mathbf{X}'-\mathbf{X}\rangle}{|\underline{\mathbf{Y}}[\mathbf{X},t]\langle \mathbf{X}'-\mathbf{X}\rangle|} & \text{otherwise} \end{cases}, \tag{3.9}$$

The bound based Peridynamic is then a special case of ordinary state-based peridynamic where the $t \equiv 1$. Two end bonds are the requirement of the OSB-PD and the BB-PD because the T must have central symmetry around each node at any given time.

Consequently, the following point needs to be addressed while implementing the bond as a data type:

- Forcing the bonds to have two ends in any implementation restricts the code to those node-based methods that required two-end bonds. Otherwise, implementation of other constitutive models with exactly one or more than two bonds will be extremely expensive or impossible.

- The bonds are the connection of the nodes in the configuration. In non-local node-based methods, the size of the neighborhood will dominate the cost of the simulation. Thus the storage size of the bonds is vital to the practicability of the code. By increasing the size of the bonds, the number of possible nodes in the domain reduces exponentially.

- Each bond needs to carry out the information of its shared volume with the horizon (i.e., the intersecting volume between the horizon and the neighbor subdomain)

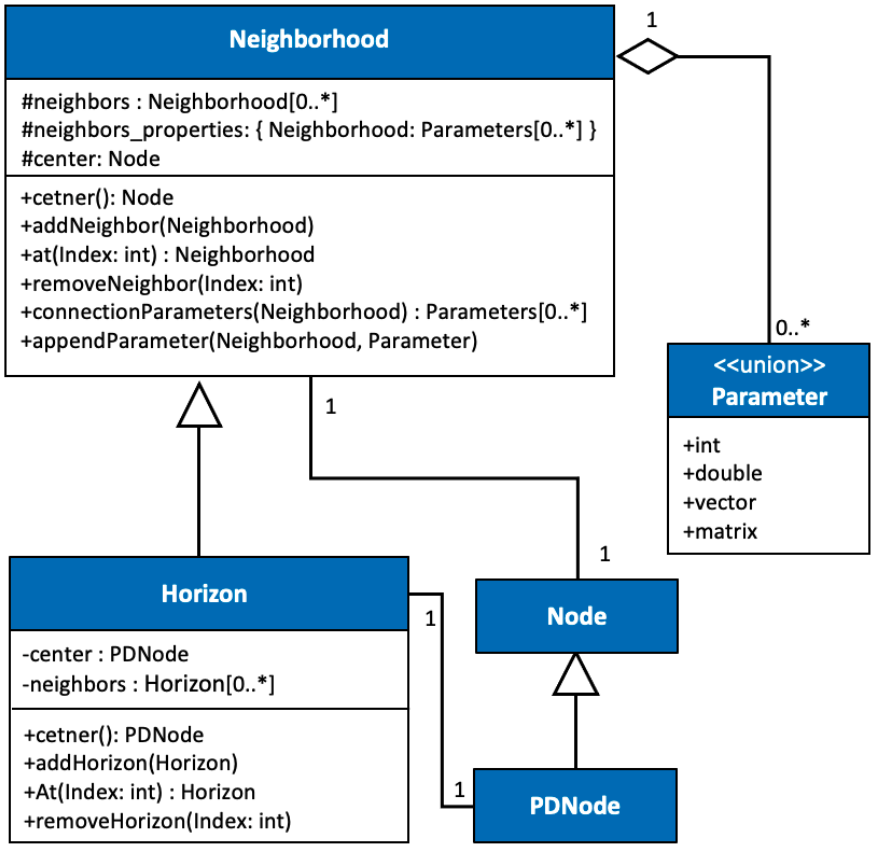



Figure 3.3: The Neighborhood and Horizon UML diagram. The Parameter, Node and, PDNode relations can be found in Figure 3.2

Figure 3.3 illustrates the proposed architecture for the Neighborhood class where Horizon is its realizations for PD. The RBS does not define any form of data structure to store the bonds. Instead, it stores the Nodes' connection inside the Neighborhood and as a form of pointers to the neighbor Nodes' Neighborhoods; this approach increases the code's extendability since more information can be accessed through

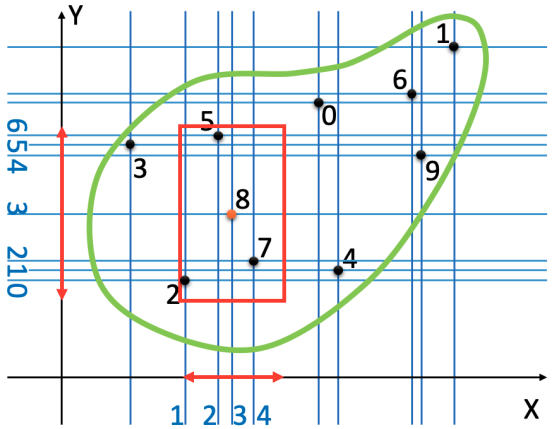



Figure 3.4: The schematic illustration of a 2D dynamic background grid with random discretization of the random domain, and the rectangular neighborhood of $N_8$.

the Neighborhood connections than Node connections. The implementation details of Neighborhood can be found under */source/configuration/Neighborhood.h* on the RBS.

Moreover, defining bonds as a connection between two Neighborhoods has several advantages,

- If the constitutive model requires access to the Neighborhood of the bond ends (i.e., neighbors of its neighbors), it does not require performing any search.
- The border (faces, edges, or corners) of the domain can be quickly found by selecting a random neighborhood and performing any tree or graph data structures search.
- If the constitutive model applies any adoptive scheme (e.g., adoptive refinement around or in front of the crack tip), it can search among the neighbor's Neighborhood instead of performing a global search on all the neighborhoods.
- It forms a network of neighborhoods, which eases the search algorithms to be implemented within the RBS. For instance, one can implement an intersection neighborhood search by only accessing one of the Neighborhood and loop through its neighbors, its neighbor's neighbors, and beyond.

#### 3.4.2.1 Coordinate Systems and Neighborhood Search

In any nonlocal node-based method implementation, the neighborhood search is considered the most expensive pre-processing procedure. A background grid can reduce the search procedure by reducing the problem size exponentially. For instance, in a 3D neighborhood search with $1000000$ uniform nodes, the problem's size will reduce

to $3 \times 100$. However, a background grid is considered bad practice since it limits the nodes to follow the background pattern and does not allow random generation. An ideal dynamic background grid allows random generation while it does not increase the neighborhood search costs. The proposed algorithm imports the nodes to the background grid coordinate system and decomposes them to the grid lines (in 3D grid planes), then maps the node index to the node's address on memory. A linear index will then be assigned to each of the grid points, which can be computed as

$$i_n = i_x + i_y n_x + i_z n_x x_y \quad 0 \leqslant i_x < n_x, \quad 0 \leqslant i_y < n_y, \tag{3.10}$$

where $n_x$, and $n_y$ are the number of gridlines (grid planes) perpendicular to $X$ and $Y$ axes, respectively. Figure 3.4 illustrates a random domain described with ten randomly generated nodes. The CoordinateSystem is then only requires to store a map of the points indexes to differentiate points from the grid point. The following map is the point map required to define the Node's initial position (i.e., CoordinateSystem's points) illustrated on Figure 3.4.

$$\begin{aligned} i_n &\rightarrow N_S \\ 1 &\rightarrow N_2 \\ 16 &\rightarrow N_4 \\ 24 &\rightarrow N_7 \\ 33 &\rightarrow N_8 \\ 48 &\rightarrow N_9 \\ 50 &\rightarrow N_3 \\ 62 &\rightarrow N_5 \\ 75 &\rightarrow N_0 \\ 87 &\rightarrow N_6 \\ 99 &\rightarrow N_1 \end{aligned} \tag{3.11}$$

where $i_n$ are the node linear indexes computed by the Eq. 3.10 and the $S^th$ node added to the background grid is noted by $N_S$.

Figure 3.5 illustrates the CoordinateSystem UML which is a singleton with a tree structure that provides a Global coordinate system at the root of the tree while finite local coordinate systems (cartesian, spherical, and cylindrical) can be defining under it. Simultaneously, each local coordinate system can have an infinite local coordinate system of type cartesian, spherical, and cylindrical coordinate systems. The implementation of the coordinate systems can be found under */source/coordinate_system/CoordinateSystem.h*.

As the implementation provides a thread-free singleton, the existence of only one coordinate system tree in the software is guaranteed, and the points of one coordinate system can then be transferred to any other coordinate system as follows:

## 3.4 Data Structure

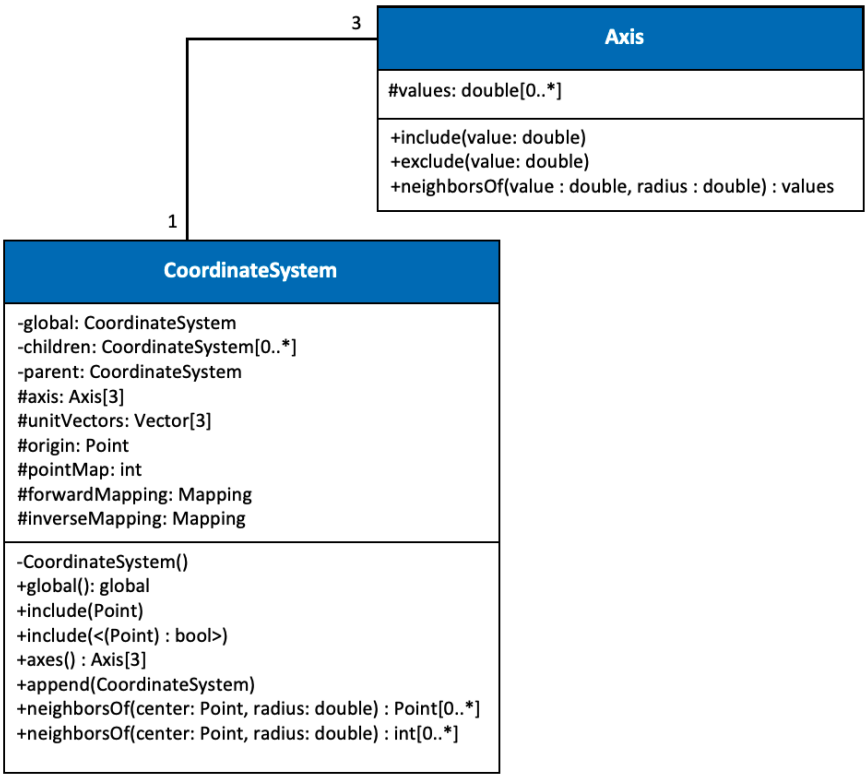



Figure 3.5: The coordinate_system::CoordinateSystem UML diagram.

```
using namespace rbs::coordinate_system;
const auto& localCartesianCoordinateSystem = CoordinateSystem::Global().appendLocal(
    CoordinateSystem::Cartesian, {1,0,0}, {0,1,0}, {0,0,1});
const auto& localCylindricalCoordinateSystem = CoordinateSystem::Global().appendLocal
    (CoordinateSystem::Cylindrical, {1,0,0}, {1, grid::toRad(90) ,0}, {0,0,1});

const auto pointInLocalCylindricalCoordinateSystem = localCylindricalCoordinateSystem
    .convert({2, 3, 4}, localCartesianCoordinateSystem);
```

The above code converts the point at $\{2, 3, 4\}$ inside a cartesian local coordinate system to a cylindrical local coordinate system. Moreover, the utilities for meshing different coordinate systems (namely, cartesian, spherical, and cylindrical) are provided in `rbs::coordinate_system::grid::generators`.

Neighborhood search in the proposed architecture is as follows:

1. Extracting the grid indexes in each direction of the coordinate system,
2. combining the indexes to extract all the grid points' linear indexes (using Eq. 3.10) inside the neighborhood,
3. remove those linear indexes that are not included in the point index map.

This approach will give a rectangular (cuboid in 3D) neighborhood around the point where further evaluation can be done within the neighbors to form a circular (spherical in 3D) or any other shape neighborhoods. The rectangular neighborhood domain for node $N_8$ is schematically illustrated in Figure 3.4. Where the included linear indexes are 1, 2, 3, 4, 11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 34, 41, 42, 43, 44, 51, 52, 53, 54, 61, 62, 63, and 64 where only 1, 24, and 62 can be found inside the map (Eq. 3.11). Thus $N_2$, $N_7$, and $N_5$ are belonging to the rectangular neighborhood of $N_8$.

The following remarks should be considered while implementing the background grid and neighborhood search.

- The background grid coordinate system may not be cartesian and may not be equivalent to the global coordinate system.
- The RBS implementation provides extra means to map the point to any external data; this will allow the constitutive model to map their Nodes to the points for defining their initial position.
- The neighborhood search can be altered by time and the position of its center. If it is changing by time, the neighborhood search needs to be triggered at the beginning of each time step.

RBS implements the proposed architecture. The following code needs to be done to search neighborhoods inside a body, given that the body has already empty Neighborhoods.

```
for(const auto& neighborhood : neighborhoods) {
    const auto centrePosition = neighborhood->centre()->initialPosition().value<space
        ::Point<3> >();
    const auto neighborIndexes = p_localCoordinateSystem->getNeighborPointIndices(
        centrePosition, searchVector, function, centrePosition);

    if ( neighborIndexes.size() ) {
        auto& neighborhoodNeighbors = neighborhood->neighbors();
        std::transform(neighborIndexes.begin(), neighborIndexes.end(), std::
            back_inserter(neighborhoodNeighbors),
           [localMapper](const coordinate_system::CoordinateSystem::LinearIndex&
               neighborCentreLinearindex){
               return localMapper.at(neighborCentreLinearindex);
           });
    }
}
```

where localMapper maps the linear index of the points of the local coordinate system to the address of the neighborhoods on memory. Note that the Part (see subsubsection 3.4.2.4) already includes a member function to perform the search; thus, the researcher is not required to perform the search manually.

#### 3.4.2.2 Geometry

The node's location defines the geometry of the PD simulation. The importing node location should be free of choice; however, by introducing a geometry data type to the software, the process of working with multiple Parts will be simplified. Two distinct geometries, namely, primary- and combined-geometries, are recommended. The primary geometry should contains a compulsory position vector, an optional unit vector, and a thickness, which allows us to define points, lines, planes, spheres, infinite-cylindrical-bars, and infinite-plates in 3D space. Figure 3.6 illustrates the recommended architecture for the primary geometry, where the distance, point status, intersection, and projection are the key features of this class. Note that the intersection and projection of two primary geometries may not be presentable by another primary geometry, for example, the intersection of two spheres or projection of a sphere on a plane; in such a situation, the class member should return empty .

The combined geometry is a combination of two other geometries with a unique set operation (i.e., $+$ operation or union; $*$ operation or intersection; $-$ operation or difference; and $\hat{}$ operation or symmetric-difference) between them. As illustrated in Figure 3.7, a Geometry interface is required to make this possible. Thus the combined- and primary-geometry should implement the Geometry interface. Then, the combined geometry will be able to store two or more geometries along with the set operation. Thus a combined-geometry will be able to combine other combined- or primary-geometries together to present a new shape in 3D. For example, the shape presented in Figure 3.8 can be created as follow:

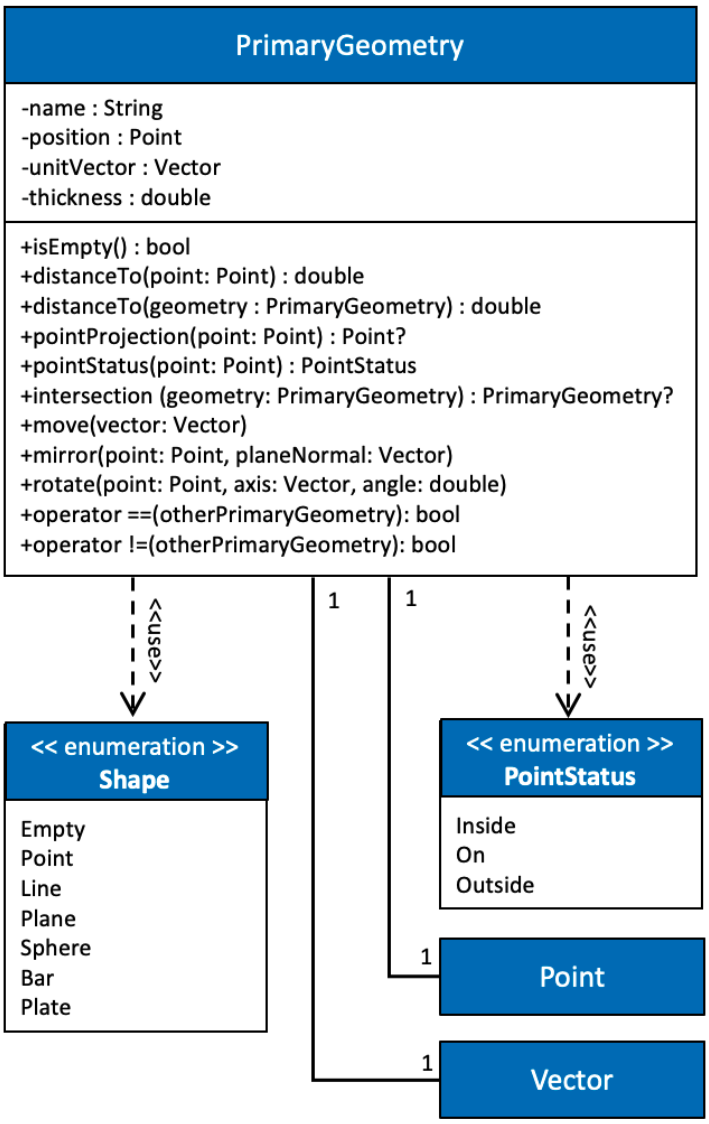



Figure 3.6: The geometry::Primary UML diagram.

```
using Operation = geometry::SetOperation;
const double innerRadius = 1.;
const double thickness = 0.1;

const auto innerBar = geometry::Primary::Bar({0,0,0}, {0,0,1}, innerRadius);
const auto outerBar = geometry::Primary::Bar({0,0,0}, {0,0,1}, innerRadius +
    thickness);

const auto infinitePipe = geometry::Combined(outerBar, Operation::Difference,
    innerBar);

const auto bottomCutter = geometry::Primary::HalfSpace({0,0,0}, {0,0,-1});
```

