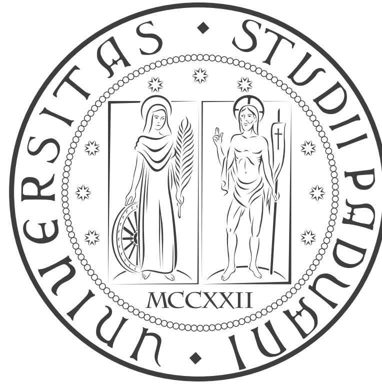


Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE
Corso di Laurea Magistrale in Ingegneria Aerospaziale

Study of computational peridynamics, explicit and implicit time integration, viscoelastic material

Tesi di Laurea Magistrale

Laureando: MARCO SPERONELLO - 1063630

Relatore: Prof. MIRCO ZACCARIOTTO
Correlatore: Prof. UGO GALVANETTO
Correlatore: Prof. ERKAN OTERKUS

Anno Accademico : 2014/2015

In collaboration with Professor Erkan Oterkus
Department of Naval Architecture and Marine Engineering
University of Strathclyde, Glasgow, UK

Abstract

This dissertation will focus on the peridynamic theory and more specifically on its numerical implementation. Peridynamic theory was first introduced in 2000 by S. A. Silling as a reformulation of the standard theory of solid mechanics. Its peculiarity concerns the use of an integral formulation instead of partial derivatives as in classical continuum mechanics. This allows for the peridynamic approach to be applied also in the presence of discontinuities in the material and during crack propagation, whereas partial derivatives are not defined.
This study will focus on the creation of a computationally efficient peridynamic solution code using the C++ programming language and will compare the performance of said code with a similar solution obtained using a code writeen in MATLAB.
Furthermore an implicit time integration scheme will be devised and it's performance will be tested against the already existing explicit time integration solution.
A formulation for a viscoelastic peridynamic material will be investigated and a viscoelastic solution code will be presented.
These codes will be used in the solution of some simple benchmark problems and the results will be compared against FEM results and analytic solutions, where possible.

Questa tesi tratta la teoria peridinamica e più specificatamente la sua implementazione numerica. La teoria peridinamica è stata introdotta nel 2000 da S. A. Silling come una riformulazione della teoria standard della meccanica dei solidi. La sua particolarità riguarda l'uso di una formulazione integrale piuttosto che alle derivate parziali, come invece avviene nella meccanica del continuo. Questo aspetto consente di applicare la teoria peridinamica anche in presenza di discontinuità nel materiale e durante la propagazione di cricche, dove invece non sono definite le derivate parziali.
Questo studio verterà sulla creazione di una soluzione numerica delle equazioni della peridinamica computazionalmente efficiente con l'uso del linguaggio C++ e sul confronto delle performance di questa soluzione con quelle ottenute da un simile codice scritto utilizzando il linguaggio MATLAB.
Inoltre sarà sviluppato uno schema di integrazione implicita e le sue performance saranno confrontate con quelle dei codici che eseguono l'integrazione esplicita.
Infine sarà analizzata una formulazione per la simulazione peridinamica di un materiale viscoelastico.
I programmi creati saranno utilizzati per la soluzione di alcuni semplici problemi di benchmark e i risultati verranno confrontati con quelli ottenuti ottenuti dal metodo FEM o da soluzione analitica, dove possibile.

Contents

1 Introduction **5**
1.1 The peridynamic theory . . . 5
1.2 Key features . . . 5
1.3 Present state . . . 6

2 Peridynamic theory **9**
2.1 Basic concept and notation . . . 9
2.1.1 Force density and stretch . . . 12
2.1.2 Strain energy density . . . 13
2.1.3 State notation . . . 13
2.1.4 Equation of motion . . . 14
2.2 Time integration . . . 15
2.2.1 Initial conditions . . . 15
2.2.2 Constraint conditions . . . 15
2.2.3 External loads . . . 16
2.3 Bond-based peridynamics . . . 17
2.4 Ordinary state-based peridynamics . . . 18
2.5 Material parameters for isotropic peridynamics . . . 19
2.5.1 1D structures . . . 21
2.5.2 2D structures . . . 21
2.5.3 3D structures . . . 21
2.5.4 Surface corrections . . . 21
2.5.5 Volume corrections . . . 25

3 Numerical solution with explicit time integration **27**
3.1 Programming languages . . . 27
3.2 Codes developed . . . 28
3.3 Program architecture . . . 28
3.3.1 Input values . . . 29
3.3.2 Initialization of position, displacement, volume and density matrices . . . 29
3.3.3 Coordinates of the material points . . . 30
3.3.4 Computation of total number of interaction and per-node number of interactions . . . 30
3.3.5 Interaction matrices initialization . . . 32
3.3.6 Relative position vectors . . . 32
3.3.7 Volume corrections . . . 32

3.3.8 Surface corrections 32
3.3.9 Peridynamic core calculations 33
3.3.10 Results elaboration and presentation 33
3.4 Benchmark problems 33
3.5 Spatial discretization 33
3.6 Explicit time integration 35
3.6.1 Integration algorithm 36
3.6.2 Application of initial and boundary conditions 37
3.6.3 Numerical stability 37
3.7 Codes developed for explicit time integration 37
3.8 Validation benchmarks results 37
3.8.1 1D benchmark 37
3.8.2 2D benchmark 39
3.8.3 3D benchmark 41
3.9 Performance analysis 43

4 Numerical solution with implicit time integration 51
4.1 Peridynamic equation of motion in implicit form 51
4.2 The Newmark-beta method for implicit integration 53
4.3 Broyden method for solving the system of equations 55
4.4 Algorithms for implicit time integration 56
4.5 Codes developed for implicit time integration 59
4.6 Validation benchmarks results 59
4.6.1 Newmark-beta benchmark results 60
4.6.2 Backward finite difference benchmark results 62
4.7 Comparison between Newmark-beta and backward finite difference implicit time integration performances 64
4.7.1 Iterative Broyden scheme performance 64
4.7.2 Computational time performance 65
4.8 Comparison between explicit and implicit time integration performances 67

5 Viscoelastic material simulation 71
5.1 Notation and definitions 71
5.2 Viscoelastic model 72
5.3 Time integration explicit algorithm 76
5.4 Benchmark results 78

6 Further developments 81

A Explicit time integration codes 83
A.1 MATLAB code 83
A.2 C++ code 85

B Implicit time integration codes 89
B.1 MATLAB Newmark-beta code 89
B.2 MATLAB backward finite difference code 93
B.3 C++ Newmark-beta code 97
B.4 C++ backward finite difference code 103

C Viscoelastic simulation codes **111**
C.1 MATLAB code 111

Chapter 1

Introduction

This chapter is intended to give an introduction to the peridynamic theory, it's key features and the specific advantages it can offer in comparison to the classical mechanics theory and the finite element method. The fields in which the theory can be applied is outlined and its present state of development is discussed in the following sections.

1.1 The peridynamic theory

The peridynamic theory is a non local theory, introduced in 2000 by S. A. Silling[14], that links the classical continuum mechanics to the molecular dynamics. The behaviour of a body is described throughout the interactions occurring between the points used to discretize the material.
In peridynamics the state of a material point is influenced by the other material points that lie within a region of finite radius called horizon. In the case of local theories these interaction are exerted at an infinitely small distance whereas in peridynamic theory the interactions are exerted non locally, within the horizon of the point, thus making it possible to consider also long-range effects.

1.2 Key features

The peridynamic theory allows to model the behaviour of bodies that present discontinuities and cracks without the need to resort to special crack growth criteria. The peridynamic governing equation's formulation makes use of integral of displacement instead of displacement derivatives, therefore it remains valid also in the presence of discontinuities and during crack initiation. The formation and propagation of cracks in a material is built into the theory[14] and doesn't require any special treatment, as instead it is demanded with the classical continuum mechanics theory.

In peridynamics the interaction between two material points is described with a response function that contains the constitutive law associated with the material. This formulation is suitable to be adapted to take into consideration various type of interactions such as interaction at the interfaces of different materials[5], study of

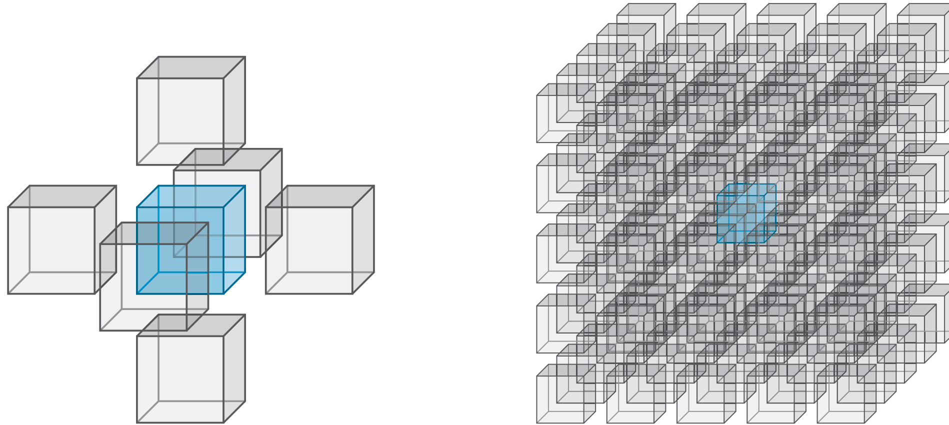


Figure 1.1: Graphic representation of interactions in local (left) and non-local (right) theories

composites behaviour[12], thermal effects and diffusion problems[6].

To obtain the solution of the peridynamic equation of motion it is necessary to calculate an integral in time and space. There are examples in literatures of analytical solution to the integro-differential peridynamic equation for simple problems. Silling used an analytical solution to solve the problem of an infinite bar subjected to a self-equilibrated load distribution[17]. A more general approach requires the numerical integration of the peridynamic equation of motion. This result can be achieved both with explicit and implicit time integration. The peridynamic equation of motion can also provide a solution to quasi-static problems by allowing the inertia term to reduce to zero[7].

1.3 Present state

Silling demonstrated in 2000[14] that the theory could correctly represent physical phenomena. In this first formulation of the theory, later called "bond-based peridynamics", the interaction between two family points is modeled as a pairwise function. This assumption requires the force between the two points to be of the same magnitude and opposite direction. A restriction on the material properties arises as a result of this formulation, requiring the Poisson's ratio to be one-forth for isotropic materials. The bond-based peridynamic theory does not distinguish between volumetric and distortional deformations, for this reason this formulation is not suited to account for plastic incompressibility or to use existing material models.

To overcome these limitations Silling et al. in 2007[15] introduced a more general formulation of the peridynamic theory, called "state-based peridynamics". This formulation is based on the concept of peridynamic state arrays: infinite dimensional arrays that contain the information pertaining the interactions between the points

of the material.

In addition to linear elastic materials, peridynamic theory has been shown to be able to account for various material behaviours such as nonlinear elastic[16], plastic[15][9], viscoelastic[10][5][19] and viscoplastic[19][3].

Peridynamic theory can also be used in coupling with the finite element method[8]. This approach can benefit from the different features of the two theories utilizing peridynamics, which is more suitable for crack prediction and damage assessment, only where a fracture in the material is expected. The FEM, which at the present level of development is more computationally efficient, can be used on the other parts of the structure. This technique results in accurate damage prediction while lowering the computational resources that a fully peridynamic simulation would require.

Peridynamic theory has been used with success in the solution of composite structures and the prediction of delamination damage and failure in composites[1]. A simple approach on this topic consists in assigning direction dependent properties to the bonds of a composite ply and modeling the interactions between the layers using inter-layer bonds.

Chapter 2

Peridynamic theory

This chapter is intended to introduce the basic notions underlying the peridynamic theory and to present the equations that are used in the numerical implementation in the following chapters. The peridynamic governing equation is presented and some of the terms that compose it are explained in further detail. For a more thorough understanding of the topic we refer to [13].

2.1 Basic concept and notation

Let us consider a body B composed of an infinite number of particles, called material points. Each particle is identified by its location in the undeformed body B given by the coordinates $\mathbf{x}_{(k)}$, where $k = 1, 2, ..., \infty$ denotes the k-th point of B, and is associated with a volume $V_{(k)}$ and a mass density $\rho(\mathbf{x}_{(k)})$.

When the body B is subjected to a deformation, each material point is subjected to a displacement, denoted as $\mathbf{u}_{(k)}(\mathbf{x}_{(k)}, t)$. This notation indicates that the displacement of a material point is a function both of position in the undeformed configuration and time. The location of the point in the deformed state is called $\mathbf{y}_{(k)}$. Equation 2.1 gives the relation between the displacement and the position of the point in deformed and undeformed configurations.

$$\mathbf{y}_{(k)} = \mathbf{x}_{(k)} + \mathbf{u}_{(k)} \tag{2.1}$$

Figure 2.1 gives a graphical representation of the position vector $\mathbf{x}_{(k)}$, the displacement vector $\mathbf{u}_{(k)}(\mathbf{x}_{(k)}$ and the deformed position vector $\mathbf{y}_{(k)}$.

Each material point can be subjected to a body load vector denoted as $\mathbf{b}_{(k)}(\mathbf{x}_{(k)}, t)$. The peridynamic theory states that the motion of the body B can be described analyzing the interaction between the infinite material points of which it is constituted. Each material points at $\mathbf{x}_{(k)}$ can interact with the material point at $\mathbf{x}_{(j)}$, where $j = 1, 2, ..., k-1, k+1, ..., \infty$ in the body B. The interaction between two material points, however, is supposed to vanish as the distance between the points in the

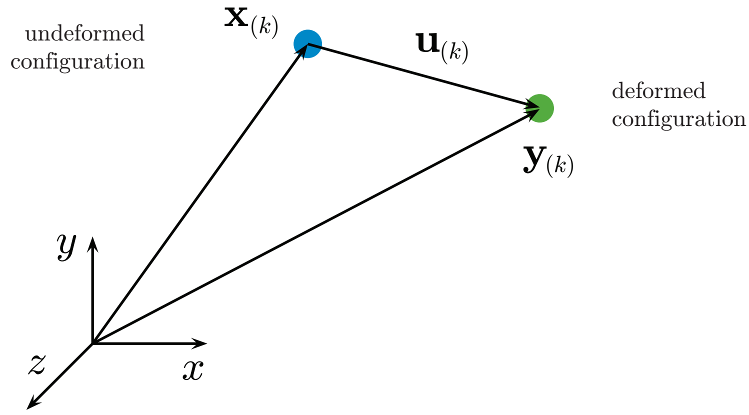


Figure 2.1: Representation of the position, displacement and deformed position vectors

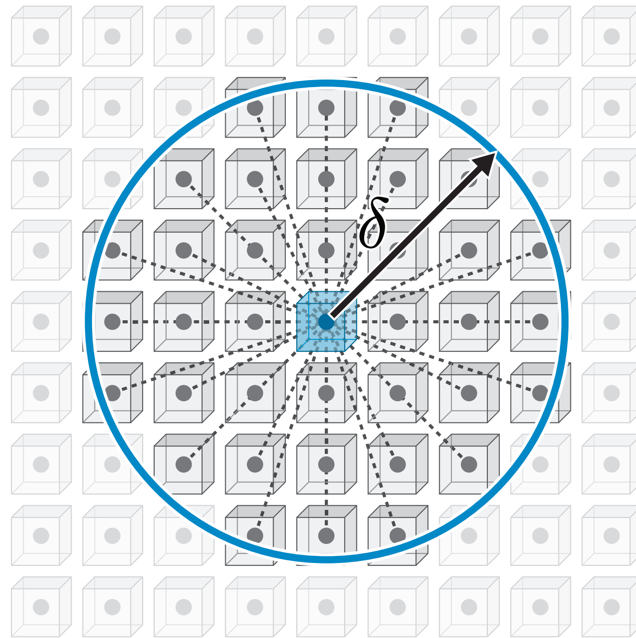


Figure 2.2: Graphic representation of the horizon $H_{\mathbf{x}_{(k)}}$ of radius δ of a point

undeformed configuration exceeds the radius of a local region, called horizon. The horizon surrounding $\mathbf{x}_{(k)}$ region is denoted as $H_{\mathbf{x}_{(k)}}$ and its radius is called δ. The value of δ is a parameter that represents the locality of the interactions. The interactions become more local as the value of δ decreases. For this reason the classical theory of elasticity can be considered as a limiting case of the peridynamic theory, as the value of δ that approaches zero. Only the material points $\mathbf{x}_{(j)}$ that lie within the horizon $H_{\mathbf{x}_{(k)}}$ can interact with the material point at $\mathbf{x}_{(k)}$. These points are referred to as the family nodes of $\mathbf{x}_{(k)}$, $H_{\mathbf{x}_{(k)}}$. The interaction between two material points is described with a function of the deformation and the constitutive properties of the material of the body.

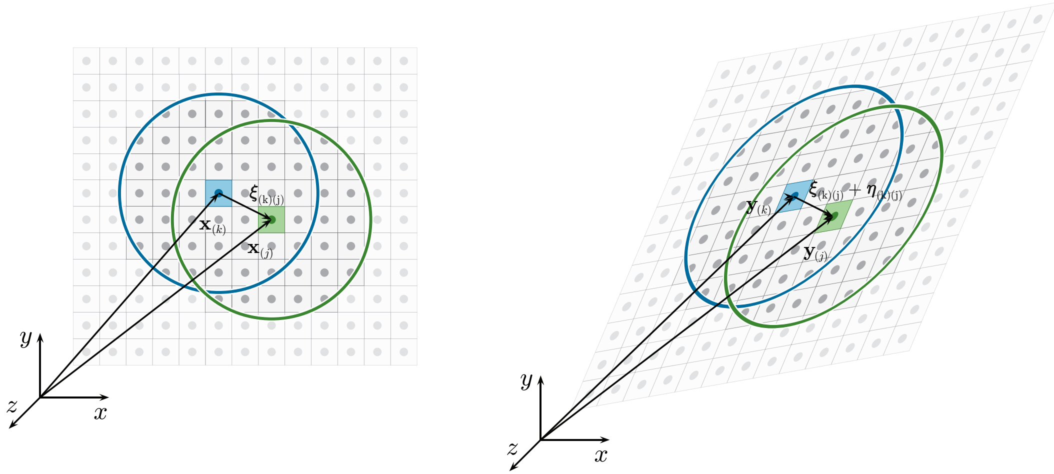


Figure 2.3: Graphic representation of the $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ vectors in deformed (left) and undeformed (right) configurations

As shown in figure 2.3 we introduce the $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ vectors, defined as described in equations 2.2 and 2.3, respectively:

$$\boldsymbol{\xi}_{(k)(j)} = \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \tag{2.2}$$

$$\boldsymbol{\eta}_{(k)(j)} = \mathbf{u}_{(j)} - \mathbf{u}_{(k)} \tag{2.3}$$

The $\boldsymbol{\xi}$ vector represents the relative position vector of the point at $\mathbf{x}_{(j)}$ with respect to the point at $\mathbf{x}_{(k)}$ in the undeformed configuration. The vector $\boldsymbol{\eta}$ represents the relative displacement vector of the same two points.
From equations 2.1 and 2.3 it follows:

$$
\begin{aligned}
\mathbf{y}_{(j)} - \mathbf{y}_{(k)} &= \mathbf{x}_{(j)} + \mathbf{u}_{(j)} - \mathbf{x}_{(k)} - \mathbf{u}_{(k)} \\
&= \boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}
\end{aligned}
\tag{2.4}
$$

2.1.1 Force density and stretch

Let us consider two material points $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$. According to the peridynamic theory points can mutually interact only if they lie within each other's horizon. That is if $\mathbf{x}_{(k)} \in H_{\mathbf{x}_{(j)}}$, which also implicates $\mathbf{x}_{(j)} \in H_{\mathbf{x}_{(k)}}$.

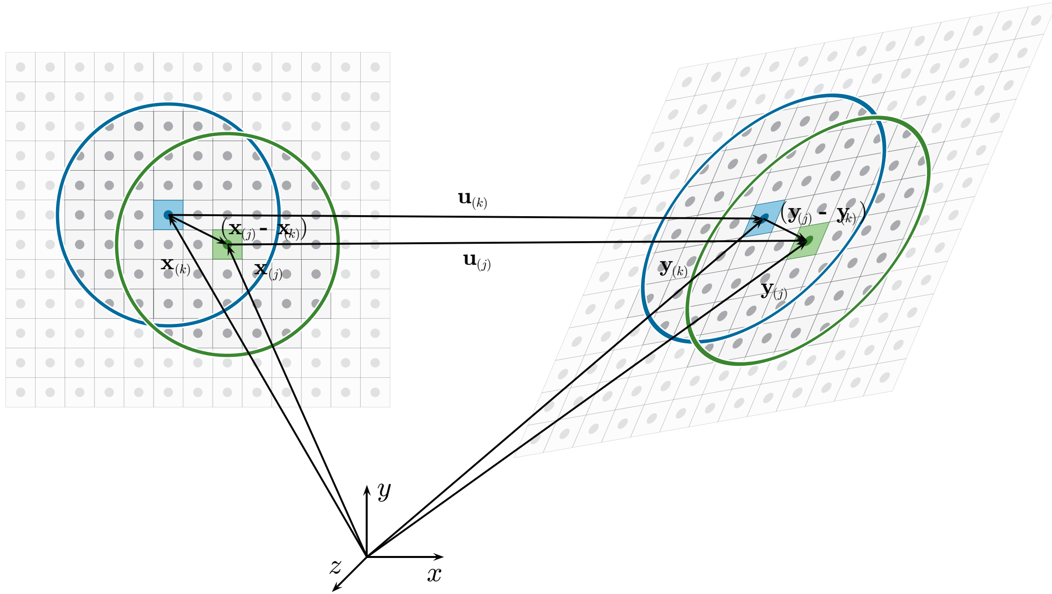


Figure 2.4: Graphic representation of interaction between points at $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$ in deformed (left) and undeformed (right) configurations

As shown in figure 2.4, the position of a material points in the deformed configuration is the result of all the interaction with the other points that lie within its horizon. This interaction is exerted throughout a force density vector called $\mathbf{t}_{(k)(j)}$, where the subscripts k and j indicate the action of the point at $\mathbf{x}_{(j)}$ on the point at $\mathbf{x}_{(k)}$.

We introduce the scalar variable stretch $s_{(k)(j)}$, defined as shown in equation 2.5:

$$s_{(k)(j)} = \frac{(|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}| - |\mathbf{x}_{(j)} - \mathbf{x}_{(k)}|)}{|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}|}$$
$$= \frac{(|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}| - |\boldsymbol{\xi}_{(k)(j)}|)}{|\boldsymbol{\xi}_{(k)(j)}|} \tag{2.5}$$

The stretch $s_{(k)(j)}$ contains information regarding how much the relative position of the points at $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$ has changed in the deformed configuration with respect to the undeformed configuration.

2.1.2 Strain energy density

The scalar function micropotential $w_{(k)(j)}$ is introduced to describe the interaction between the material points at $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$. This function depends on the properties of the material and the stretch between $\mathbf{x}_{(k)}$ and all the other points within its horizon. For this reason we can observe that the value of $w_{(k)(j)}$ is generally different from the value of $w_{(j)(k)}$ because the horizon of the two points does not coincide.

The strain energy density $W_{(k)}$, associated with the material point at $\mathbf{x}_{(k)}$, is expressed as show in equation 2.6:

$$W_{(k)} = \frac{1}{2} \sum_{\substack{j=1 \\ j \neq k}}^{\infty} \frac{1}{2} \left(w_{(k)(j)} + w_{(j)(k)} \right) V_{(j)} \tag{2.6}$$

2.1.3 State notation

Being $\mathbf{y}_{(k)}$ the position of the in the deformed configuration, the infinite-dimensional array $\underline{\mathbf{Y}}$, called deformation state, is introduced. This array contains the information of all the relative position vectors pertaining the deformed configuration. The deformation state associated with the material point at $\mathbf{x}_{(k)}$ is defined as shown in equation 2.7:

$$\underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t) = \left\{ \begin{array}{c} (\mathbf{y}_{(1)} - \mathbf{y}_{(k)}) \\ \vdots \\ (\mathbf{y}_{(\infty)} - \mathbf{y}_{(k)}) \end{array} \right\} \tag{2.7}$$

The force state $\underline{\mathbf{T}}$ is similarly defined. This infinite-dimensional array contains all the information regarding the force density vectors $\mathbf{t}_{(k)(j)}$ associated with the material point at $\mathbf{x}_{(k)}$ and is defined by equation 2.8:

$$\underline{\mathbf{T}}(\mathbf{x}_{(k)}, t) = \begin{Bmatrix} \mathbf{t}_{(k)(1)} \\ \vdots \\ \mathbf{t}_{(k)(\infty)} \end{Bmatrix} \tag{2.8}$$

As shown by the previous two equations, both the deformation and force state are time dependent arrays.
The information on the relative position vector $(\mathbf{y}_{(j)} - \mathbf{y}_{(k)})$ can be obtained by operating the the deformation state $\underline{\mathbf{Y}}$ as follows:

$$\begin{aligned} (\mathbf{y}_{(j)} - \mathbf{y}_{(k)}) &= \underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle \\ &= \underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t)\langle \boldsymbol{\xi}_{(k)(j)} \rangle \end{aligned} \tag{2.9}$$

Similarly the force density vector $\mathbf{t}_{(k)(j)}$ is obtained by operating the force state $\underline{\mathbf{T}}$ as follows:

$$\begin{aligned} \mathbf{t}_{(k)(j)}(\mathbf{u}_{(j)} - \mathbf{u}_{(k)}, \mathbf{x}_{(j)} - \mathbf{x}_{(k)}, t) &= \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle \\ &= \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \boldsymbol{\xi}_{(k)(j)} \rangle \end{aligned} \tag{2.10}$$

2.1.4 Equation of motion

Let us consider the kinetic energy T and the potential energy U in a body. The integration of the equation of the virtual work principle leads to the well known Lagrange's equation:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{\mathbf{u}}_{(k)}}\right) - \frac{\partial L}{\partial \mathbf{u}_{(k)}} = 0 \tag{2.11}$$

where L is defined as $L = T - U$.
Using the peridynamic expressions of T and U and substituting them into Lagrange's equation 2.11 we obtain the peridynamic equation of motion:

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int\limits_H \left[\mathbf{t}(\mathbf{u}' - \mathbf{u}, \mathbf{x}' - \mathbf{x}, t) - \mathbf{t}'(\mathbf{u} - \mathbf{u}', \mathbf{x} - \mathbf{x}', t)\right] \mathrm{d}H + \mathbf{b}(\mathbf{x},t) \tag{2.12}$$

and in state notation:

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int\limits_H \left[\underline{\mathbf{T}}(\mathbf{x},t)\langle\mathbf{x}' - \mathbf{x}\rangle - \underline{\mathbf{T}}(\mathbf{x},t)\langle\mathbf{x} - \mathbf{x}'\rangle\right] \mathrm{d}H + \mathbf{b}(\mathbf{x},t) \tag{2.13}$$

2.2 Time integration

Equations 2.12 and 2.13 describe how the material points of the body B interact with each other. The displacement $\mathbf{u}(\mathbf{x},t)$ is the result of this interaction. Analyzing the equations we also notice that they contain differentiation with respect to time and integration in a spatial domain. They indeed do not contain spatial derivatives, making them valid everywhere in the domain of the body, regardless of discontinuities that may occur, as previously anticipated.

As pointed out, the displacement is a function of the initial position of the points in the undeformed configuration and of external loadings and constraints. In order to proceed with the integration of equations 2.12 and 2.13 it is necessary to address the problem of how initial conditions, constraint conditions and external loads can be represented in the peridynamic theory.

2.2.1 Initial conditions

The time integration of $\ddot{\mathbf{u}}$ from equations 2.12 and 2.13 requires two initial conditions to be specified: one on the initial displacement and one on its first derivative:

$$\mathbf{u}(\mathbf{x}, t = 0) = \mathbf{u}^*(\mathbf{x}) \tag{2.14}$$

$$\dot{\mathbf{u}}(\mathbf{x}, t = 0) = \mathbf{v}^*(\mathbf{x}) \tag{2.15}$$

2.2.2 Constraint conditions

Even though constraint conditions are not required to solve the integral in equations 2.12 and 2.13, they might be imposed as external conditions on position or velocity of the boundary of the body.

Displacement constraint

Because there are no spatial derivates in the peridynamic equation of motion, displacement conditions can't be directly applied on the boundary of the body. In order to apply these condition a fictitious material region, called R_c, is added to the body and the displacement conditions are applied to its material points as follows:

$$\mathbf{u}(\mathbf{x}, t) = \mathbf{U_0} \qquad \text{for } \mathbf{x} \in R_c \tag{2.16}$$

Velocity constraints

Similarly to what shown in the previous paragraph, velocity constraints can be introduced using the same fictitious layer approach:

$$\dot{\mathbf{u}}(\mathbf{x}, t) = \mathbf{V_0}(t) \qquad \text{for } \mathbf{x} \in R_c \tag{2.17}$$

2.2.3 External loads

As with the constraint conditions, also external loads can't be directly introduced into the peridynamic equation of motion. External forces can be applied only as body forces acting on the material points. Therefore boundary forces cannot be applied on the boundary of a body but need to be applied on a layer of material with a finite thickness. Similarly with what done in the previous paragraph, a boundary layer, called R_l, of thickness Δ is added to the body.