### 3.4 Data Structure

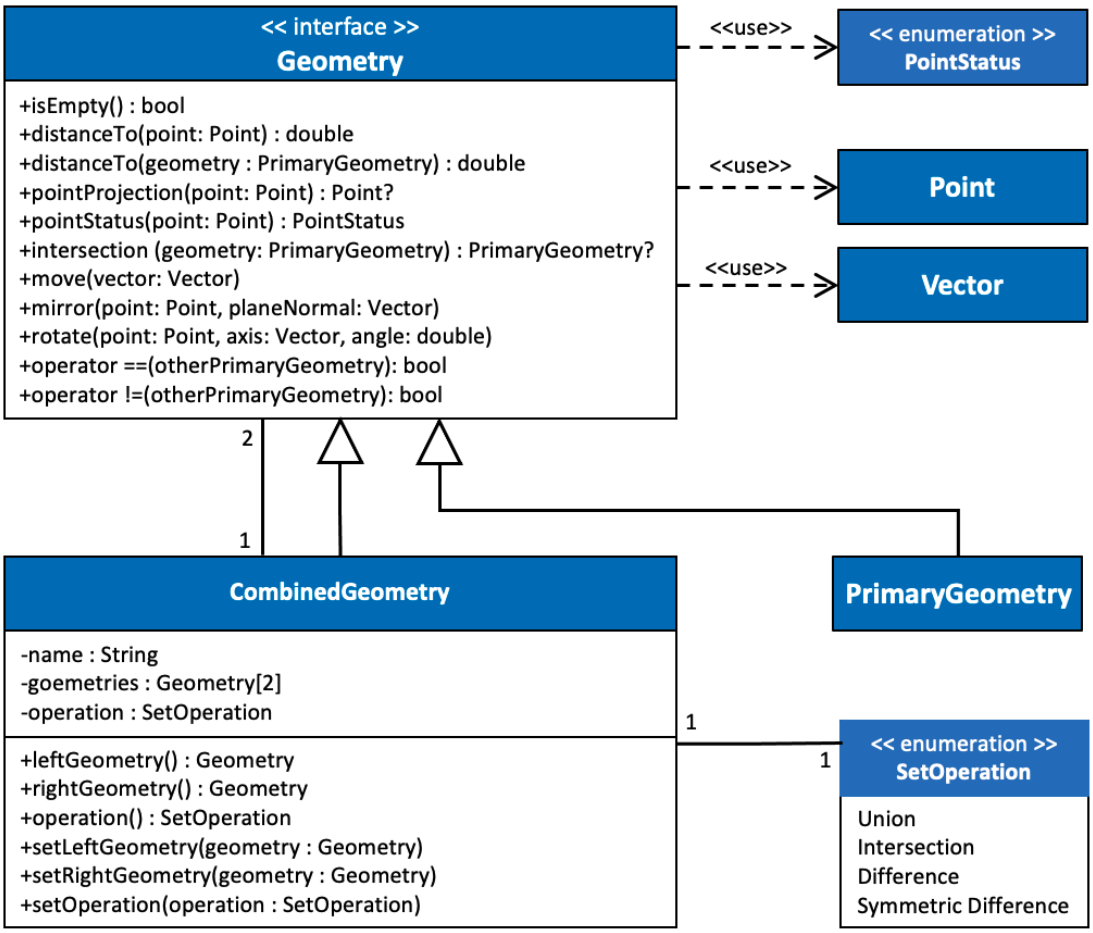



Figure 3.7: The geometry::Combined UML diagram.

```
const auto upperCutter = geometry::Primary::HalfSpace({0,0,1}, {0,3,4});
const auto cutter = geometry::Combined(upperCutter, Operation::Union, upperCutter);

const auto shape = geometry::Combined(infinitePipe, Operation::Difference, cutter);
```

RBS implements the above suggested architecture in geometry::Primary and geometry::Combined classes. The geometry::Primary and geometry::Combined implementation in RBS provides easy to use static constructors to create different shapes. Figure 3.9 illustrates six static constructors available for creating geometry::Primary. Nine shapes illustrated in Figure 3.10 are the provided static constructors of geometry::Combined. Since, RBS's development is ongoing, more static constructors are expected to be added in the future. The geometries' implementation details can be found under */source/geometry* and the documentation under */documentation/geometry* on the RBS repository.

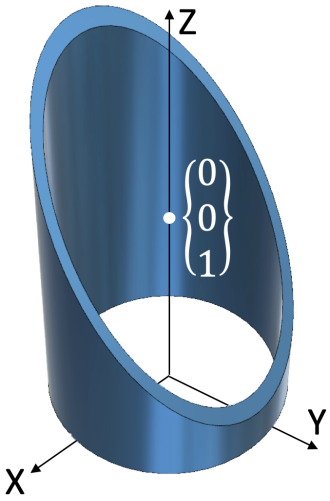



Figure 3.8: A cut of a pipe geometry created by geometry::Combined.

#### 3.4.2.3 Complex Geometry

There are two possible ways to create complex geometries that cannot be presented by the geometry::Primary or geometry::Combined.

- First, one can introduce a new geometry type (e.g., geometry::Complex class) that inherits the geometry::Geometry interface and implements the new form of geometries.
- Second, combining the geometry::Primary or geometry::Combined with the coordinate systems.

The first approach should be considered if the new geometry::Complex class would cover a wide range of well-known geometries that are likely to be used in engineering applications. However, the second approach is more preferred if the geometry is unique in terms of mathematical parameters. Here we create an ellipsoid and an elliptical paraboloid to present the second approach.

***Ellipsoid***

First, we define a new custom local coordinate system by appending one to the Global coordinate system. The new coordinate system is cartesian, where its X-axis unit vector (i.e., i vector) has a length of 2.

```
using namespace coordinate_system;
auto& globalCS = CoordinateSystem::Global();
auto& localCS = globalCS.appendCustomLocal(
    {0, 0, 0},  // origin
```

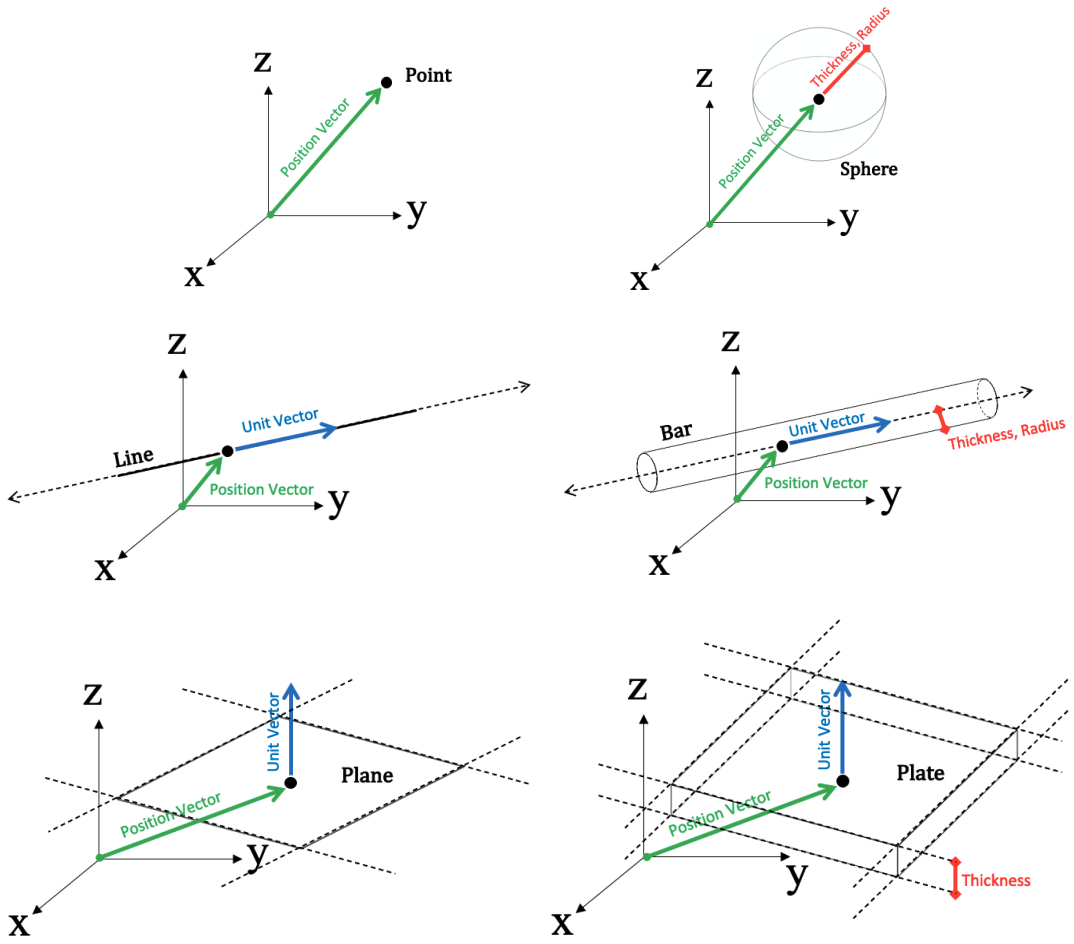



Figure 3.9: The provided static constructors for geometry::Primary.

```
    {2, 0, 0},
    {0, 1, 0},
    {0, 0, 1}, // i, j, k vectors
    convertors::cartesian::toCartesian(),        // mapping
    convertors::cartesian::toCartesianInverse() // inverse mapping
);
```

Next, we create a sphere that later we will use inside the local coordinate system.

```
auto sphere = geometry::Primary::Sphere({0, 0, 0}, 2.5);
```

By meshing the local coordinate system and including the points inside the sphere locally, we would have our ellipsoid in the global coordinate system.

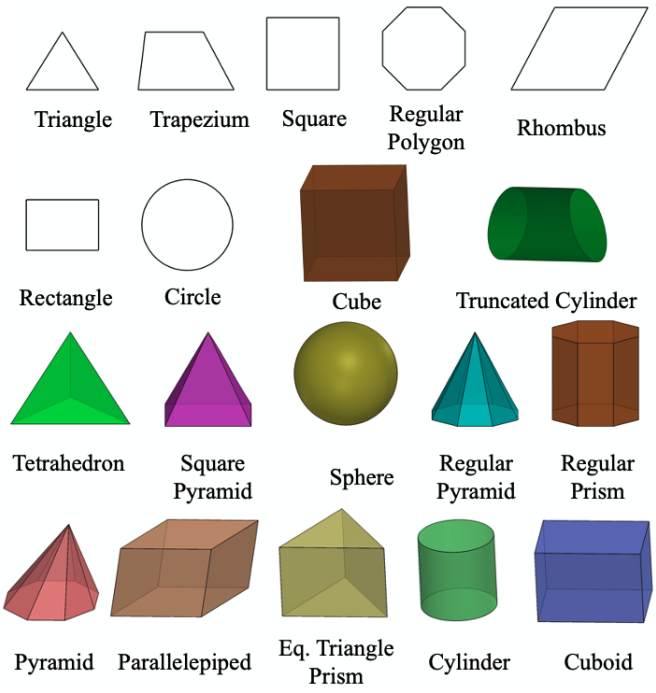



Figure 3.10: The provided static constructors for geometry::Combined.

```
grid::cartesian::uniformAroundOrigin(space::consts::one3D * 0.05, space::consts::
    one3D * sphere.thickness() + 0.05, localCS.axes());

localCS.include([sphere](const space::Point<3>& point) {
    return sphere.pointStatus(point) != geometry::PointStatus::Outside;
});
```

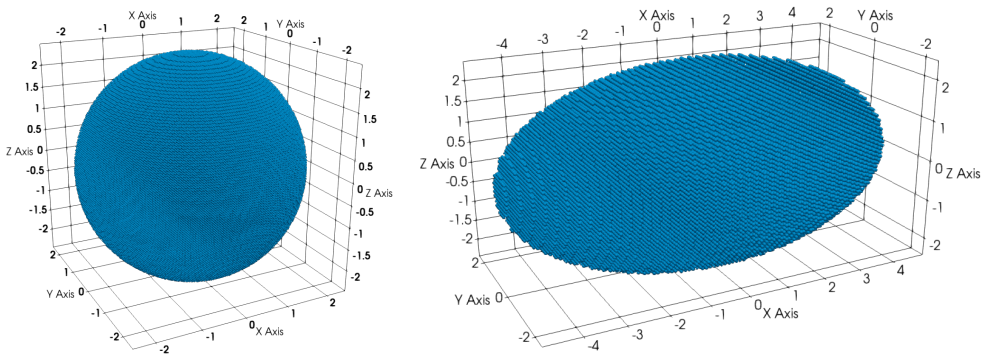
To present this, we can export the local and global coordinate systems as

```
const auto& localPoints = localCS.getAllPoints();

auto file_local = exporting::VTKFile(path, "local", "vtk");
file_local.appendCell(exporting::vtk::Cell::PolyVertex, exporting::vtk::convertors::
    convertToVertexes(localPoints));
file_local.assemble();

auto file_global = exporting::VTKFile(path, "global", "vtk");
for(const auto& localPoint : localPoints) {
    const auto globalPoint = globalCS.convert(localPoint, localCS);
    file_global.appendCell(exporting::vtk::Cell::Vertex, exporting::vtk::convertors::
        convertToVertexes({globalPoint}));
}
file_global.assemble();
```

which result to the configurations illustrated in Figure 3.11.




(a) (b)

Figure 3.11: The ellipsoid in (a) local and (b) global coordinate systems.

***Elliptical Paraboloid***

Unlike the ellipsoid, one cannot convert any geometry::Primary or geometry::Combined to an elliptical paraboloid plate by changing the coordinate system unit vectors. Here we need to create a new coordinate system by implementing required mapping and inverse mapping functions.

First, we need to define the mapping and inverse mapping.

```
using Point = space::Point<3>;
using Vector = space::Vector<3>;

const auto mapping =  [](const Point& point, const Point& origin, const Vector& i,
    const Vector& j, const Vector& k) {
    const auto& positionVector = point.positionVector(); // The position vector of
        the point in parent coordinate system.
    return Point{
        positionVector[0],
        positionVector[1],
        0
    };
};

const auto inverseMapping = [](const Point& point, const Point& origin, const Vector&
     i, const Vector& j, const Vector& k) {
    const auto& positionVector = point.positionVector(); // The position vector of
        the point in child coordinate system.
    return Point{
        positionVector[0],
        positionVector[1],
        (pow(positionVector[0], 2) + pow(positionVector[1] / 2, 2))
    };
};
```

Next, we define a new custom local coordinate system by appending one to the Global coordinate system. The new coordinate system is a custom coordinate system,

where its origin and unit vector are the same as the global coordinate system, while its mapping and inverse mapping are the above functions.

```
using namespace coordinate_system;
auto& globalCS = CoordinateSystem::Global();
auto& localCS = globalCS.appendCustomLocal(space::consts::o3D,
                                           space::consts::i3D,
                                           space::consts::j3D,
                                           space::consts::k3D,
                                           mapping,
                                           inverseMapping);
```

Next, we create a plate that we will later use inside the local coordinate system.

```
const auto plateThickness = 0.04;
auto plate = geometry::Primary::Plate({0, 0, -plateThickness / 2}, space::consts::k3D
    , plateThickness);
```

By meshing the local coordinate system and including the points inside the sphere locally, we will have our ellipsoid in the global coordinate system.

```
grid::cartesian::uniformAroundOrigin(space::consts::one3D * 0.02, {5, 10,
    plateThickness}, localCS.axes());

localCS.include([plate](const space::Point1<3>& point)
    return plate.pointStatus(point) != geometry::PointStatus::Outside;
});
```

To present this, we can export the local and global coordinate systems as before will result to the configurations illustrated in Figure 3.12.

#### 3.4.2.4 Parts

A flexible computation method should allow simulations containing different constitutive models and possibly rigid bodies where they may need to be analyzed parallelly or while interacting with each other. Introducing the concept of ”Part,” where each Part contains a local coordinate system, a geometry, neighborhoods of a body or part of the problem domain, and neighborhoods containing neighbors from other Part reduces the complexity of implementing any constitutive models by hiding the neighborhood search algorithm following the Facade Design pattern. The material points inside the Part have the same material behavior, same neighborhood geometry (same horizon shape for PD problems), same neighborhood search function, and same discretization routine. Each Part can have a unique time integration scheme and a constitutive model. The parts may also overlap (e.g., simulating porous materials). Figure 3.13 illustrates the Part UML diagram that satisfies the points mentioned above.