In the case of a distributed pressure $p(\mathbf{x}, t)$, the equivalent peridynamic representation becomes:

$$\mathbf{b}(\mathbf{x}, t) = \frac{1}{\Delta} p(\mathbf{x}, t)\mathbf{n} \tag{2.18}$$

In the case of a point force $\mathbf{P}(t)$ acting on the surface S_l of the body, the equivalent peridynamic representation becomes:

$$\mathbf{b}(\mathbf{x}, t) = \frac{1}{S_l \Delta} \mathbf{P}(t) \tag{2.19}$$

2.3 Bond-based peridynamics

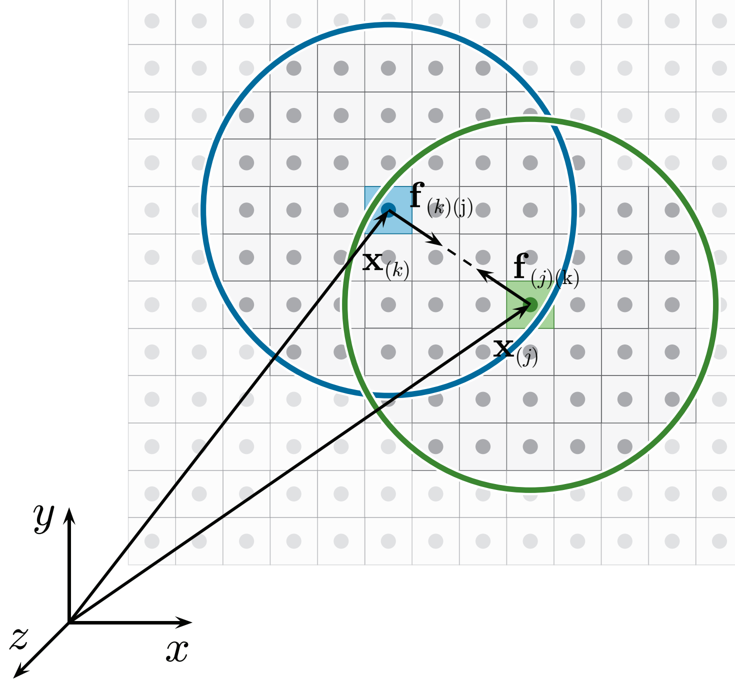


Figure 2.5: Graphic representation of the interaction between material points in the bond-based peridynamic theory

In bond-based peridynamic theory it is assumed that the interaction between two particles lying within each other's horizon is equal in magnitude and lying on the direction of the relative position vector, as shown in figure 2.5.

This approach guarantees the satisfaction of the balance of angular momentum. The interaction between two particles, being $\frac{\mathbf{y}'-\mathbf{y}}{|\mathbf{y}'-\mathbf{y}|}$ the direction of the vector, can be expressed as:

$$\begin{aligned} \mathbf{t}(\mathbf{u}'-\mathbf{u},\mathbf{x}'-\mathbf{x},t) = \underline{\mathbf{T}}(\mathbf{x},t)\langle\mathbf{x}'-\mathbf{x}\rangle &= \frac{1}{2}C\frac{\mathbf{y}'-\mathbf{y}}{|\mathbf{y}'-\mathbf{y}|} \\ &= \frac{1}{2}\mathbf{f}(\mathbf{u}'-\mathbf{u},\mathbf{x}'-\mathbf{x},t) \end{aligned} \tag{2.20}$$

and:

$$\mathbf{t}'(\mathbf{u}-\mathbf{u}',\mathbf{x}-\mathbf{x}',t) = \underline{\mathbf{T}}(\mathbf{x}',t)\langle\mathbf{x}-\mathbf{x}'\rangle = -\frac{1}{2}C\frac{\mathbf{y}'-\mathbf{y}}{|\mathbf{y}'-\mathbf{y}|} = -\frac{1}{2}\mathbf{f}(\mathbf{u}'-\mathbf{u},\mathbf{x}'-\mathbf{x},t) \tag{2.21}$$

The parameter C contains the properties of the material and will be discussed in the next sections.
Substituting equations 2.20 and 2.21 into the peridynamic equation of motion, given by 2.13, we obtain the equation of motion specific to the bond-based peridynamic:

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int_H \mathbf{f}(\mathbf{u}'-\mathbf{u},\mathbf{x}'-\mathbf{x},t)\,\mathrm{d}H + \mathbf{b}(\mathbf{x},t) \tag{2.22}$$

2.4 Ordinary state-based peridynamics

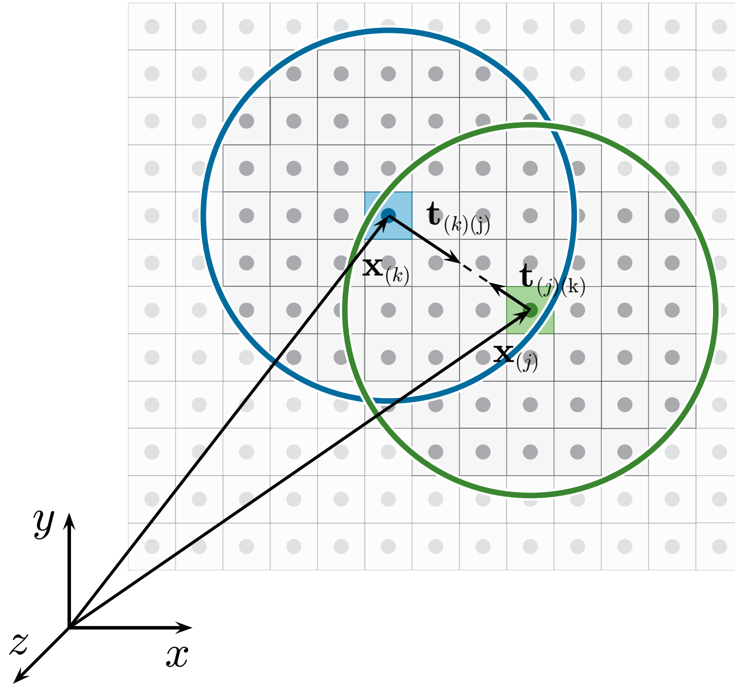


Figure 2.6: Graphic representation of the interaction between material points in the state-based peridynamic theory

In the state-based peridynamic theory it is assumed that the interaction between two particles lying within each other's horizon is still aligned to the direction of the relative position vector in the deformed state, as in bond-based peridynamic. However, the magnitude of the force vector that the point at $\mathbf{x}'$ exerts on $\mathbf{x}$ can differ from the magnitude of the force vector that $\mathbf{x}$ exerts on $\mathbf{x}'$. Figure 2.6 gives a representation of the interaction between two particles in the state-based peridynamics. The satisfaction of the balance of angular momentum is still guaranteed, as the force vectors are aligned on the same direction.

The force density vectors are respectively defined by:

$$\mathbf{t}(\mathbf{u}' - \mathbf{u}, \mathbf{x}' - \mathbf{x}, t) = \underline{\mathbf{T}}(\mathbf{x}, t)\langle \mathbf{x}' - \mathbf{x} \rangle = \frac{1}{2} A \frac{\mathbf{y}' - \mathbf{y}}{|\mathbf{y}' - \mathbf{y}|} \tag{2.23}$$

and:

$$\mathbf{t}(\mathbf{u}' - \mathbf{u}, \mathbf{x}' - \mathbf{x}, t) = \underline{\mathbf{T}}(\mathbf{x}, t)\langle \mathbf{x}' - \mathbf{x} \rangle = \frac{1}{2} B \frac{\mathbf{y}' - \mathbf{y}}{|\mathbf{y}' - \mathbf{y}|} \tag{2.24}$$

where A and B are parameters that depend on the material properties, the deformation field and the horizon. The state-based peridynamics allows to overcome the limitation on the Poisson's ration imposed by the bond-based theory.

2.5 Material parameters for isotropic peridynamics

The following equation gives the expression of the state-based force density vector:

$$\begin{aligned} \mathbf{t}_{(k)(j)} &= 2\delta \left\{ d \frac{\Lambda_{(k)(j)}}{|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}|} \left(a\theta_{(k)} - \frac{1}{2} a_2 T_{(k)} \right) + b \left(s_{(k)(j)} - \alpha T_{(k)} \right) \right\} \frac{\mathbf{y}_{(j)} - \mathbf{y}_{(k)}}{|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}|} \\ &= 2\delta \left\{ d \frac{\Lambda_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \left(a\theta_{(k)} - \frac{1}{2} a_2 T_{(k)} \right) + b \left(s_{(k)(j)} - \alpha T_{(k)} \right) \right\} \frac{\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}\right|} \end{aligned} \tag{2.25}$$

and for bond-based peridynamics:

$$\mathbf{t}_{(k)(j)} = 2\delta b \left(s_{(k)(j)} - \alpha T_{(k)}\right) \frac{\mathbf{y}_{(j)} - \mathbf{y}_{(k)}}{\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right|}$$
$$= 2\delta b \left(s_{(k)(j)} - \alpha T_{(k)}\right) \frac{\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}\right|} \tag{2.26}$$

Where the the parameter $\Lambda_{(k)(j)}$ is:

$$\Lambda_{(k)(j)} = \frac{\mathbf{y}_{(j)} - \mathbf{y}_{(k)}}{\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right|} \cdot \frac{\mathbf{x}_{(j)} - \mathbf{x}_{(k)}}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|}$$
$$= \frac{\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}\right|} \cdot \frac{\boldsymbol{\xi}_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \tag{2.27}$$

The $\theta_{(k)(j)}$ term is defined as:

$$\theta_{(k)} = d \sum_{j=1}^{N} w_{(k)(j)} \left(s_{(k)(j)} - \alpha T_{(k)}\right) \frac{\mathbf{y}_{(j)} - \mathbf{y}_{(k)}}{\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right|} \cdot \left(\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right) V_{(j)} + 3\alpha T_{(k)} \tag{2.28}$$

Where $T_{(k)}$ is the temperature change at point $\mathbf{x}_{(k)}$, α is the thermal expansion coefficient of the material and $w_{(k)(j)}$ is a non-dimensional influence function that depends on the distance between the points $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$ and regulates the influence of material points away from $\mathbf{x}_{(k)}$ defined as:

$$w_{(k)(j)} = \frac{\delta}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|}$$
$$= \frac{\delta}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \tag{2.29}$$

To evaluate the peridynamic equations it is necessary to know the value of the parameters a, a_2, a_3, b and d. In order to calculate them, a body is subjected to the simple loading conditions of isotropic expansion and simple shear. The strain components resulting from the loading conditions are used to determine the values of θ and W with the classical mechanics and the peridynamic theories. These values are then compared to obtain the peridynamic parameters as a function of the

engineering material constants.
This process is performed for the mono-dimensional, bi-dimensional and three-dimensional idealizations. In the following three sections are reported the values of the peridynamic coefficients obtained this way.

2.5.1 1D structures

For the mono-dimensional idealization the peridynamic parameters are:

$$a = a_2 = a_3 = 0 \qquad d = \frac{1}{2\delta^2 A} \qquad b = \frac{E}{2A\delta^3} \qquad c = \frac{2E}{\delta^2 A} \tag{2.30}$$

2.5.2 2D structures

For the bi-dimensional idealization the peridynamic parameters are:

$$a = \frac{1}{2}\left(\kappa - 2\mu\right) \quad a_2 = 4\alpha a \quad a_3 = 4\alpha^2 a \qquad d = \frac{2}{\pi h \delta^3} \qquad b = \frac{6\mu}{\pi h \delta^4} \tag{2.31}$$

2.5.3 3D structures

For the three-dimensional idealization the peridynamic parameters are:

$$a = \frac{1}{2}\left(\kappa - \frac{5}{3}\mu\right) \quad a_2 = 6\alpha a \quad a_3 = 9\alpha^2 a \qquad d = \frac{9}{4\pi\delta^4} \qquad b = \frac{15\mu}{2\pi\delta^5} \tag{2.32}$$

2.5.4 Surface corrections

The peridynamic parameters a, b and d, calculated in the previous sections, have been obtained with the assumption that the horizon of the point taken into account to calculate the dilatation and the strain energy density is fully contained within the body's boundary. This assumption can't be valid for all the material points in the body discretization. As shown in figure 2.7 there are some points whose horizon extends beyond the boundary of the body and therefore is incomplete.
Surface correction are necessary to avoid a factitious softening of the material properties that would occur in proximity of the body's boundary.
Since the configuration of the mesh is different for each geometry, it's impractical to try to achieve an analytical solution to this problem. To overcome this issue a surface correction is applied. The value of this parameter is obtained numerically

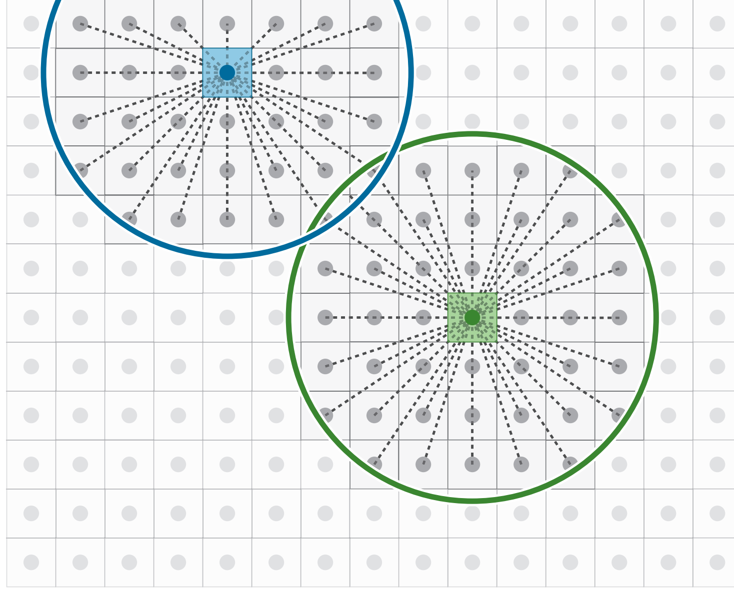


Figure 2.7: The blue node is close to the surface of the body and its horizon is incomplete, whereas the green node is inside the material and its horizon is complete

integrating the dilatation and the strain energy for a simple loading condition, this value is then compared to the one calculated from classical continuum mechanics.

The first simple loading condition considered is uniaxial stretch. The body is subjected to a uniform displacement gradient in all three directions x, y and z. The resulting displacement is then:

$$\mathbf{u}_1(\mathbf{x}) = \begin{Bmatrix} \frac{\partial u_x^*}{\partial x}x \\ 0 \\ 0 \end{Bmatrix} \qquad \mathbf{u}_2(\mathbf{x}) = \begin{Bmatrix} 0 \\ \frac{\partial u_y^*}{\partial y}y \\ 0 \end{Bmatrix} \qquad \mathbf{u}_3(\mathbf{x}) = \begin{Bmatrix} 0 \\ 0 \\ \frac{\partial u_z^*}{\partial z}z \end{Bmatrix} \tag{2.33}$$

Where $\frac{\partial u_\alpha^*}{\partial \alpha} = \zeta$, with $\alpha = x, y, z$, is the displacement gradient.
The peridynamic dilatation therm $\theta_m^{PD}(\mathbf{x}_{(k)})$ where $m = 1, 2, 3$ can be computed from equation 2.28:

$$\theta_m^{PD}(\mathbf{x}_{(k)}) = d\delta \sum_{j=1}^{N} s_{(k)(j)} \Lambda_{(k)(j)} V_{(j)} \tag{2.34}$$

While the corresponding classical continuum mechanics $\theta_m^{CM}(\mathbf{x}_{(k)})$ value is:

$$\theta_m^{CM}(\mathbf{x}_{(k)}) = \zeta \tag{2.35}$$

The dilatation correction factor in the m direction is then defined as:

$$D_{m(k)} = \frac{\theta_m^{CM}(\mathbf{x}_{(k)})}{\theta_m^{PD}(\mathbf{x}_{(k)})} = \frac{\zeta}{d\delta \sum_{j=1}^{N} s_{(k)(j)} \Lambda_{(k)(j)} V_{(j)}} \tag{2.36}$$

The second loading case considered is simple shear. The resulting displacement vectors are:

$$\mathbf{u}_1(\mathbf{x}) = \begin{Bmatrix} \frac{1}{2}\frac{\partial u^*_{x'}}{\partial y'} x \\ -\frac{1}{2}\frac{\partial u^*_{x'}}{\partial y'} y \\ 0 \end{Bmatrix} \quad \mathbf{u}_2(\mathbf{x}) = \begin{Bmatrix} 0 \\ \frac{1}{2}\frac{\partial u^*_{y'}}{\partial z'} y \\ -\frac{1}{2}\frac{\partial u^*_{y'}}{\partial z'} z \end{Bmatrix} \quad \mathbf{u}_3(\mathbf{x}) = \begin{Bmatrix} -\frac{1}{2}\frac{\partial u^*_{z'}}{\partial x'} x \\ 0 \\ \frac{1}{2}\frac{\partial u^*_{z'}}{\partial z'} z \end{Bmatrix} \tag{2.37}$$

Where $\frac{\partial u^*_{\alpha'}}{\partial \beta'} = \zeta$, with $\alpha \neq \beta \quad \alpha, \beta = x', y', z'$, is the displacement gradient, and x', y', z' are respectively oriented by an angle of -45^0 with respect to $(x-y)$, $(x-z)$ and $(y-z)$.
The peridynamic strain energy $W_m^{PD}(\mathbf{x}_{(k)})$ is obtained as follows:

$$W_m^{PD}(\mathbf{x}_{(k)}) = a\left(\theta_m^{PD}(\mathbf{x}_{(k)})\right)^2 + b\delta \sum_{j=1}^{N} \frac{1}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|} \left(\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| - \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|\right)^2 V_{(j)} \tag{2.38}$$

Where $m = 1, 2, 3$. The term $\theta_m^{PD}(\mathbf{x}_{(k)})$ is expected to vanish for this loading condition, therefore the peridynamic strain energy would become:

$$W_m^{PD}(\mathbf{x}_{(k)}) = b\delta \sum_{j=1}^{N} \frac{1}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|} \left(\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| - \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|\right)^2 V_{(j)} \tag{2.39}$$

The dilatation and strain energy density values in classical continuum mechanics notation are:

$$\theta_m^{CM}(\mathbf{x}_{(k)}) = 0 \qquad W_m^{CM}(\mathbf{x}_{(k)}) = \frac{1}{2}\mu\zeta^2 \tag{2.40}$$

Thus the correction term is:

$$S_{m(k)} = \frac{W_m^{CM}(\mathbf{x}_{(k)})}{W_m^{PD}(\mathbf{x}_{(k)})} = \frac{\frac{1}{2}\mu\zeta^2}{b\delta \sum_{j=1}^{N} \frac{1}{\left|\mathbf{x}_{(j)}-\mathbf{x}_{(k)}\right|} \left(\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| - \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|\right)^2 V_{(j)}} \tag{2.41}$$

The correction terms, computed for all the three directions, can be arranged in a vector, as shown in equation 2.42 and used as the principal values of an ellipsoid.

$$\mathbf{g}_{(d)}(\mathbf{x}_{(k)}) = \begin{Bmatrix} g_{x(d)(k)} \\ g_{y(d)(k)} \\ g_{z(d)(k)} \end{Bmatrix} = \begin{Bmatrix} D_{1(k)} \\ D_{2(k)} \\ D_{3(k)} \end{Bmatrix} \quad \mathbf{g}_{(b)}(\mathbf{x}_{(k)}) = \begin{Bmatrix} g_{x(b)(k)} \\ g_{y(b)(k)} \\ g_{z(b)(k)} \end{Bmatrix} = \begin{Bmatrix} S_{1(k)} \\ S_{2(k)} \\ S_{3(k)} \end{Bmatrix} \tag{2.42}$$

This approach allows to calculate the correction factors in all directions. Let us consider two material points $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$, $\mathbf{n}$ is the unit vector in the direction of their relative position, defined as follows:

$$\mathbf{n} = \frac{\mathbf{x}_{(j)} - \mathbf{x}_{(k)}}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|} = \begin{Bmatrix} n_x \\ n_y \\ n_z \end{Bmatrix} \tag{2.43}$$

The correction factors used as principal values of the ellipsoid for the $(k)(j)$ interaction are defined as the mean value of the respective correction factors:

$$\bar{\mathbf{g}}_{(\beta)(k)(j)} = \begin{Bmatrix} \bar{g}_{x(\beta)(k)(j)} \\ \bar{g}_{y(\beta)(k)(j)} \\ \bar{g}_{z(\beta)(k)(j)} \end{Bmatrix} = \frac{\mathbf{g}_{(\beta)(k)} + \mathbf{g}_{(\beta)(j)}}{2} \qquad \beta = d, b \tag{2.44}$$

The correction factor is then computed as follows:

$$G_{(\beta)(k)(j)} = \left(\left[\frac{n_x}{\bar{g}_{x(\beta)(k)(j)}} \right]^2 + \left[\frac{n_y}{\bar{g}_{y(\beta)(k)(j)}} \right]^2 + \left[\frac{n_z}{\bar{g}_{z(\beta)(k)(j)}} \right]^2 \right)^{-\frac{1}{2}} \tag{2.45}$$

Finally the expressions for the dilatation term $\theta_{(k)}$ and the strain energy density $W_{(k)}$ are modified as follows to account for the surface correction factors:

$$\theta_{(k)} = d\delta \sum_{j=1}^{N} G_{(d)(k)(j)} s_{(k)(j)} \Lambda_{(k)(j)} V_{(j)} \tag{2.46}$$

and:

$$\begin{aligned} W_{(k)} = a\theta_{(k)}^2 - a_2\theta_{(k)}T_{(k)} + a_3T_{(k)}^2 \dots \\ \dots + b\sum_{j=1}^{N} G_{(d)(k)(j)} \frac{1}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|} \left(\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| - \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|\right)^2 V_{(j)} \end{aligned} \tag{2.47}$$

2.5.5 Volume corrections

The peridynamic equation of motion 2.13 evaluated at point $\mathbf{x}_{(k)}$ contains the summation of all the volumes of the points $\mathbf{x}_{(j)}$ contained within its horizon. However, as shown in figure 2.8, some of the volume referring to nodes lying within $\mathbf{x}_{(k)}$ horizon is actually outside of it.

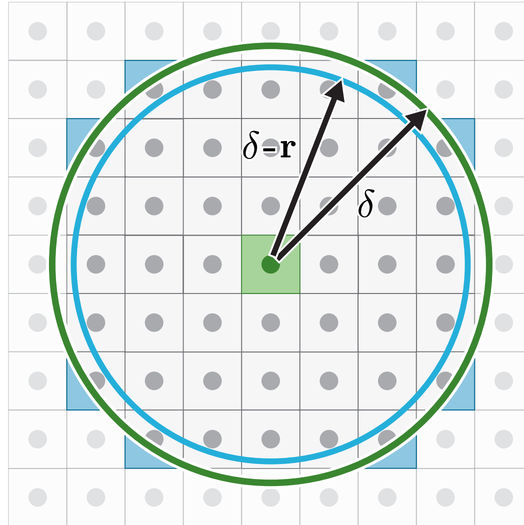


Figure 2.8: Method for identifying nodes that need volume corrections

For this reason it is necessary to correct the value of the volume $V_{(j)}$ to account for this eventuality:

$$\rho_{(k)}\ddot{\mathbf{u}}_{(k)} = \sum_{\substack{j=1 \\ j\neq k}}^{\infty} \left[\underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle - \underline{\mathbf{T}}(\mathbf{x}_{(j)}, t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle\right] \left(v_{c(j)}V_{(j)}\right) + \mathbf{b}_{(k)} \tag{2.48}$$

As shown in picture 2.8, the value of the correction $v_{c(j)}$ needs to be calculated for each of the nodes lying between the horizon and the sphere of radius $\delta - r$. For a uniform spacing between the nodes of value Δ the value of r is assumed to be equal to $\frac{\Delta}{2}$. For all the points $\mathbf{x}_{(j)}$, where $\delta - r < \xi_{(k)(j)} < \delta$, the volume correction is calculated as follows:

$$v_{c(j)} = \frac{\delta + r - \xi_{(k)(j)}}{2r} \tag{2.49}$$

As pointed out before, this technique for the volume correction calculation is valid only in presence of a uniform discretization. A different kind of mesh will need to be treated with a more sophisticated approach.

Chapter 3

Numerical solution with explicit time integration

In this chapter the current approach for explicit time integration solution scheme is presented. This method is then applied in different solution codes written in MATLAB and in C++ for mono, bi and three-dimensional idealization. The implementation of various techniques to improve the solver efficiency are discussed. The solution accuracy is tested with a benchmark problem. The performance of the two programming languages are analyzed.

3.1 Programming languages

The simulations are performed using two different programming languages: MATLAB, developed by MathWorks, and C++. The logic underlying the codes and the operations they perform is as similar as possible. However some tasks may be achieved in a slightly different way, due to the peculiarities of each of the programming languages.

The MATLAB programming language is an easy-to-use tool, designed specifically for numerical computation and to perform efficient matrix operations. The applications written in this language are executed inside the Matlab software, which is available for Microsoft Windows, Mac OS X and the Linux environments. The use of different operating systems may result in different execution performance.

C++ is an object oriented programming language, designed for high efficiency and performance. The current revision of this programming language, which is the one that is used in this study, is C++ 11. Even though there are some OS specific C++ tools and hardware specific APIs that can be implemented, a program written in standard C++ would be compatible with almost all the operating systems, once compiled for the target OS. Once again it is necessary to acknowledge that different operating systems may result in different performance, even when they are being executed on the same hardware.

3.2 Codes developed

With the purpose of creating a working tool for solution of the peridynamic equations, several codes are developed and tested. Bond-based and ordinary state-based peridynamics are implemented in a single program, as state-based peridynamics can revert to bond-based when the poisson ration is equal to 0.25.

All the codes developed allows for the solution of tridimensional problems. The tridimensional implementation is also suitable for the solution of mono and bidimensional idealizations. No change to the structure of the programs is required, as only the value of the peridynamic parameters and a variable that sets the number of dimensions of the problem need to be changed.

3.3 Program architecture

The programs are divided in sequential tasks. The workflow, illustrated in figure 3.1 and explained in the following sections, is relatively common to all the codes that are presented. The different features of the two programming languages result, on occasion, in slightly different code implementation.

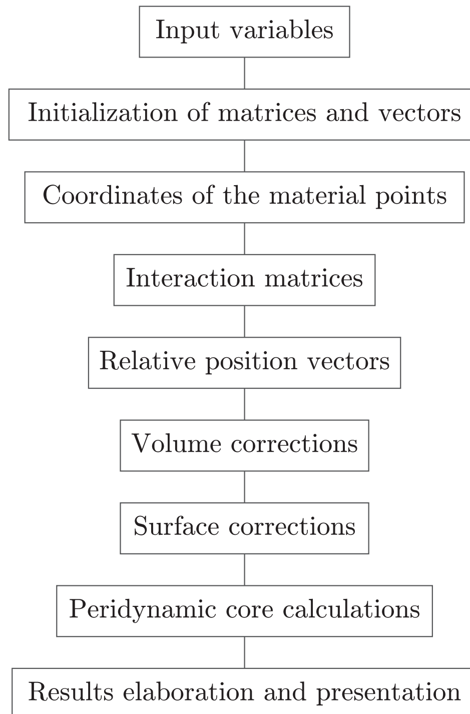


Figure 3.1: The tasks performed by the programs

3.3.1 Input values

The input values of the simulation are found in the first section of the codes. These values can be pre-assigned to variables or require a user input. They include the geometric parameters of the discretization, the material parameters, the peridynamic parameters and the integration parameters. Here the number of dimensions used in the idealization is also defined throughout a specific variable.

3.3.2 Initialization of position, displacement, volume and density matrices

The initialization of the vectors and the matrices used to store the data of the simulations is necessary to allocate the required memory and to speed up the execution of the script. Some of the variables are initialized in this section of the script, some others will be initialized in the following stages because the dimension of the required memory is yet to be known at this stage.
The C++ and MATLAB languages differ in the way they handle matrices. While MATLAB allows the initialization and operation between bi-dimensional matrices, in C++ there's no "proper" matrix in the way we are used to think about it. What is used instead is memory locations that can be used to store values or vectors. While this approach is less intuitive and slightly more difficult to utilize than the one provided by MATLAB, it offers a much greater control over memory allocation, thus allowing to allocate exactly just the right amount of memory required in a very easy way. In figure 3.2 it is graphically shown how the two programming languages can handle the storage of data in matrices. The blue dots represents a data value, it is possible to see that in order to store the same quantity of data MATLAB allocates also a portion of memory that is then left unused.

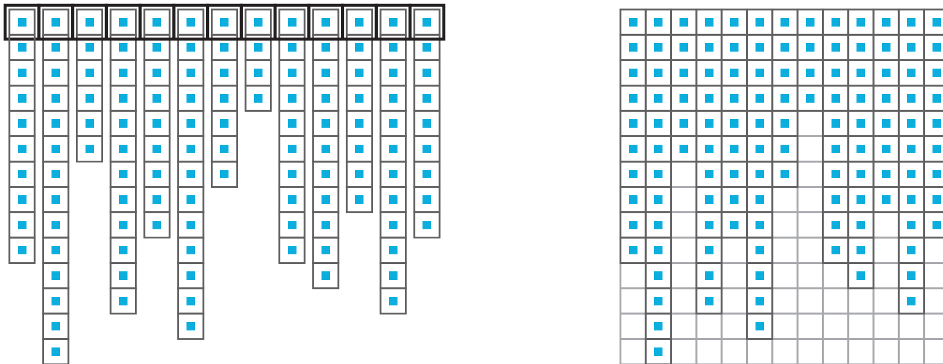


Figure 3.2: On the left a representation of how it is possible to store data in a matrix-like configuration. On the right is the way MATLAB allows to store data in a matrix.

The bigger the number of nodes in the simulation and the most severe the problem of over-allocated memory can become. In order to solve this inconvenience, the use of cell in MATLAB has been investigated. This option was discarded due to poor performance in the script execution time.

Even though C++ seems to be superior when it comes to efficient memory handling, MATLAB has some built-in features for matrix operations that are very easy to implement and is specifically designed for high performance matrix calculus.