The RBS uses the Facade Design pattern to hide the complexity of the performing

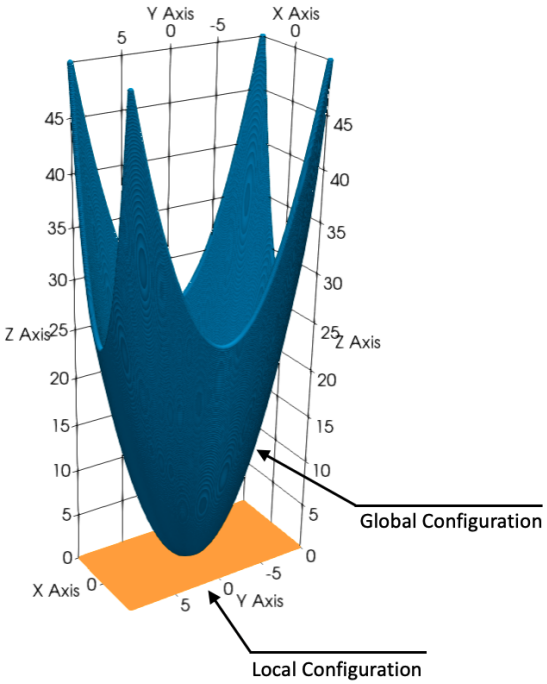



Figure 3.12: The elliptical paraboloid in local and global coordinate systems.

neighborhood. Each Part comes with only one local coordinate system, one neighborhood search algorithm, and one geometry. The parts then can be combined to create a bigger domain if required. Although the Part's implementation can be ignored, it is highly recommended, not only because it simplifies the interface for using the neighborhood search and local coordinate system, but also because it secures the neighborhoods from unexpected alteration and eases the exportation of the part. The implementation details of Part can be found under */source/configuration/Part.h* on the RBS.

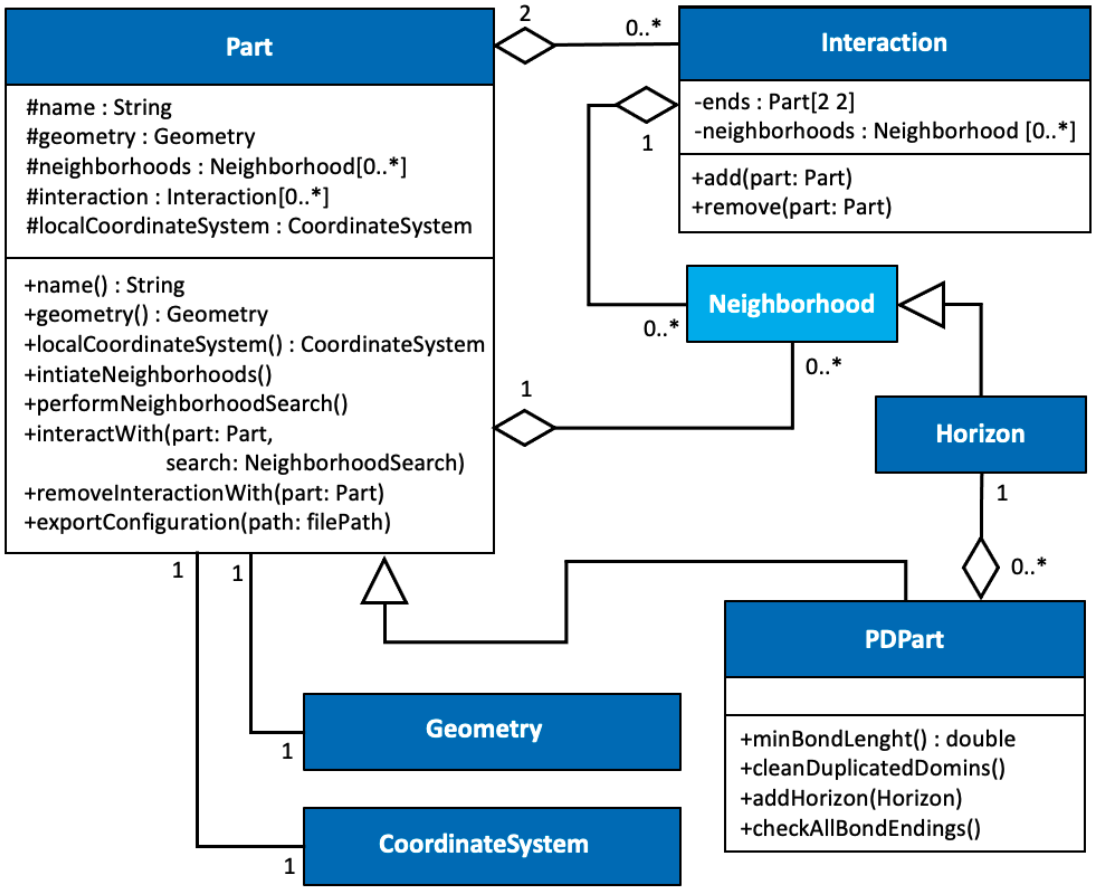



Figure 3.13: The Part UML diagram. The Neighborhood, and Horizon UML diagrams can be found in Figures 3.3.

## 3.5 Relations

The Relation allows changing data (Feedee) in the event of alternation of other or even the same data (Feeder). The event can be handled with two distinct approaches. First, by throwing a message from the Feeder whenever it receives a change, receiving the message by the Relation to alter the Feedee. Second, by looping through the existing Relations at each time step. The first approach is preferable in terms of multiprocessing, but the order of the data structure alternation is not guaranteed. Although it is possible to secure the process through switchers, the author preferred the second approach because of its simplicity and guaranteed security on the order of Relations execution. The RBS uses a singleton called Analyse to register the Relations in the preferred order and execute them at each timestep. The Applicable interface is utilized, which is realized by the Relation to make the connection between Relations and Analyses possible.

Figure 3.14 illustrates the UML diagram for Relation. The solid transition and solid rotation can simply be achieved by defining a Relation with its Feeder as the transition vector and rotation center and vector, respectively. The Feedee will be a Part whose local coordinate system will move or rotate at each execution. The constitutive models (e.g., Bond-Based Peridynamic) are Relation between Time (i.e., Feeder) to Part (i.e., Feedee) where at each time step the Part's Nodes' forces changes based on the Analyse Time. Note that the time integration is also a Relation between Time to Part with the difference that the Part's Nodes' parameters will be updated. It is also possible to include the time integrations within the constitutive models, but this approach is not recommended since it increases the maintenance cost. The implementation details of Relation and Applicable can be found under */source/relations/* on the RBS.

The provided relations by RBS are not intended to cover all situations and possibilities but rather the most common ones. If one requires the creation of a new type of Relation, specific to a problem, they should implement their Relation from scratch, and by inheriting the relations::Relation. For instance, the following will introduce a dynamic peridynamic contact model.

```
class DynamicSearch: public relations::Relation<Part, Part> {
public:
    DynamicSearch(const Part& neighborPart, Part& part, const double horizonRadius)
        : Relation<Part, Part>(neighborPart, part, [horizonRadius](const Part&
            neighborPart, Part& part) {
            part.searchNeighborsWith(neighborPart, horizonRadius, true);
        }) {
    }
};
```

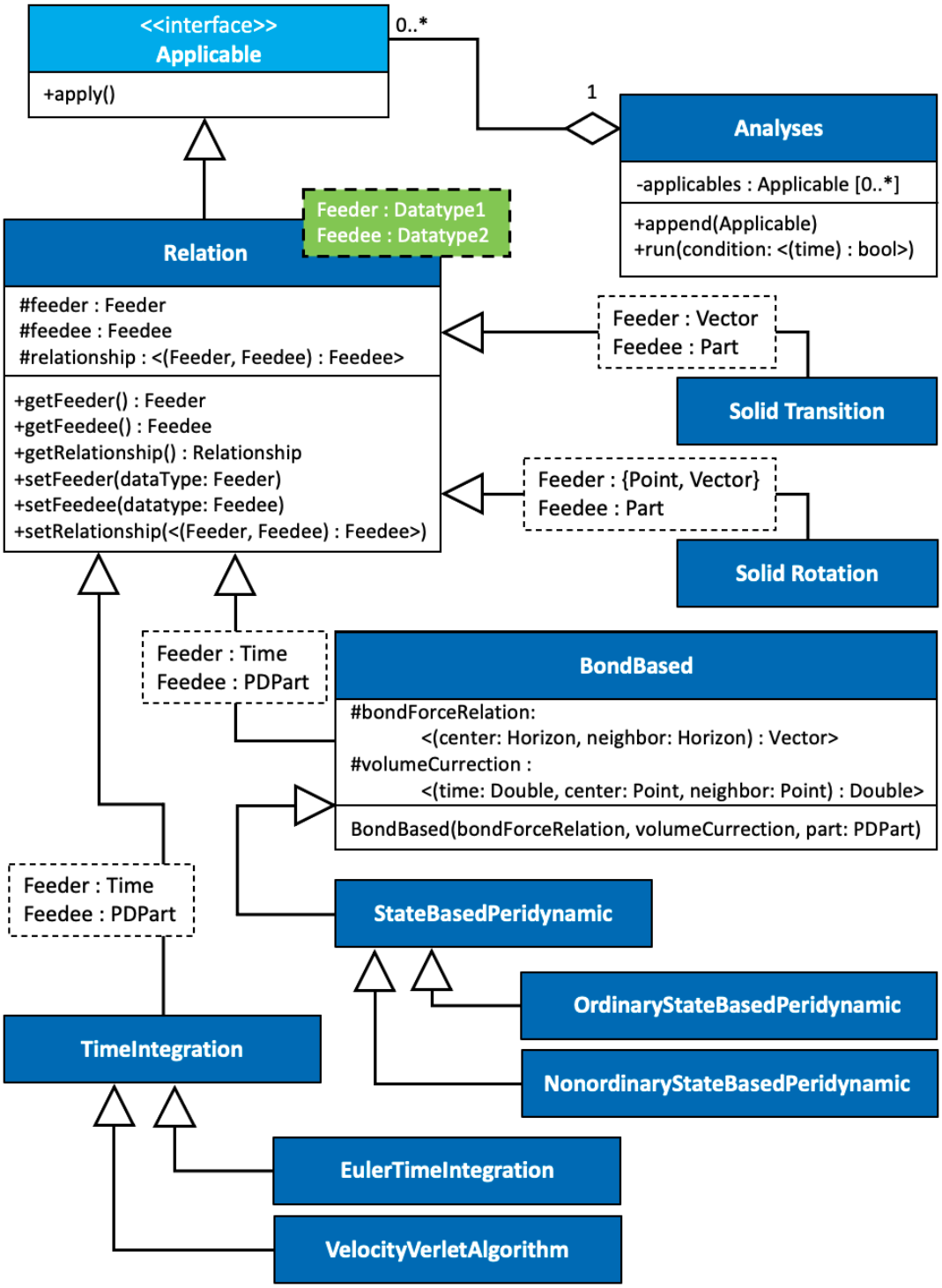



Figure 3.14: The Relation UML diagram. The Parameter UML diatgram can be found in Figure 3.2.

### 3.5.1 RBS Architecture

Figure 3.15 gives an overall view of the RBS architecture and its extensible components. A comparison between, Figure 3.1 and Figure 3.15 give a perspective on the advancement and flexibility of the RBS compare to its protestors.

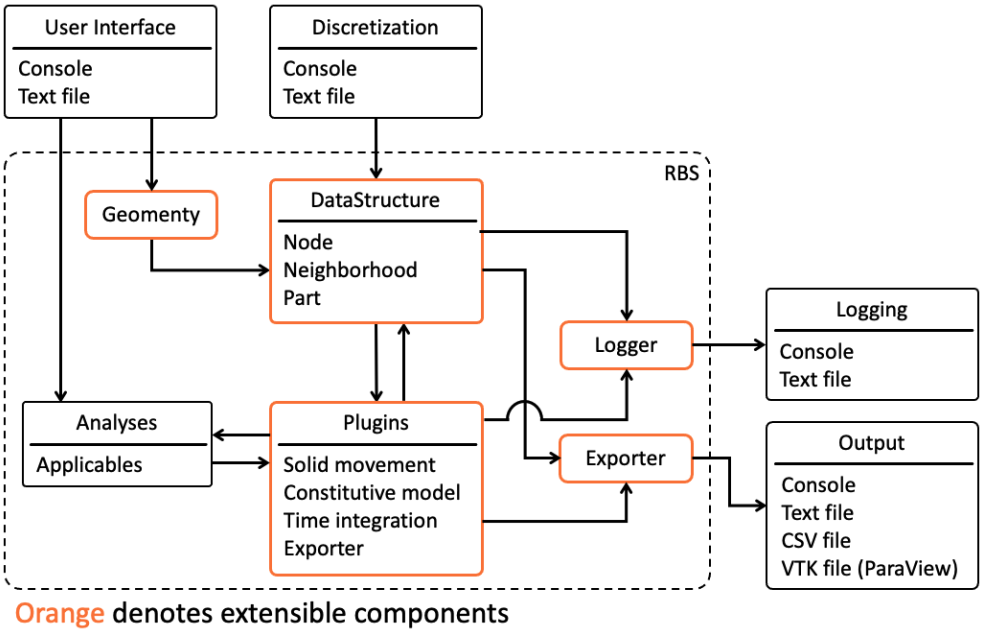



Figure 3.15: RBS architecture and its extendable areas.

# Chapter 4

# RBS in Practice

This chapter is dedicated to presenting the applications of the RBS in practice. The examples are provided in full detail at the GitHub repository [1]; here, we discuss the basic principles and advantages of the RBS architecture in three examples. First, we will examine the wave propagation inside a plate utilizing PD, and we discuss the basics of PD as a plugin to RBS. Next, a fracture problem with preexisting crack is simulated to demonstrate the flexibility of the RBS in terms of defining crack geometry and customizing the bond-force relation of PD. Finally, a complex simulation of polymer is presented where the complex Ordinary State-Based Peridynamic simulation takes place to model the fracture initiation and growth in a mesoscale problem.

## 4.1 Wave Propagation

Dally *et al.*[39] reported the wave propagation inside a plate (see Figure 4.1) under explosion at the plate's top edge midpoint. Nishawala *et al.*[40] used a triangular impulse load with a maximum amplitude of 20.7E3 at 10 $\mu sec$ and pulse width of 20 $\mu sec$ as it illustrated in Figure 4.2 to simulate the wave propagation. They modeled the experiment with both Cellular Automata and Peridynamic. The reported distribution of the displacement at 1.07 microseconds can be found in Figure 4.3. The full description, RBS simulation results, and the completed code, ready to execute, are provided in the GitHub repository [2].

The RBS adopts the same folder structure as the namespace structure. Each plugin has a folder and a forward declaration file, which simplifies the use of the plugin. For instance, the structure of the Peridynamic is as follows,

[1] https://github.com/alijenabi/RelationBasedSoftware/tree/master/simulations/

[2] "https://github.com/alijenabi/RelationBasedSoftware/tree/master/simulations/ElasticWavePropagationinPlate"

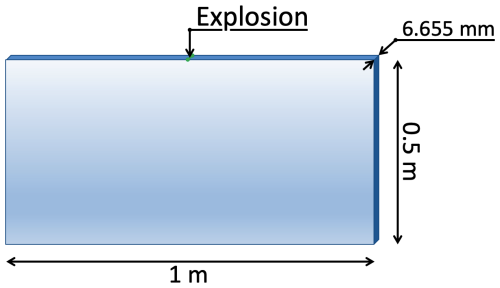



Figure 4.1: The wave propagation experiment geometry.

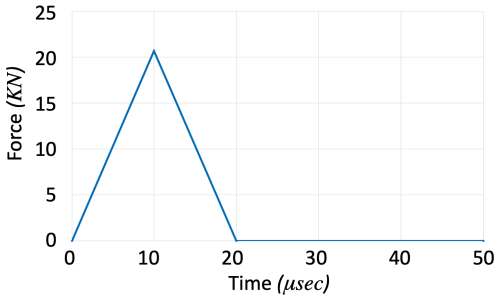



Figure 4.2: The triangular impulse load.

```
RBS
└─ source
   └─ relations
      └─ peridynamic
         ├─ time_integrations
         │  ├─ PDEuler ..... class
         │  └─ PDVelocityVerletAlgorithm ..... class
         ├─ BondBased ..... class
         ├─ BoundaryDomain ..... class
         ├─ Exporter ..... class
         ├─ OrdinaryStateBased ..... class
         ├─ Property ..... class
         └─ peridynamic ..... forward declaration
```

where ”class” denotes both the header and the source files.