3.3.3 Coordinates of the material points

In this phase of the program the coordinates of the nodes of the discretization are calculated and stored in the pre-initiated matrix. The operation performed in this section are specific to the geometry of the problem. Different geometries or idealization would result in different implementations.

node ID →	1	2	3	4	5	6	7	8	9	10
coordinates	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}

Figure 3.3: Matrix containing the nodes coordinates

Each node of the discretization is numbered with a unique identification number and that ID is used to identify it's coordinates in the position matrix, as shown in figure 3.3.
The coordinates of the nodes may be provided by and external file or database.

3.3.4 Computation of total number of interaction and per-node number of interactions

A sweep throughout all the nodes of the discretization is performed, in order to calculate the relative position vectors for all the interactions. For each node is checked whether the distance between the node, identified as (k) and the neighbouring nodes, identified as (j), is less than the value of δ, the horizon dimension. If this is true the node ID is added to a matrix where the ID of the interacting nodes are stored.

As shown in figure 3.4, the number of the column identifies the ID of the k-th node, while on the rows are stored the IDs of the j-th nodes. This matrix is used to identify the nodes that can interact with each other.

A progressive ID number is assigned to each interaction. As shown in figure 3.5, the matrix that stores the interaction IDs is structured in the same way as the matrix shown in figure 3.4.

The combined information contained in these two matrices allows to identify the nodes interacting with each other and the ID that will be used to store all the other information pertaining to that interaction.

node (k) ID →	1	2	3	4	5	6	7	8	9	10
node (j) IDs	id_{j1}	id_{j6}	id_{j13}	id_{j16}	id_{j22}	id_{j26}	id_{j31}	id_{j37}	id_{j40}	id_{j47}
	id_{j2}	id_{j7}	id_{j14}	id_{j17}	id_{j23}	id_{j27}	id_{j32}	id_{j38}	id_{j41}	id_{j48}
	id_{j3}	id_{j8}	id_{j15}	id_{j18}	id_{j24}	id_{j28}	id_{j33}	id_{j39}	id_{j42}	id_{j49}
	id_{j4}	id_{j9}		id_{j19}	id_{j25}	id_{j29}	id_{j34}		id_{j43}	id_{j50}
	id_{j5}	id_{j10}		id_{j20}		id_{j30}	id_{j35}		id_{j44}	
		id_{j11}		id_{j21}			id_{j36}		id_{j45}	
		id_{j12}							id_{j46}	

Figure 3.4: Matrix containing the IDs of the (j) nodes interacting with the (k) node identified by the column number

node (k) ID →	1	2	3	4	5	6	7	8	9	10
interaction IDs	1	6	13	16	22	26	31	37	40	47
	2	7	14	17	23	27	32	38	41	48
	3	8	15	18	24	28	33	39	42	49
	4	9		19	25	29	34		43	50
	5	10		20		30	35		44	
		11		21			36		45	
		12							46	

Figure 3.5: Matrix containing the IDs of the interaction; the k-th node is identified by the column number

3.3.5 Interaction matrices initialization

Now that the total number of interaction and the number of interaction per node is known, the remaining matrices can be initialized and the correct amount of memory is allocated.

3.3.6 Relative position vectors

The information regarding the relative position vectors is stored in a matrix, figure 3.6, where the column number is identified by the interaction ID. The rows of the matrix contain the three components of the vectors. Similarly so for the position vector modulus.

interaction ID →	1	2	3	4	5	6	7	8	9	10
$\xi_{(k)(j)}$ vector coordinates	ξ_{1x}	ξ_{2x}	ξ_{3x}	ξ_{4x}	ξ_{5x}	ξ_{6x}	ξ_{7x}	ξ_{8x}	ξ_{9x}	ξ_{10x}
	ξ_{1y}	ξ_{2y}	ξ_{3y}	ξ_{4y}	ξ_{5y}	ξ_{6y}	ξ_{7y}	ξ_{8y}	ξ_{9y}	ξ_{10y}
	ξ_{1z}	ξ_{2z}	ξ_{3z}	ξ_{4z}	ξ_{5z}	ξ_{6z}	ξ_{7z}	ξ_{8z}	ξ_{9z}	ξ_{10z}

Figure 3.6: Matrix containing the relative position vectors' components

3.3.7 Volume corrections

Volume corrections are calculated for each interaction. Since all the problems studied have a regular discretization, the value of the corrections is computed implementing the simple equation 2.8. These values are then stored as shown in figure 3.7.

interaction ID →	1	2	3	4	5	6	7	8	9	10
volume correction →	v_{c1}	v_{c2}	v_{c3}	v_{c4}	v_{c5}	v_{c6}	v_{c7}	v_{c8}	v_{c9}	v_{c10}

Figure 3.7: Matrix containing the volume corrections

3.3.8 Surface corrections

In order to calculate the surface corrections, the body is subjected to a fictitious deformation along the directions of reference coordinate system (depending on the idealization used, the direction of the deformation may be two or just one). For each deformation the value of the peridynamic strain energy is computed and compared with the one obtained from the classical continuum mechanics theory. The corrections along the directions and the unit vector of the direction of the interaction are used to calculate the value of the correction for each interaction. These values are stored in a vector, as shown in figure 3.8.

interaction ID →	1	2	3	4	5	6	7	8	9	10
surface correction →	$G_{\beta 1}$	$G_{\beta 2}$	$G_{\beta 3}$	$G_{\beta 4}$	$G_{\beta 5}$	$G_{\beta 6}$	$G_{\beta 7}$	$G_{\beta 8}$	$G_{\beta 9}$	$G_{\beta 10}$

Figure 3.8: Matrix containing the surface corrections

3.3.9 Peridynamic core calculations

The tasks explained so far are common to all the codes developed. This phase instead can change substantially, depending on the nature of the problem and the solution techniques adopted. In particular different approaches will be implemented for explicit and implicit time integration. The detailed explanation of how the programs work will be given in the next sections.
The simulation progress percentage is shown at fixed intervals, to show the user the progress of the time integration.

3.3.10 Results elaboration and presentation

The results collected in the previous phases can be outputted to files or stored in temporary memory. During this stage the results are elaborated and presented. The program allows to specify the nodes for which the displacement output is required and the number of outputs that should be collected during the simulation.

3.4 Benchmark problems

To validate the results of the peridynamic codes and to test their performance different strategies can be implemented. One approach can be to achieve the solution of a simple problem with another numerical method, such as the finite element method, and to compare the results against the ones provided by the peridynamic simulation. Another strategy is to implement a peridynamic solution of a benchmark problem for which the analytical solution is known and then to validate the results. In both cases the problem analyzed may be dynamical or static.

In figures 3.9, 3.10 and 3.11 are presented three examples of simple geometries respectively for the mono, bi and three-dimensional idealization. These geometries are used in the following benchmark problems.

3.5 Spatial discretization

A volume is associated with each material point that forms the discretization. These points can be regularly spaced or unevenly placed, as shown in figure 3.12, depending on the level of accuracy desired, computational resources available and geometry of the body.
Since the focus of this study is not on the creation of the peridynamic mesh, only simple geometries are used. Instead of creating a discretization starting from the

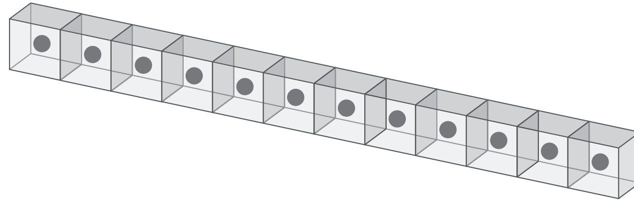


Figure 3.9: Mono-dimensional example of a bar discretization

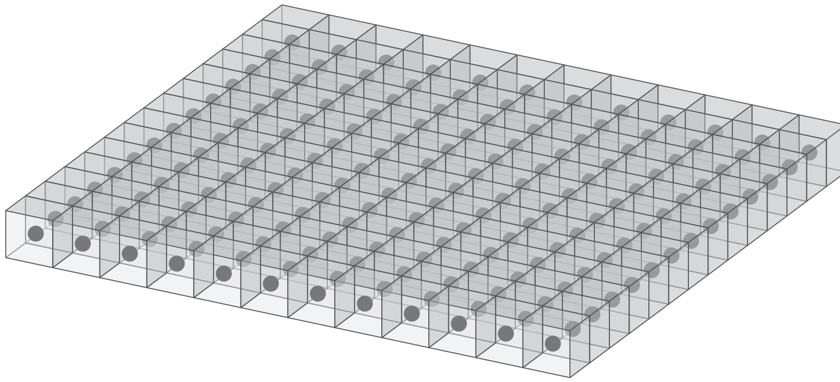


Figure 3.10: Bi-dimensional example of a plate discretization

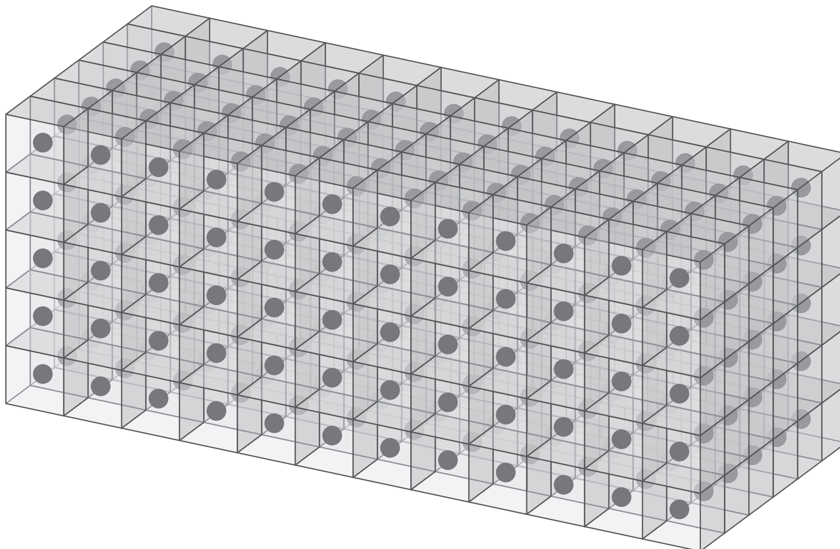


Figure 3.11: Tri-dimensional example of a block discretization

body's geometry, the geometry itself will be created by putting together the points of the discretization, as well shown in figures 3.9, 3.10 and 3.11.

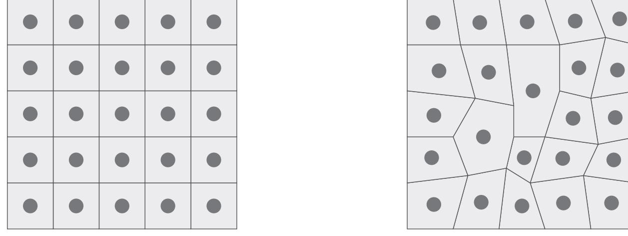


Figure 3.12: Example of regular discretization (left) and unevenly placed material points (right)

3.6 Explicit time integration

Explicit time integration is the simplest form of solution to the peridynamic equation of motion for dynamic simulations. Initial conditions on the displacement of each point are required in order to start the time integration procedure. Second and first derivatives of the displacement are approximated using the finite difference technique.

Using the forward finite difference approximation for the second derivative of the k-th material point we obtain:

$$\ddot{\mathbf{u}}^{n}_{(k)} = \frac{\dot{\mathbf{u}}^{n+1}_{(k)} - \dot{\mathbf{u}}^{n}_{(k)}}{\Delta t} \tag{3.1}$$

from which we obtain the value of the first derivative at the next time step:

$$\dot{\mathbf{u}}^{n+1}_{(k)} = \ddot{\mathbf{u}}^{n}_{(k)}\Delta t + \dot{\mathbf{u}}^{n}_{(k)} \tag{3.2}$$

Similarly for the displacement:

$$\dot{\mathbf{u}}^{n}_{(k)} = \frac{\mathbf{u}^{n+1}_{(k)} - \mathbf{u}^{n}_{(k)}}{\Delta t} \tag{3.3}$$

$$\mathbf{u}^{n+1}_{(k)} = \dot{\mathbf{u}}^{n}_{(k)}\Delta t + \mathbf{u}^{n}_{(k)} \tag{3.4}$$

3.6.1 Integration algorithm

In figure 3.13 it is shown a scheme of how the explicit time integration algorithm works.

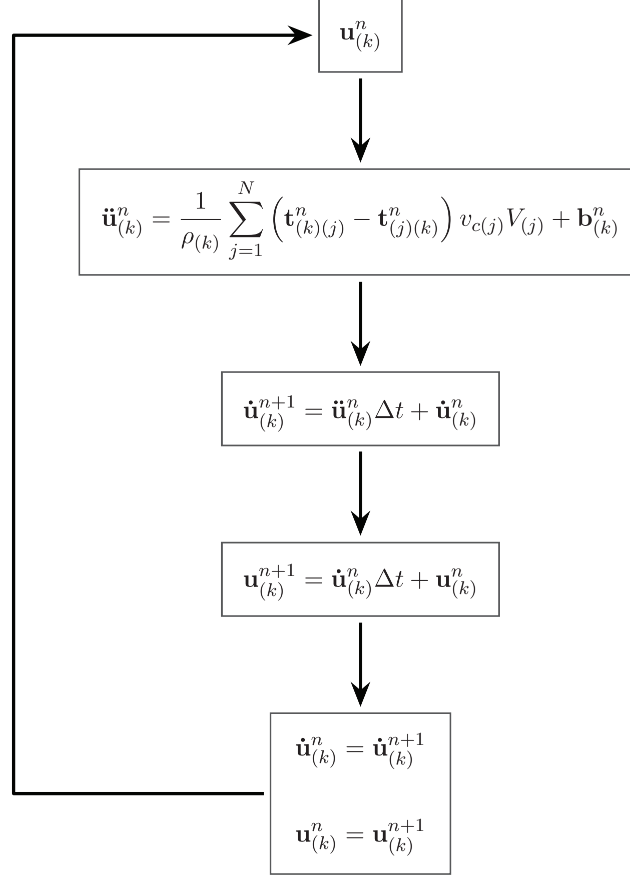


Figure 3.13: Explicit time integration algorithm

The force density functions $\mathbf{t}^n_{(k)(j)}$ and $\mathbf{t}^n_{(j)(k)}$ are computed, from an initial value of the displacement, for all the nodes within the horizon of the k-th point. These values are then used to calculate the second derivative $\ddot{\mathbf{u}}^n_{(k)}$ at the current time step. The first derivative $\dot{\mathbf{u}}^{n+1}_{(k)}$ and the displacement $\mathbf{u}^{n+1}_{(k)}$ vectors at the next time step are obtained using the explicit forward finite difference technique, as previously shown. These values are used as the new values of $\dot{\mathbf{u}}^n_{(k)}$ and $\mathbf{u}^n_{(k)}$ to restart the algorithm.

At each step of the integration, the time of the simulation is incremented of a Δt quantity. Limitations on the Δt value are required, as will be shown hereinafter, to ensure the stability of the integration procedure.

The numerical error from the integration procedure is of the order of $O(\Delta t^2)$ while the error from spatial integration is of the order of $O(\Delta^2)$, as shown in [13] 7.3.

3.6.2 Application of initial and boundary conditions

Initial condition on the displacement vectors **u** are required to start the integration algorithm. They can be set to zero or have a non-zero value, depending on the problem analyzed. Boundary conditions can be enforced by setting the displacement value of the bounded nodes after the integration procedure and before restarting the algorithm.

3.6.3 Numerical stability

The explicit time integration algorithm is subjected to stability issues with the increase of the time step Δt. Performing a von Neumann stability analysis, as shown in [13] 7.4, the following stability condition on the time step arises:

$$\Delta t < \sqrt{\frac{2\rho_{(k)}}{\sum_j \left(2ad\delta \frac{\left(d\delta \sum_l \left(\frac{1}{\left|\boldsymbol{\xi}_{(l)(k)}\right|} + \frac{1}{\left|\boldsymbol{\xi}_{(l)(j)}\right|} \right) V_l \right)}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} + \frac{4b\delta}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \right) \left(v_{c(j)} V_{(j)} \right)}} \tag{3.5}$$

3.7 Codes developed for explicit time integration

In the appendix A will be presented the full MATLAB and C++ peridynamic codes used to obtain the results that are discussed in the following sections. Each line of code is commented to provide a better understanding of the operations that it performs.

3.8 Validation benchmarks results

To ensure the validity of the results produced by the peridynamic codes, developed for explicit time integration, it is necessary to compare them to analytical or FEM solutions. This task is performed solving simple benchmark problems, whose analytical solution is known.

Three benchmarks are performed: one with a mono-dimensional geometry idealization, one with a bi-dimensional geometry and one with a three-dimensional body.

3.8.1 1D benchmark

The problem analyzed for the mono-dimensional benchmark can be found in [13]. A bar, represented with a mono-dimensional discretization, is subjected to the following uniform initial displacement field:

$$\mathbf{u}_{(k)} = \varepsilon \mathbf{x}_{(k)} \tag{3.6}$$

When the bar is left free to move it vibrates along the direction of it's axis. The analytical value of the longitudinal displacement of a point of the bar is given by the following equation:

$$u_x(x,t) = \frac{8\varepsilon L}{\pi^2} \sum_{n=o}^{\infty} \frac{(-1)^n}{(2n+1)^2} \sin\left(\frac{(2n+1)\,\pi x}{2L}\right) \cos\left(\sqrt{\frac{E}{\rho}} \frac{(2n+1)\,\pi}{2L} t\right) \tag{3.7}$$

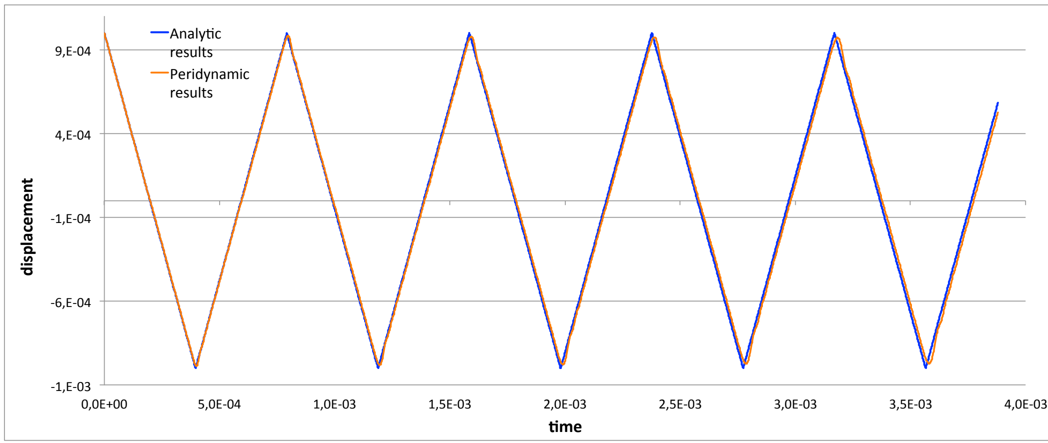


Figure 3.14: Peridynamic (orange) and analytic (blue) results for 1D benchmark with explicit time integration at $x = L$

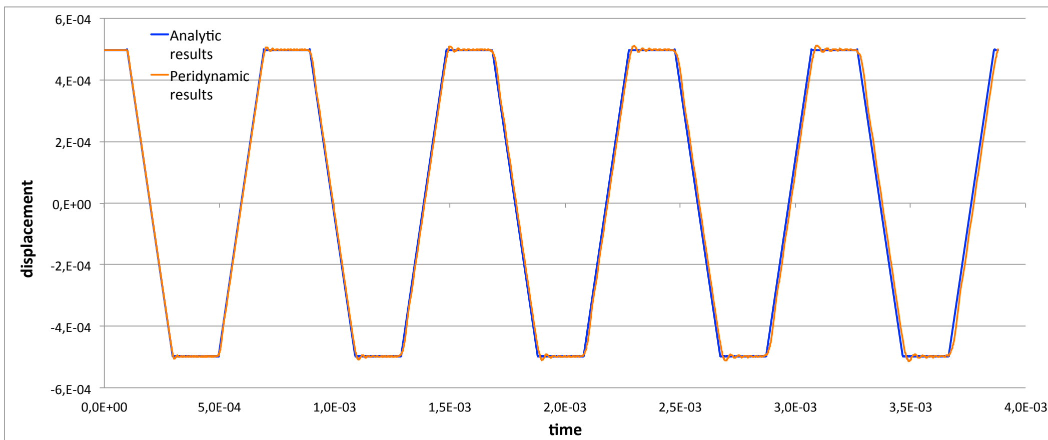


Figure 3.15: Peridynamic (orange) and analytic (blue) results for 1D benchmark with explicit time integration at $x = \frac{1}{2}L$

In figures 3.14 and 3.15 is shown a comparison between the analytic and peridynamic results of two points in the vibrating bar.

This benchmark also proves that the output of the MATLAB program matches perfectly the results provided by the C++ solution code.

As it can be noticed from the figures, the peridynamic results show excellent agreement with the analytic results at the beginning of the simulation and they tend to deviate slightly as the time of the simulation progresses. This behaviour is easily explainable as the integration error builds up as the simulation time progresses.

The bar has been meshed with 1000 nodes and the time step used for the explicit integration process is 1.94 $10^{-7}s$, as suggested in [13] to ensure the stability of the integration process.

3.8.2 2D benchmark

The same test problem previously illustrated is used to perform a benchmark on the 2D solution sequence of the codes. The simulation is performed with a thin plate instead of a bar. The plate is discretized with a bi-dimensional mesh of 200 by 20 nodes, as shown in figure 3.16. The results of the simulations are shown in figures 3.17 and 3.18.

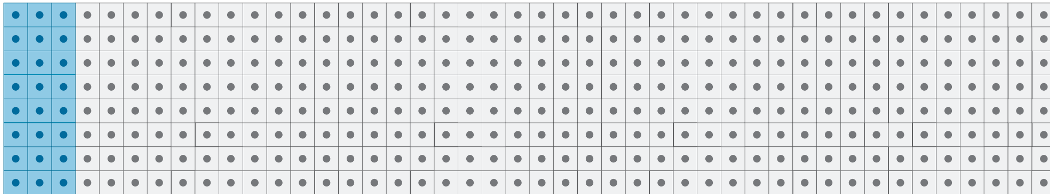


Figure 3.16: Representation of the 2D model used for the simulations (this image doesn't show the real number of nodes)

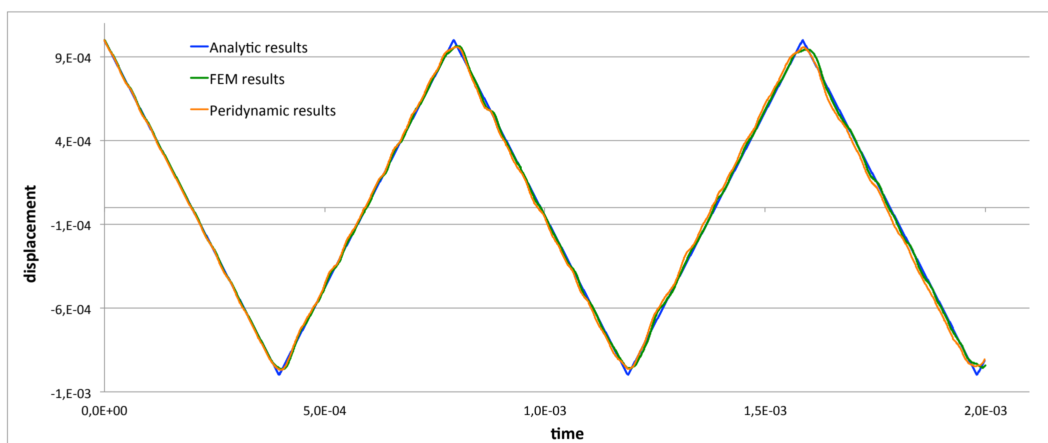


Figure 3.17: Peridynamic (orange) and analytic (blue) results for 2D benchmark with explicit time integration at $x = L$

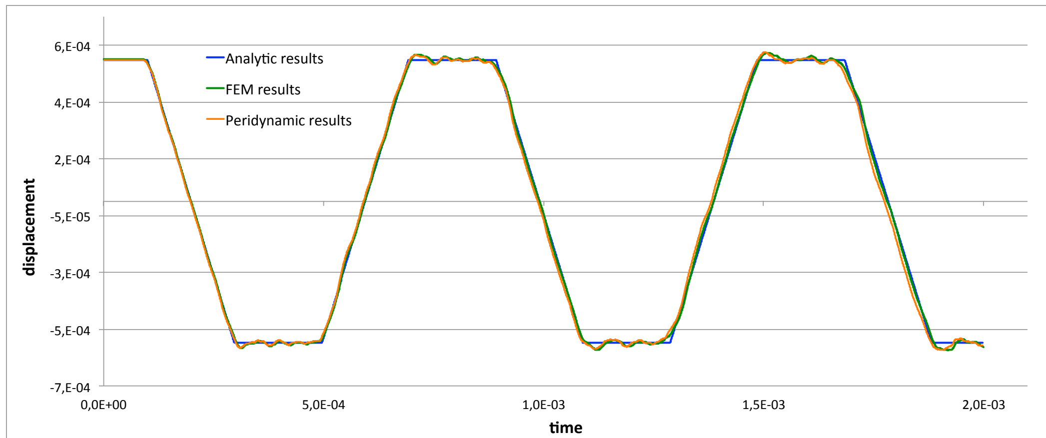


Figure 3.18: Peridynamic (orange) and analytic (blue) results for 2D benchmark with explicit time integration at $x = \frac{1}{2}L$

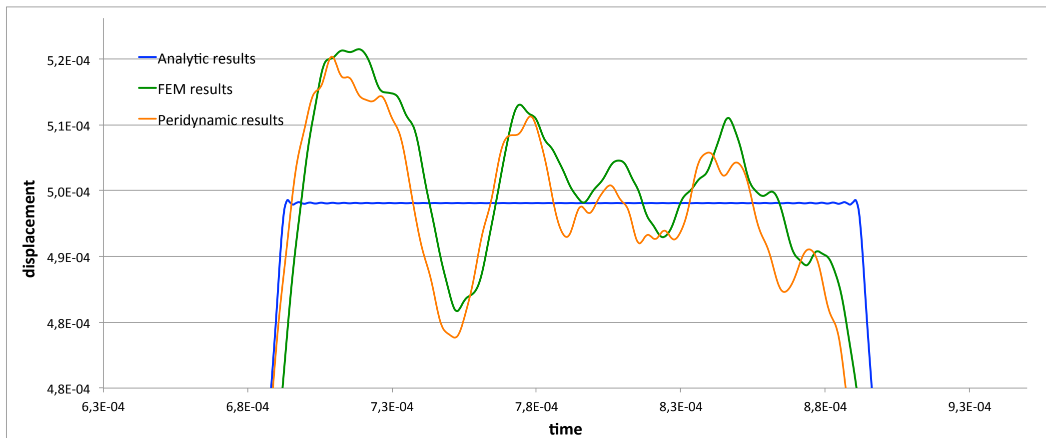


Figure 3.19: Zoom of the overshoot area for 2D benchmark with explicit time integration at $x = \frac{1}{2}L$

A finite element simulation of the benchmark problem has provided further data to compare to the peridynamic solution. As it is particularly clear from figure 3.18, the FEM solution matches very well the peridynamic results.

In figure 3.19 we can see a zoom of the time-displacement plot of figure 3.18. We can notice that both the peridynamic and the FEM solution present an initial overshoot and then an oscillation around the value predicted from the analytic solution. This behaviour might be ascribable to the size of the mesh used to discretize the body.

Once again the benchmark is used to check that the results provided by the MATLAB solution code correspond to the what obtained with the C++ solution program.

3.8.3 3D benchmark

To test the behaviour of the 3D program, once again the vibrating bar benchmark is performed. The bar is discretized with a mesh of 200 by 10 by 10 nodes. The results of the simulation are shown in figures 3.20 and 3.21.

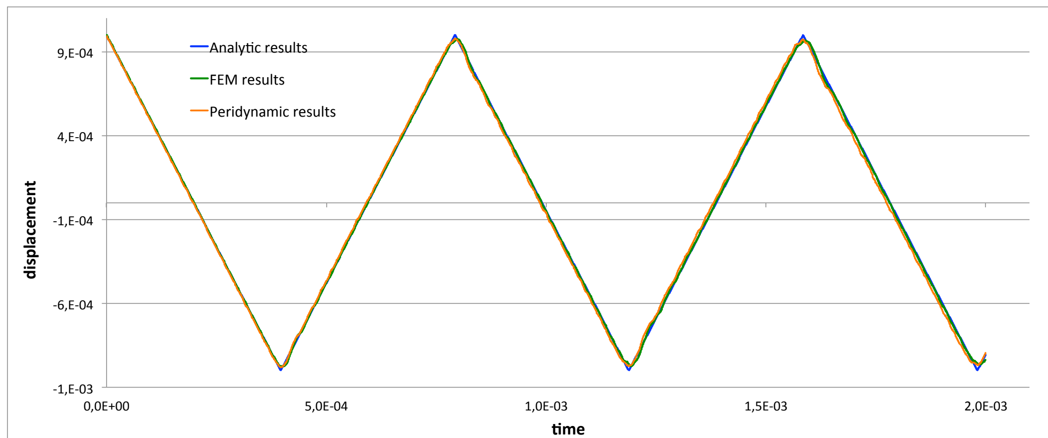


Figure 3.20: Peridynamic (orange) and analytic (blue) results for 3D benchmark with explicit time integration at $x = L$

From the plots it can be noticed that the peridynamic results are in good agreement with the analytic and FEM solutions.

In figure 3.22 we can see a zoom of the time-displacement plot of figure 3.21. We can notice also in this case that both the peridynamic and the FEM solution present an initial overshoot and then an oscillation around the value predicted from the analytic solution. The behaviour of the FEM and peridynamic solution is similar and it might be ascribable to the size of the mesh used to discretize the body.

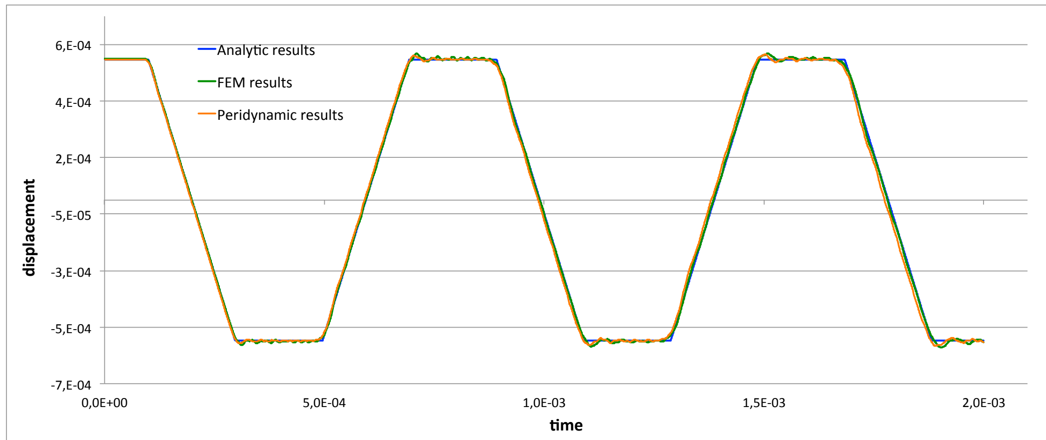


Figure 3.21: Peridynamic (orange) and analytic (blue) results for 3D benchmark with explicit time integration at $x = \frac{1}{2}L$

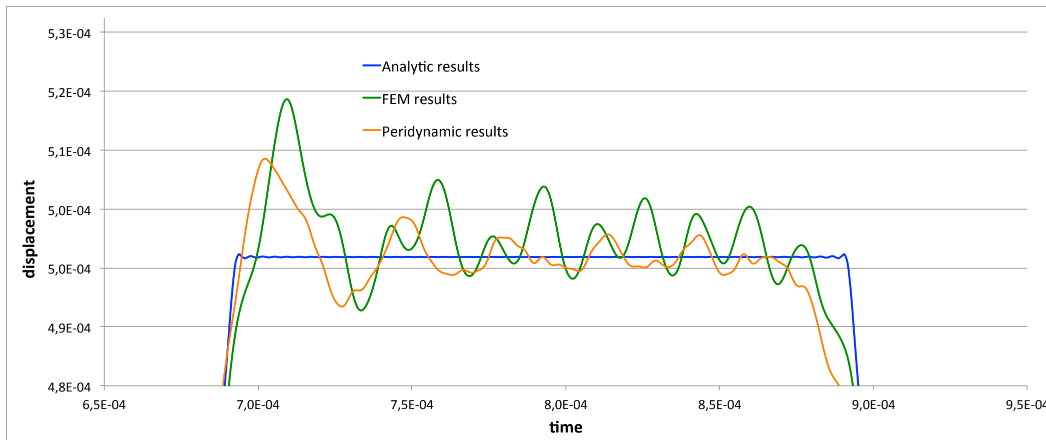


Figure 3.22: Zoom of the time-displacement plot for 3D benchmark with explicit time integration at $x = \frac{1}{2}L$

3.9 Performance analysis

The difference in performance between the MATLAB and C++ solution codes is evaluated by measuring the CPU time of some significant tasks of the program. The hardware and software configuration of the testing machine is expected to affect the performance measurements. In table 3.1 it is shown the testing set up.

CPU	Intel Core i7 Quad Core 2.0 GHz
RAM memory	16 Gb DDR3 1333 Mhz
Hard Drive	SSD Crucial M4 265 Gb on SATA3
Operating System	OSX Yosemite 10.10.1
MATLAB Version	R2011b 64-bit
C++ Version	11 compiled with Xcode 6

Table 3.1: Configuration of the testing computer

In order to provide accurate measurements, each testing session is carried out after a system reboot and all the unnecessary applications are shut down before starting the test. Furthermore the output collecting and processing is deactivated during the performance measurements. To ensure the validity of the results the measures are repeated three times. Hereinafter will be reported the mean values of the three measures.
These are the tests that are going to be performed on the solution codes:

1. Total simulation time with respect to the number of nodes
2. Total simulation time with respect to the number of interactions
3. Time of integration over one time-step with respect to the number of nodes
4. Time of integration over one time-step with respect to the number of interactions
5. Time of pre-integrations tasks of the MATLAB 3D program
6. Time of pre-integrations tasks of the C++ 3D program

In table 3.2 is recorded the total CPU time of the simulation of a mono-dimensional bar performed with 1000 time steps and in table 3.3 the average CPU time of one integration time step. In figures 3.23 and 3.24 are reported respectively the plots of the total CPU time and one step CPU time with respect to the number of nodes (left) and number of interactions (right).

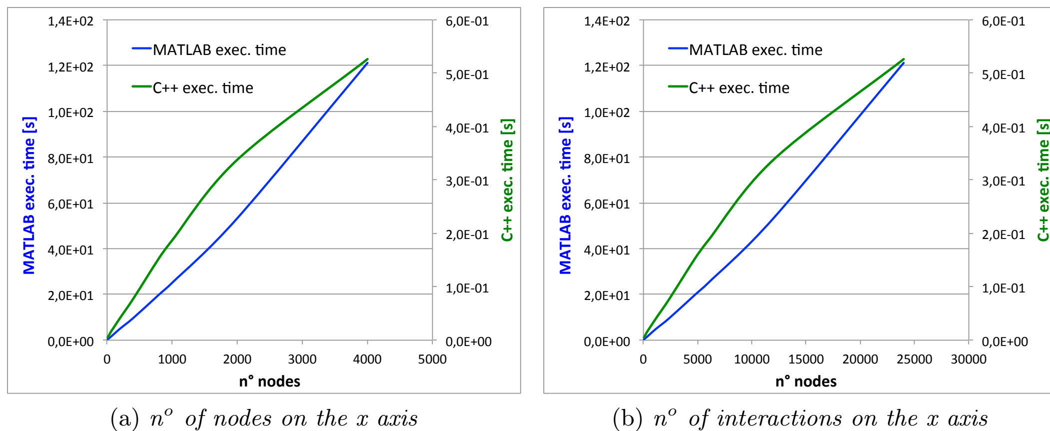
As we can notice from the plots both the total CPU time and one step CPU time are linearly proportional to the number of nodes and the number of interactions. We can also point out that the CPU time of the C++ code is roughly 2 orders of

n° nodes	n° interactions	MATLAB CPU time [s]	C++ CPU time [s]
13	66	3.572e-1	5.078e-3
53	306	1.329e+0	1.445e-2
103	606	2.593e+0	2.377e-2
203	1206	5.088e+0	4.212e-2
403	2406	9.669e+0	7.767e-2
803	4806	2.002e+1	1.544e-1
1003	6006	2.512e+1	1.874e-1
2003	12006	5.332e+1	3.381e-1
4003	24006	1.212e+2	5.263e-1

Table 3.2: Total time of the simulation for 1D structure with 1000 time steps

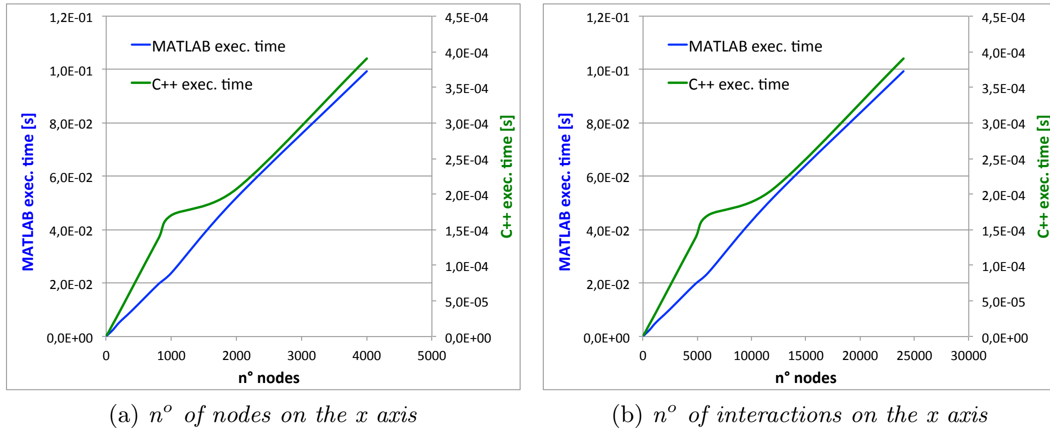
n° nodes	n° interactions	MATLAB CPU time [s]	C++ CPU time [s]
13	66	3.168e-4	2.003e-6
53	306	1.274e-3	8.935e-6
103	606	2.460e-3	1.752e-5
203	1206	5.290e-3	3.428e-5
403	2406	9.891e-3	6.887e-5
803	4806	1.964e-2	1.378e-4
1003	6006	2.389e-2	1.704e-4
2003	12006	5.206e-2	2.074e-4
4003	24006	9.941e-2	3.906e-4

Table 3.3: Average CPU time of one integration time step for a 1D structure



(a) *n° of nodes on the x axis* (b) *n° of interactions on the x axis*

Figure 3.23: Plot of the total CPU time for a 1D simulation



(a) *n° of nodes on the x axis* (b) *n° of interactions on the x axis*

Figure 3.24: Plot of the CPU time of one integration time step for a 1D simulation

magnitude shorter than the MATLAB CPU time for an equivalent simulation. This conclusion has to be referred to the testing configuration of table 3.1 in which the simulations have been carried out. More tests with different hardware and software configurations would allow to further prove the soundness this inference.

The same test is carried out on a 2D structure, the results of the total CPU time and one integration time step CPU time are recorded in tables 3.4 and 3.5. In figures 3.25 and 3.26 are once again reported the plots of the values of the previous tables.

n° nodes	n° interactions	MATLAB CPU time [s]	C++ CPU time [s]
46	470	1.360e+0	1.566e-2
172	3160	7.290e+0	7.151e-2
664	15352	3.283e+1	3.197e-1
1030	24808	5.144e+1	5.168e-1
2295	58248	1.260e+2	1.219e+0
4060	105688	2.321e+2	2.286e+0

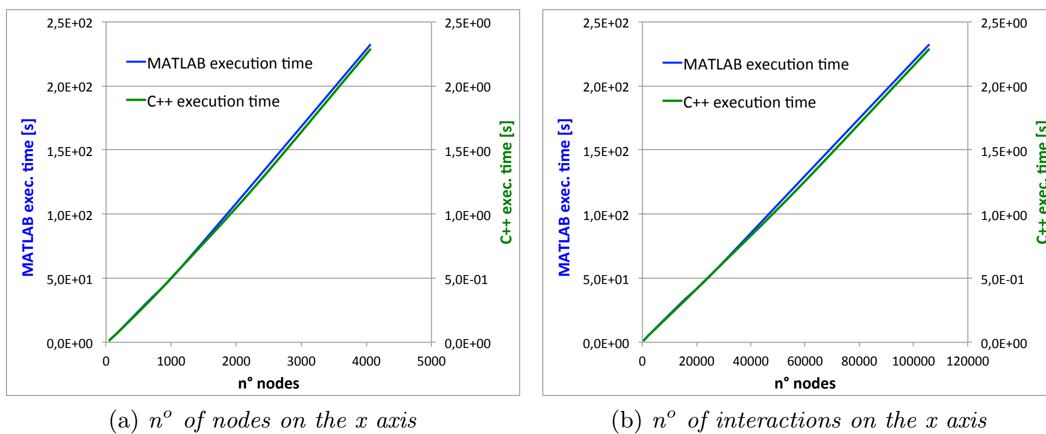
Table 3.4: Total time of the simulation for 2D structure with 500 time steps

From figures 3.25 and 3.26 we can see very clearly the direct proportionality relation between the simulation time and the number of nodes or interactions. Once again the C++ simulations appear to run 100 times faster than the MATLAB counterpart.

The total CPU time and one integration time step CPU time for a tridimensional simulation are recorded respectively in tables 3.6 and 3.7. Figures 3.27 and 3.28 show the graphs of the CPU times with respect to the number of nodes and the number of interactions between the nodes. Also for a 3D structure we can conclude that the C++ simulation is performed 100 times faster than the same simulation in MATLAB. The plots of the CPU times with respect to the number of interactions

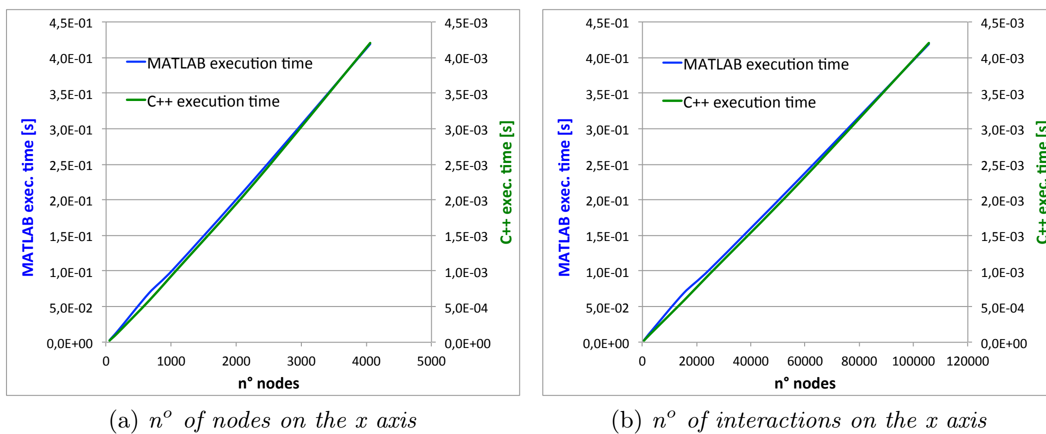
n° nodes	n° interactions	MATLAB CPU time [s]	C++ CPU time [s]
46	470	2.463e-3	1.943e-5
172	3160	1.497e-2	1.240e-4
664	15352	6.940e-2	5.864e-4
1030	24808	1.022e-1	9.587e-4
2295	58248	2.309e-1	2.253e-3
4060	105688	4.188e-1	4.204e-3

Table 3.5: Average CPU time of one integration time step for a 2D structure



(a) *n° of nodes on the x axis* (b) *n° of interactions on the x axis*

Figure 3.25: Plot of the total CPU time for a 2D simulation



(a) *n° of nodes on the x axis* (b) *n° of interactions on the x axis*

Figure 3.26: Plot of the CPU time of one integration time step for a 2D simulation

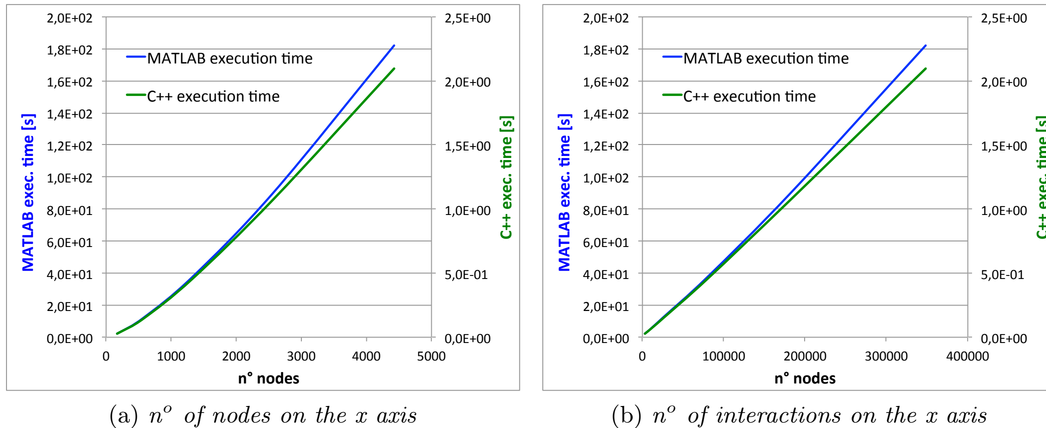
show a clear direct proportionality relation between these two variables.

n° nodes	n° interactions	MATLAB CPU time [s]	C++ CPU time [s]
172	3492	1.920e+0	2.514e-2
576	25054	1.212e+1	1.453e-1
1328	80132	3.762e+1	4.558e-1
2575	183022	9.039e+1	1.078e+0
4428	348364	1.822e+2	2.097e+0

Table 3.6: Total time of the simulation for 3D structure with 100 time steps

n° nodes	n° interactions	MATLAB CPU time [s]	C++ CPU time [s]
172	3492	1.609e-2	1.825e-4
576	25054	1.067e-1	1.286e-3
1328	80132	3.444e-1	4.146e-3
2575	183022	7.875e-1	9.727e-3
4428	348364	1.469e+0	1.860e-2

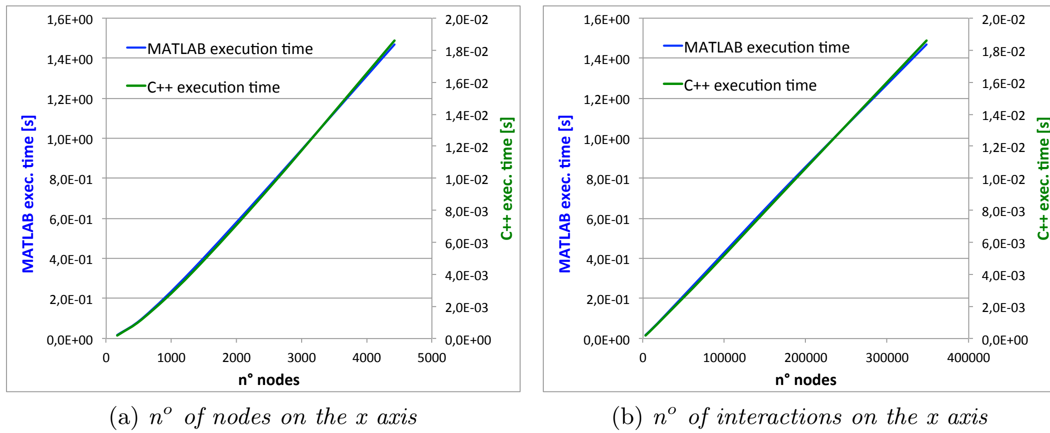
Table 3.7: Average CPU time of one integration time step for a 3D structure



(a) *n° of nodes on the x axis* (b) *n° of interactions on the x axis*

Figure 3.27: Plot of the total CPU time for a 3D simulation

From the performance tests performed so far we can point out that the C++ solution code appears to be performing much better under the hardware and software configuration considered.
In light of the direct proportionality relation between the CPU times and the number of nodes or interactions of the simulation, we can infer that, given the required number of material points and the total time of integration, it is possible to estimate the computational time required to run a simulation.

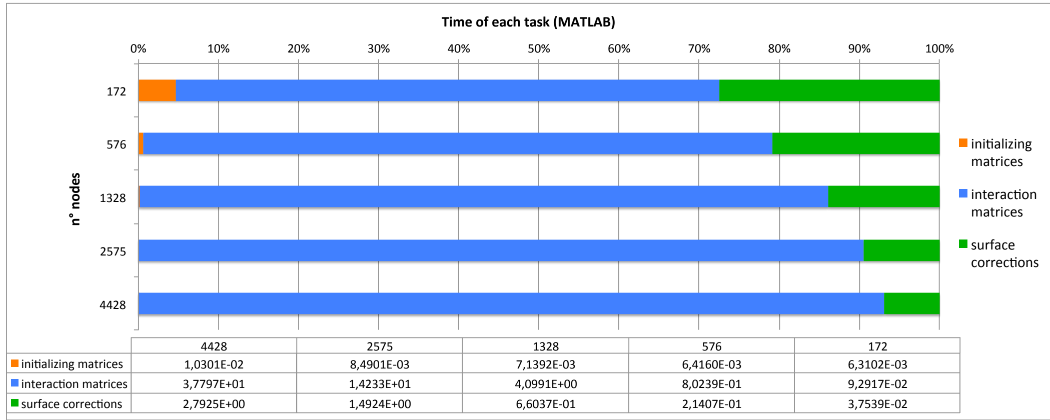


(a) *n° of nodes on the x axis*

(b) *n° of interactions on the x axis*

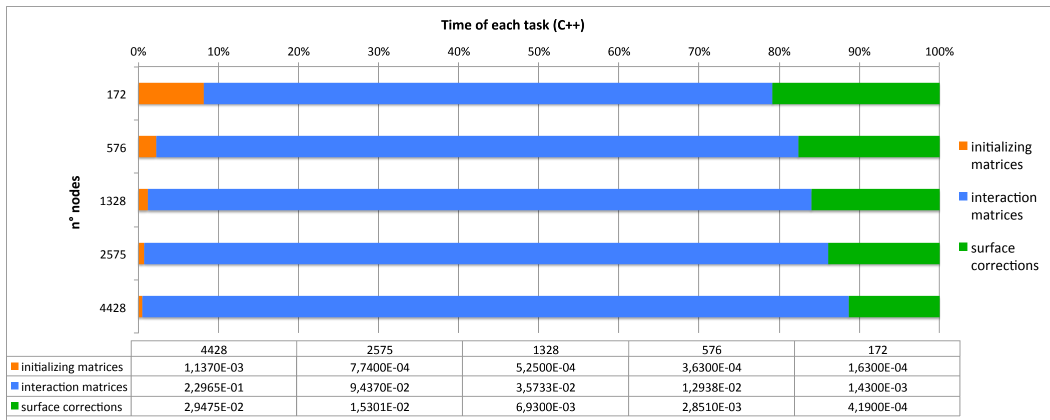
Figure 3.28: Plot of the CPU time of one integration time step for a 3D simulation

Figures 3.29 and 3.30 show the CPU time of each task that precedes the integration algorithm, respectively for the MATLAB and C++ codes. We can notice that the task of building the interaction matrices requires the majority of the computational time. As the number of nodes increases the percentage of time required to fill the interaction matrices, calculated on the total time of the pre-integration tasks, increases as well. This consideration is valid for both the programming languages.



	4428	2575	1328	576	172
initializing matrices	1,0301E-02	8,4901E-03	7,1392E-03	6,4160E-03	6,3102E-03
interaction matrices	3,7797E+01	1,4233E+01	4,0991E+00	8,0239E-01	9,2917E-02
surface corrections	2,7925E+00	1,4924E+00	6,6037E-01	2,1407E-01	3,7539E-02

Figure 3.29: Pre-integration tasks CPU times for the MATLAB 3D code



	4428	2575	1328	576	172
initializing matrices	1,1370E-03	7,7400E-04	5,2500E-04	3,6300E-04	1,6300E-04
interaction matrices	2,2965E-01	9,4370E-02	3,5733E-02	1,2938E-02	1,4300E-03
surface corrections	2,9475E-02	1,5301E-02	6,9300E-03	2,8510E-03	4,1900E-04

Figure 3.30: Pre-integration tasks CPU times for the C++ 3D code

Chapter 4

Numerical solution with implicit time integration

Implicit time integration offers the advantage of being unconditionally stable, with respect to the time step, even though the accuracy of the solution is influenced by the size of the time step. This allows to perform a dynamic simulation with a bigger time step than what would be possible with an explicit integration scheme.
In this chapter an integration scheme for implicit time integration is devised. This solution is implemented in both the MATLAB and the C++ peridynamic solvers. The accuracy and performance of these codes is tested against the results obtained with the explicit solvers.

4.1 Peridynamic equation of motion in implicit form

Being $\mathbf{u}_{(k)}(t)$ the displacement of a point located at $\mathbf{x}_{(k)}$ at time t, the value of $\mathbf{u}_{(k)}\left(t+i\Delta t\right)$ can be written as a series of Taylor as follows:

$$\mathbf{u}_{(k)}\left(t+i\Delta t\right)=\sum_{n=o}^{\infty} i^{n}\frac{\Delta t}{n!}\mathbf{u}_{(k)}^{(n)} \tag{4.1}$$

where $\mathbf{u}_{(k)}^{(n)}$ is the n-th order displacement derivative. The derivative of the displacement can be approximated with the following expression:

$$\frac{\Delta t^{d}}{d!}\mathbf{u}_{(k)}^{(d)}+O\left(\Delta t^{d+p}\right)=\sum_{i=i_{min}}^{i_{max}} C_{i}\mathbf{u}_{(k)}\left(t+i\Delta t\right) \tag{4.2}$$

where d is the order of the derivative, p is the desired order of the error of the approximation, C_i are some coefficients that need to be determined. For a backward difference approximation we have that $i_{min}=-(d+p-1)$ and $i_{max}=0$.

Substituting equation 4.2 into equation 4.1 and multiplying by $\frac{d!}{\Delta t^d}$ we get:

$$\mathbf{u}_{(k)}^{(d)} = \frac{d!}{\Delta t^d} \sum_{n=0}^{d+p-1} \left(\sum_{i=i_{min}}^{i_{max}} i^n C_i \right) \frac{\Delta t^n}{n!} \mathbf{u}_{(k)}^{(n)} + O\left(\Delta t^p\right) \tag{4.3}$$

For the previous equation to be satisfied it needs to be:

$$\sum_{i=i_{min}}^{i_{max}} i^n C_i = \begin{cases} 0 & \text{for } 0 \leq n \leq d+p-1 \quad \text{and } n \neq d \\ 1 & \text{for } n = d \end{cases} \tag{4.4}$$

to approximate the second derivative of the displacement with an order of the error of $O\left(\Delta t^2\right)$ we have $d = 2$ and $p = 2$. Thus equation 4.3 becomes:

$$\ddot{\mathbf{u}}_{(k)} = \frac{2}{\Delta t^2} \sum_{n=0}^{3} \left(\sum_{i=-3}^{0} i^n C_i \right) \frac{\Delta t^n}{n!} \mathbf{u}_{(k)}^{(n)} + O\left(\Delta t^2\right) \tag{4.5}$$

From equation 4.4 we obtain the following system of equations:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -3 & -2 & -1 & 0 \\ 9 & 4 & 1 & 0 \\ -27 & -8 & -1 & 0 \end{bmatrix} \begin{Bmatrix} C_{-3} \\ C_{-2} \\ C_{-1} \\ C_0 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{Bmatrix} \tag{4.6}$$

Its solution is $C_{-3} = -\frac{1}{2}$ $\quad C_{-2} = 2$ $\quad C_{-1} = -\frac{5}{2}$ $\quad C_0 = 1$. So the backward finite difference approximation of $\ddot{\mathbf{u}}_{(k)}$ at the n-th time step is:

$$\ddot{\mathbf{u}}_{(k)}^{n} = \frac{2\mathbf{u}_{(k)}^{n} - 5\mathbf{u}_{(k)}^{n-1} + 4\mathbf{u}_{(k)}^{n-2} - \mathbf{u}_{(k)}^{n-3}}{\Delta t^2} \tag{4.7}$$

Using the implicit integration scheme, the peridynamic governing equation is:

$$
\rho \ddot{\mathbf{u}}^n_{(k)} = \sum_{j=1}^{N} \left(2ad\delta G_{(d)(k)(j)} \frac{\Lambda^n_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \left(\theta^n_{(k)} + \theta^n_{(j)}\right) \dots \right.
$$

$$
\left. + 4b\delta G_{(b)(k)(j)} s^n_{(k)(j)} \right) v_{c(j)} V_{(j)} \frac{\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}^n_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}^n_{(k)(j)}\right|} + \mathbf{b}^n_{(k)} \tag{4.8}
$$

Substituting equation 4.7 into 4.8 we obtain:

$$
\rho \frac{2\mathbf{u}^n_{(k)} - 5\mathbf{u}^{n-1}_{(k)} + 4\mathbf{u}^{n-2}_{(k)} - \mathbf{u}^{n-3}_{(k)}}{\Delta t^2} = \dots
$$

$$
\dots = \sum_{j=1}^{N} \left(2ad\delta G_{(d)(k)(j)} \frac{\Lambda^n_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \left(\theta^n_{(k)} + \theta^n_{(j)}\right) \dots \right. \tag{4.9}
$$

$$
\left. + 4b\delta G_{(b)(k)(j)} s^n_{(k)(j)} \right) v_{c(j)} V_{(j)} \frac{\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}^n_{(k)(j)}}{\left|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}^n_{(k)(j)}\right|} + \mathbf{b}^n_{(k)}
$$

where $\Lambda^n_{(k)(j)}$, $\theta^n_{(k)}$, $\theta^n_{(j)}$, $s^n_{(k)(j)}$ and $\boldsymbol{\eta}^n_{(k)(j)}$ are all function of the displacement at the n-th time step $\mathbf{u}^n_{(k)}$, which also appears on the left side of the equation. To calculate the displacement is then necessary to solve a system of equations in the form of:

$$
\begin{aligned}
&[\mathbf{K}]\{\mathbf{u}\} = \{\mathbf{r}\} \\
&[\mathbf{K}]\{\mathbf{u}\} - \{\mathbf{r}\} = 0
\end{aligned} \tag{4.10}
$$

A solution to this problem will be presented later.

4.2 The Newmark-beta method for implicit integration

The Newmark-beta method is a different approach that can be used to obtain an implicit form of the problem[11]. The accuracy and performance of this technique will be investigated and compared to the backward finite difference method.