## 4.1 Wave Propagation

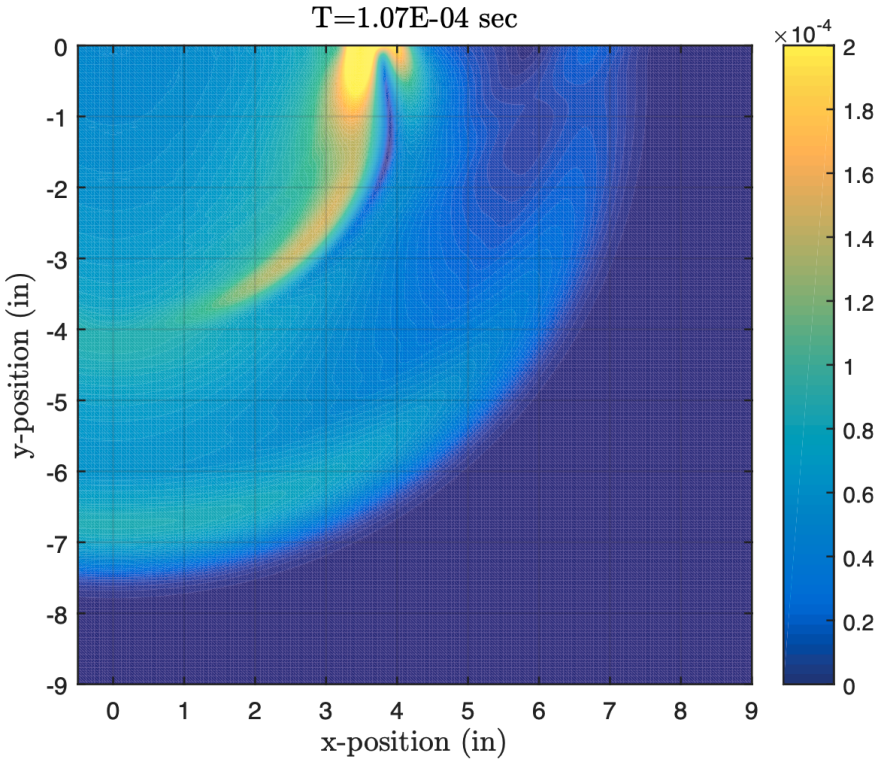



Figure 4.3: The PD displacement plot from [40].

Let us define the problem description first.

```
const std::string path = "/any-file-path/";

const double dencity = 1300;           // kg / m^3
const double youngsModulus = 3.85e9;   // GPa
const double poissonRasio = 1. / 3.;

const auto plateHeight = 0.5;          // 0.5 m
const auto plateWidth  = 1.0;          // 1.0 m
const auto plateThickness = 0.006655;  // 6.655 mm

const auto horizonRasio = 3.;
const auto gridSpacing = std::min({plateWidth / 1024, plateHeight / 512});
const auto horizonRadius = horizonRasio * gridSpacing;
```

Each plugin brings not only its own constitutive model but also the time-integration schemes and its exporters. Thus, those researchers that develop a new constitutive model must define how their method works with the core functionality of the RBS. Other researchers can securely use the provided constitutive models without requiring a deep understanding of its implementation details. For instance, since the Bond-Based Peridynamic provides it the definition of the PDPart, researchers are not required to

adopt the Part defined by the RBS core to the Peridynamic. For defining PDPart, we can write

```
using CS = coordinate_system::CoordinateSystem;
using BondBasedPeridynamic = rbs::relations::peridynamic::BondBased;
auto platePart = BondBasedPeridynamic::PDPart("Plate", CS::Global().appendLocal(CS::
    Cartesian));
```

where the defined part can then be used as the Feedee of other Relations provided by the peridynamic plugin.

Next, we need to define the geometry and assign it to the plate part.

```
const auto plateShape = geometry::Combined::Cuboid({-plateWidth / 2, -plateHeight, -
    plateThickness / 2},
                                                    space::vec3{plateWidth, 0, 0},
                                                    space::vec3{0, plateHeight, 0},
                                                    space::vec3{0, 0, plateThickness
                                                        });
platePart.setGeometry(plateShape);
```

where cuboid is one of the static Constructors of Combined geometry class.

Using the provided grid meshing functions in */source/coordinate_system/grid.h* on the RBS. One can mesh the coordinate system as follows. Note that for creating one mesh in Z-direction, we can override the mesh as done in line 6 or change the distance-vector third component to { gridSpacing, gridSpacing, plateThickness }.

```
1 const auto distanceVector = gridSpacing * space::consts::one3D;
2 const auto startPoint = space::Point<3>{-plateWidth / 2, -plateHeight, -
      plateThickness / 2} + distanceVector / 2;
3 const auto endPoint   = space::Point<3>{plateWidth / 2, 0, plateThickness / 2} -
      distanceVector / 2;
4
5 coordinate_system::grid::cartesian::uniformDirectional(startPoint, endPoint,
      distanceVector, platePart.local().axes());
6 platePart.local().axes()[2] = std::set<double>{0};
```

As explained in section 3.4.2.1, the grid points that are the Part's Nodes' location should be marked by including them to the Node map. This is our first encounter with functional programming benefits. The Part's local coordinate system's included member function is capable of receiving an include function, which will be executed at each grid point to include or exclude that point as Part's Node. The included function requires a specific signature. In the following code, it is defined as lambda function between line 2 to 4. The include function should receive the grid point position as input and return a boolean that defines whether the grid point is inside the part or not.

```
1 const auto& plateShape = platePart.geometry();
2 cosnt auto& includeFunction = [&plateShape](const auto& localPoint) -> bool {
3     return plateShape.pointStatus(localPoint) == geometry::Inside;
```

```
};

platePart.local().include(includeFunction);
```

Since the plate Part in this simulation should include all of the grid points, the above code can be simplified as following

```
platePart.local().include([](const auto&) { return true; });
```

Next, we initialize the neighborhoods, add density and volume, and search the neighborhoods. The inheritance of object-oriented programming explained in section 3.2.1 becomes extremely useful here. Since the PDPart is a subclass of Part, one can use all of the provided functionalities by Part class without considering whether the PDPart overrides them or uses the same algorithm.

```
\\ Neighborhood initialization
platePart.initiateNeighborhoods();

\\ Adding volume and density
const double gridVolume = pow(gridSpacing, 2) * plateThickness;
const auto& neighborhoods = platePart.neighborhoods();
std::for_each(neighborhoods.begin(), neighborhoods.end(), [gridVolume, dencity](const
     Part::NeighborhoodPtr& neighborhood) {
    using Property = relations::peridynamic::Property;
    auto& centre = *neighborhood->centre();
    centre.at(Property::Volume).setValue(gridVolume);
    centre.at(Property::Density).setValue(dencity);
});

\\ Neighborhood search
platePart.searchInnerNeighbors(horizonRadius);
```

Since the platePart is a PDPart, we can create other relations inside the PD. For instance, we can creat a velocity Verlet algorithm and a elastic Bond-Based Peridynamic Relations as

```
auto timeIntegration = relations::peridynamic::time_integration::
    VelocityVerletAlgorithm(platePart);
auto platePDRelation = relations::peridynamic::BondBased::Elastic(materialConstant,
    gridSpacing, horizonRadius, platePart, false);
```

RBS combines the above benefits of Object-Oriented Programming (OOP) with functional programming to maximize the code's extendability while benefiting OOP's simplicity. For instance, the BoundaryDomain Relation of peridynamic plugin allows changing the PDPart's Nodes' properties in each timestep. This class provides a function called Conditioner, where it can be passed to the constructor to apply any boundary domain to the peridynamic simulation. The Conditioner receives the time of the sim-

ulation and the nodes one by one. Nishawala *et al.*[40] applied the triangular impulse load illustrated in Figure 4.2 to four Nodes at the top edge of the plate (see Figure 4.1). To implement the same load in RBS, one can utilize the BoundaryDomain and its Conditioner as follows

```
const auto maxForcePerNode = 4 * 20.7e3 / ( plateThickness * gridSpacing );
const auto conditioner = [gridSpacing, maxForcePerNode](const double time,
    configuration::Node& node) {
    using Property = relations::peridynamic::Property;
    auto& postion = node.initialPosition().value<space::Point<3> >().positionVector()
        ;

    if (-gridSpacing <= postion[0] && postion[0] <= gridSpacing && -gridSpacing * 2.1
         < postion[1]) {
        if (time <= 10e-6) {
            node.at(Property::Force) = -space::vec3{0, time * maxForcePerNode / 10e
                -6, 0};
        } else if(time <= 20e-6) {
            node.at(Property::Force) = -space::vec3{0, maxForcePerNode - (time - 10e
                -6) * maxForcePerNode / 10e-6, 0};
        } else {
            node.at(Property::Force) = space::vec3{0, 0, 0};
        }
    } else {
        if (node.has(Property::Force))
            node.at(Property::Force) = space::vec3{0, 0, 0};
    }
};
auto load = relations::peridynamic::BoundaryDomain(conditioner, platePart);
```

To reduce the memory cost, we can change the else clause as follows

```
if (-gridSpacing <= postion[0] && postion[0] <= gridSpacing && -gridSpacing * 2.1 <
    postion[1]) {
    ... same as above ...
} else {
    if (node.has(Property::Force))
        node.at(Property::Force) = space::consts::o3D;
}
```

This will also reduce the computation costs since the BondBased and VelocityVerletAlgorithm will avoid computing if no parameter is stored on the Node.

For defining an elastic bond-based peridynamic constitutive model, we can use the Elastic static member of the BondBased class.

```
const double bulkModulus = youngsModulus / ( 3 * (1 - 2 * poissonRasio));
const double materialConstant = 12 * bulkModulus / (M_PI * gridSpacing * pow(
    horizonRadius, 3));
auto platePDRelation = relations::peridynamic::BondBased::Elastic(materialConstant,
    gridSpacing, horizonRadius, platePart);
```

Finally, we can add the relations above to the Analyse and run it.

```
auto& analysis = Analyse::current();
analysis.setTimeSpan(0.125e-6);
analysis.setMaxTime(208e-6);
analysis.appendRelation(load);
analysis.appendRelation(platePDRelation);
analysis.appendRelation(timeIntegration);

return analysis.run();
```

Thanks to the procedural programming approach on modeling steps, the researchers can follow the sequence of the simulation and their connection. The full CPP code of this example can be found in Appendix A.

The illustrated PD displacement and velocity plots in Figure 4.4 and Figure 4.5 respectively were achieved. The similarity of the pattern and wave speed inside the domain was observed to be the same as those reported by Dally *et al.*[39] and Nishawala *et al.*[40].

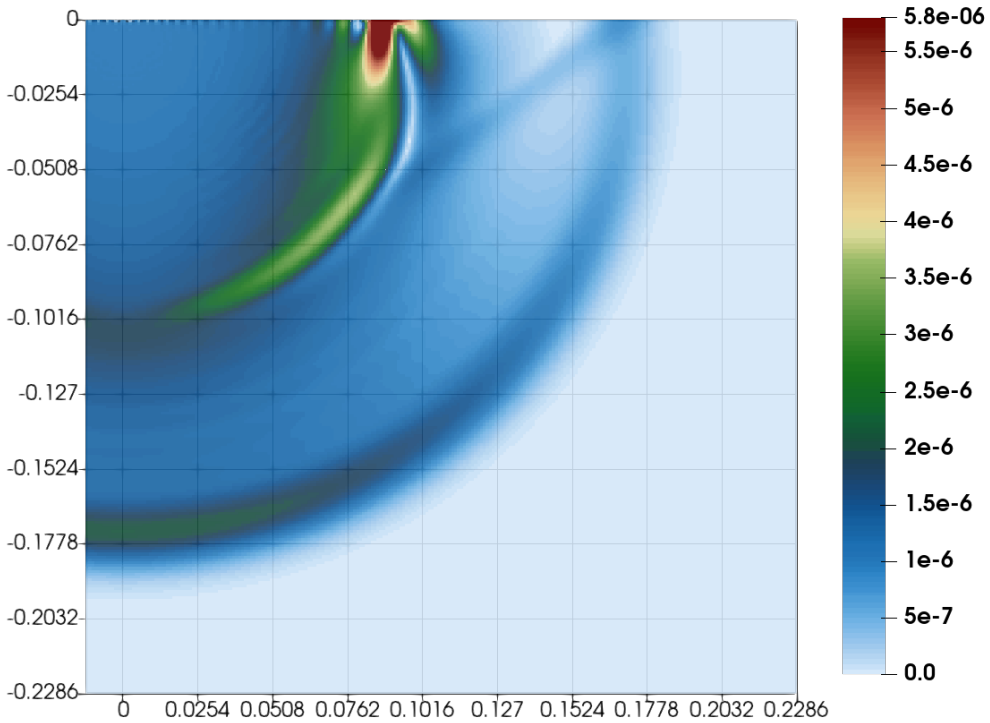



Figure 4.4: The PD displacement plot.

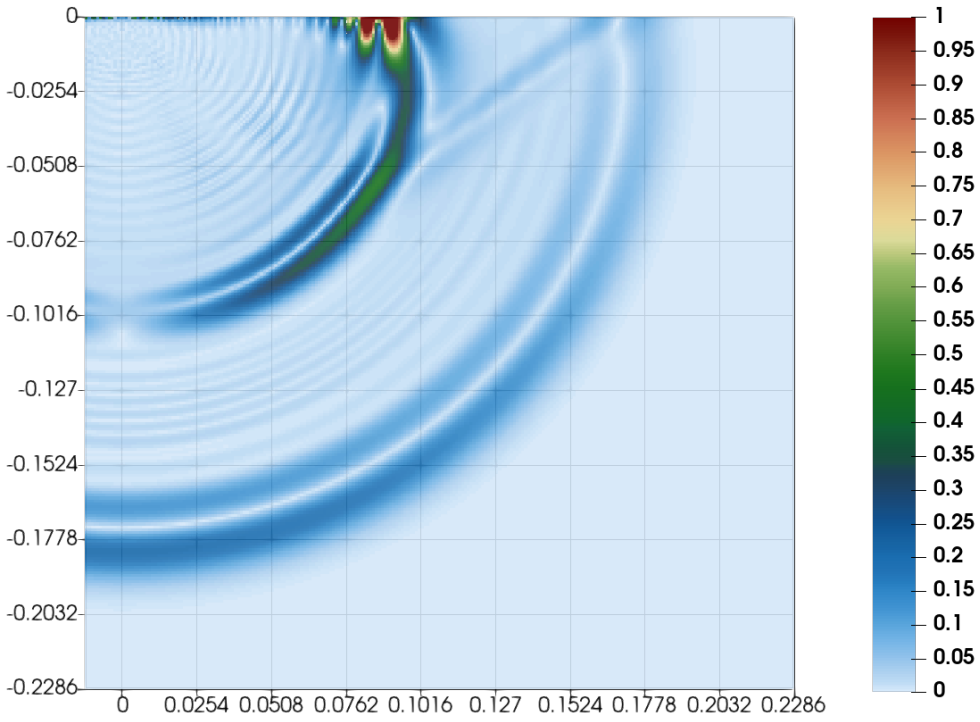



Figure 4.5: The PD velocity plot.

## 4.2 Fracture in plate with Pre-existing Crack

A squared plate under tension from both sides with a pre-existing crack is simulated and presented in this section. The loading is applied as constant velocity with a steady rate of $20\frac{m}{s}$ and $50\frac{m}{s}$ at the plate's top and bottom edge. The plate has a pre-existing crack at the center and perpendicular to the boundary condition, as illustrated in Figure 4.6. The plate's material properties can be found in Table 4.1 and peridyanmic parameters in Table 4.2.

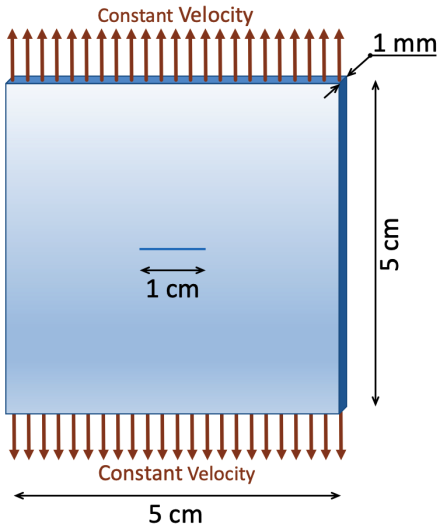



Figure 4.6: The plate under tensile loading.

Although one can manually add the Nodes and Neighborhoods to the Part's local coordinate and the Parts, the RBS provides "geometry" and "constitutive" namespaces to ease the assembly. Establishing a pre-existing crack becomes cumbersome if the horizon's radius is greater than the pre-existing crack's thickness because the neighborhood search will include the nodes on the other side of the crack to the Nodes' Horizons. RBS open data structure allows the researcher to manipulate the neighborhood as it suits them. We can remove the Neighborhoods that one end is above the pre-existing crack while another end is below it. Utilizing the halfspace primary geometry, we can define the space above and below the pre-existing crack. We can then use the neighborhood centers' position status to see if they match the condition of being cut by the pre-existing crack. The following code illustrates the above steps for all of the plate part's Nodes.

```
const auto halfSpace = geometry::Primary::HalfSpace(space::Point<3>{0, 0, 0}, space::
    consts::j3D);
const auto& neighborhoods = platePart.neighborhoods();
std::for_each(neighborhoods.begin(), neighborhoods.end(), [&halfSpace, preCrackLength
    , horizonRadius](const Part::NeighborhoodPtr& neighborhood) {
```

Table 4.1: The material properties of plate with pre-existing crack.

| elastic modulus | Poisson's ratio | density |
|---|---|---|
| 192GPa | $\frac{1}{3}$ | $8000\frac{kg}{m^3}$ |

Table 4.2: The peridynamic simulation parameters.

| grid spacing $\Delta$ | horizon radius $\delta$ | critical stretch |
|---|---|---|
| $0.1mm$ | $3.015 \times \Delta$ | 0.04472 |

```
    const auto& centerPos = neighborhood->centre()->initialPosition().value<space::
        Point<3> >();
    const auto& centerPosVec = centerPos.positionVector();
    if (-preCrackLength / 2 < centerPosVec[0] && centerPosVec[0] < preCrackLength / 2
        && -horizonRadius * 2 < centerPosVec[1] && centerPosVec[1] < horizonRadius *
            2) {
        const auto centerStatus = halfSpace.pointStatus(centerPos);
        auto& neighbors = neighborhood->neighbors();

        neighbors.erase(
            std::remove_if(neighbors.begin(), neighbors.end(), [&halfSpace,
                centerStatus](const Part::NeighborhoodPtr& neighbor) -> bool{
                const auto neighborStatus = halfSpace.pointStatus(neighbor->centre()
                    ->initialPosition().value<space::Point<3> >());
                return neighborStatus != centerStatus || (neighborStatus ==
                    centerStatus && neighborStatus == geometry::On);
            }),
            neighbors.end());
    }
});
```

The bond force-stretch is a linear function with a cut-off at the negative and positive critical stretch (see Figure 4.7). The adaptation of functional programming by RBS becomes handy when such bond force-stretch is required to be implemented. The researcher can create a bond-based or ordinary state-based peridynamic and passing a bond force-stretch and volume correction method to the constitutive model. The following code presents the implementation of the bond force-stretch illustrated in Figure 4.7.

```
using BondBasedPeridynamic = relations::peridynamic::BondBased;
const auto bondForceStretch =
  [materialConstant, criticalStretch] (
    const double,
    const Vector& initial, const Vector& deformation,
    const HorizonPtr& centerHorizon, const HorizonPtr& neighborHorizon) -> Vector
  {
    if (deformation.isZero() || initial.isZero())
        return space::consts::o3D;

    if (centerHorizon->hasStatus(neighborHorizon, Property::Damage))
        return space::consts::o3D;
```

```
    const auto initialLength = initial.length();
    const auto currentBondVec = initial + deformation;
    const auto bondStretch = (currentBondVec.length() - initialLength) /
        initialLength;

    if ( -criticalStretch <= bondStretch && bondStretch <= criticalStretch )
        return materialConstant * bondStretch * (initial + deformation).unit();
    centerHorizon->setStatus(neighborHorizon, Property::Damage, int(1));
    return space::consts::o3D;
});

const auto constitutiveModel = BondBasedPeridynamic(
  bondForceStretch,
  BondBasedPeridynamic::VolumeCorrection,
  platePart, true);
```

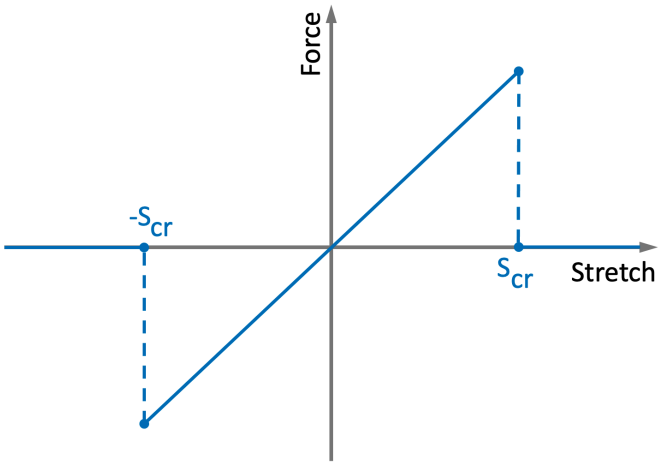


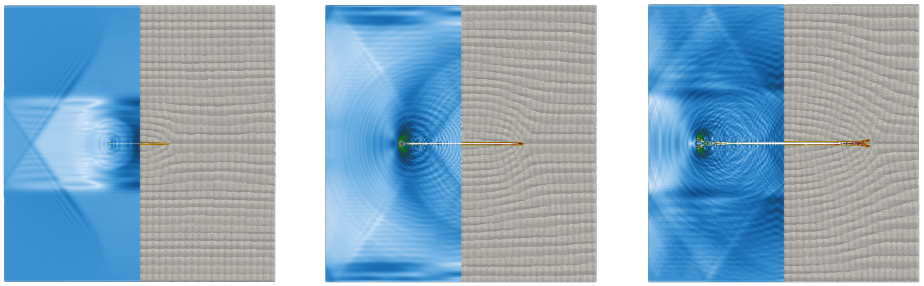

Figure 4.7: The bond force-stretch relation for the plate at Figure 4.6.

The same process as the example in the section 4.1 is employed to simulate the boundary condition and time integration scheme. Figures 4.8 and 4.9 illustrate the wave propagation and fracture growth for $20\frac{m}{s}$ and $50\frac{m}{s}$ constant velocity as boundary conditions, respectively.

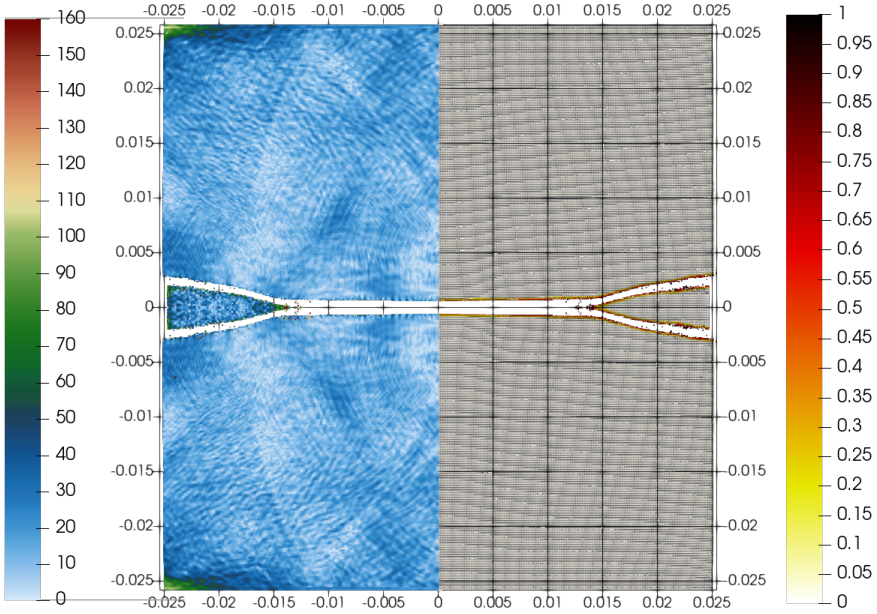
RBS logger is off by default. One can turn it on to store the process of the simulation in a LOG file.

```
auto& logger = report::Logger::centre();
logger.setFileLevel(logger.Debug,   path + "logs/", "Debug");
```

Part of the simulation log file can be found in Appendix B.

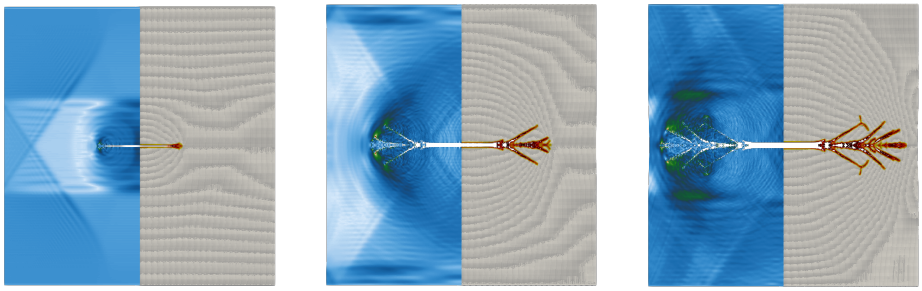


(a) $t = 450ns$ (b) $t = 720ns$ (c) $t = 900ns$

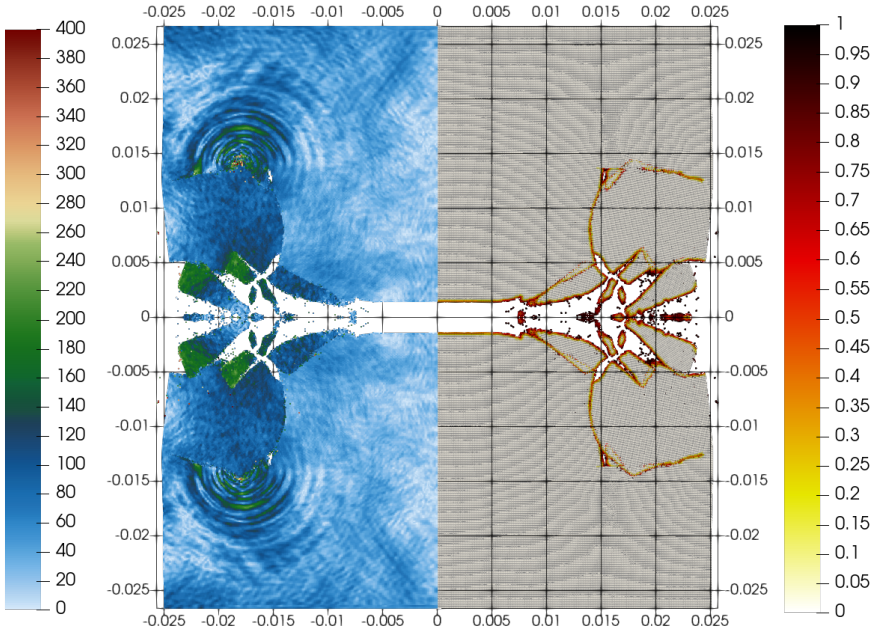



(d) $t = 1800ns$

Figure 4.8: The wave propagation (left half) and fracture growth (right half) for $20\frac{m}{s}$ constant velocity as boundary conditions applied to the plate illustrated in Figure 4.6.



(a) $t = 450ns$ (b) $t = 720ns$ (c) $t = 900ns$




(d) $t = 1800ns$

Figure 4.9: The wave propagation (left half) and fracture growth (right half) for $50\frac{m}{s}$ constant velocity as boundary conditions applied to the plate illustrated in Figure 4.6.

## 4.3 Polymer Fracture

A polymer matrix cube with a size-length of 10 000 $nm$ (volume of $10^{12}$ $nm^3$) is simulated. The polymer particles with a diameter of 800 $nm$ are randomly distributed within the matrix. Figure 4.10 illustrates a polymer matrix composite's initial configuration with 500 randomly located particles and boundary domains at the top and the bottom. The particles, the matrix, and the boundaries are assumed to have no imperfection. As explained in chapter 2, it is often the case to select the horizon radius three times bigger than the grid spacing. The polymer is under tension with the velocity of 25 $nm/s$ from top and bottom. Thus we need to impose boundary conditions (BC) by extending the domain over at least one horizon radius and then applying the equivalent BC to all nodes within this horizon (see [13]). Previously, in chapter 3, we called those domains '*boundary domains*' (BD). For implementing the boundary domains, two PDParts called boundary domains (BD) were created with the three layers of Nodes (i.e., horizon radius of the matrix). The BDs' Neighborhoods are then included in the matrix neighborhood and vice versa.

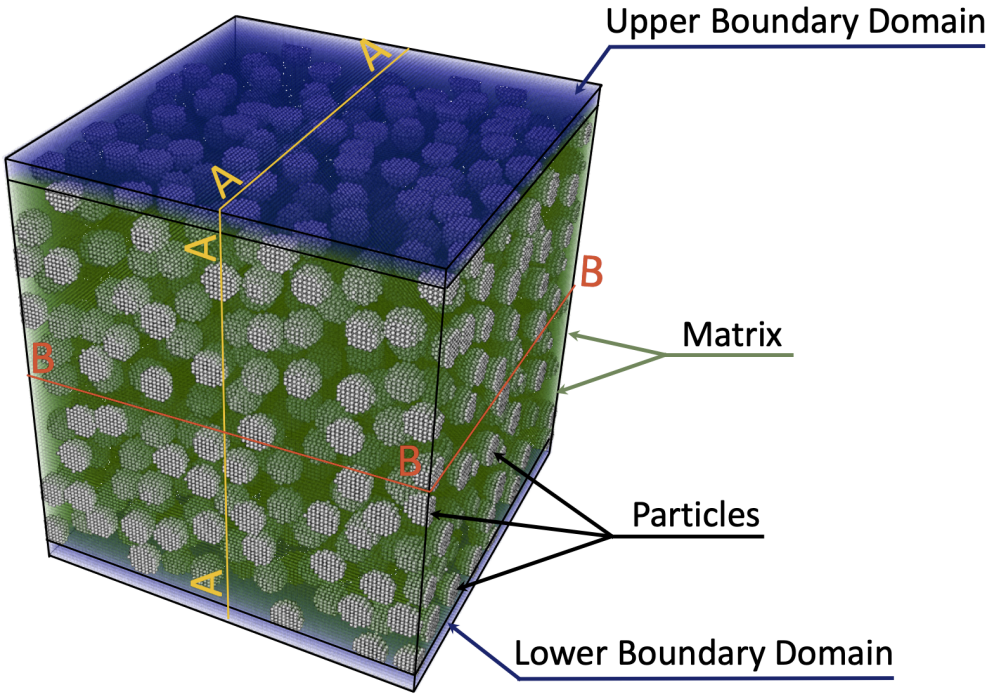



Figure 4.10: Initial configuration of a polymer matrix composite, 500 particles, and the boundary condition domains at two ends.

According to [41, 42], we assume a Poisson's ratio of 0.42, the density of 955 $gr/cm^3$, and elastic modulus equal to 1.035 $N/mm^2$. To be able to examine RBS

performance, different elastic modulus and volume fractions ($Vp$) for particles is considered. The $E_{Particle}/E_{Polyethylene}$ = 2, 3, 4, 5, 10, 100, and 1000 are selected and the $Vp$ varied in a range from 0.2%, 1.0%, 2.5%, 15%, 35%, up to 50% which will be correspond to 10, 50, 100, 500, 1000, and 1350 particles inside the matrix volume. Since the particles can be partially outside of the matrix cube, and since their position is random, most likely, we do not achieve the exact Vps as above.

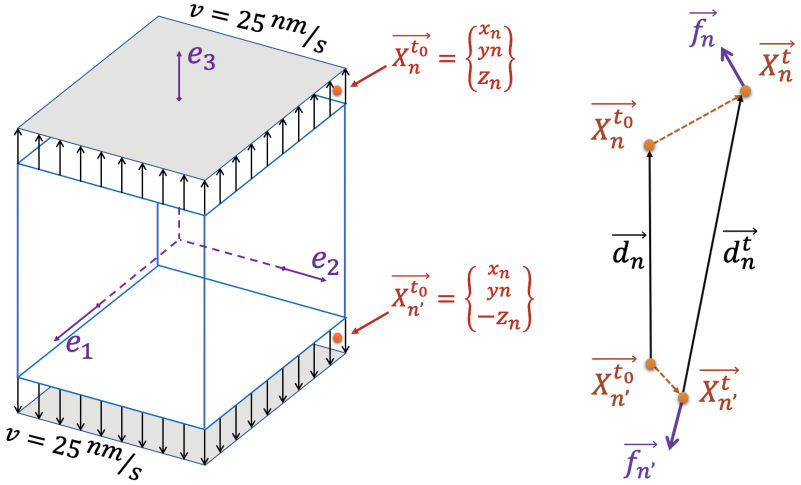



Figure 4.11: Schematic presentation of a) the applied velocity on the boundary domains and an example of counterpart nodes of the boundary domains, b) the displacement vectors between counterpart nodes of the boundary domains, and their reaction forces at initial configuration $t_0$ and after $t$ seconds.

A Schematic presentation of the problem state can be found in Figure 4.11a. We are interested in predicting the macroscopic uniaxial stress-strain curve, which can be extracted from measuring the boundary domain nodes' reaction forces and displacements. To achieve this, each upper domain node's displacements and reaction forces, along with their counterpart Node, should be computed in each time step. See node $n$ and its counterpart $n'$ in Figure 4.11a, which are located at $\vec{X}_n^t$ and $\vec{X}_{n'}^t$, respectively. The stresses should be computed from the reaction forces at the end of each time step as

$$\sigma_t = \frac{\sum_n f_n^t + f_{n'}^t}{A} \tag{4.1}$$

where the cross-sectional area $A$ is in our case equal to $10^8 nm^2$, and the reaction forces $f_n^t$ and their counterpart $f_{n'}^t$ are schematically illustrated in Figure 4.11b for node $n$ and its counterpart $n'$. The strain can be obtained by the quotient of the average change in

length of counterpart nodes difference vector ($\vec{d}^t$) in the z-direction as

$$\varepsilon_t = \frac{\sum_n^N \frac{(\vec{d}_n^t - \vec{d}_n)\cdot e_3}{\vec{d}_n \cdot e_3}}{N} \tag{4.2}$$

The particles elastic modulus assume to be 100 times the matrix elastic modulus. The material properties of the particle and matrix used in the simulations can be found in Table 4.3. The particle-particle bond and particle-boundary domain bonds have the same force-stretch relation as well as the matrix-matrix and matrix-boundary domain bonds. However, the particle-matrix bonds have a cut off force-stretch relation at the critical tensile stretch while it is elastic in compression. A Schematic presentation of the bond force-stretch relations can be found in Figure 4.12. A similar approach to the example presented in Section 4.2 has been taken to implement the ordinary state-based peridynamic constitutive model. It is worth mentioning that both BondBased and OrdinaryStateBased classes provide easy-to-use static constructors for defining linear bond force-stretch relations with and without cut-offs.