Using an extended version of the Cauchy's mean value theorem, the displacement first derivative can be written as follows:

$$
\dot{\mathbf{u}}^n_{(k)} = \dot{\mathbf{u}}^{n-1}_{(k)} + \Delta t \ddot{\mathbf{u}}^{\gamma}_{(k)} \tag{4.11}
$$

where :

$$\ddot{\mathbf{u}}^{\gamma}_{(k)} = (1-\gamma)\,\ddot{\mathbf{u}}^{n-1}_{(k)} + \gamma\ddot{\mathbf{u}}^{n}_{(k)} \qquad 0 \leq \gamma \leq 1 \tag{4.12}$$

Substituting $\ddot{\mathbf{u}}^{\gamma}_{(k)}$ from equation 4.12 into equation 4.11 we get:

$$\dot{\mathbf{u}}^{n}_{(k)} = \dot{\mathbf{u}}^{n-1}_{(k)} + (1-\gamma)\,\Delta t\ddot{\mathbf{u}}^{n-1}_{(k)} + \gamma\Delta t\ddot{\mathbf{u}}^{n}_{(k)} \tag{4.13}$$

Using the extended mean value theorem again we can obtain the following expression for the displacement:

$$\mathbf{u}^{n}_{(k)} = \mathbf{u}^{n-1}_{(k)} + \Delta t\dot{\mathbf{u}}^{n-1}_{(k)} + \frac{1}{2}\delta t^2\ddot{\mathbf{u}}^{\beta}_{(k)} \tag{4.14}$$

where:

$$\ddot{\mathbf{u}}^{\beta}_{(k)} = (1-2\beta)\,\ddot{\mathbf{u}}^{n-1}_{(k)} + 2\beta\ddot{\mathbf{u}}^{n}_{(k)} \qquad 0 \leq 2\beta \leq 1 \tag{4.15}$$

Assuming $\gamma = \frac{1}{2}$ equation 4.13 can be rewritten as:

$$\dot{\mathbf{u}}^{n}_{(k)} = \dot{\mathbf{u}}^{n-1}_{(k)} + \frac{\Delta t}{2}\left(\ddot{\mathbf{u}}^{n-1}_{(k)} + \ddot{\mathbf{u}}^{n}_{(k)}\right) \tag{4.16}$$

and substituting $\ddot{\mathbf{u}}^{\beta}_{(k)}$ from equation 4.15 into equation 4.14 we get the final expression for the displacement vector:

$$\mathbf{u}^{n}_{(k)} = \mathbf{u}^{n-1}_{(k)} + \Delta t\dot{\mathbf{u}}^{n-1}_{(k)} + \frac{1-2\beta}{2}\Delta t^2\ddot{\mathbf{u}}^{n-1}_{(k)} + \beta\Delta t^2\ddot{\mathbf{u}}^{n}_{(k)} \tag{4.17}$$

In equation 4.17 both $\mathbf{u}^{n}_{(k)}$ and $\ddot{\mathbf{u}}^{n}_{(k)}$ are unknown, and the latter is function of the displacement at the n-th time step, as shown in equation 4.8. Once again it is necessary to solve a system of equations to calculate the displacement value.

4.3 Broyden method for solving the system of equations

To find the displacement at each time step, the system of equations presented in the two previous sections needs to be solved. The Broyden's method is an iterative quasi-Newton method developed by Broyden in 1965[2].

Let us consider the problem given by equation 4.10 and let us define the function $F(\mathbf{u})$ as:

$$F(\mathbf{u}) = [\mathbf{K}]\{\mathbf{u}\} - \{\mathbf{r}\} = 0 \tag{4.18}$$

The solution of the equation $F(\mathbf{u}) = 0$ is the displacement that we are looking for. The approximation of the Taylor expansion of $F(\mathbf{u})$ about $\mathbf{u}^0$ is:

$$F(\mathbf{u}) = F\left(\mathbf{u^0}\right) + J\left(\mathbf{u}^0\right)\left(\mathbf{u} - \mathbf{u}^0\right) + O\left(\mathbf{u} - \mathbf{u}^0\right)^2 \tag{4.19}$$

where $J\left(\mathbf{u}^0\right)$ is the Jacobian matrix of $F(\mathbf{u})$ for $\mathbf{u} = \mathbf{u}^0$. If $\mathbf{u}^*$ is the root of $F(\mathbf{u})$ then $F(\mathbf{u}^*) = 0$ and substituting this in equation 4.19 we get:

$$0 \approx F\left(\mathbf{u^0}\right) + J\left(\mathbf{u}^0\right)\left(\mathbf{u}^* - \mathbf{u}^0\right) \tag{4.20}$$

and so:

$$\mathbf{u}^* \approx \mathbf{u}^0 - J\left(\mathbf{u}^0\right)^{-1} F\left(\mathbf{u}^0\right) \tag{4.21}$$

From equation 4.21 we can devise the well known Newton iterative process:

$$\mathbf{u}^n = \mathbf{u}^{n-1} - J\left(\mathbf{u}^{n-1}\right)^{-1} F\left(\mathbf{u}^{n-1}\right) \tag{4.22}$$

A setback of the Newton iterative scheme is the need to calculate the inverse of the Jacobian matrix at each iteration This process can be computationally expensive, particularly so when the function $F(\mathbf{u})$ is complicated to evaluate, as it is in our case.
The Broyden's method uses an approximation of the inverse Jacobian matrix, which

is less expensive to calculate. The iterative process of equation 4.22 then becomes:

$$\mathbf{u}^n = \mathbf{u}^{n-1} - B_{n-1}^{-1} F\left(\mathbf{u}^{n-1}\right) \tag{4.23}$$

With each iteration, the matrix B_{n-1}^{-1} becomes closer and closer to $J\left(\mathbf{u}^{n-1}\right)^{-1}$ and so $\mathbf{u}^n$ becomes closer to the exact solution $\mathbf{u}^*$.
Broyden proposes[2] to update the value of B_{n-1}^{-1}, using the Sherman-Morrison derivation, as shown in the following equation:

$$B_n^{-1} = B_{n-1}^{-1} + \frac{\left(\left(\mathbf{u}^n - \mathbf{u}^{n-1}\right) - B_{n-1}^{-1}\left(F\left(\mathbf{u}^n\right) - F\left(\mathbf{u}^{n-1}\right)\right)\right)\left(\mathbf{u}^n - \mathbf{u}^{n-1}\right)^T B_{n-1}^{-1}}{\left(\mathbf{u}^n - \mathbf{u}^{n-1}\right)^T B_{n-1}^{-1}\left(F\left(\mathbf{u}^n\right) - F\left(\mathbf{u}^{n-1}\right)\right)} \tag{4.24}$$

For the first iteration, an approximation of B_0^{-1} may be the identical matrix multiplied by a scalar value $\alpha\,[I]$, where α can be chose to ensure a faster convergence of the method.

4.4 Algorithms for implicit time integration

The previously introduced techniques can be implemented in the form of iterative algorithms. Unlike with explicit time integration, the implicit solution requires two nested iteration processes, one to perform the time integration and one to perform the solution of the system of equations.

In figure 4.1 it is presented the algorithm for the implicit integration with the backward finite difference approximation of the second derivative of the displacement vector. This algorithm uses the Broyden's method to solve the system of equations.

In figure 4.2 it is presented the algorithm for the implicit integration were the Newmark-beta method is used to approximate the displacement second derivative. Also in this case the Broyden's method is used to solve the system of equations and calculate the displacement vector.

From the two algorithms schemes we can notice that the number of variables that need to be stored, compared with the explicit integration scheme, increases. In figure 4.1 we see that the displacement of the three previous time steps is required to calculate the F function whereas in figure 4.2 we can see that the displacement, its first and second derivative at the previous time step need to be stored.

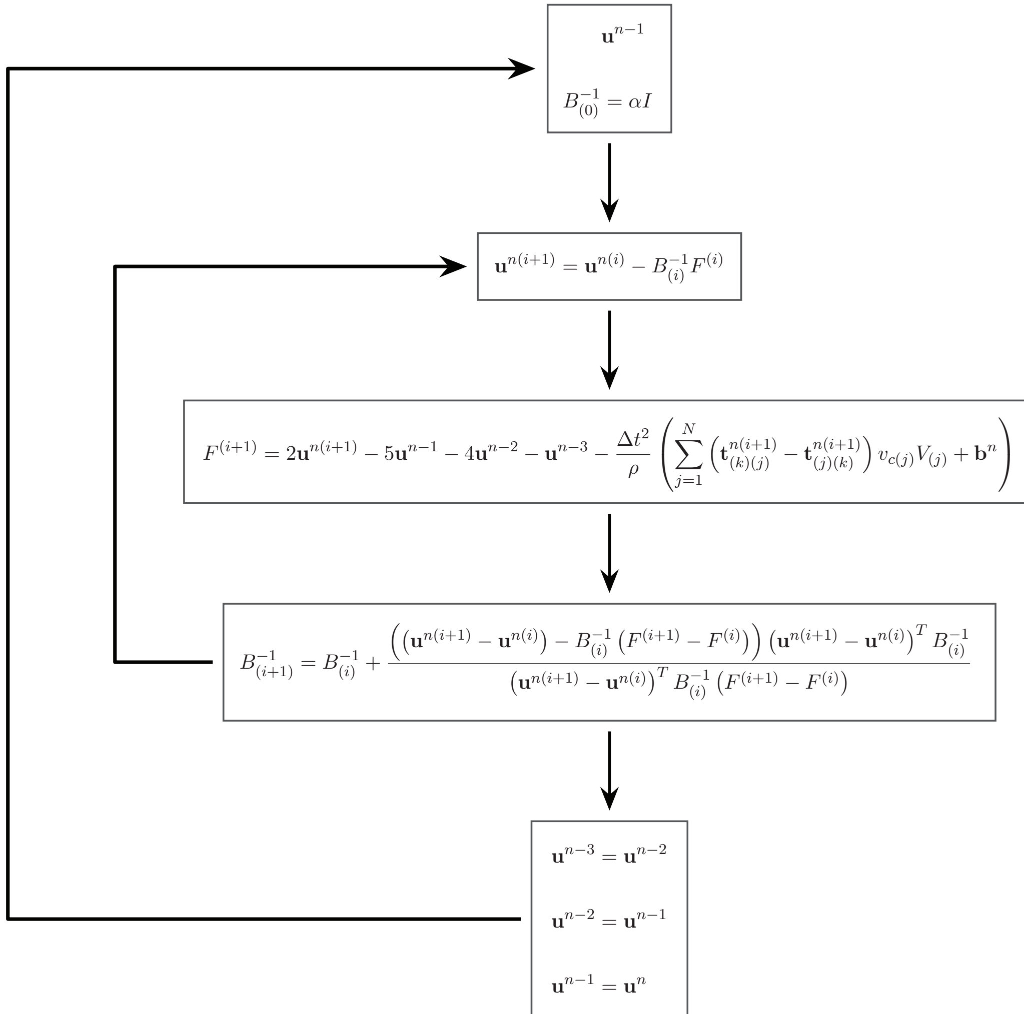


Figure 4.1: Implicit time integration algorithm with backward finite difference approximation

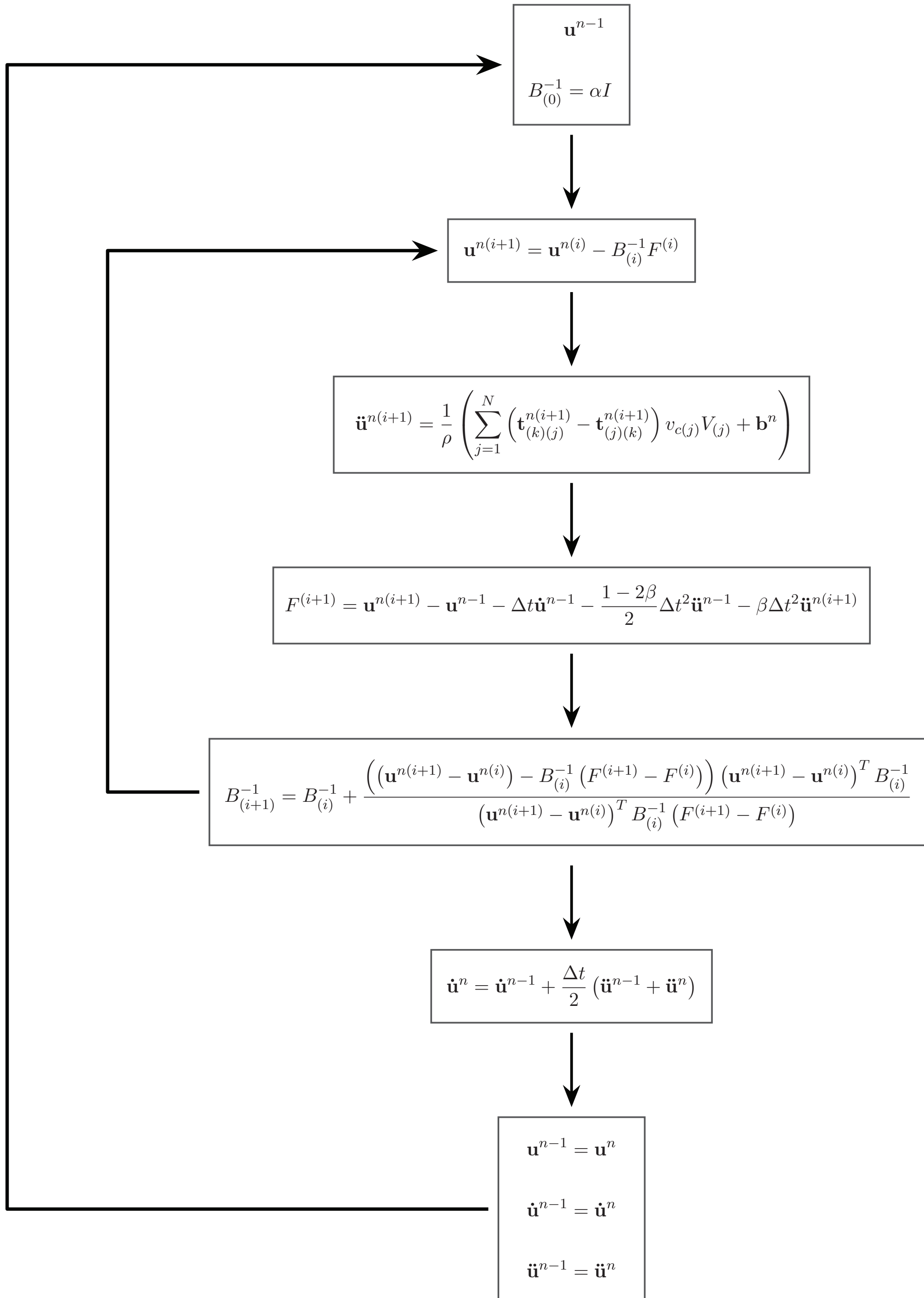


Figure 4.2: Implicit time integration algorithm with Newmark-beta method

4.5 Codes developed for implicit time integration

In the appendix C are presented the MATLAB and C++ implicit peridynamic codes used to obtain the results that will be discussed in the following sections. Two different codes have been developed: one that uses the backward finite difference method and another one that adopts the Newmark-beta method. Each line of code is commented to provide a better understanding of the operations that it performs.

4.6 Validation benchmarks results

A different benchmark is devised to validate the results of the implicit time integration codes. Instead of imposing an initial displacement to the nodes of the discretization, an external force is applied to one end of a bar in the direction of the bar's axis. This force is time dependent as shown in the following equation:

$$b_x = A \sin(wt) \tag{4.25}$$

In figure 4.3 it's shown a plot of the b_x force density value against time.

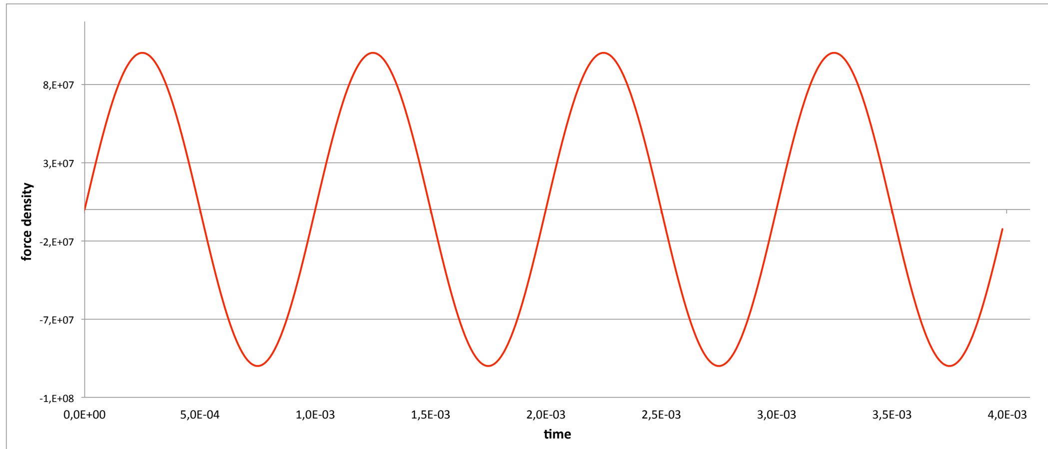


Figure 4.3: Force b_x against time

An analysis of this problem is also conducted with a FEM software and with the explicit integration code previously developed. The results provided by these two solutions will be compared to the results produced by the implicit integration codes. The benchmarks are performed with both the Newmark-beta method and the backward second order finite difference method.
The test is performed on a mono-dimensional bar of length $L = 1$ discretized with 100 nodes and with $E = 200e + 9$ and $\rho = 7850$.

4.6.1 Newmark-beta benchmark results

This test is performed with two different time step sizes: one with a step size of $\Delta t = 1e - 5\,s$ and one with $\Delta t = 2e - 5\,s$. The explicit integration results are obtained with a time step of $\Delta t = 1e - 6\,s$ In the graphs that will follow the displacement of the nodes at $x = L$ and $x = 0.5L$ is plotted against time. In figure 4.4 the implicit peridynamic results obtained with the Newmark-beta method are compared to the FEM results while in figure 4.5 they are compared to the explicit time integration results, for a time step size of $\Delta t = 2e - 5\,s$.

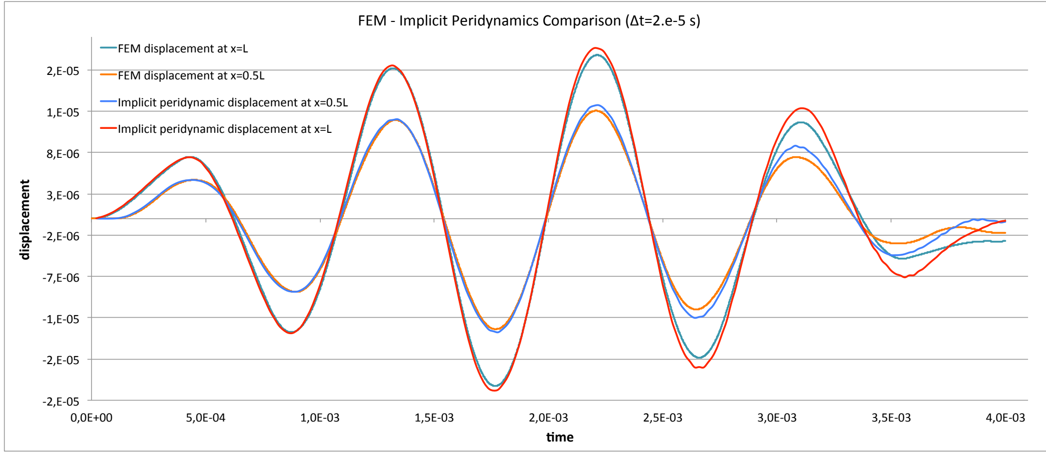


Figure 4.4: Implicit peridynamic with Newmark-beta and FEM results for 1D benchmark with $\Delta t = 2e - 5\,s$

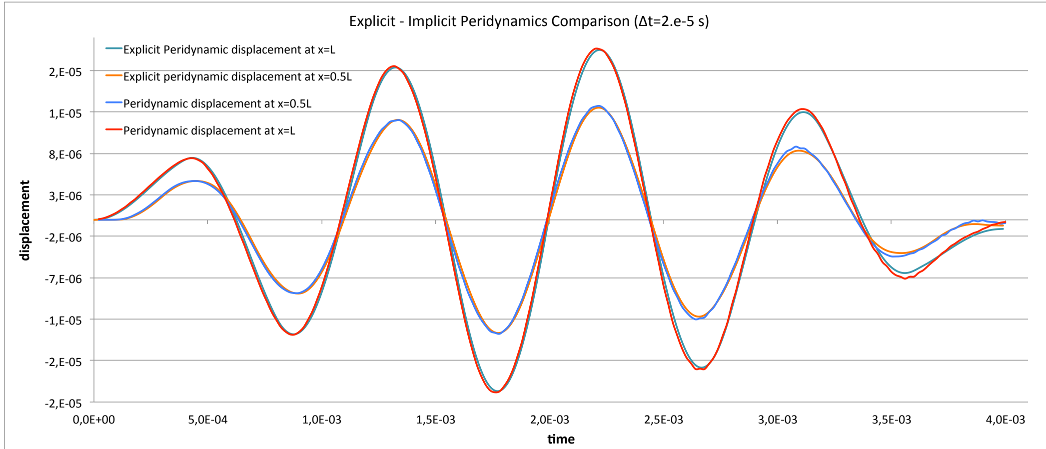


Figure 4.5: Implicit with Newmark-beta and explicit peridynamic results for 1D benchmark with $\Delta t = 2e - 5\,s$

In figure 4.6 the implicit peridynamic results are compared to the FEM results while in figure 4.7 they are compared to the explicit time integration results, for a time

step size of $\Delta t = e - 5\,s$.

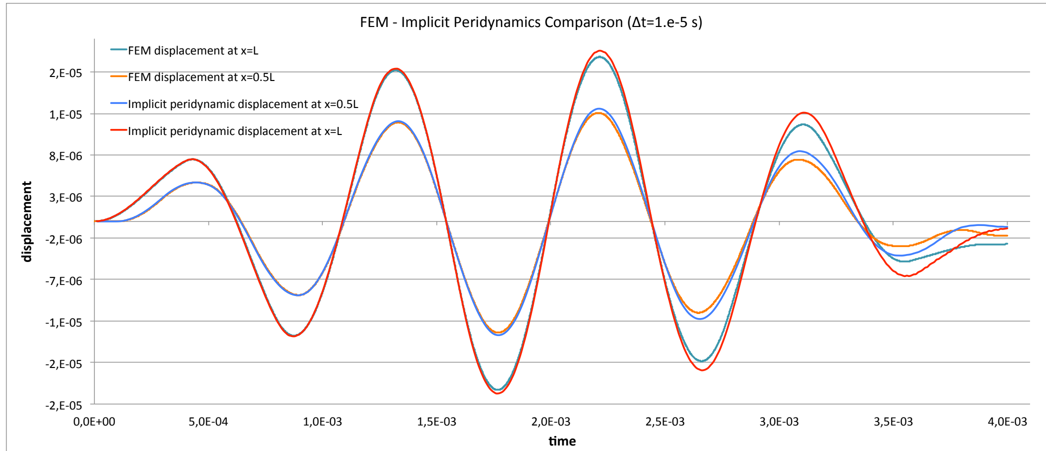


Figure 4.6: Implicit peridynamic with Newmark-beta and FEM results for 1D benchmark with $\Delta t = 1e - 5\,s$

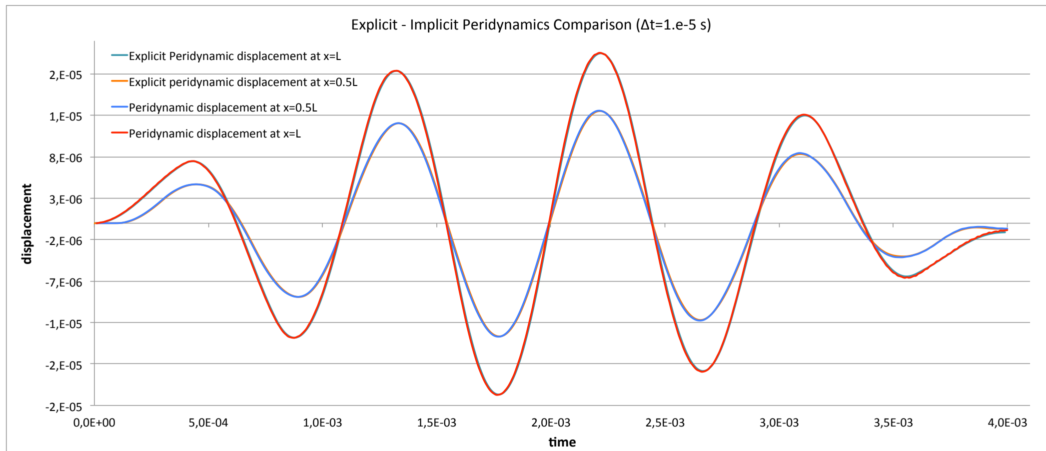


Figure 4.7: Implicit with Newmark-beta and explicit peridynamic results for 1D benchmark with $\Delta t = 1e - 5\,s$

We can notice from figures 4.4 and 4.6 that the discrepancy between the FEM and the implicit time integration results is very narrow at the beginning of the simulation and it tends to widen as the time proceeds.
Instead, looking at figures 4.5 and 4.7, we can see that the implicit integration results match very well the explicit integration results. This is particularly clear in figure 4.7, where the implicit time integration has been performed with the smaller time step.

4.6.2 Backward finite difference benchmark results

Also this benchmark is performed with two different time step sizes: $\Delta t = 1e - 5\,s$ and $\Delta t = 2e - 5\,s$. In figure 4.8 the implicit peridynamic results, obtained with the backward finite difference method, are compared to the FEM results while in figure 4.9 they are compared to the explicit time integration results, for a time step size of $\Delta t = 2e - 5\,s$.

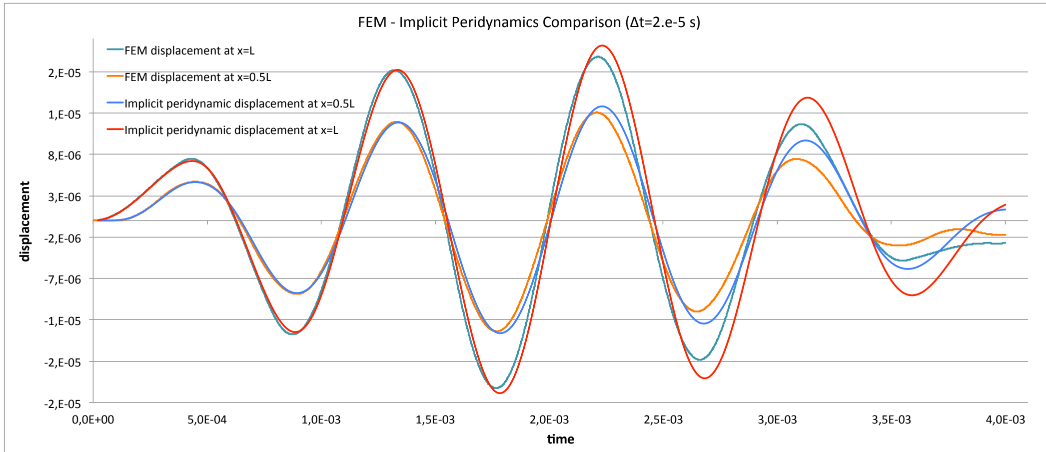


Figure 4.8: Implicit peridynamic with backward finite difference and FEM results for 1D benchmark with $\Delta t = 2e - 5\,s$

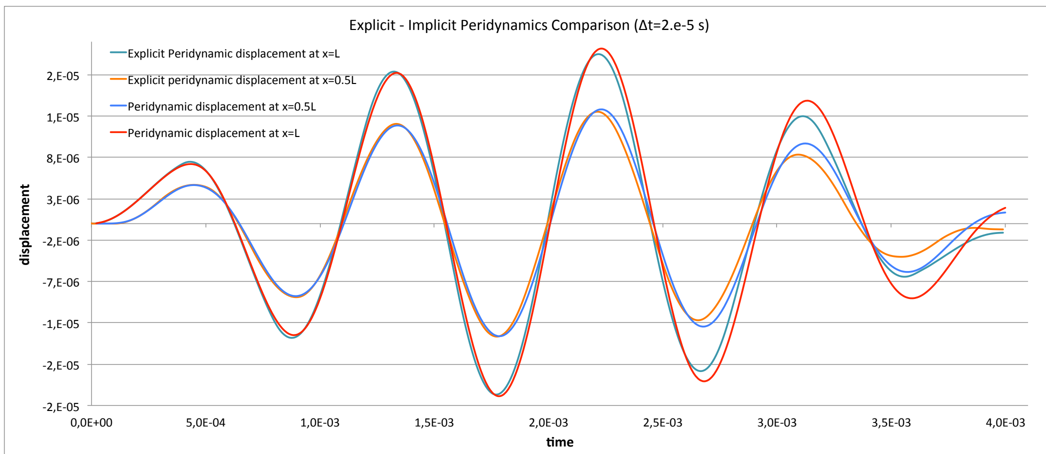


Figure 4.9: Implicit with backward finite difference and explicit peridynamic results for 1D benchmark with $\Delta t = 2e - 5\,s$

In figure 4.10 the implicit backward finite difference peridynamic results are compared to the FEM results while in figure 4.11 they are compared to the explicit time integration results, for a time step size of $\Delta t = e - 5\,s$.