Table 4.3: All material parameters used in the polymer simulations

| Material | Polyethylene (Matrix) | Particles |
|---|---|---|
| Poisson's ratio $\nu$ | 0.42 | 0.42 |
| density $\rho$ | 0.955 $gr/cm^3$ | $\alpha\ \rho_{matrix}$ |
| Young's modulus $E$ | 1.035 $N/mm^2$ | $\alpha\ E_{matrix}$ |

$\alpha = 2, 3, 4, 5, 10, 100, 1000$

The required displacement and the reaction forces for evaluating the Eq. 4.1 and Eq. 4.2 have been done by searching a neighborhood centered at each node of the bottom BD containing one bond that, at another end, is a Node in the top BD (see Figure 4.13). Then, at the end of each time step, we computed the Eq. 4.1 and Eq. 4.2 and reported them in a CSV file. This procedure is particularly chosen to emphasize the flexibility of the RBS in implementing the complicated processes. The following steps have been taken.

1. Introducing a new subclass called StressStrainPart of Part and override its neighborhood search() method to fit the neighborhood search requirement and use it at initial configuration to find all of the Neighborhoods.

2. Introducing a new subclass of Relation with Time as Feeder and StressStrainPart as Feedee that evaluates the Eq. 4.1 and Eq. 4.2 and stores the stress-strain values of each pair inside the center Node of the bottom BD Neighborhoods.

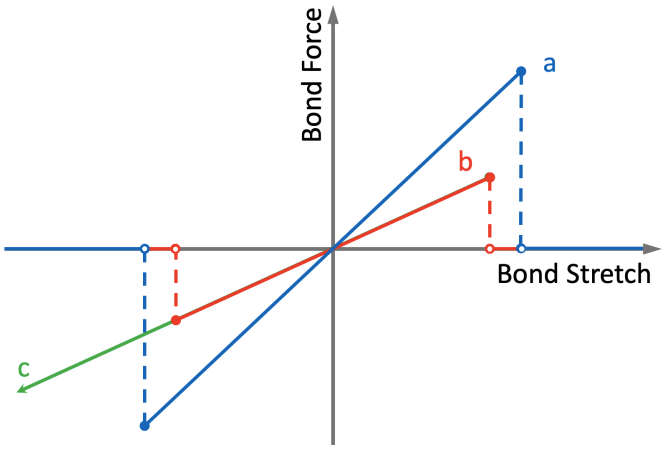



Figure 4.12: Schematic presentation of different force-stretch behavior of the bonds. a) particle-particle and particle-boundary bonds, b) matrix-matrix, and matrix-boundary bonds, c) particle-matrix bonds

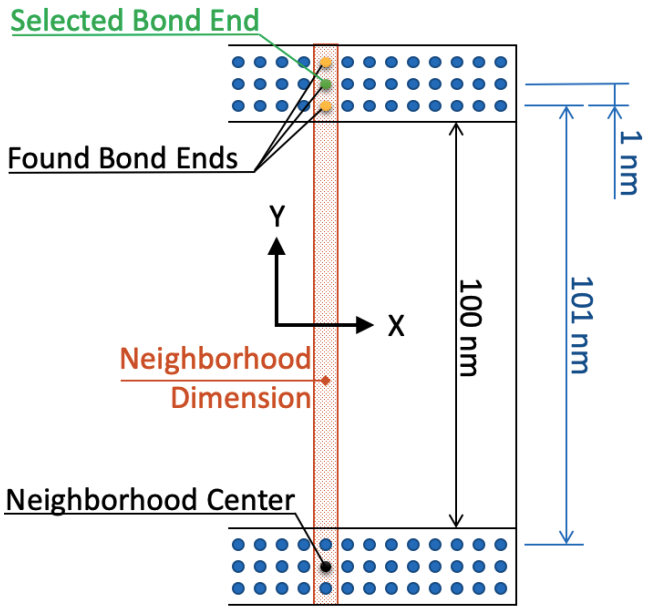


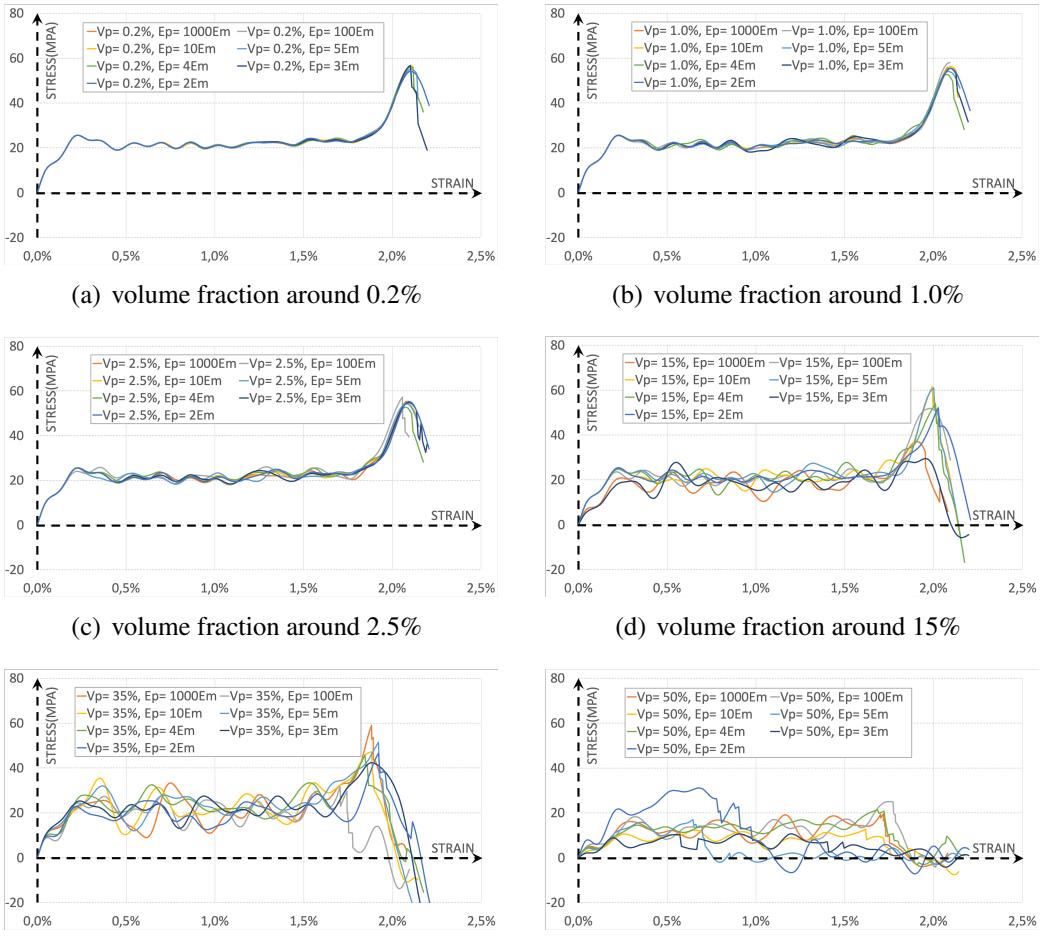

Figure 4.13: Schematic presentation of neighborhood search for computing the stress-strain diagram of the polymer specimen.

3. Introducing a new subclass of Relation with StressStrainPart as Feeder and CSV-File as Feedee to average the stresses and strains at each center Node of the bottom BD Neighborhoods and export them to the CSVFile.

4. Use the new defined Relations at each time step in the same sequence and after

the constitutive model's Relation and time integration scheme's Relation.

After the implementation of the above steps, the code provided the series of text files (with CSV format) exported, which was then added to Microsoft Excel to produce the diagram illustrated in Figure 4.14. The extracted curves show the influence of the particle density on polymer behavior. For particle densities below 2.5%, the stiffness barely influences the stress-strain curve. At a volume fraction of 15%, the influence gets more irrational while the stress-strain response for a volume fraction of 50% looks quite erratic and unrealistic.




(a) volume fraction around 0.2%

(b) volume fraction around 1.0%

(c) volume fraction around 2.5%

(d) volume fraction around 15%

(e) volume fraction around 35%

(f) volume fraction around 50%

Figure 4.14: The stress-strain curves for simulations with similar volume fraction but different particle-matrix stiffness ratios.

Though the response of each specimen is assumed to be linear, the illustrated macroscopic stress-strain behavior in Figure 4.14 is nonlinear. The nonlinear behavior

is caused by the fractured bonds, reducing the cross-sectional area, which finally yields to a reduced stiffness. Initially, the matrix fractures rapidly, causing a long plateau in the stress-strain curve. Once nearly all the matrix-bonds in a specific cross-section are broken, the stiffer particles carry the load resulting in the hardening behavior at strain levels around 2%. As long as the particles are not completely separated from the matrix, they absorb the load that cannot be carried by the matrix due to the broken bonds. The interface fracture separating the particles from the matrix finally causes the strain-softening observed in the latter part of the curve. This behavior is nearly independent of the volume fraction (and particle stiffness) since we did not vary the interface strength. Since the properties of the particle-matrix interface are identical for all simulations and since the interface bonds break before the particle-particle bonds, the stress-strain response for all simulations (with the same $V_p$) exhibit a similar behavior.

Table 4.4: Configurations of Computers hosting the polymer simulation.

| Operating system | macOS | openSUSE | Ubuntu |
|---|---|---|---|
| OS Version | 10.14 | 13.2 | 16.4 |
| Memory | 16 GB | 32 GB | 32 GB |
| Hard Drive | SSD | HDD | HDD |
| Processor | 1.6 GHz | 3.2 GHz | 3.2 GHz |

A grid spacing of 100 $nm$ utilized for the simulation, produced a total of 1 100 000 nodes and over 900 000 000 bonds in each model. Note that, due to the change of the stiffness of the system, the sound speed within the material differs between simulations, and so does the critical time step. The simulations are performed in three computer sets. The configuration of the computers and their operating system can be found in Table 4.4.

Table 4.5: RBS performance on different computer sets.

| | | Operating system | | |
|---|---|---|---|---|
| | Unit | macOS | openSUSE | Ubuntu |
| Number of simulations | - | 8 | 17 | 17 |
| Discritation time | milliseconds | 992 | 630 | 768 |
| Neighborhood search | seconds | 26 | 32 | 32 |
| Contact search | seconds per particle | 49 | 38 | 39 |
| Initial configuration | minutes | 89 | 74 | 72 |
| Time step | seconds | 194 | 188 | 190 |
| Exporting | seconds | 77 | 92 | 98 |

RBS produces a logging file during the simulation. The sole purpose of the logging files is to give the user a sense of what is currently under progress. An example of the logging files can be found in Appendix B. Note that the polymer simulation provides a relatively long logging section on each time iteration since there are more than 2000 OrdinaryStateBased Relations and 2000 TimeIntegration Relations. Each particle requires one OrdinaryStateBased Relation, one TimeIntegration Relation, and two OrdinaryStateBased Relations for interactions (i.e., matrix-particle and particle-matrix Relations), plus, two OrdinaryStateBased Relations for particle-boundary domain interactions for particles close to the boundary domains. One OrdinaryStateBased Relation and one TimeIntegration Relation for the matrix are required too. The logging files also provide us with the timing of the simulation. The times of each part of simulations are recorded, and the average of them on each computer can be found in Table 4.5. The RBS PD plugin can also export the configuration of each Part to the simulation to a text file, which can then be used to read the exported information of each Node and perform desired post-processing on them. RBS is also capable of exporting the configuration of the simulation to VTK format, which is a convenient format for ParaView software [1].

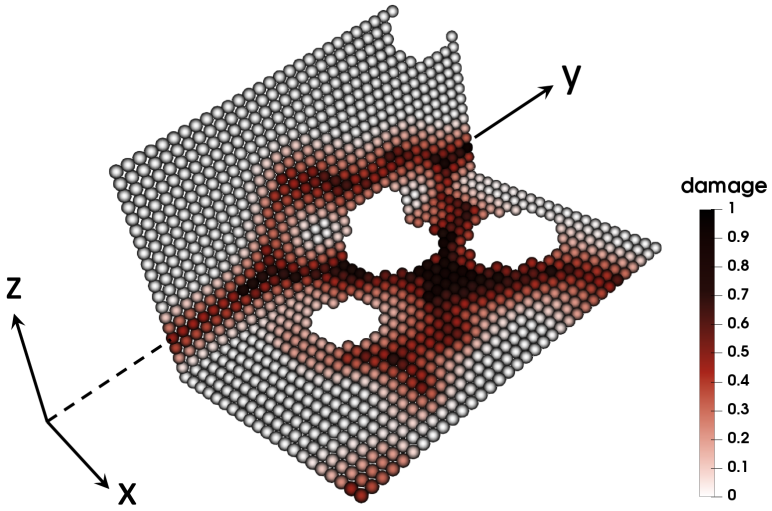



Figure 4.15: Effect of particle-matrix interface on the crack propagation

Because the particles have a higher stiffness than the matrix, the bonds between particles have a higher fracture critical stretch (compare Figures 4.12a and 4.12b). Consequently, in all the simulations, the crack path should not cross through the particles. Instead, the cracks must propagate around the particles. Figure 4.15 Illustrates the crack path in the $x-y$ and $y-z$ plane around three random particles for a simulation with $V_p \simeq 35\%$ and $E_{Particle}/E_{Matrix}$ = 2. The increase of the particle-particle

[1] https://www.paraview.org

bond stiffness did not affect the fracture propagation. A similar behavior is found in the experimental studies by Haque and Ali [43]. The particle-matrix interface forces the crack propagation to follow its path (i.e., the particles face), thus, by increasing the $V_p$, the particle-matrix interface failure causes random alternations in the plateau part of stress-strain responses. The observed erratic stress-strain behavior of the simulation with $V_p \simeq 50\%$ is a testament to this fact.

In the second half of the plateau of the stress-strain curve (see Figure 4.14), the cracks start to appear around the mid-cross-section. Subsequently, the cracks continue to propagate through the matrix. The second hardening of the stress-strain curve begins when the majority of the matrix-matrix bonds around the mid-cross-section are broken; the load is mainly carried by a combination of particle-particle and particle-matrix bonds. The simulation with $V_p \simeq 15\%$ and $E_{Particle}/E_{Matrix}$ = 100 is selected to present the damage distribution and crack propagation (see Figure 4.16).

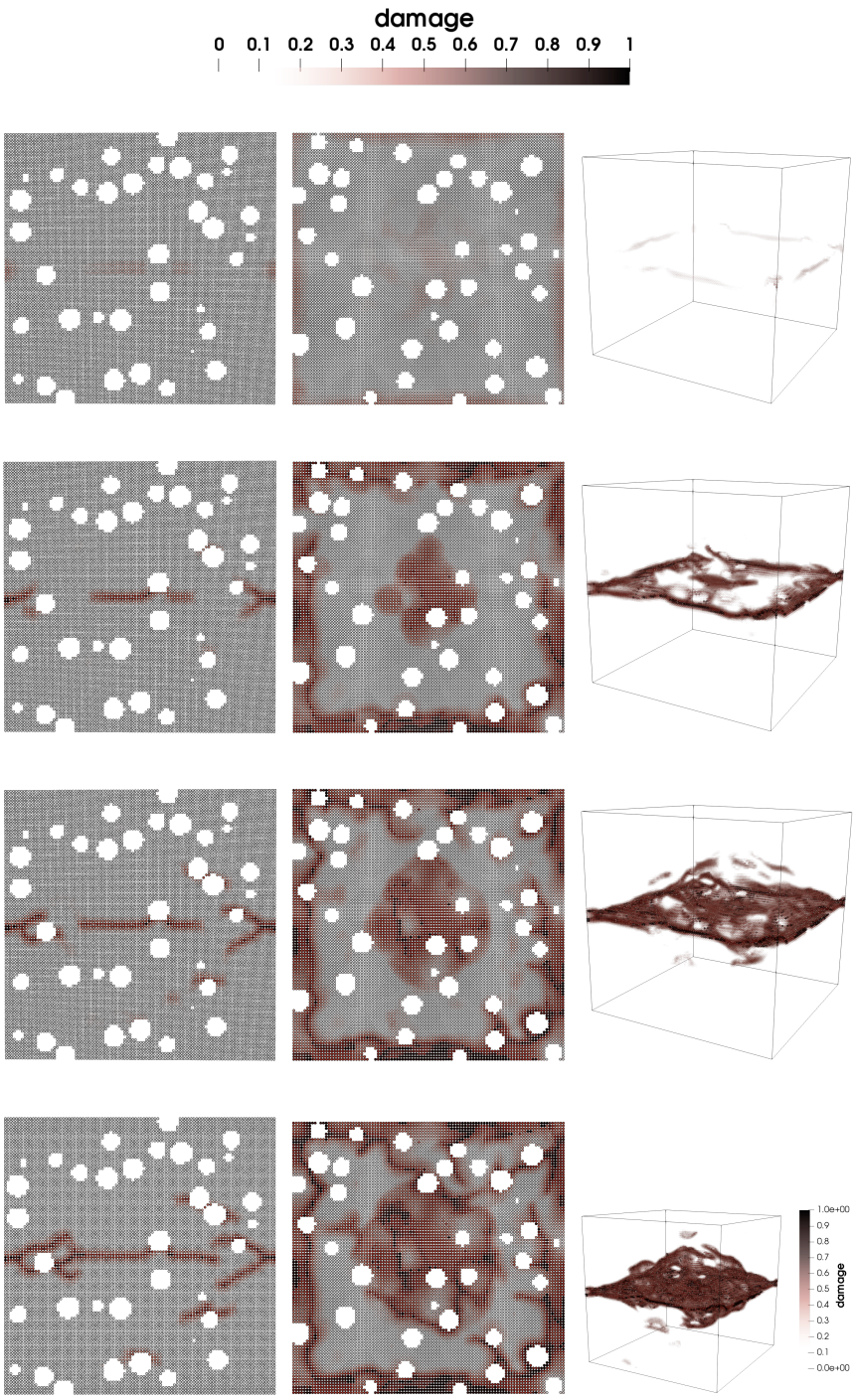



Figure 4.16: The crack distribution at the A-A cross-section (left side), and the B-B cross-section (middel), and 3D view (right side). The cross-sections are illustrated in Figure 4.10.