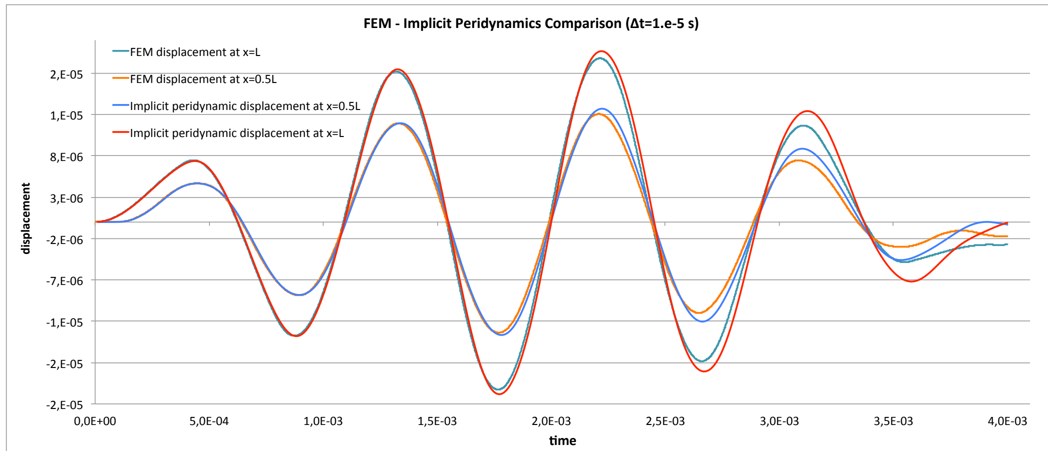


Figure 4.10: Implicit peridynamic with backward finite difference and FEM results for 1D benchmark with $\Delta t = 1e - 5\,s$

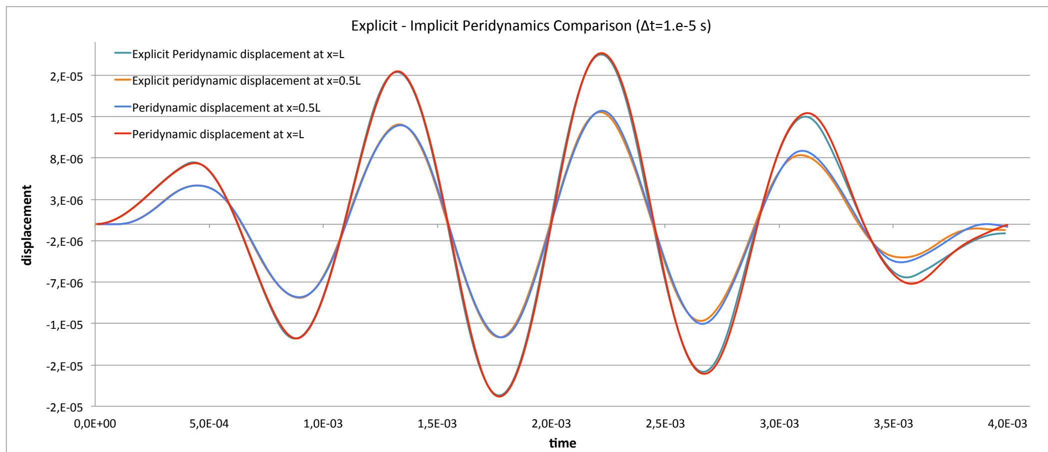


Figure 4.11: Implicit with backward finite difference and explicit peridynamic results for 1D benchmark with $\Delta t = 1e - 5\,s$

Once again we can see that the implicit peridynamic results tend to deviate from the FEM results toward the end of the simulation while they generally show a better agreement with the explicit time integration results. From figure 4.11 and also particularly form figure 4.9 we notice that the implicit results obtained with this method show a greater discrepancy with the explicit results than what we observed in the implicit Newmark-beta method.

4.7 Comparison between Newmark-beta and backward finite difference implicit time integration performances

The first performance test on the two methods is measuring the speed of the iterative Broyden scheme for the solution of the system of equations. This is measured by the number of iterative steps needed to reach convergency on the system solution. There is no need to perform this test on both the MATLAB and C++ codes, as they are going to give the same results.
The second performance test consists in measuring the computational time required to complete the simulation. This test will be performed also on the C++ code, as we expect a some difference in terms of computational time between the MATLAB code and the C++ code.
All the performance tests have been conducted on the mono-dimensional codes.

4.7.1 Iterative Broyden scheme performance

With each time step a system of equations has to be solved. The root of the system of equations gives the displacement's components at the current time step. The solution of the system is achieved through a Broyden iterative process. In table 4.1 are recorded the total number of iterations necessary to find the root of the equations and the mean value of iterations per time step, obtained dividing the total number of iterations by the number of time steps.

	Newmark-beta		Backward finite difference	
time steps	tot. iterations	iterations per step	tot. iterations	iterations per step
10	46	4.60	32	3.20
50	242	4.84	256	5.12
100	492	4.92	576	5.76
200	992	4.96	1304	6.53
400	1992	4.98	2707	6.77
800	3992	4.99	5532	6.92
1600	7992	5.00	11311	7.07

Table 4.1: Speed of convergency of Broyden iterative method for 1D idealization with 100 nodes and $\Delta t = 1e-5\,s$

From the previous table we can notice that the Broyden iterative process converges to the solution faster when the Newmark-beta method is used, with a mean value of 5 iterations per cycle instead of the 7 required by the backward finite difference

method.
We have to point out that these results are referred to the specific problem used to perform the tests. Different problems may give different results in terms of convergency speed.

4.7.2 Computational time performance

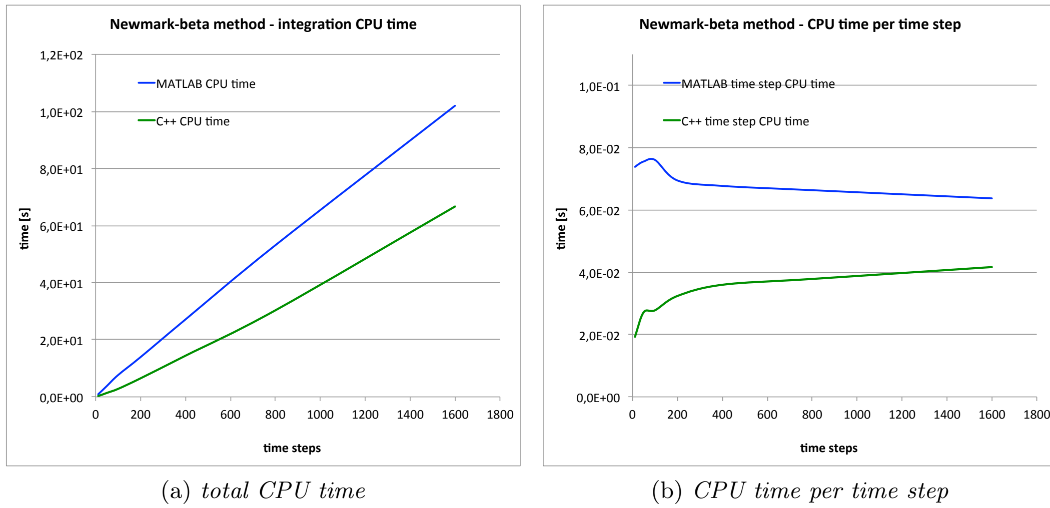
This test is performed by measuring the computational time required to accomplish the integration procedure, from the start of the integration to its end, without considering the pre and post integration tasks. The simulation is carried out on the mono-dimensional bar discretized with 100 nodes with increasing number of time steps. This test is conducted on the Newmark-beta and the backward finite difference method for both the MATLAB and C++ codes.
As stated in the previous chapter the computational times measured with this tests are referred to the testing configuration of table 3.1. To ensure that the measures are not influenced by external factors, each testing session has been carried out after a system reboot and all the unnecessary applications have been shut down before starting the test. Some of the operations performed by MATLAB, like matrix multiplications, make use of parallel computing. Here will be recorded the actual CPU execution time, or in other words the sum of the time each processor spent performing the tasks. All the CPU times recorded are intended as the mean value over three measures.

	MATLAB		C++	
time steps	CPU time NB [s]	CPU time BFD [s]	CPU time NB [s]	CPU time BFD [s]
10	7.40e-1	5.50e-1	1.93e-1	1.01e-1
50	3.78e+0	3.54e+0	1.36e+0	1.48e+0
100	7.61e+0	7.83e+0	2.78e+0	3.86e+0
200	1.39e+1	1.68e+1	6.48e+0	8.94e+0
400	2.71e+1	3.41e+1	1.44e+1	1.81e+1
800	5.31e+1	6.82e+1	3.03e+1	3.73e+1
1600	1.02e+2	1.36e+2	6.67e+1	7.72e+1

Table 4.2: Computational times of the implicit integration process. NB indicates the Newmark-beta method and BFD indicated the backward finite difference method.

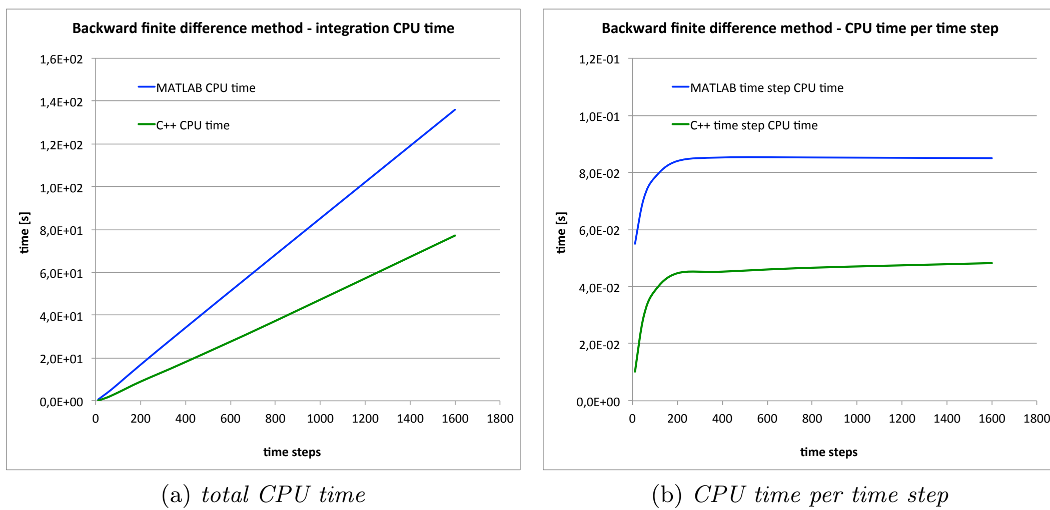
In figures 4.12 and 4.13 are reported the plots of the total integration CPU time (left) and the CPU time for one integration time step (right) for the Newmark-beta and the backward finite difference methods respectively.

We can notice that the C++ code is more performant than the MATLAB counterpart, even though the performance are of the same order of magnitude.



(a) *total CPU time* (b) *CPU time per time step*

Figure 4.12: Performance of the Newmark-beta method MATLAB and C++ codes



(a) *total CPU time* (b) *CPU time per time step*

Figure 4.13: Performance of the backward finite difference method MATLAB and C++ codes

4.8 Comparison between explicit and implicit time integration performances

The implicit and explicit integration performances are tested on two different mono-dimensional simulations. The first problem is a bar subjected to a force applied on one end of the bar as shown in equation 4.25 and figure 4.3. The second problem is a mono-dimensional bar subjected to an initial displacement as described by equation 3.6. The simulations are performed with a discretization of 50, 100 and 200 nodes and for increasing values of the total simulation time.

The time step sizes used for the first simulations are recorded in table 4.3. For the explicit integration process the maximum stable time step size that ensures the stability of the integration is adopted. For each time step of implicit integration ten time steps of explicit integration will be performed, so that the total time of the simulation remains the same.

n° nodes	explicit Δt [s]	implicit Δt [s]
50	7.00e-6	7.00e-5
100	3.50e-6	3.50e-5
200	1.75e-6	1.75e-5

Table 4.3: Time step size used in explicit and implicit integration for the first benchmark

In table 4.4 are recorded the CPU times of the explicit integration, in tables 4.5 and 4.6 are respectively recorded the CPU times of the implicit integration with the Newmark-beta method and the backward finite difference method for the first benchmark.

	50 nodes		100 nodes		200 nodes	
steps	MATLAB [s]	C++ [s]	MATLAB [s]	C++ [s]	MATLAB [s]	C++ [s]
100	1.60e-1	2.85e-3	2.90e-1	3.22e-3	5.80e-1	6.76e-3
500	7.50e-1	9.24e-3	1.46e+0	1.41e-2	3.11e+0	2.23e-2
1000	1.55e+0	1.41e-2	2.96e+0	2.31e-2	5.81e+0	4.02e-2
5000	6.79e+0	5.01e-2	1.29e+1	9.35e-2	2.54e+1	1.79e-1

Table 4.4: Explicit CPU time for the first benchmark

From these data we can observe that the explicit time integration seems to be more efficient for the solution of the problem analyzed in this circumstance. Moreover we can see that the C++ implementation of the implicit time integration becomes less and less efficient with increasing number of nodes with respect to the MATLAB counterpart.
However the implicit integration scheme requires the solution of a system of equations multiple times for each integration step. This solution involves multiplications between matrices which are performed by MATLAB using parallel computing. As a result the real time required to perform each integration step is shorter than the

steps	50 nodes MATLAB [s]	50 nodes C++ [s]	100 nodes MATLAB [s]	100 nodes C++ [s]	200 nodes MATLAB [s]	200 nodes C++ [s]
10	4.80e-1	3.16e-1	2.26e+0	1.27e+0	2.97e+0	7.15e+0
50	2.09e+0	1.62e+0	1.16e+1	8.08e+0	1.77e+1	4.44e+1
100	4.39e+0	3.13e+0	2.41e+1	1.57e+1	3.76e+1	9.38e+1
500	1.93e+1	1.66e+1	1.43e+2	8.44e+1	2.28e+2	5.46e+2

Table 4.5: Explicit-implicit CPU time comparison with the Newmark-beta method for the first benchmark

steps	50 nodes MATLAB [s]	50 nodes C++ [s]	100 nodes MATLAB [s]	100 nodes C++ [s]	200 nodes MATLAB [s]	200 nodes C++ [s]
10	6.01e-1	3.86e-1	2.54e+0	1.61e+0	2.87e+0	4.81e+0
50	2.42e+0	1.96e+0	1.39e+1	1.11e+1	1.82e+1	4.32e+1
100	5.53e+0	3.89e+0	2.73e+1	2.02e+1	4.02e+1	1.06e+2
500	2.35e+1	2.01e+1	1.28e+2	9.47e+1	2.07e+2	5.55e+2

Table 4.6: Explicit-implicit CPU time comparison with the backward finite difference method for the first benchmark

CPU time here reported, as more than one CPU is used at the same time. The table 4.7 shows a comparison between the real elapsed time of the explicit and implicit Newmark-beta codes performed with MATLAB while in table 4.8 it is shown the same comparison obtained with the backward finite difference method.

steps	50 nodes explicit [s]	50 nodes implicit [s]	100 nodes explicit [s]	100 nodes implicit [s]	200 nodes explicit [s]	200 nodes implicit [s]
100-10	1.51e-1	5.43e-1	2.67e-1	6.16e-1	5.30e-1	9.91e-1
500-50	7.09e-1	2.42e+0	1.33e+0	3.19e+0	2.66e+0	5.25e+0
1000-100	1.39e+0	4.86e+0	2.60e+0	6.56e+0	5.09e+0	1.11e+1
5000-500	6.77e+0	2.44e+1	1.23e+1	4.05e+1	2.47e+1	6.18e+1

Table 4.7: Explicit and implicit Newmark-beta real execution times for the first benchmark obtained with the MATLAB codes

As we can notice from tables 4.7 and 4.8, the explicit integration MATLAB code still requires less time to be executed than it's implicit counterparts.

The second simulation is performed with the time steps shown in table 4.9. Once again the size of the time step used in the implicit simulation is ten times the one used in the explicit simulation.
As shown previously, in table 4.10 are recorded the CPU times of the explicit integration, in tables 4.11 and 4.12 are respectively recorded the CPU times of the implicit integration with the Newmark-beta method and the backward finite differ-

steps	50 nodes explicit [s]	50 nodes implicit [s]	100 nodes explicit [s]	100 nodes implicit [s]	200 nodes explicit [s]	200 nodes implicit [s]
100-10	1.51e-1	5.13e-1	2.67e-1	6.84e-1	5.30e-1	7.30e-1
500-50	7.09e-1	2.57e+0	1.33e+0	3.48e+0	2.66e+0	4.86e+0
1000-100	1.39e+0	5.11e+0	2.60e+0	7.24e+0	5.09e+0	1.06e+1
5000-500	6.77e+0	2.28e+1	1.23e+1	3.33e+1	2.47e+1	5.27e+1

Table 4.8: Explicit and implicit backward finite difference real execution times for the first benchmark obtained with the MATLAB codes

n^o nodes	explicit Δt [s]	implicit Δt [s]
50	1.0e-6	1.0e-5
100	7.0e-7	7.0e-6
200	4.0e-7	4.0e-6

Table 4.9: Time step size used in explicit and implicit integration for the second benchmark

ence method for the second benchmark.

steps	50 nodes MATLAB [s]	50 nodes C++ [s]	100 nodes MATLAB [s]	100 nodes C++ [s]	200 nodes MATLAB [s]	200 nodes C++ [s]
100	2.20e-1	2.79e-3	3.70e-1	4.49e-3	6.80e-1	7.26e-3
500	9.60e-1	8.85e-3	1.57e+0	1.35e-2	3.19e+0	2.27e-2
1000	1.78+0	1.38e-2	3.12e+0	2.37e-2	5.59e+0	4.09e-2
5000	7.17+0	4.93e-2	1.31e+1	9.64e-2	2.52e+1	1.79e-1

Table 4.10: Explicit CPU time for the second benchmark

steps	50 nodes MATLAB [s]	50 nodes C++ [s]	100 nodes MATLAB [s]	100 nodes C++ [s]	200 nodes MATLAB [s]	200 nodes C++ [s]
10	4.70e-1	4.58e-2	1.10e+0	4.99e-1	1.73e+0	2.71e+0
50	2.13e+0	2.48e-1	5.25e+0	2.55e+0	8.51e+0	1.59e+1
100	4.38e+0	4.83e-1	1.02e+1	4.96e+0	1.73e+1	2.89e+1
500	2.07e+1	2.70e+0	4.71e+1	2.48e+1	9.87e+1	1.81e+2

Table 4.11: Explicit-implicit CPU time comparison with the Newmark-beta method for the second benchmark

We can observe also from tables 4.11 and 4.12 how still the C++ implementation becomes less and less efficient, with respect to the MATLAB codes, as the number of nodes increases.

steps	50 nodes MATLAB [s]	C++ [s]	100 nodes MATLAB [s]	C++ [s]	200 nodes MATLAB [s]	C++ [s]
10	1.60e-1	6.81e-2	1.13e+0	5.68e-1	1.89e+0	4.25e+0
50	4.60e-1	2.65e-1	3.50e+0	2.20e+0	6.81e+0	1.68e+1
100	9.70e-1	4.76e-1	6.22e+0	4.42e+0	1.22e+1	3.58e+1
500	4.08e+0	2.31e+0	2.74e+1	2.31e+1	5.44e+1	1.84e+2

Table 4.12: Explicit-implicit CPU time comparison with the backward finite difference method for the second benchmark

The tables 4.13 and 4.14 show the comparison in the total execution time of the explicit and implicit MATLAB codes for the second benchmark problem.

steps	50 nodes explicit [s]	implicit [s]	100 nodes explicit [s]	implicit [s]	200 nodes explicit [s]	implicit [s]
100-10	1.36e-1	1.17e-1	2.62e-1	2.68e-1	5.58e-1	4.17e-1
500-50	6.91e-1	5.08e-1	1.38e+0	1.19e+0	2.71e+0	1.98e+0
1000-100	1.37e+0	9.86e-1	2.64e+0	2.40e+0	5.32e+0	4.11e+0
5000-500	6.53e+0	4.85e+0	1.27e+1	1.12e+1	2.41e+1	2.33e+1

Table 4.13: Explicit and implicit Newmark-beta real execution times for the first benchmark obtained with the MATLAB codes

steps	50 nodes explicit [s]	implicit [s]	100 nodes explicit [s]	implicit [s]	200 nodes explicit [s]	implicit [s]
100-10	1.36e-1	1.38e-1	2.62e-1	2.74e-1	5.58e-1	4.69e-1
500-50	6.91e-1	4.16e-1	1.38e+0	8.62e-1	2.71e+0	1.67e+0
1000-100	1.37e+0	7.77e-1	2.64e+0	1.52e+0	5.32e+0	3.09e+0
5000-500	6.53e+0	3.45e+0	1.27e+1	6.70e+0	2.41e+1	1.34e+1

Table 4.14: Explicit and implicit backward finite difference real execution times for the first benchmark obtained with the MATLAB codes

On the contrary to what pointed out from the first benchmark, we can see from tables 4.13 and 4.14 that the implicit time integration is performed in a slightly shorter amount of time than the explicit integration. We can deduce that in some conditions the implicit time integration can be preferable to the explicit one. We can also infer that an efficient solution method for the system of equations is key in obtaining a performant implicit integration.

Chapter 5

Viscoelastic material simulation

In this chapter the peridynamic viscoelastic material description by Mitchell[10] is presented. His approach is used in the creation of an explicit peridynamic solution scheme for the bond-based theory formulation. A benchmark problem is devised to test the results of the peridynamic viscoelastic solution.

5.1 Notation and definitions

Let $e_{(k)(j)}$ be defined as the scalar extension state:

$$e_{(k)(j)} = \left|Y_{(k)(j)}\right| - \left|X_{(k)(j)}\right| \tag{5.1}$$

where $\left|Y_{(k)(j)}\right|$ and $\left|X_{(k)(j)}\right|$ are define respectively as:

$$\left|Y_{(k)(j)}\right| = \left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| \qquad \left|X_{(k)(j)}\right| = \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right| \tag{5.2}$$

The scalar extension state of equation 5.1 can be decomposed additively in its spherical and deviatoric parts:

$$e_{(k)(j)} = e^{i}_{(k)(j)} + e^{d}_{(k)(j)} \tag{5.3}$$

The spherical extension state is defined as:

$$e^{i}_{(k)(j)} = \frac{\theta_{(k)(j)} \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|}{3} \tag{5.4}$$

where $\theta_{(k)(j)}$ is the dilatation term. Substituting equation 5.4 into 5.3 we obtain the following expression for the deviatoric part $e^d_{(k)(j)}$:

$$e^d_{(k)(j)} = e_{(k)(j)} - e^i_{(k)(j)} = \left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| - \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right| - \frac{\theta_{(k)(j)}\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|}{3} \tag{5.5}$$

5.2 Viscoelastic model

In the viscoelastic model presented by Mitchell in [10] the viscoelasticity concept is applied to the deviatoric part of the scalar extension state. The addictive decomposition is extended once more to the deviatoric extension state to include a back extension state.

The interaction between two nodes is now modeled as shown in figure 5.1.

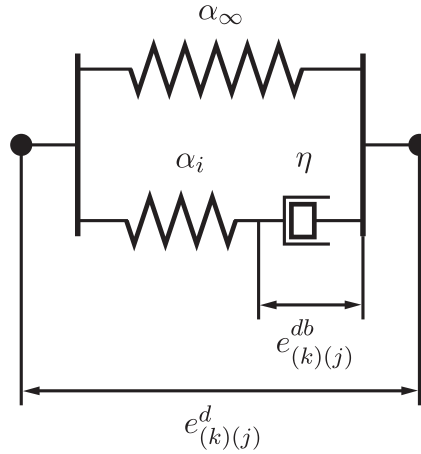


Figure 5.1: Model of viscoelastic interaction

The deviatoric part is decomposed into its elastic and back parts as follows:

$$e^d_{(k)(j)} = e^{de}_{(k)(j)} + e^{db}_{(k)(j)} \tag{5.6}$$

The elastic strain energy density then becomes:

$$W_{(k)(j)} = \frac{k\theta^2}{2} + \frac{\alpha_\infty}{2} e^d_{(k)(j)} \bullet w e^d_{(k)(j)} + \frac{\alpha_i}{2}\left(e^d_{(k)(j)} - e^{db}_{(k)(j)}\right) \bullet w \left(e^d_{(k)(j)} - e^{db}_{(k)(j)}\right) \tag{5.7}$$

Where the dot product $\bullet$ between two states $\underline{\mathbf{A}}$ and $\underline{\mathbf{B}}$ is defined in [15] as:

$$\underline{\mathbf{A}} \bullet \underline{\mathbf{B}} = \int_H \left(\underline{\mathbf{AB}}\right) \langle \boldsymbol{\xi} \rangle \, \mathrm{d}V_{\boldsymbol{\xi}} \tag{5.8}$$

and the point product of the two states $\left(\underline{\mathbf{AB}}\right) \langle \boldsymbol{\xi} \rangle = \underline{\mathbf{A}}\langle \boldsymbol{\xi} \rangle \underline{\mathbf{B}}\langle \boldsymbol{\xi} \rangle$.

The volumetric and deviatoric components o the scalar force state t are obtained from the function W as follows:

$$t^i_{(k)(j)} = \frac{\partial W_{(k)(j)}}{\partial \theta} \frac{\partial \theta}{\partial e^i_{(k)(j)}} \qquad t^d_{(k)(j)} = \frac{\partial W_{(k)(j)}}{\partial e^d_{(k)(j)}} \tag{5.9}$$

Evaluating equation 5.9 for the bond-based peridynamics we obtain the following value of the force:

$$t_{(k)(j)} = \left(\alpha_\infty + \alpha_i\right) w e^d_{(k)(j)} - \alpha_i w e^{db}_{(k)(j)} \tag{5.10}$$

The value of $\alpha_\infty w$ in bond-based peridynamic theory is given by:

$$\alpha_\infty w = \frac{2\delta b}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \tag{5.11}$$

The value of $\alpha_\infty w$ expressed in equation 5.11 ensures also that the viscoelastic solution reverts back to the steady state solution after a sufficient amount of time.

Let us define $\alpha = \alpha_\infty + \alpha_i$ and $\alpha_i = \lambda_i \alpha$. From these definitions we can obtain the following relations:

$$\alpha_\infty = \left(1 - \lambda_i\right) \alpha \tag{5.12}$$

$$\alpha = \frac{\alpha_\infty}{1 - \lambda_i} \qquad \alpha_i = \frac{\lambda_i}{1 - \lambda_i} \alpha_\infty \tag{5.13}$$

Substituting the value of α_i from equation 5.13 and the value of α_∞ from equation 5.11 into equation 5.10 we obtain the following expression for the bond-based scalar force:

$$t_{(k)(j)} = \frac{2\delta b}{\left|\boldsymbol{\xi}_{(k)(j)}\right|} \frac{1}{1-\lambda_i} \left(\left|\boldsymbol{\xi}_{(k)(j)} + \boldsymbol{\eta}_{(k)(j)}\right| - \left|\boldsymbol{\xi}_{(k)(j)}\right| - \lambda_i e^{db}_{(k)(j)} \right) \tag{5.14}$$

To evaluate the previous expression we need first to know the value of e^{db}. The force t^d in the dashpot, by Newton's third law, is also the force in the spring, as shown in figure 5.2.

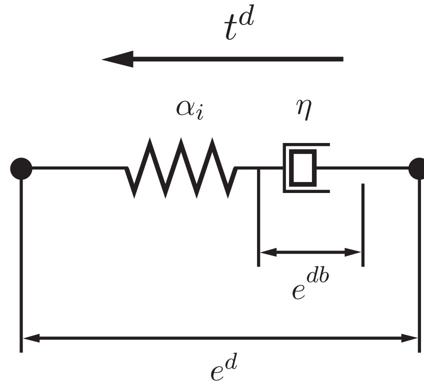


Figure 5.2: Deviatoric force in the dashpot and spring

The value of the force is given by:

$$\begin{aligned} t^d &= \eta \dot{e}^{db} \\ &= \alpha_i \left(e^d - e^{db} \right) \end{aligned} \tag{5.15}$$

The evolution of e^{db} is then given by the following equation:

$$\dot{e}^{db} = \frac{1}{\tau^b} \left(e^d - e^{db} \right) \tag{5.16}$$

where $\tau^b = \frac{\eta}{\alpha_i}$ is the time constant associated with the material response. Equation 5.16 highlights that the state of a viscoelastic material is history dependent. The

solution of equation 5.16 is assumed to exist and to take the form of:

$$e^{db} = \frac{1}{\tau^b} \int_{-\infty}^{t} e^{\frac{-(t-s)}{\tau^b}} e^d(s)\, \mathrm{d}s \tag{5.17}$$

In order to solve this integral it would be necessary to know all the history of $e^d(t)$ which is not feasible for a practical implementation. In numeric solutions e^d is known only in discrete points in time. Let us indicate with t_n the time of the simulation at the n-th step and with $t_{n+1} = t_n + \Delta t$ the time at the next time step. The values of e^d_{n+1}, e^d_n and e^{db}_n are known and we need to find the value of e^{db}_{n+1}. We assume that $e^{db}(t) \to 0$ as $t \to -\infty$. The integral of equation 5.17 is rewritten as follows:

$$e^{db}_{n+1} = e^d_{n+1} - \int_{-\infty}^{t_n} e^{\frac{-(t_n-s)}{\tau^b}} \dot{e}^d(s)\, \mathrm{d}s \tag{5.18}$$

We substitute $e^d_{n+1} = e^d_n + \Delta e^d$ and we break the integral in two parts as follows:

$$e^{db}_{n+1} = e^d_n + \Delta e^d - e^{\frac{-\Delta t}{\tau^b}} \int_{-\infty}^{t_n} e^{\frac{-(t_n-s)}{\tau^b}} \dot{e}^d(s)\, \mathrm{d}s - \int_{t_n}^{t_{n+1}} e^{\frac{-(t_{n+1}-s)}{\tau^b}} \dot{e}^d(s)\, \mathrm{d}s \tag{5.19}$$

The first integral in the previous expression can be rewritten as:

$$e^{\frac{-\Delta t}{\tau^b}} \int_{-\infty}^{t_n} e^{\frac{-(t_n-s)}{\tau^b}} \dot{e}^d(s)\, \mathrm{d}s = e^{\frac{-\Delta t}{\tau^b}} \left(e^d_n - e^{db}_n\right) \tag{5.20}$$

Substituting this expression into equation 5.19 we get:

$$e^{db}_{n+1} = e^d_n \left(1 - e^{\frac{-\Delta t}{\tau^b}}\right) + e^{db}_n e^{\frac{-\Delta t}{\tau^b}} + \Delta e^d - \int_{t_n}^{t_{n+1}} e^{\frac{-(t_{n+1}-s)}{\tau^b}} \dot{e}^d(s)\, \mathrm{d}s \tag{5.21}$$

The scalar deviatoric extension state $e^d(t)$ is estimated by linearly interpolating over the time step:

$$e^d(t) \approx e^d_n + \frac{\Delta e^d}{\Delta t}(t - t_n) \qquad t_n \le t \le t_{n+1} \tag{5.22}$$

this assumption allows to evaluate the integral in equation 5.21 as:

$$\int_{t_n}^{t_{n+1}} e^{\frac{-(t_{n+1}-s)}{\tau^b}} \dot{e}^d(s)\,\mathrm{d}s = \frac{\Delta e^d}{\Delta t}\left(1 - e^{\frac{-\Delta t}{\tau^b}}\right)\tau^b \tag{5.23}$$

So expression 5.21 becomes:

$$\begin{aligned} e^{db}_{n+1} &= e^d_n\left(1 - e^{\frac{-\Delta t}{\tau^b}}\right) + e^{db}_n e^{\frac{-\Delta t}{\tau^b}} + \Delta e^d - \frac{\Delta e^d}{\Delta t}\left(1 - e^{\frac{-\Delta t}{\tau^b}}\right)\tau^b \\ &= e^d_n\left(1 - e^{\frac{-\Delta t}{\tau^b}}\right) + e^{db}_n e^{\frac{-\Delta t}{\tau^b}} + \Delta e^d\left(1 - \frac{\tau^b}{\Delta t}\left(1 - e^{\frac{-\Delta t}{\tau^b}}\right)\right) \end{aligned} \tag{5.24}$$

5.3 Time integration explicit algorithm

Using equations 5.14 and 5.24 we can now build an iterative algorithm for of the peridynamic equation of motion with a viscoelastic material.

To start the integration algorithm it is necessary to know the initial value of $e^{db\,0}$ for all the bonds in the body. As suggested in [10] this initial value can be simply taken as 0, assuming all the material points of the discretization are stationary at the beginning of the simulation.
Using the initial condition of $e^{db\,0} = 0$ we can compute the initial value of the force t^0. Using the explicit integration technique we can calculate the displacement and the value of the deviatoric back extension at the next time step and start again the algorithm.

In figure 5.3 it is shown how the explicit integration algorithm works.

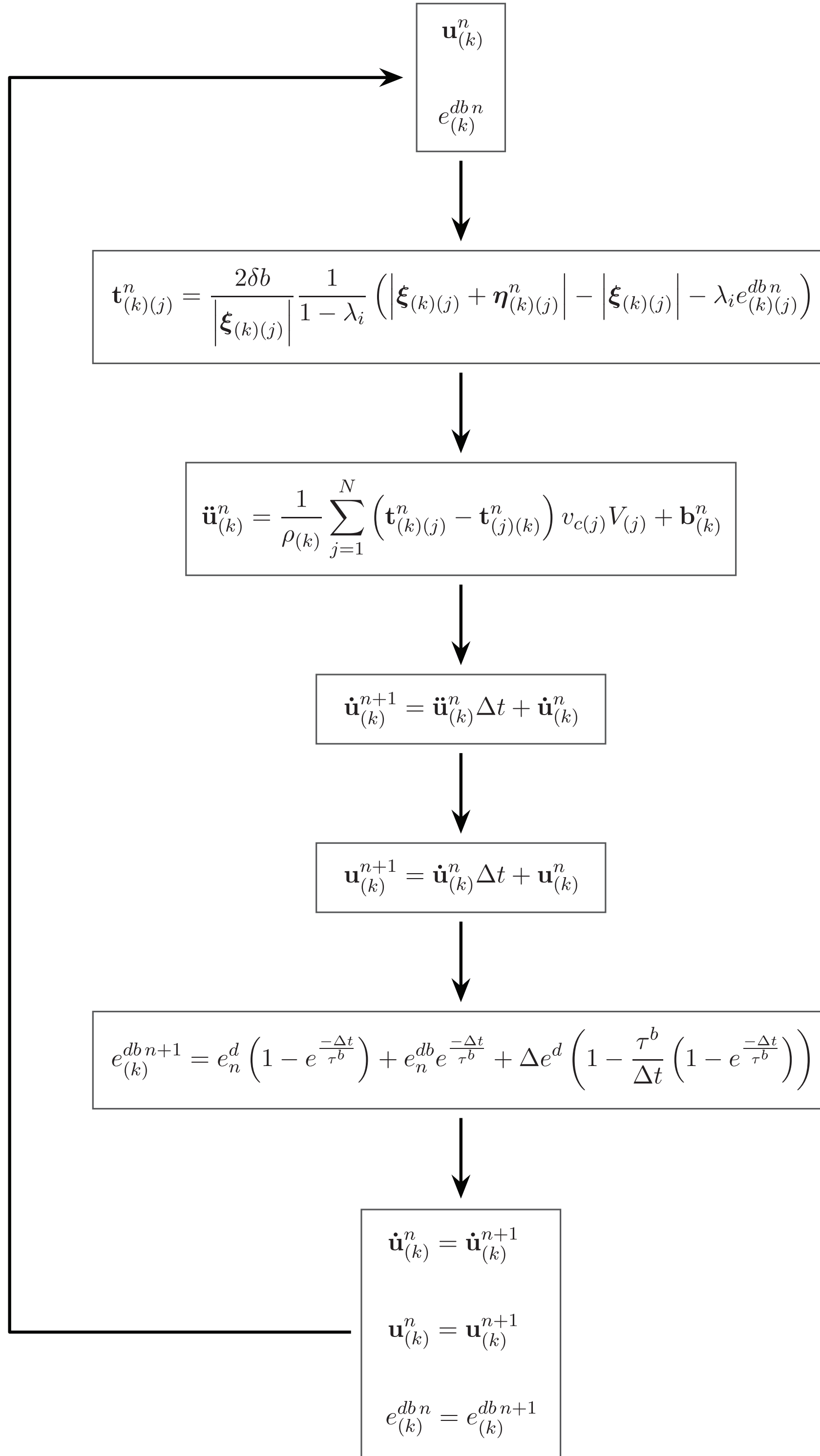


Figure 5.3: Explicit time integration algorithm for viscoelastic peridynamic

5.4 Benchmark results

The results produced by the peridynamic viscoelastic code are tested against the FEM results in two benchmark problems.

The first benchmark consists in a mono dimensional bar of length $L = 1$ which is discretized with 100 nodes. A constant force $F = 1$ is applied at one and of the bar in the direction of its axis. In figures 5.4 and 5.5 is shown a comparison of the peridynamic and FEM results for points located at $x = L$ and $x = 0.5L$ respectively.

We can notice from the plots that the peridynamic results match very well the FEM results. We can also see that the displacement of the node at $x = L$ tends toward $5.e - 6$ as the time of the simulation progresses. This value is exactly the final displacement value expected from equations 5.25:

$$u_{x=L} = \varepsilon_{x=L} L \qquad \varepsilon_{x=L} = \frac{\sigma}{E} \qquad \sigma = \frac{F}{A}$$
$$u_{x=L} = \frac{FL}{EA} = \frac{1}{1.e - 6\, 2.e + 11} = 5.e - 6 \tag{5.25}$$

We can also see in figure 5.6 the relative error between the FEM and the Peridynamic results against the time of the simulation.

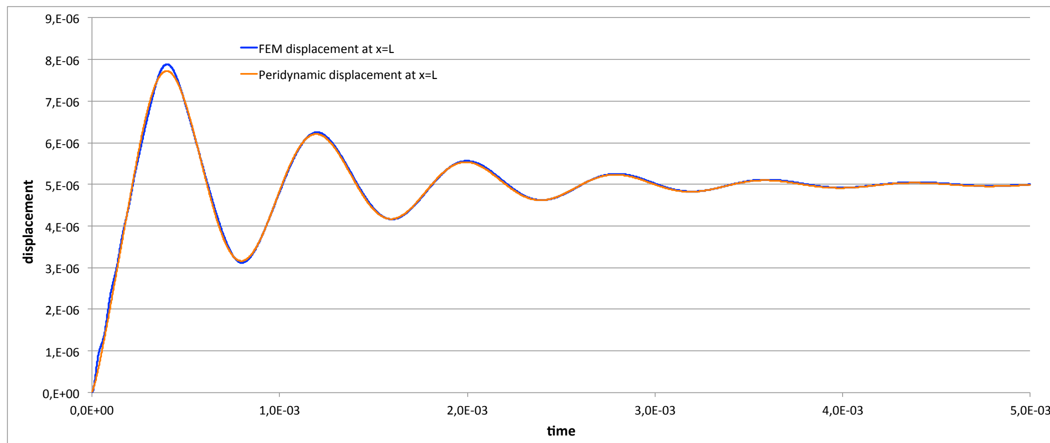


Figure 5.4: FEM (blue) and peridynamic (orange) displacement at $x = L$ of the first benchmark

In the second benchmark the constant force is replaced by a sinusoidal force described by equation 4.25. In figures 5.7 and 5.8 are shown the comparison of the FEM and peridynamic results for points located at $x = L$ and $x = 0.5L$, respectively. The green line in the plots represents the force value in the x direction.

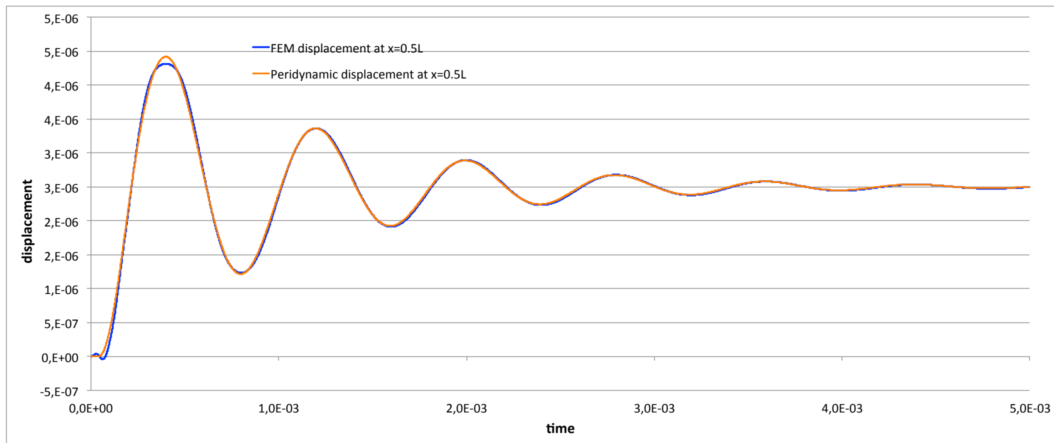


Figure 5.5: FEM (blue) and peridynamic (orange) displacement at $x = 0.5L$ of the first benchmark

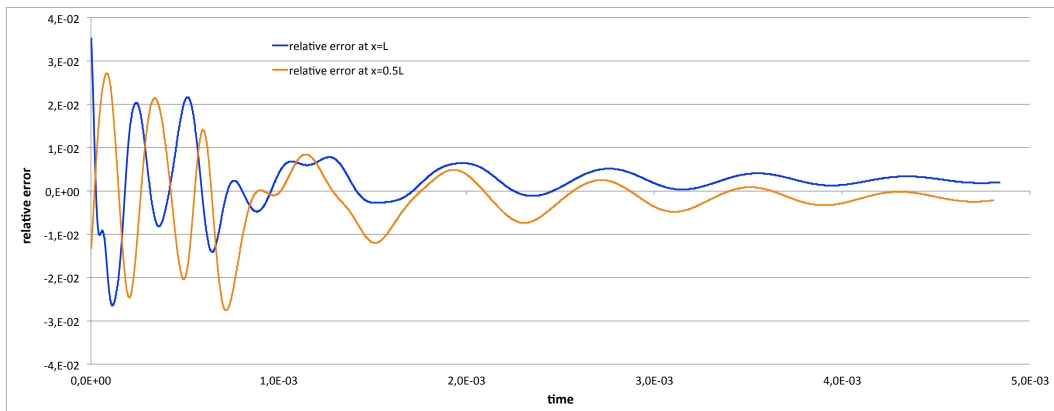


Figure 5.6: Relative FEM-Peridynamic error at x=0.5L (orange) and at x=L (blue) for the first benchmark

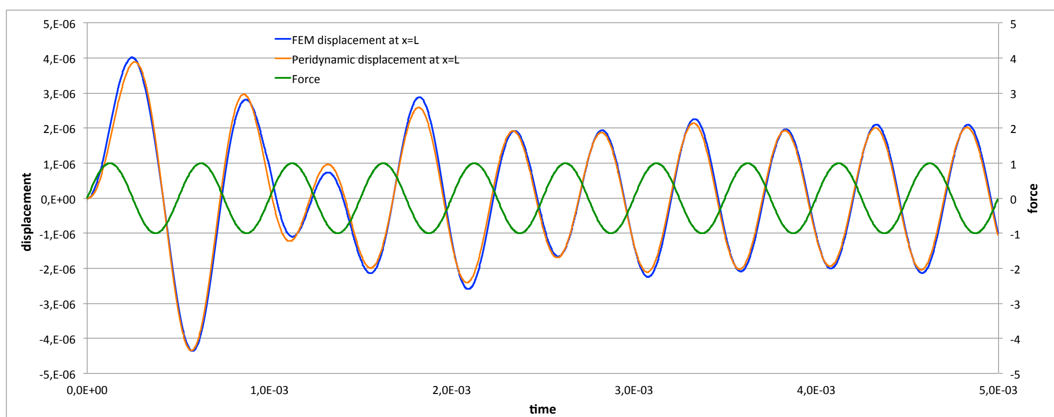


Figure 5.7: FEM (blue) and peridynamic (orange) displacement at $x = L$ of the second benchmark

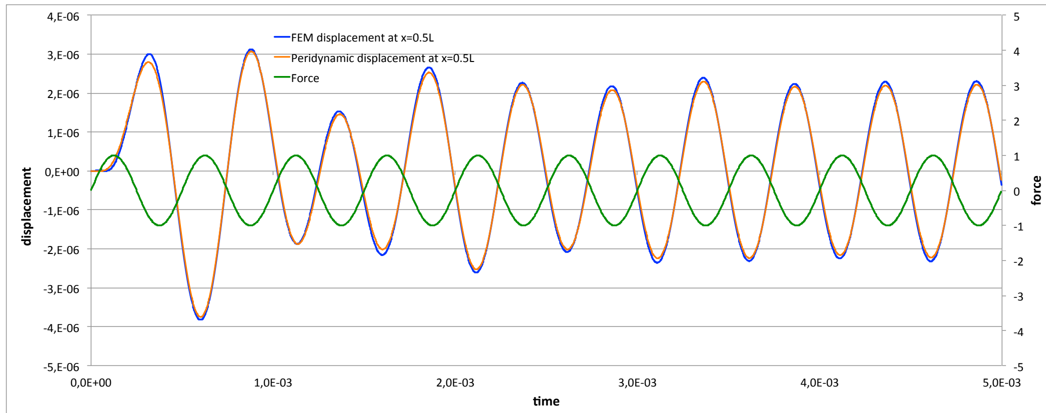


Figure 5.8: FEM (blue) and peridynamic (orange) displacement at $x = 0.5L$ of the second benchmark

Once again we can notice from figures 5.7 and 5.8 that the peridynamic results match very well the FEM results.

This viscoelastic model could be used in the solution of a fracture problem, to simulate how the material dampening can affect the crack growth and propagation.

Chapter 6

Further developments

When implementing a scheme for the implicit time integration of the peridynamic equation of motion it is necessary to solve a system of non linear equations. The greater the number of nodes in the discretization and the more complex it becomes to devise a solution to the system of equations. Also increasing the time step of the implicit simulation results in an increase in the number of iteration required to perform the solution of the system of equations. For this reason a robust and efficient method for the solution of the system of equations is of key importance in the development of an implicit integration scheme. So far only the Broyden's iterative method has been investigated. A future development might be to use different solution methods and compare their performance and stability.

The viscoelastic solution algorithm previously presented uses an explicit integration scheme to solve the peridynamic equation of motion. An explicit solution scheme could be applied also to this type of simulation and its stability and performance could be checked against the explicit solution algorithm.
Furthermore peridynamics allows to predict when and where damage will occur and the growth path of a crack. The viscoelastic material model can be used in the solution of a crack initiation and propagation problem to compare the different behaviour of a viscoelastic and a non-viscoelastic material.

Appendix A

Explicit time integration codes

Hereinafter will be provided the portion of the peridynamic codes that performs the explicit time integration. Both the MATLAB and C++ version of the codes will be shown. Every step of the code is commented to provide a better understanding of the operations performed.

A.1 MATLAB code

```
0001 fprintf('Starting time integration algorithm \n');
0002
0003 % initializing the variables for output saving
0004 perc_print=0;
0005 output_index=0;
0006
0007 % core calculation of peridynamic equation
0008 for n=1:time_steps
0009
0010     % performing a sweep of all (k) nodes
0011     for k=1:n_nodes
0012
0013         % initializing to zero the theta(k) value (if required)
0014         if a ~= 0
0015             theta(1,k)=0;
0016         end
0017
0018         % performing a sweep of all (j) nodes in the neighbourhood of the
0019         % (k) node
0020         for i=1:max_interactions_per_node
0021
0022             % checking if the ID of the (j) node is valid
0023             if  j_nodes_id(i,k) ~= 0
0024
0025                 % saving the ID of the (j) node in the j variable
0026                 j=j_nodes_id(i,k);
0027
0028                 % saving the ID of the interaction in the ind_id variable
0029                 int_id=interaction_id(i,k);
0030
0031                 % calculating the eta(k)(j) vector
0032                 eta_vector(:,int_id)=U(:,j)-U(:,k);
0033
0034                 % calculating the norm of the csi+eta vector
0035                 norm_csi_eta=norm( csi_vector(:,int_id)+eta_vector(:,int_id) );
0036
0037                 % calculating the stretch(k)(j) value
0038                 stretch(1,int_id)=( norm_csi_eta - csi_modulus(1,int_id) )/csi_modulus(1,int_id);
```

```
0039
0040                % calculating the unit vector of the interaction direction
0041                int_direction_unit(:,int_id)=( csi_vector(:,int_id)+ ...
0042                                             eta_vector(:,int_id) )/norm_csi_eta;
0043
0044                % theta calculation (if required)
0045                if a ~= 0
0046
0047                    % calculating the lambda(k)(j) value
0048                    lambda(1,int_id)=dot( int_direction_unit(:,int_id) ,...textcolorcomment
0049                                    csi_vector(:,int_id)/csi_modulus(1,int_id) );
0050
0051                    %calculating the theta(k) value
0052                    theta(1,k)=theta(1,k)+d*delta*G_d(1,int_id)*stretch(1,int_id)* ...
0053                               lambda(1,int_id)*v_c(1,int_id)*V(1,j);
0054
0055                end
0056
0057            end
0058        end
0059    end
0060
0061    % initializing to zero the force value
0062    force(:,:)=0;
0063
0064    % performing a second sweep of all (k) nodes to calculate the force
0065    % acting on the node
0066    for k=1:n_nodes
0067
0068        % setting the zero the variable that will contain the theta term
0069        % (if a has non-zero value)
0070        theta_term=0;
0071
0072        % performing a sweep of all (j) nodes in the neighbourhood of the
0073        % (k) node
0074        for i=1:max_interactions_per_node
0075
0076            % checking if the ID of the (j) node is valid
0077            if  j_nodes_id(i,k) ~= 0
0078
0079                % saving the ID of the (j) node in the j variable
0080                j=j_nodes_id(i,k);
0081
0082                % saving the ID of the interaction in the ind_id variable
0083                int_id=interaction_id(i,k);
0084
0085                % the theta term needs to be calculated only if a has a
0086                % non-zero value
0087                if a ~= 0
0088                    theta_term=2*a*d*delta*G_d(1,int_id)*lambda(1,int_id)/ ...
0089                               csi_modulus(1,int_id)*( theta(1,k)+theta(1,j) );
0090                end
0091
0092                % incrementing the value of the force
0093                force(:,k)=force(:,k) + ( theta_term + 4*b*delta*G_b(1,int_id)* ...
0094                           stretch(1,int_id) )*int_direction_unit(:,int_id)*( v_c(1,int_id)*V(1,j) );
0095
0096            end
0097        end
0098
0099        % calculating the acceleration vector
0100        U_dotdot(:,k)=( force(:,k)+body_f(:,k) )/ro(1,k);
0101
0102    end
0103
0104    % calculating the velocity vectors at the next time step
0105    U_dot=U_dotdot*delta_t+U_dot;
```

```
0106
0107     % calculating the displacement vectors at the next time step
0108     U=U_dot*delta_t+U;
0109
0110
0111     % setting to zero the displacement value of the boundary nodes
0112     for k=1:i_boundarynodes
0113         U(:,k)=0;
0114     end
0115
0116
0117     % saving the displacement of the output nodes
0118     if n >= output_index*time_steps/num_out_results && num_out_results>=output_index+1
0119         output_index=output_index+1;
0120         for i=1:size(output_nodes,2)
0121             displacement_peridynamic(i,output_index)=U(1,output_nodes(i));
0122         end
0123         time(1,output_index)=n*delta_t;
0124     end
0125
0126
0127     if n >= perc_print*time_steps/100
0128
0129         % calculating the percentage of progress of time integration
0130         perc_progress=n/time_steps*100;
0131
0132         % showing the time integration progress
0133         fprintf('%.0f%% completed \n',perc_progress);
0134         perc_print=perc_print+1;
0135
0136     end
0137
0138 end
```

A.2 C++ code

```
0001 cout <<  "Starting time integration algorithm \n" << endl;
0002 cout.precision(0);
0003
0004 // initializing auxiliary variables
0005 double perc_print=0., perc_progress, theta_term;
0006 int output_index=0;
0007
0008 // core calculation of peridynamic equation
0009 for (int n=0; n<time_steps; n++) {
0010
0011     // performing a sweep of all (k) nodes
0012     for (int k=0; k<j_nodes_id.size(); k++) {
0013
0014         // initializing to zero the theta(k) value (if required)
0015         if (fabs(a)>1.E-4) {
0016             theta[k]=0.;
0017         }
0018
0019         // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0020         for (int i=0; i<j_nodes_id[k].size(); i++) {
0021
0022             // saving the ID of the (j) node and the interaction id in two variables for clarity
0023             j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0024
0025             // initializing the norm of the csi+eta vector
0026             norm_csi_eta=0.;
0027
0028             // calculating the eta(k)(j) vector
0029             for (int e=0; e<num_dimensions; e++) {
```

```
0030
0031                eta_vector[e][int_id]=U[e][j]-U[e][k];
0032
0033                // incrementing the norm of the csi+eta vector
0034                norm_csi_eta+=pow(csi_vector[e][int_id]+eta_vector[e][int_id],2.);
0035
0036            }
0037
0038            // calculating the norm of the csi+eta vector
0039            norm_csi_eta=sqrt(norm_csi_eta);
0040
0041            // calculating the stretch(k)(j) value
0042            stretch[int_id]=(norm_csi_eta-csi_modulus[int_id])/csi_modulus[int_id];
0043
0044            // calculating the unit vector of the interaction direction
0045            for (int e=0; e<num_dimensions; e++) {
0046                int_direction_unit[e][int_id]=(csi_vector[e][int_id]+
0047                                                    eta_vector[e][int_id])/norm_csi_eta;
0048            }
0049
0050            // theta calculation (if required)
0051            if (fabs(a)>1.E-4) {
0052
0053                // initializing to zero the lambda value
0054                lambda[int_id]=0.;
0055
0056                // calculating the lambda(k)(j) value
0057                for (int e=0; e<num_dimensions; e++) {
0058                    lambda[int_id]+=int_direction_unit[e][int_id]*csi_vector[e][int_id]/
0059                                        csi_modulus[int_id];
0060                }
0061
0062                // calculating the theta(k) value
0063                theta[k]+=d*delta*G_d[int_id]*stretch[int_id]*lambda[int_id]*v_c[int_id]*V[j];
0064
0065            }
0066        }
0067    }
0068
0069    // performing a second sweep of all (k) nodes to calculate the force acting on the node
0070    for (int k=0; k<j_nodes_id.size(); k++) {
0071
0072        // initializing to zero the force(k) value
0073        for (int e=0; e<num_dimensions; e++) {
0074            force[e][k]=0.;
0075        }
0076
0077        // setting the zero the variable that will contain the theta term
0078        theta_term=0.;
0079
0080        // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0081        for (int i=0; i<j_nodes_id[k].size(); i++) {
0082
0083            // saving the ID of the (j) node and the interaction id in two variables for clarity
0084            j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0085
0086            // theta term calculation (if required)
0087            if (fabs(a)>1.E-4) {
0088                theta_term=2*a*d*delta*G_d[int_id]*lambda[int_id]/csi_modulus[int_id]*
0089                                (theta[k]+theta[j]);
0090            }
0091
0092            // incrementing the value of the force
0093            for (int e=0; e<num_dimensions; e++) {
0094                force[e][k]+=(theta_term+4*b*delta*G_b[int_id]*stretch[int_id])*
0095                                int_direction_unit[e][int_id]*v_c[int_id]*V[j];
0096            }
```

```
0097
0098        }
0099
0100    }
0101
0102
0103    for (int k=i_boundarynodes; k<j_nodes_id.size(); k++) {
0104        for (int e=0; e<num_dimensions; e++) {
0105
0106            // calculating the acceleration vector
0107            U_dotdot[e][k]=(force[e][k]+body_f[e][k])/ro[k];
0108
0109            // calculating the velocity vector at the next time step
0110            U_dot[e][k]=U_dotdot[e][k]*delta_t+U_dot[e][k];
0111
0112            // calculating the displacement vectors at the next time step
0113            U[e][k]=U_dot[e][k]*delta_t+U[e][k];
0114
0115        }
0116    }
0117
0118    // saving the displacement of the output nodes
0119    if (n>=output_index*time_steps/num_out_results && num_out_results>=output_index) {
0120
0121        for (int i=0; i<output_nodes.size(); i++) {
0122            displacement_peridynamic[i][output_index]=U[0][output_nodes[i]];
0123        }
0124        time[output_index]=n*delta_t;
0125        output_index++;
0126
0127    }
0128
0129    if (n>=perc_print*time_steps/100) {
0130
0131        // calculating the percentage of progress of time integration
0132        perc_progress=n/(double)time_steps*100;
0133
0134        // showing the time integration progress
0135        cout << fixed << perc_progress << "% completed \n " << endl;
0136        perc_print++;
0137
0138    }
0139
0140 }
```

Appendix B

Implicit time integration codes

Hereinafter will be provided the portion of the peridynamic codes that performs the implicit time integration using the Newmark-beta and the backward finite difference methods. Both the MATLAB and C++ version of the codes will be shown. Every step of the code is commented to provide a better understanding of the operations performed.