# Chapter 5

# Concluding Remarks and Future Prospects

The nature of physical problems are often nonlocal; the local methods that are traditionally developed simplifies those physical phenomena to reduce the problem to the less complex formulation. This simplification was necessitated due to our limited power of computation before computers became personal devices. The early works on implementing the computational methods clearly show the authors' concern about reducing the memory and computation cost as much as possible so that they can extend the size of their models. One of the approaches to reduce the computation costs was to refine the domain in the area of interest, which led to the introduction of the multiscale simulation. The refinement and multiscaling are still an open area of research. Several revolutions in computer hardware engineering during the 80s and early 90s introduced the new possibility to the programmers, which lead to the introduction of new software development methods and architecture. Those methods were gradually employed by researchers to model even more complex problems and to develop more expensive computational methods. The use of nonlocal models like molecular dynamics has been widely reported by researchers in the last three decades. The consideration of nonlocal methods in continuum problem has led to the introduction of Peridynamic, but by nature, it has also inherited high computational costs. The idea of refinement is also applicable to peridynamic, but due to the special integration form of the governing equation, refining PD in its natural way causes higher computation costs. The initial approaches of refining the peridynamic domain while using a smaller integration domain for its governing equation failed, due to the creation of so-called ghost forces. Ghost forces are responsible for introducing an artificial wave on the refinement bound. Several successful studies can be found in the literature. While some of those methods require changes to the constitutive model, others demand extra computational costs.

The reduction of the costs of the peridynamic simulation is the main focus of the

current study. Three main areas are addressed to achieve this goal. First, a new refinement method is proposed for reducing the computational costs by allowing smaller node spacing in some part of the problem domain while preserving the horizon-node spacing ratio. Second, a new software architecture developed to reduce the PD computation cost by considering the new programming paradigms and methods. Third, an opensource code called RBS is developed using the proposed architecture and by following agile development manifesto. RBS reduces the cost of implementing new constitutive methods or modification of the PD by separating the modules from each other. Consequently, the researcher can simply develop or modify an individual part of the code while utilizing the rest of the provided functionality securely. While accomplishing the above goals, the following achievements have also been obtained:

- The proposed refinement method eliminated the existence of so-called ghost forces.

- In the search for expanding the new refinement method to a higher dimension, the mesh sensitivity of the PD obtained is shown to be extremely effective. These effects are studied and reported.

- A simple approach for reducing the mesh sensitivity of the PD is proposed, and its effectiveness tested by applying them to to a crack propagation problem.

- The proposed approach for reducing the mesh sensitivity proved to have less computation cost than those available in the literature.

- The PD extended for simulating mesoscale problems using RBS's new features.

## 5.1 Future Research Prospects

The present thesis establishes a new approach to refinement of the PD and introduces new possibilities for further development of nonlocal nodal based simulations. A few possible extensions to the current work can be suggested as follows:

- Utilizing the new refinement method in a higher dimension requires complex geometrical solver. Such solvers are opensource and can be combined with the proposed scheme.

- The developed opensource software (RBS) does not support complex geometrical functionalies; extending it will allow implementation of the Multi-Horizon Peridynamic to higher dimensions.

- The smooth horizon method can address other reported areas where mesh sensitivity of PD dominates the simulations.
- The Relations in RBS provide a new opportunity to create a parallel time integration scheme since Relation can become the Feedee and Feeder of other Relations.

# References


[1] Stewart A Silling. Reformulation of elasticity theory for discontinuities and long-range forces. *Journal of the Mechanics and Physics of Solids*, 48(1):175–209, 2000. 1, 5, 6, 14

[2] Stewart A Silling, M Epton, O Weckner, J Xu, and E Askari. Peridynamic states and constitutive modeling. *Journal of Elasticity*, 88(2):151–184, 2007. 1, 8, 21, 38, 39

[3] Stewart A Silling and Ebrahim Askari. A meshfree method based on the peridynamic model of solid mechanics. *Computers and structures*, 83(17-18):1526–1535, 2005. 3, 8, 9, 37

[4] E Askari, F Bobaru, RB Lehoucq, ML Parks, SA Silling, and O Weckner. Peridynamics for multiscale materials modeling. In *Journal of Physics: Conference Series*, volume 125,1, page 012078. IOP Publishing, 2008. 3

[5] Florin Bobaru and Wenke Hu. The meaning, selection, and use of the peridynamic horizon and its relation to crack branching in brittle materials. *International journal of fracture*, 176(2):215–222, 2012. 3

[6] Florin Bobaru, Mijia Yang, Leonardo Frota Alves, Stewart A Silling, Ebrahim Askari, and Jifeng Xu. Convergence, adaptive refinement, and scaling in 1d peridynamics. *International Journal for Numerical Methods in Engineering*, 77(6):852–877, 2009. 3

[7] Huilong Ren, Xiaoying Zhuang, Yongchang Cai, and Timon Rabczuk. Dual-horizon peridynamics. *International Journal for Numerical Methods in Engineering*, 108(12):1451–1476, 2016. 3, 13, 22

[8] Florin Bobaru and Youn Doh Ha. Adaptive refinement and multiscale modeling in 2d peridynamics. *Mechanical and Materials Engineering Faculty Publications*, 2011. 3, 9

[9] SW Han, C Diyaroglu, S Oterkus, Erdogan Madenci, E Oterkus, Y Hwang, and H Seol. Peridynamic direct concentration approach by using ansys. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 544–549. IEEE, 2016. 3, 32

[10] Erdogan Madenci, Cagan Diyaroglu, and Nam D Phan. Ansys implementation of peridynamics for deformation of orthotropic materials. In *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, page 1463, 2018. 3

[11] R Beckmann, R Mella, and MR Wenman. Mesh and timestep sensitivity of fracture from thermal strains using peridynamics implemented in abaqus. *Computer methods in applied mechanics and engineering*, 263:71–80, 2013. 3

[12] Xiaohua Huang, Zhiwu Bie, Lifeng Wang, Yanli Jin, Xuefeng Liu, Guoshao Su, and Xiaoqiao He. Finite element method of bond-based peridynamics and its abaqus implementation. *Engineering Fracture Mechanics*, 206:408–426, 2019. 3, 32

[13] Erdogan Madenci and Erkan Oterkus. *Peridynamic theory and its applications*, volume 17. Springer, 2014. 3, 33, 72

[14] David John Littlewood, Michael L Parks, John Anthony Mitchell, and Stewart Andrew Silling. The peridigm framework for peridynamic simulations. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2013. 3, 31

[15] Michael L Parks, Pablo Seleson, Steven J Plimpton, Stewart A Silling, and Richard B Lehoucq. Peridynamics with lammps: a user guide, v0. 3 beta. *Sandia Report (2011–8253)*, page 3532, 2011. 4, 31

[16] Selda Oterkus, Erdogan Madenci, and Abigail Agwai. Fully coupled peridynamic thermomechanics. *Journal of the Mechanics and Physics of Solids*, 64:1–23, 2014. 6

[17] Florin Bobaru and Monchai Duangpanya. A peridynamic formulation for transient heat conduction in bodies with evolving discontinuities. *Journal of Computational Physics*, 231(7):2764–2785, 2012.

[18] C Diyaroglu, E Oterkus, S Oterkus, and Erdogan Madenci. Peridynamics for bending of beams and plates with transverse shear deformation. *International Journal of Solids and Structures*, 69:152–168, 2015. 6

[19] Steven F Henke and Sachin Shanbhag. Mesh sensitivity in peridynamic simulations. *Computer Physics Communications*, 185(1):181–193, 2014. 7, 20

[20] John T Foster, Stewart A Silling, and Weinong Chen. An energy based failure criterion for use with peridynamic states. *International Journal for Multiscale Computational Engineering*, 9(6), 2011. 8

[21] Pablo Seleson. Improved one-point quadrature algorithms for two-dimensional peridynamic models based on analytical calculations. *Computer Methods in Applied Mechanics and Engineering*, 282:184–217, 2014. 9

[22] Pablo Seleson and David J Littlewood. Convergence studies in meshfree peridynamic simulations. *Computers & Mathematics with Applications*, 71(11):2432–2448, 2016. 9

[23] Stewart A Silling. Linearized theory of peridynamic states. *Journal of Elasticity*, 99(1):85–111, 2010. 9

[24] David J Littlewood. Simulation of dynamic fracture using peridynamics, finite element modeling, and contact. In *ASME 2010 International Mechanical Engineering Congress and Exposition*, pages 209–217. American Society of Mechanical Engineers Digital Collection, 2010. 9

[25] Sheng-Wei Chi, Chung-Hao Lee, Jiun-Shyan Chen, and Pai-Chen Guan. A level set enhanced natural kernel contact algorithm for impact and penetration modeling. *International Journal for Numerical Methods in Engineering*, 102(3-4):839–866, 2015. 10

[26] Ted Belytschko, Wing Kam Liu, Brian Moran, and Khalil Elkhodary. *Nonlinear finite elements for continua and structures*. John wiley & sons, 2013. 10

[27] Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012. 10

[28] David John Littlewood, Timothy Shelton, and Jesse David Thomas. Estimation of the critical time step for peridynamic models. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2013. 11

[29] Daniele Dipasquale, Mirco Zaccariotto, and Ugo Galvanetto. Crack propagation with adaptive grid refinement in 2d peridynamics. *International Journal of Fracture*, 190(1-2):1–22, 2014. 13

[30] Marco Pasetto, Yu Leng, Jiun-Shyan Chen, John T Foster, and Pablo Seleson. A reproducing kernel enhanced approach for peridynamic solutions. *Computer Methods in Applied Mechanics and Engineering*, 340:1044–1078, 2018. 13

[31] P Lindsay, ML Parks, and A Prakash. Enabling fast, stable and accurate peridynamic computations using multi-time-step integration. *Computer Methods in Applied Mechanics and Engineering*, 306:382–405, 2016. 13

[32] Huilong Ren, Xiaoying Zhuang, and Timon Rabczuk. Dual-horizon peridynamics: A stable solution to varying horizons. *Computer Methods in Applied Mechanics and Engineering*, 318:762–782, 2017. 13

[33] Andris Freimanis and Ainars Paeglitis. Mesh sensitivity in peridynamic quasi-static simulations. *Procedia Engineering*, 172:284–291, 2017. 20

[34] Daniele Dipasquale, Giulia Sarego, Mirco Zaccariotto, and Ugo Galvanetto. Dependence of crack paths on the orientation of regular 2d peridynamic grids. *Engineering Fracture Mechanics*, 160:248–263, 2016. 20, 24, 29

[35] Paul Demmie and Stewart Silling. An approach to modeling extreme loading of structures using peridynamics. *Journal of Mechanics of Materials and Structures*, 2(10):1921–1945, 2007. 31

[36] Sierra Solid Mechanics Team. Sierra/solid mechanics 4.22 user's guide. *SAND2011-7597, Sandia National Laboratories*, 2011. 31

[37] Georg C Ganzenmüller, Martin O Steinhauser, Paul Van Liedekerke, and Katholieke Universtiteit Leuven. The implementation of smooth particle hydrodynamics in lammps. *Paul Van Liedekerke Katholieke Universiteit Leuven*, 1:1–26, 2011. 31

[38] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm users' guide v1. 0.0. *SAND Report*, 7800, 2012. 31

[39] JW Dally and SA Thau. Observations of stress wave propagation in a half-plane with boundary loading. *International Journal of Solids and Structures*, 3(3):293–308, 1967. 59, 65

[40] Vinesh V Nishawala, Martin Ostoja-Starzewski, Michael J Leamy, and Paul N Demmie. Simulation of elastic wave propagation using cellular automata and peridynamics, and comparison with experiments. *Wave Motion*, 60:73–83, 2016. 59, 61, 64, 65

[41] Werner Mueller and Ines Jakob. Oxidative resistance of high-density polyethylene geomembranes. *Polymer Degradation and Stability*, 79(1):161–172, 2003. 72

[42] N Kiass, R Khelif, L Boulanouar, and K Chaoui. Experimental approach to mechanical property variability through a high-density polyethylene gas pipe wall. *Journal of applied polymer science*, 97(1):272–281, 2005. 72

[43] A Haque and M Ali. High strain rate responses and failure analysis in polymer matrix composites–an experimental and finite element study. *Journal of composite materials*, 39(5):423–450, 2005. 79

# Appendix A

# RBS Codes

## A.1 Wave Propagation Simulation Code

The following is the main function for simulating the wave propagation example explained in detail in section 4.1.

```
1  //
2  //  WaveDispersionAndPropagation.cpp
3  //  Relation-Based Simulator (RBS)
4  //
5  //  Created by Ali Jenabidehkordi on 10.10.2020.
6  //  Copyright 2020 Ali Jenabidehkordi. All rights reserved.
7  //
8
9  #include "coordinate_system/grid.h"
10 #include "configuration/Part.h"
11 #include "relations/peridynamic.h"
12 #include <iostream>
13 #include <time.h>
14
15 using namespace rbs;
16 using namespace rbs::configuration;
17 using namespace std;
18
19 /**
20  * @brief Presents the construction process of work done by Nishawal et.al. on their paper:
21  *        "Simulation of Elastic Wave Propagation using Cellular Automata and Peridynamics, and Comparison with
        Experiments."
22  * @details: Find more information and step by step description on https://github.com/alijenabi/
        RelationBasedShttps://github.com/alijenabi/RelationBasedSoftware/blob/master/simulations/Elastic%20Wave%20
```

```
      Propagation%20in%20Plate/Elastic%20Wave%20Propagation%20in%20Plate.md
 * @note Examples are suitable for clang compiler. Running them suing other compilers may require modification.
 * @note Examples are suitable for mac file-system. Running them on other operating systems may require
      modification.
 * @return EXIT_SUCCESS if all examples are build successfuly, EXIT_FAILURE otherwise.
 */
int main() {

    const std::string path = "/<An existing folder path>/";
    auto& logger = report::Logger::centre();

    /* - Uncomment the code below if you like to log the simulation.- */
//    logger.setFileLevel(logger.Debug,   path + "logs/", "Debug");
//    logger.setFileLevel(logger.Timing,  path + "logs/", "Timing");
//    logger.setFileLevel(logger.Process, path + "logs/", "Process");
//    logger.setFileLevel(logger.Warning, path + "logs/", "Warning");

    using BC = report::Logger::Broadcast;
    logger.log(BC::Block, "Problem definition");

    const double dencity = 1300;            // kg / m^3
    const double youngsModulus = 3.85e9;    // GPa
    const double poissonRasio = 1. / 3.;

    const auto plateHeight = 0.5;           // 0.5 m
    const auto plateWidth  = 1.0;           // 1.0 m
    const auto plateThickness = 0.006655;   // 6.655 mm

    const auto horizonRasio = 3.;           // m = 3
    const auto gridSpacing = std::min({plateWidth / 1024, plateHeight / 512});
    const auto horizonRadius = horizonRasio * gridSpacing;

    logger.log(BC::Process, "Initiating the plate's Part.\n");
    using CS = coordinate_system::CoordinateSystem;
    auto platePart = Part("Plate", CS::Global().appendLocal(CS::Cartesian));

    logger.log(BC::Process, "Creating the part geometry.");
    {
        const auto plateShape = geometry::Combined::Cuboid({-plateWidth / 2, -plateHeight, -plateThickness / 2},
                                                           space::vec3{plateWidth, 0, 0},
                                                           space::vec3{0, plateHeight, 0},
                                                           space::vec3{0, 0, plateThickness});
        platePart.setGeometry(plateShape);
    }

    logger.log(BC::Block, "Meshing the part's coordinate system.");
    {
        const auto distanceVector = gridSpacing * space::consts::one3D;
        const auto startPoint = space::Point<3>{-plateWidth / 2, -plateHeight, -plateThickness / 2} +
              distanceVector / 2;
```

## A.1 Wave Propagation Simulation Code

```
70          const auto endPoint   = space::Point<3>{plateWidth / 2, 0, plateThickness / 2} - distanceVector / 2;
71
72          coordinate_system::grid::cartesian::uniformDirectional(startPoint, endPoint,distanceVector, platePart.
                local().axes());
73          platePart.local().axes()[2] = std::set<double>{0};
74      }
75
76      logger.log(BC::Block, "Including the points to the coordiante system.");
77      logger.log(BC::Process, "Including the grid points that are inside the part shape.\n");
78      {
79          const auto& plateShape = platePart.geometry();
80          platePart.local().include([&plateShape](const auto& localPoint) {
81              return plateShape.pointStatus(localPoint) == geometry::Inside;
82          });
83      }
84
85      logger.log(BC::Block, "Neighborhood search");
86
87      platePart.initiateNeighborhoods();
88
89      logger.log(BC::Process, "Adding the volume and the density.");
90      const double gridVolume = pow(gridSpacing, 2) * plateThickness;
91      const auto& neighborhoods = platePart.neighborhoods();
92      std::for_each(neighborhoods.begin(), neighborhoods.end(), [gridVolume, dencity](const Part::NeighborhoodPtr&
            neighborhood) {
93          using Property = relations::peridynamic::Property;
94          auto& centre = *neighborhood->centre();
95          centre.at(Property::Volume).setValue(gridVolume);
96          centre.at(Property::Density).setValue(dencity);
97      });
98
99      platePart.searchInnerNeighbors(horizonRadius);
100
101   /* - Uncomment the code below to check the configuration and neighborhood search checkpoints. - */
102 //    logger.log(BC::Block, "Exporting to VTK");
103 //    logger.log(BC::ProcessStart, "");
104 //    platePart.exportConfiguration(path + "vtks/");
105 //    platePart.exportCheckpointNeighborhoods(path + "vtks/");
106 //    logger.log(BC::ProcessEnd, "");
107
108     logger.log(BC::Block, "Defining the relations.");
109     const auto maxForcePerNode = 20.7e3 / ( plateThickness * gridSpacing );
110     const auto boundaryConditioner = [gridSpacing, maxForcePerNode](const double time, configuration::Node& node)
            {
111         using Property = relations::peridynamic::Property;
112         auto& postion = node.initialPosition().value<space::Point<3> >().positionVector();
113
114         if (-gridSpacing <= postion[0] && postion[0] <= gridSpacing && -gridSpacing * 2.1 < postion[1]) {
115             if (time <= 10e-6) {
116                 node.at(Property::Force) = -space::vec3{0, time * maxForcePerNode / 10e-6, 0};
```

```
117             } else if(time <= 20e-6) {
118                 node.at(Property::Force) = -space::vec3{0, maxForcePerNode - (time - 10e-6) * maxForcePerNode / 10
                        e-6, 0};
119             } else {
120                 node.at(Property::Force) = space::consts::o3D;
121             }
122         } else {
123             if (node.has(Property::Force))
124                 node.at(Property::Force) = space::consts::o3D;
125         }
126     };
127     auto load = relations::peridynamic::BoundaryDomain(boundaryConditioner, platePart);
128
129     auto timeIntegration = relations::peridynamic::time_integration::VelocityVerletAlgorithm(platePart);
130
131     const double bulkModulus = youngsModulus / (3 * (1 - 2 * poissonRasio));
132     const double materialConstant = 12 * bulkModulus / (M_PI * pow(horizonRadius, 3) * plateThickness);
133     logger.log(BC::Process, "Bulk Modulus = " + to_string(bulkModulus));
134     logger.log(BC::Process, "Material Constant = " + to_string(materialConstant));
135     auto platePDRelation = relations::peridynamic::BondBased::Elastic(materialConstant, gridSpacing, horizonRadius
          , platePart, false);
136
137     using P = relations::peridynamic::Property;
138     using On = relations::peridynamic::Exporter::Target;
139     const std::set properties = {P::Displacement, P::Velocity, P::Acceleration, P::Force};
140
141     auto plateExporterCC = relations::peridynamic::Exporter(properties, On::CurrentConfiguration, platePart, path
          + "vtks/", "PlatePartOnCurrentConfig");
142     plateExporterCC.setCondition([](const double, const size_t timeStep) {
143         return timeStep % 10 == 0;
144     });
145
146     const auto exportationCondition = [](const double, const size_t timeStep) {
147         return timeStep == 857;
148     };
149     auto plateExporterAt107ms = relations::peridynamic::Exporter(properties, On::InitialConfiguration, platePart,
           path + "vtks/", "plateExporterAt107ms");
150     auto plateExporterAt107msCurrent = relations::peridynamic::Exporter(properties, On::InitialConfiguration,
           platePart, path + "vtks/", "plateExporterAt107msCurrent");
151     plateExporterAt107ms.setCondition(exportationCondition);
152     plateExporterAt107msCurrent.setCondition(exportationCondition);
153
154     const auto maxSondSpeed = sqrt(youngsModulus / dencity);
155     const auto maxTimeSpan = std::min({0.125e-6, gridSpacing / maxSondSpeed * 0.8}); // .5 Safty Factor.
156     logger.log(BC::Process, "Maximum Sound Speed = " + to_string(maxSondSpeed));
157     logger.log(BC::Process, "Maximum Time Span = " + report::date_time::duration::formated(maxTimeSpan, 3));
158
159     auto& analysis = Analyse::current();
160     analysis.setTimeSpan(maxTimeSpan);
161     analysis.setMaxTime(308e-6);
```

## A.1 Wave Propagation Simulation Code

```
162     analysis.appendRelation(load);
163     analysis.appendRelation(platePDRelation);
164     analysis.appendRelation(timeIntegration);
165     analysis.appendRelation(plateExporterCC);
166     analysis.appendRelation(plateExporterAt107ms);
167     analysis.appendRelation(plateExporterAt107msCurrent);
168
169     return analysis.run();
170 }
```

# Appendix B

# RBS Progress Reports

## B.1 Fracture in Plate with Pre-existing Crack

The following is a part of the logging file with Debug RecieverLevel, which is saved on a file while simulating the fracture propagation on a plate with a pre-existing crack in section 4.2. The computer set is a mac whose characteristics can be found in Table 4.4.

```
1  ================================================================================
2
3                              Relation-Based Simulator (RBS)
4
5                              Version: 1.0.0
6                              Date: 12 February 2021, Friday
7                              Time: 21:45:37 PM +0100
8
9  ================================================================================
10
11 --------------------------------------------------------------------------------Task
12 |
13 |  * The broadcasted information with a higher or equal level to Debug will be writing to
14 |    the /documents/RBSExampels/fracture_in_plate_with_precrack_20/logs/Debug.log.
15 |-------------------------------------------------------------------| 327 microseconds
16
17 ------------------------------------------------------------------Problem_definition
18 |
19 |+- Initiating the plate's Part.
20 |-------------------------------------------------------------------| 363 microseconds
21
22 ---------------------------------------------------Meshing_the_part's_coordinate_system.
```

```
23 |
24 |-------------------------------------------------| 3 milliseconds and 192 microseconds
25
26 -------------------------------------------------Including_the_points_to_the_coordiante
27 |                                                                                system.
28 |
29 |---------------------------------------------------------| 1 second and 779 milliseconds
30
31 ---------------------------------------------------------------------Neighborhood_search
32 |
33 |+- Initiating "Plate" neighborhoods.
34 |+-- 253000 neighborhoods initiated.
35 |+--- Done in 4 seconds, 284 milliseconds, and 464 microseconds.
36 |+--- 16 CPU Clock per neighborhood.
37 |+--- 16 microseconds and 934 nanoseconds per neighborhood.
38 |+- Adding the volume and the density.
39 |+- Searching for inner neighbors of "Plate" Part.
40 |+-- 7300820 neighhors found.
41 |+--- Done in 2 minutes, 48 seconds, and 778 milliseconds.
42 |+--- 23 CPU Clock per neighhor.
43 |+--- 23 microseconds and 117 nanoseconds per neighhor.
44 |+- Removeing the bonds that are passing through the notch.
45 |+- 3600 bonds removed.
46 |-----------------------------------------------------------| 2 minutes and 54 seconds
47
48 ---------------------------------------------------------------Defining_the_relations.
49 |
50 |+- Young's Modulus = 192000000000.000000
51 |+- Material Constant = 401385714496010320317775872.000000
52 |+- Maximum Sound Speed = 4898.979486
53 |+- Maximum Time Span = 9 nanoseconds
54 |-------------------------------------------------------------------| 500 microseconds
55
56 ---------------------------------------------------------------Starting_the_Analyses
57 |
58 |-------------------------------------------------------------------| 47 microseconds
59
60 -------------------------------------------------------------------Time_Iteration_#0
61 |
62 |+- Analyse time: zero
63 |+- Applying boundary condition to the "Plate" Part.
64 |+-- 253000 Nodes updated.
65 |+--- Done in 81 milliseconds and 440 microseconds.
66 |+--- 0 CPU clock per Node.
67 |+--- 321 nanoseconds per Node.
68 |+- Applying bond-based peridynamic to "Plate" Part.
69 |+-- 253000 Nodes exported.
70 |+--- Done in 13 seconds, 247 milliseconds, and 19 microseconds.
71 |+--- 52 CPU clock per Node.
72 |+--- 52 microseconds and 359 nanoseconds per Node.
```

## B.1 Fracture in Plate with Pre-existing Crack

```
|+- Applying Euler Time-Integration to "Plate" Part.
|+-- 6000 node's properties updated.
|+--- Done in 849 milliseconds and 599 microseconds.
|+--- 141 CPU Clock per neighborhood.
|+--- or 141 microseconds and 599 nanoseconds per node's property.
|+- 253000 neighborhood updated.
|+-- Done in 849 milliseconds and 812 microseconds.
|+-- 3 CPU Clock per neighborhood.
|+-- or 3 microseconds and 358 nanoseconds per neighborhood.
|+- Exporting "Plate" Part neighbors.
|+-- to: /documents/RBSExampels/fracture_in_plate_with_precrack_20/vtks/CurrentConfig_0.vtk
|+-- 253000 Nodes exported.
|+--- Done in 43 seconds, 734 milliseconds, and 397 microseconds.
|+--- 172 CPU clock per Node.
|+--- 172 microseconds and 863 nanoseconds per Node.
|-------------------------------------------------------| 57 seconds and 913 milliseconds

------------------------------------------------------------------------------Time_Iteration_#1
|
|+- Analyse time: 13 nanoseconds
|+- Applying boundary condition to the "Plate" Part.
|+-- 253000 Nodes updated.
|+--- Done in 69 milliseconds and 575 microseconds.
|+--- 0 CPU clock per Node.
|+--- 275 nanoseconds per Node.
|+- Applying bond-based peridynamic to "Plate" Part.
|+-- 253000 Nodes exported.
|+--- Done in 12 seconds, 907 milliseconds, and 168 microseconds.
|+--- 51 CPU clock per Node.
|+--- 51 microseconds and 16 nanoseconds per Node.
|+- Applying Euler Time-Integration to "Plate" Part.
|+-- 18000 node's properties updated.
|+--- Done in 848 milliseconds and 538 microseconds.
|+--- 47 CPU Clock per neighborhood.
|+--- or 47 microseconds and 141 nanoseconds per node's property.
|+- 253000 neighborhood updated.
|+-- Done in 848 milliseconds and 792 microseconds.
|+-- 3 CPU Clock per neighborhood.
|+-- or 3 microseconds and 354 nanoseconds per neighborhood.
|-------------------------------------------------------| 13 seconds and 826 milliseconds
```

```
123
124
125
126
127 ------------------------------------------------------------------Time_Iteration_#1998
128 |
129 |+- Analyse time: 26 microseconds and 707 nanoseconds
130 |+- Applying boundary condition to the "Plate" Part.
131 |+-- 253000 Nodes updated.
132 |+--- Done in 68 milliseconds and 257 microseconds.
133 |+--- 0 CPU clock per Node.
134 |+--- 269 nanoseconds per Node.
135 |+- Applying bond-based peridynamic to "Plate" Part.
136 |+-- 253000 Nodes exported.
137 |+--- Done in 27 seconds, 755 milliseconds, and 228 microseconds.
138 |+--- 109 CPU clock per Node.
139 |+--- 109 microseconds and 704 nanoseconds per Node.
140 |+- Applying Euler Time-Integration to "Plate" Part.
141 |+-- 759000 node's properties updated.
142 |+--- Done in 1 second and 435 milliseconds.
143 |+--- 1 CPU Clock per neighborhood.
144 |+--- or 1 microsecond and 891 nanoseconds per node's property.
145 |+- 253000 neighborhood updated.
146 |+-- Done in 1 second, 436 milliseconds, and 1 microsecond.
147 |+-- 5 CPU Clock per neighborhood.
148 |+-- or 5 microseconds and 675 nanoseconds per neighborhood.
149 |----------------------------------------------------| 29 seconds and 260 milliseconds
150
151 ------------------------------------------------------------------Time_Iteration_#1999
152 |
153 |+- Analyse time: 26 microseconds and 720 nanoseconds
154 |+- Applying boundary condition to the "Plate" Part.
155 |+-- 253000 Nodes updated.
156 |+--- Done in 65 milliseconds, 878 microseconds, and 999 nanoseconds.
157 |+--- 0 CPU clock per Node.
158 |+--- 260 nanoseconds per Node.
159 |+- Applying bond-based peridynamic to "Plate" Part.
160 |+-- 253000 Nodes exported.
161 |+--- Done in 26 seconds, 879 milliseconds, and 912 microseconds.
162 |+--- 106 CPU clock per Node.
163 |+--- 106 microseconds and 244 nanoseconds per Node.
164 |+- Applying Euler Time-Integration to "Plate" Part.
165 |+-- 759000 node's properties updated.
166 |+--- Done in 1 second and 535 milliseconds.
167 |+--- 2 CPU Clock per neighborhood.
168 |+--- or 2 microseconds and 23 nanoseconds per node's property.
169 |+- 253000 neighborhood updated.
170 |+-- Done in 1 second, 536 milliseconds, and 70 microseconds.
171 |+-- 6 CPU Clock per neighborhood.
172 |+-- or 6 microseconds and 71 nanoseconds per neighborhood.
```

## B.1 Fracture in Plate with Pre-existing Crack

```
|-------------------------------------------------------| 28 seconds and 482 milliseconds

-----------------------------------------------------------------------Time_Iteration_#2000
|
|+- Analyse time: 26 microseconds and 733 nanoseconds
|+- Applying boundary condition to the "Plate" Part.
|+-- 253000 Nodes updated.
|+--- Done in 78 milliseconds and 799 microseconds.
|+--- 0 CPU clock per Node.
|+--- 311 nanoseconds per Node.
|+- Applying bond-based peridynamic to "Plate" Part.
|+-- 253000 Nodes exported.
|+--- Done in 29 seconds, 96 milliseconds, and 631 microseconds.
|+--- 115 CPU clock per Node.
|+--- 115 microseconds and 6 nanoseconds per Node.
|+- Applying Euler Time-Integration to "Plate" Part.
|+-- 759000 node's properties updated.
|+--- Done in 1 second and 429 milliseconds.
|+--- 1 CPU Clock per neighborhood.
|+--- or 1 microsecond and 883 nanoseconds per node's property.
|+- 253000 neighborhood updated.
|+-- Done in 1 second, 429 milliseconds, and 870 microseconds.
|+-- 5 CPU Clock per neighborhood.
|+-- or 5 microseconds and 651 nanoseconds per neighborhood.
|+- Exporting "Plate" Part neighbors.
|+-- to:
|    /documents/RBSExampels/fracture_in_plate_with_precrack_20/vtks/CurrentConfig_200.vtk
|+-- 253000 Nodes exported.
|+--- Done in 52 seconds, 286 milliseconds, and 431 microseconds.
|+--- 206 CPU clock per Node.
|+--- 206 microseconds and 665 nanoseconds per Node.
|-------------------------------------------------------------| 1 minute and 23 seconds

=============================================================================================

                            Relation-Based Simulator (RBS)

                            Version: 1.0.0
                            Date: 13 February 2021, Saturday
                            Time: 23:08:22 PM +0100

====================================| Finished after 18 hours, 52 minutes, and 2 seconds
```

# Academic Curriculum Vitae


## Ali Jenabidehkordi

Institute of Structural Mechanics
Bauhaus-University Weimar
Marienstrasse 15, 99423 Weimar, Germany
Email: ali.Jenabidehkordi@uni-weimar.de


## Education

- Ph.D : Computational Mechanics, Institute of Structural Mechanics, Bauhaus-University Weimar, Germany, 2013-2021.
- M.Sc: Structural Engineering, Urmia University, Iran, 2008-2012.
- B.Sc: Civil Engineering, Azad University, Iran, 2003-2008.

## Publications


1. A. Jenabidehkordi, R. Abadi, T. Rabczuk. *Computational modeling of meso-scale fracture in polymer matrix composites employing peridynamics*. Composite Structures, 253 (2020). doi:10.1016/j.compstruct.2020.112740.
2. A. Jenabidehkordi, T. Rabczuk, *The Multi-Horizon Peridynamics*. CMES-Computer Modeling in Engineering & Sciences, 121 (2019) 493–500. doi:10.32604/cmes.2019.07942.
3. A. Jenabidehkordi, *Computational methods for fracture in rock: a review and recent advances*. Frontiers of Structural and Civil Engineering, 13 (2019) 273-287. doi:10.1007/s11709-018-0459-5.