B.1 MATLAB Newmark-beta code

```
0001 fprintf('Starting time integration algorithm \n');
0002
0003 % initializing the variables for output saving
0004 perc_print=0;
0005 output_index=0;
0006
0007 % core calculation of peridynamic equation
0008 for n=1:time_steps
0009
0010     % saving the previous values of displacement
0011     U_m1=U;
0012
0013     % initializing the inverse matrix B and the system solution parameters
0014     B_1=eye(n_nodes)*0.25;
0015     max_displacement_difference=1;
0016     iteration=0;
0017
0018
0019     % calculating the initial value of F function
0020     % performing a sweep of all (k) nodes
0021     for k=1:n_nodes
0022
0023         % initializing to zero the theta(k) value (if required)
0024         if a ~= 0
0025             theta(1,k)=0;
0026         end
0027
0028         % performing a sweep of all (j) nodes in the neighbourhood of the
0029         % (k) node
0030         for i=1:max_interactions_per_node
0031
0032             % checking if the ID of the (j) node is valid
0033             if  j_nodes_id(i,k) ~= 0
0034
0035                 % saving the ID of the (j) node in the j variable
0036                 j=j_nodes_id(i,k);
0037
```

```
0038                % saving the ID of the interaction in the ind_id variable
0039                int_id=interaction_id(i,k);
0040
0041                % calculating the eta(k)(j) vector
0042                eta_vector(:,int_id)=U(:,j)-U(:,k);
0043
0044                % calculating the norm of the csi+eta vector
0045                norm_csi_eta=norm( csi_vector(:,int_id)+eta_vector(:,int_id) );
0046
0047                % calculating the stretch(k)(j) value
0048                stretch(1,int_id)=( norm_csi_eta - csi_modulus(1,int_id) )/csi_modulus(1,int_id);
0049
0050                % calculating the unit vector of the interaction direction
0051                int_direction_unit(:,int_id)=( csi_vector(:,int_id)+ ...
0052                    eta_vector(:,int_id) )/norm_csi_eta;
0053
0054                % theta calculation (if required)
0055                if a ~= 0
0056
0057                    % calculating the lambda(k)(j) value
0058                    lambda(1,int_id)=dot( int_direction_unit(:,int_id) ,...
0059                        csi_vector(:,int_id)/csi_modulus(1,int_id) );
0060
0061                    %calculating the theta(k) value
0062                    theta(1,k)=theta(1,k)+d*delta*G_d(1,int_id)*stretch(1,int_id)* ...
0063                        lambda(1,int_id)*v_c(1,int_id)*V(1,j);
0064
0065                end
0066
0067            end
0068        end
0069    end
0070
0071
0072    % performing a second sweep of all (k) nodes to calculate the force
0073    % acting on the node
0074    for k=1:n_nodes
0075
0076        % initializing to zero the force(k) value
0077        force(:,k)=0;
0078
0079        % theta term initialization
0080        theta_term=0;
0081
0082        % performing a sweep of all (j) nodes in the neighbourhood of the
0083        % (k) node
0084        for i=1:max_interactions_per_node
0085
0086            % checking if the ID of the (j) node is valid
0087            if  j_nodes_id(i,k) ~= 0
0088
0089                % saving the ID of the (j) node in the j variable
0090                j=j_nodes_id(i,k);
0091
0092                % saving the ID of the interaction in the ind_id variable
0093                int_id=interaction_id(i,k);
0094
0095                % theta term calculation (if required)
0096                if a ~= 0
0097                    theta_term=2*a*d*delta*G_d(1,int_id)*lambda(1,int_id)/ ...
0098                        csi_modulus(1,int_id)*( theta(1,k)+theta(1,j) );
0099                end
0100
0101                % incrementing the value of the force
0102                force(:,k)=force(:,k) + ( theta_term + 4*b*delta*G_b(1,int_id)* ...
0103                    stretch(1,int_id) )*int_direction_unit(:,int_id)*( v_c(1,int_id)*V(1,j) );
0104
```

```
0105            end
0106        end
0107
0108        % calculating the second displacement derivative vector
0109        U_dotdot(:,k)=( force(:,k)+body_f(:,k) )/ro(1,k);
0110
0111    end
0112
0113    U_dotdot_m1=U_dotdot;
0114
0115    % calculating the value of F
0116    F=U-U_m1-delta_t*U_dot_m1-(1-2*beta)/2*delta_t^2*U_dotdot-beta*delta_t^2*U_dotdot_m1;
0117
0118
0119    while max_displacement_difference>displacement_tolerance && iteration<200
0120
0121        % saving the previous values of F and displacement
0122        F_old=F;
0123        U_old=U;
0124
0125        % calculating the next value of the displacement
0126        U=U_old-(B_1*F_old')';
0127
0128        % setting to zero the displacement value of the boundary nodes
0129        for k=1:i_boundarynodes
0130            U(:,k)=0;
0131        end
0132
0133        % performing a sweep of all (k) nodes
0134        for k=1:n_nodes
0135
0136            % initializing to zero the theta(k) value (if required)
0137            if a ~= 0
0138                theta(1,k)=0;
0139            end
0140
0141            % performing a sweep of all (j) nodes in the neighbourhood of the
0142            % (k) node
0143            for i=1:max_interactions_per_node
0144
0145                % checking if the ID of the (j) node is valid
0146                if  j_nodes_id(i,k) ~= 0
0147
0148                    % saving the ID of the (j) node in the j variable
0149                    j=j_nodes_id(i,k);
0150
0151                    % saving the ID of the interaction in the ind_id variable
0152                    int_id=interaction_id(i,k);
0153
0154                    % calculating the eta(k)(j) vector
0155                    eta_vector(:,int_id)=U(:,j)-U(:,k);
0156
0157                    % calculating the norm of the csi+eta vector
0158                    norm_csi_eta=norm( csi_vector(:,int_id)+eta_vector(:,int_id) );
0159
0160                    % calculating the stretch(k)(j) value
0161                    stretch(1,int_id)=( norm_csi_eta - csi_modulus(1,int_id) )/csi_modulus(1,int_id);
0162
0163                    % calculating the unit vector of the interaction direction
0164                    int_direction_unit(:,int_id)=( csi_vector(:,int_id)+ ...
0165                        eta_vector(:,int_id) )/norm_csi_eta;
0166
0167                    % theta calculation (if required)
0168                    if a ~= 0
0169
0170                        % calculating the lambda(k)(j) value
0171                        lambda(1,int_id)=dot( int_direction_unit(:,int_id) ,...
```

```
0172                        csi_vector(:,int_id)/csi_modulus(1,int_id) );
0173
0174                    %calculating the theta(k) value
0175                    theta(1,k)=theta(1,k)+d*delta*G_d(1,int_id)*stretch(1,int_id)* ...
0176                        lambda(1,int_id)*v_c(1,int_id)*V(1,j);
0177
0178                end
0179
0180            end
0181        end
0182    end
0183
0184
0185    % performing a second sweep of all (k) nodes to calculate the force
0186    % acting on the node
0187    for k=1:n_nodes
0188
0189        % initializing to zero the force(k) value
0190        force(:,k)=0;
0191
0192        % theta term initialization
0193        theta_term=0;
0194
0195        % performing a sweep of all (j) nodes in the neighbourhood of the
0196        % (k) node
0197        for i=1:max_interactions_per_node
0198
0199            % checking if the ID of the (j) node is valid
0200            if  j_nodes_id(i,k) ~= 0
0201
0202                % saving the ID of the (j) node in the j variable
0203                j=j_nodes_id(i,k);
0204
0205                % saving the ID of the interaction in the ind_id variable
0206                int_id=interaction_id(i,k);
0207
0208                % theta term calculation (if required)
0209                if a ~= 0
0210                    theta_term=2*a*d*delta*G_d(1,int_id)*lambda(1,int_id)/ ...
0211                        csi_modulus(1,int_id)*( theta(1,k)+theta(1,j) );
0212                end
0213
0214                % incrementing the value of the force
0215                force(:,k)=force(:,k) + ( theta_term + 4*b*delta*G_b(1,int_id)* ...
0216                    stretch(1,int_id) )*int_direction_unit(:,int_id)*( v_c(1,int_id)*V(1,j) );
0217
0218            end
0219        end
0220
0221        % calculating the second displacement derivative vector
0222        U_dotdot(:,k)=( force(:,k)+body_f(:,k) )/ro(1,k);
0223
0224    end
0225
0226    % calculating the value of F
0227    F=U-U_m1-delta_t*U_dot_m1-(1-2*beta)/2*delta_t^2*U_dotdot-beta*delta_t^2*U_dotdot_m1;
0228
0229    % calculating the maximum difference between the displacement at
0230    % the current iteration step and at the previous one
0231    max_displacement_difference=max(abs(U-U_old));
0232
0233    % calculating the value of B_1 for the next iteration
0234    B_1=B_1+((U-U_old)'-B_1*(F-F_old)')/((U-U_old)*B_1*(F-F_old)')*((U-U_old)*B_1);
0235
0236    % incrementing the iteration counter
0237    iteration=iteration+1;
0238
```

```
0239        % checking if the maximum number of iteration has been reached
0240        if iteration==200
0241            fprintf('Convergency not reached with maximum number of iteration, shutting down.');
0242            return;
0243        end
0244
0245    end
0246
0247    % calculating the next value of the displacement
0248    U=U_old-(B_1*F')';
0249
0250    % setting to zero the displacement value of the boundary nodes
0251    for k=1:i_boundarynodes
0252        U(:,k)=0;
0253    end
0254
0255    % calculating the displacement's first derivative
0256    U_dot_m1=U_dot_m1+delta_t/2*(U_dotdot+U_dotdot_m1);
0257
0258
0259
0260    % saving the displacement of the output nodes
0261    if n >= output_index*time_steps/num_out_results && num_out_results>=output_index+1
0262        output_index=output_index+1;
0263        for i=1:size(output_nodes,2)
0264            displacement_peridynamic(i,output_index)=U(1,output_nodes(i));
0265        end
0266        time(1,output_index)=n*delta_t;
0267    end
0268
0269
0270    if n >= perc_print*time_steps/100
0271
0272        % calculating the percentage of progress of time integration
0273        perc_progress=n/time_steps*100;
0274
0275        % showing the time integration progress
0276        fprintf('%.0f%% completed \n',perc_progress);
0277        perc_print=perc_print+1;
0278
0279    end
0280
0281 end
```

B.2 MATLAB backward finite difference code

```
0001 fprintf('Starting time integration algorithm \n');
0002
0003 % initializing the variables for output saving
0004 perc_print=0;
0005 output_index=0;
0006
0007 % core calculation of peridynamic equation
0008 for n=1:time_steps
0009
0010    % initializing the inverse matrix B and the system solution parameters
0011    B_1=eye(n_nodes)*0.025;
0012    max_displacement_difference=1;
0013    iteration=0;
0014
0015
0016    % calculating the initial value of F function
0017    % performing a sweep of all (k) nodes
0018    for k=1:n_nodes
0019
```

```
0020        % initializing to zero the theta(k) (if required)
0021        if a ~= 0
0022            theta(1,k)=0;
0023        end
0024
0025        % performing a sweep of all (j) nodes in the neighbourhood of the
0026        % (k) node
0027        for i=1:max_interactions_per_node
0028
0029            % checking if the ID of the (j) node is valid
0030            if  j_nodes_id(i,k) ~= 0
0031
0032                % saving the ID of the (j) node in the j variable
0033                j=j_nodes_id(i,k);
0034
0035                % saving the ID of the interaction in the ind_id variable
0036                int_id=interaction_id(i,k);
0037
0038                % calculating the eta(k)(j) vector
0039                eta_vector(:,int_id)=U(:,j)-U(:,k);
0040
0041                % calculating the norm of the csi+eta vector
0042                norm_csi_eta=norm( csi_vector(:,int_id)+eta_vector(:,int_id) );
0043
0044                % calculating the stretch(k)(j) value
0045                stretch(1,int_id)=( norm_csi_eta - csi_modulus(1,int_id) )/csi_modulus(1,int_id);
0046
0047                % calculating the unit vector of the interaction direction
0048                int_direction_unit(:,int_id)=( csi_vector(:,int_id)+ ...
0049                    eta_vector(:,int_id) )/norm_csi_eta;
0050
0051                % theta calculation (if required)
0052                if a ~= 0
0053
0054                    % calculating the lambda(k)(j) value
0055                    lambda(1,int_id)=dot( int_direction_unit(:,int_id) ,...
0056                        csi_vector(:,int_id)/csi_modulus(1,int_id) );
0057
0058                    %calculating the theta(k) value
0059                    theta(1,k)=theta(1,k)+d*delta*G_d(1,int_id)*stretch(1,int_id)* ...
0060                        lambda(1,int_id)*v_c(1,int_id)*V(1,j);
0061
0062                end
0063
0064            end
0065        end
0066    end
0067
0068
0069    % performing a second sweep of all (k) nodes to calculate the force
0070    % acting on the node
0071    for k=1:n_nodes
0072
0073        % initializing to zero the force(k) value
0074        force(:,k)=0;
0075
0076        % theta term initialization
0077        theta_term=0;
0078
0079        % performing a sweep of all (j) nodes in the neighbourhood of the
0080        % (k) node
0081        for i=1:max_interactions_per_node
0082
0083            % checking if the ID of the (j) node is valid
0084            if  j_nodes_id(i,k) ~= 0
0085
0086                % saving the ID of the (j) node in the j variable
```

```
0087                j=j_nodes_id(i,k);
0088
0089                % saving the ID of the interaction in the ind_id variable
0090                int_id=interaction_id(i,k);
0091
0092                % theta term calculation (if required)
0093                if a ~= 0
0094                    theta_term=2*a*d*delta*G_d(1,int_id)*lambda(1,int_id)/ ...
0095                        csi_modulus(1,int_id)*( theta(1,k)+theta(1,j) );
0096                end
0097
0098                % incrementing the value of the force
0099                force(:,k)=force(:,k) + ( theta_term + 4*b*delta*G_b(1,int_id)* ...
0100                    stretch(1,int_id) )*int_direction_unit(:,int_id)*( v_c(1,int_id)*V(1,j) );
0101
0102            end
0103        end
0104
0105        % calculating the second displacement derivative vector
0106        U_dotdot(:,k)=( force(:,k)+body_f(:,k) )/ro(1,k);
0107
0108    end
0109
0110    % calculating the value of F
0111    F=2*U-5*U_m1+4*U_m2-U_m3-delta_t^2*U_dotdot;
0112
0113
0114    while max_displacement_difference>displacement_tolerance && iteration<200
0115
0116        % saving the previous values of F and displacement
0117        F_old=F;
0118        U_old=U;
0119
0120        % calculating the next value of the displacement
0121        U=U_old-(B_1*F_old')';
0122
0123        % setting to zero the displacement value of the boundary nodes
0124        for k=1:i_boundarynodes
0125            U(:,k)=0;
0126        end
0127
0128        % performing a sweep of all (k) nodes
0129        for k=1:n_nodes
0130
0131            % initializing to zero the theta(k) value (if required)
0132            if a ~= 0
0133                theta(1,k)=0;
0134            end
0135
0136            % performing a sweep of all (j) nodes in the neighbourhood of the
0137            % (k) node
0138            for i=1:max_interactions_per_node
0139
0140                % checking if the ID of the (j) node is valid
0141                if  j_nodes_id(i,k) ~= 0
0142
0143                    % saving the ID of the (j) node in the j variable
0144                    j=j_nodes_id(i,k);
0145
0146                    % saving the ID of the interaction in the ind_id variable
0147                    int_id=interaction_id(i,k);
0148
0149                    % calculating the eta(k)(j) vector
0150                    eta_vector(:,int_id)=U(:,j)-U(:,k);
0151
0152                    % calculating the norm of the csi+eta vector
0153                    norm_csi_eta=norm( csi_vector(:,int_id)+eta_vector(:,int_id) );
```

```
0154
0155                    % calculating the stretch(k)(j) value
0156                    stretch(1,int_id)=( norm_csi_eta - csi_modulus(1,int_id) )/csi_modulus(1,int_id);
0157
0158                    % calculating the unit vector of the interaction direction
0159                    int_direction_unit(:,int_id)=( csi_vector(:,int_id)+ ...
0160                        eta_vector(:,int_id) )/norm_csi_eta;
0161
0162                    % theta term calculation (if required)
0163                    if a ~= 0
0164
0165                        % calculating the lambda(k)(j) value
0166                        lambda(1,int_id)=dot( int_direction_unit(:,int_id) ,...
0167                            csi_vector(:,int_id)/csi_modulus(1,int_id) );
0168
0169                        %calculating the theta(k) value
0170                        theta(1,k)=theta(1,k)+d*delta*G_d(1,int_id)*stretch(1,int_id)* ...
0171                            lambda(1,int_id)*v_c(1,int_id)*V(1,j);
0172
0173                    end
0174
0175                end
0176            end
0177        end
0178
0179
0180        % performing a second sweep of all (k) nodes to calculate the force
0181        % acting on the node
0182        for k=1:n_nodes
0183
0184            % initializing to zero the force(k) value
0185            force(:,k)=0;
0186
0187            % theta term initialization
0188            theta_term=0;
0189
0190            % performing a sweep of all (j) nodes in the neighbourhood of the
0191            % (k) node
0192            for i=1:max_interactions_per_node
0193
0194                % checking if the ID of the (j) node is valid
0195                if  j_nodes_id(i,k) ~= 0
0196
0197                    % saving the ID of the (j) node in the j variable
0198                    j=j_nodes_id(i,k);
0199
0200                    % saving the ID of the interaction in the ind_id variable
0201                    int_id=interaction_id(i,k);
0202
0203                    % theta term calculation (if required)
0204                    if a ~= 0
0205                        theta_term=2*a*d*delta*G_d(1,int_id)*lambda(1,int_id)/ ...
0206                            csi_modulus(1,int_id)*( theta(1,k)+theta(1,j) );
0207                    end
0208
0209                    % incrementing the value of the force
0210                    force(:,k)=force(:,k) + ( theta_term + 4*b*delta*G_b(1,int_id)* ...
0211                        stretch(1,int_id) )*int_direction_unit(:,int_id)*( v_c(1,int_id)*V(1,j) );
0212
0213                end
0214            end
0215
0216            % calculating the second displacement derivative vector
0217            U_dotdot(:,k)=( force(:,k)+body_f(:,k) )/ro(1,k);
0218
0219        end
0220
```

```
0221         % calculating the value of F
0222         F=2*U-5*U_m1+4*U_m2-U_m3-delta_t^2*U_dotdot;
0223
0224         % calculating the maximum difference between the displacement at
0225         % the current iteration step and at the previous one
0226         max_displacement_difference=max(abs(U-U_old));
0227
0228         % calculating the value of B_1 for the next iteration
0229         B_1=B_1+((U-U_old)'-B_1*(F-F_old)')/((U-U_old)*B_1*(F-F_old)')*((U-U_old)*B_1);
0230
0231         % incrementing the iteration counter
0232         iteration=iteration+1;
0233
0234         % checking if the maximum number of iteration has been reached
0235         if iteration==200
0236             fprintf('Convergency not reached with maximum number of iteration, shutting down.');
0237             return;
0238         end
0239
0240     end
0241
0242     % calculating the next value of the displacement
0243     U=U_old-(B_1*F')';
0244
0245     % setting to zero the displacement value of the boundary nodes
0246     for k=1:i_boundarynodes
0247         U(:,k)=0;
0248     end
0249
0250     % saving the values of displacement at previous steps
0251     U_m3=U_m2;
0252     U_m2=U_m1;
0253     U_m1=U;
0254
0255
0256
0257     % saving the displacement of the output nodes
0258     if n >= output_index*time_steps/num_out_results && num_out_results>=output_index+1
0259         output_index=output_index+1;
0260         for i=1:size(output_nodes,2)
0261             displacement_peridynamic(i,output_index)=U(1,output_nodes(i));
0262         end
0263         time(1,output_index)=n*delta_t;
0264     end
0265
0266
0267     if n >= perc_print*time_steps/100
0268
0269         % calculating the percentage of progress of time integration
0270         perc_progress=n/time_steps*100;
0271
0272         % showing the time integration progress
0273         fprintf('%.0f%% completed \n',perc_progress);
0274         perc_print=perc_print+1;
0275
0276     end
0277
0278 end
```

B.3 C++ Newmark-beta code

```
0001 cout << "Starting time integration algorithm \n" << endl;
0002 cout.precision(0);
0003
0004 // initializing auxiliary variables
```

```
0005 double perc_print=0., perc_progress, theta_term, max_displacement_difference;
0006 int output_index=0, iteration=0, tot_iterations=0, max_iterations=200;
0007
0008 // core calculation of peridynamic equation
0009 for (int n=1; n<(time_steps+1); n++) {
0010
0011     // initializing the inverse matrix B and the system solution parameters
0012     for (int e=0; e<num_dimensions; e++) {
0013         for (int i=0; i<n_nodes; i++) {
0014             for (int j=0; j<n_nodes; j++) {
0015                 if (i==j) {
0016                     B_1[e][i][j]=alpha;
0017                 } else {
0018                     B_1[e][i][j]=0.;
0019                 }
0020             }
0021         }
0022     }
0023
0024     // initializing the max displacement difference variable and the number of iterations variable
0025     max_displacement_difference=2.*displacement_tolerance;
0026     iteration=0;
0027
0028
0029     // calculating the initial value of F function
0030
0031     // performing a sweep of all (k) nodes
0032     for (int k=0; k<j_nodes_id.size(); k++) {
0033
0034         // initializing to zero the theta(k) value (if required)
0035         if (fabs(a)>1.E-4) {
0036             theta[k]=0.;
0037         }
0038
0039         // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0040         for (int i=0; i<j_nodes_id[k].size(); i++) {
0041
0042             // saving the ID of the (j) node and the interaction id in two variables for clarity
0043             j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0044
0045             // initializing the norm of the csi+eta vector
0046             norm_csi_eta=0.;
0047
0048             // calculating the eta(k)(j) vector
0049             for (int e=0; e<num_dimensions; e++) {
0050
0051                 eta_vector[e][int_id]=U[e][j]-U[e][k];
0052
0053                 // incrementing the norm of the csi+eta vector
0054                 norm_csi_eta+=pow(csi_vector[e][int_id]+eta_vector[e][int_id],2.);
0055
0056             }
0057
0058             // calculating the norm of the csi+eta vector
0059             norm_csi_eta=sqrt(norm_csi_eta);
0060
0061             // calculating the stretch(k)(j) value
0062             stretch[int_id]=(norm_csi_eta-csi_modulus[int_id])/csi_modulus[int_id];
0063
0064             // calculating the unit vector of the interaction direction
0065             for (int e=0; e<num_dimensions; e++) {
0066                 int_direction_unit[e][int_id]=(csi_vector[e][int_id]+
0067                                                eta_vector[e][int_id])/norm_csi_eta;
0068             }
0069
0070             // theta calculation (if required)
0071             if (fabs(a)>1.E-4) {
```

```
0072
0073                // initializing to zero the lambda value
0074                lambda[int_id]=0.;
0075
0076                // calculating the lambda(k)(j) value
0077                for (int e=0; e<num_dimensions; e++) {
0078                    lambda[int_id]+=int_direction_unit[e][int_id]*
0079                                        csi_vector[e][int_id]/csi_modulus[int_id];
0080                }
0081
0082                // calculating the theta(k) value
0083                theta[k]+=d*delta*G_d[int_id]*stretch[int_id]*lambda[int_id]*v_c[int_id]*V[j];
0084
0085            }
0086        }
0087    }
0088
0089    // performing a second sweep of all (k) nodes to calculate the force acting on the node
0090    for (int k=0; k<j_nodes_id.size(); k++) {
0091
0092        // initializing to zero the force(k) value
0093        for (int e=0; e<num_dimensions; e++) {
0094            force[e][k]=0.;
0095        }
0096
0097        // theta term initialization
0098        theta_term=0.;
0099
0100        // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0101        for (int i=0; i<j_nodes_id[k].size(); i++) {
0102
0103            // saving the ID of the (j) node and the interaction id in two variables for clarity
0104            j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0105
0106            // theta term calculation (if required)
0107            if (fabs(a)>1.E-4) {
0108                theta_term=2*a*d*delta*G_d[int_id]*lambda[int_id]/
0109                                csi_modulus[int_id]*(theta[k]+theta[j]);
0110            }
0111
0112            // incrementing the value of the force
0113            for (int e=0; e<num_dimensions; e++) {
0114                force[e][k]+=(theta_term+4*b*delta*G_b[int_id]*stretch[int_id])*
0115                                int_direction_unit[e][int_id]*v_c[int_id]*V[j];
0116            }
0117
0118        }
0119
0120    }
0121
0122    for (int e=0; e<num_dimensions; e++) {
0123        for (int k=i_boundarynodes; k<j_nodes_id.size(); k++) {
0124            // calculating the displacement second derivative
0125            U_dotdot[e][k]=(force[e][k]+body_f[e][k])/ro[k];
0126        }
0127    }
0128
0129    // saving the previous values of displacement and its second derivative
0130    U_m1=U;
0131    U_dotdot_m1=U_dotdot;
0132
0133    // calculating the value of F
0134    for (int e=0; e<num_dimensions; e++) {
0135        for (int k=0; k<n_nodes; k++) {
0136            F[e][k]=U[e][k]-U_m1[e][k]-delta_t*U_dot_m1[e][k]-(1.-2.*beta)/2.*pow(delta_t,2.)*
0137                        U_dotdot[e][k]-beta*pow(delta_t,2.)*U_dotdot_m1[e][k];
0138        }
```

```
0139    }
0140
0141    // starting the Broyden iterative process
0142    while (max_displacement_difference>displacement_tolerance && iteration<max_iterations) {
0143
0144        // saving the previous values of F and displacement
0145        F_old=F;
0146        U_old=U;
0147
0148        // calculating the next value of the displacement
0149        for (int e=0; e<num_dimensions; e++) {
0150            for (int k=0; k<n_nodes; k++) {
0151                U[e][k]=U_old[e][k];
0152                for (int j=0; j<n_nodes; j++) {
0153                    U[e][k]+=-B_1[e][k][j]*F[e][j];
0154                }
0155            }
0156
0157            // setting to zero the displacement value of the boundary nodes
0158            for (int k=0; k<i_boundarynodes; k++) {
0159                U[e][k]=0.;
0160            }
0161        }
0162
0163        // performing a sweep of all (k) nodes
0164        for (int k=0; k<j_nodes_id.size(); k++) {
0165
0166            // initializing to zero the theta(k) value (if required)
0167            if (fabs(a)>1.E-4) {
0168                theta[k]=0.;
0169            }
0170
0171            // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0172            for (int i=0; i<j_nodes_id[k].size(); i++) {
0173
0174                // saving the ID of the (j) node and the interaction id in two variables for clarity
0175                j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0176
0177                // initializing the norm of the csi+eta vector
0178                norm_csi_eta=0.;
0179
0180                // calculating the eta(k)(j) vector
0181                for (int e=0; e<num_dimensions; e++) {
0182
0183                    eta_vector[e][int_id]=U[e][j]-U[e][k];
0184
0185                    // incrementing the norm of the csi+eta vector
0186                    norm_csi_eta+=pow(csi_vector[e][int_id]+eta_vector[e][int_id],2.);
0187
0188                }
0189
0190                // calculating the norm of the csi+eta vector
0191                norm_csi_eta=sqrt(norm_csi_eta);
0192
0193                // calculating the stretch(k)(j) value
0194                stretch[int_id]=(norm_csi_eta-csi_modulus[int_id])/csi_modulus[int_id];
0195
0196                // calculating the unit vector of the interaction direction
0197                for (int e=0; e<num_dimensions; e++) {
0198                    int_direction_unit[e][int_id]=(csi_vector[e][int_id]+
0199                                                    eta_vector[e][int_id])/norm_csi_eta;
0200                }
0201
0202                // theta calculation (if required)
0203                if (fabs(a)>1.E-4) {
0204
0205                    // initializing to zero the lambda value
```

```
0206                    lambda[int_id]=0.;
0207
0208                    // calculating the lambda(k)(j) value
0209                    for (int e=0; e<num_dimensions; e++) {
0210                        lambda[int_id]+=int_direction_unit[e][int_id]*
0211                                            csi_vector[e][int_id]/csi_modulus[int_id];
0212                    }
0213
0214                    // calculating the theta(k) value
0215                    theta[k]+=d*delta*G_d[int_id]*stretch[int_id]*lambda[int_id]*v_c[int_id]*V[j];
0216
0217                }
0218            }
0219        }
0220
0221        // performing a second sweep of all (k) nodes to calculate the force acting on the node
0222        for (int k=0; k<j_nodes_id.size(); k++) {
0223
0224            // initializing to zero the force(k) value
0225            for (int e=0; e<num_dimensions; e++) {
0226                force[e][k]=0.;
0227            }
0228
0229            // theta term initialization
0230            theta_term=0.;
0231
0232            // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0233            for (int i=0; i<j_nodes_id[k].size(); i++) {
0234
0235                // saving the ID of the (j) node and the interaction id in two variables for clarity
0236                j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0237
0238                // theta term calculation (if required)
0239                if (fabs(a)>1.E-4) {
0240                    theta_term=2*a*d*delta*G_d[int_id]*lambda[int_id]/
0241                                    csi_modulus[int_id]*(theta[k]+theta[j]);
0242                }
0243
0244                // incrementing the value of the force
0245                for (int e=0; e<num_dimensions; e++) {
0246                    force[e][k]+=(theta_term+4*b*delta*G_b[int_id]*stretch[int_id])*
0247                                    int_direction_unit[e][int_id]*v_c[int_id]*V[j];
0248                }
0249
0250            }
0251
0252        }
0253
0254
0255        for (int k=i_boundarynodes; k<j_nodes_id.size(); k++) {
0256            for (int e=0; e<num_dimensions; e++) {
0257                // calculating the displacement second derivative
0258                U_dotdot[e][k]=(force[e][k]+body_f[e][k])/ro[k];
0259            }
0260        }
0261
0262        // calculating the value of F
0263        for (int e=0; e<num_dimensions; e++) {
0264            for (int k=0; k<n_nodes; k++) {
0265                F[e][k]=U[e][k]-U_m1[e][k]-delta_t*U_dot_m1[e][k]-(1.-2.*beta)/2.*pow(delta_t,2.)*
0266                            U_dotdot[e][k]-beta*pow(delta_t,2.)*U_dotdot_m1[e][k];
0267            }
0268        }
0269
0270        // calculating the U-U_old difference
0271        for (int e=0; e<num_dimensions; e++) {
0272            for (int k=0; k<n_nodes; k++) {
```

```
0273                U_Uold_diff[e][k]=fabs(U[e][k]-U_old[e][k]);
0274            }
0275        }
0276
0277        // calculating the maximum difference between the displacement
0278        // at the current iteration step and at the previous one
0279        max_displacement_difference=0.;
0280        int k=0;
0281        int e=0;
0282        while (max_displacement_difference<displacement_tolerance && e<num_dimensions && k<n_nodes) {
0283            max_displacement_difference=U_Uold_diff[e][k];
0284            k++;
0285            if (k==n_nodes) {
0286                e++;
0287                k=0;
0288            }
0289
0290        }
0291
0292        // calculating the value of B_1 for the next iteration
0293        for (int e=0; e<num_dimensions; e++) {
0294
0295            temp3=0.;
0296            for (int i=0; i<n_nodes; i++) {
0297                temp1[e][i]=0.;
0298                for (int j=0; j<n_nodes; j++) {
0299                    temp1[e][i]+=B_1[e][i][j]*(F[e][j]-F_old[e][j]);
0300                }
0301
0302                if (U_Uold_diff[e][k]>=displacement_tolerance) {
0303                    for (int j=0; j<n_nodes; j++) {
0304                        temp2[e][i][j]=(U[e][i]-U_old[e][i]-temp1[e][i])*(U[e][j]-U_old[e][j]);
0305                    }
0306                }
0307
0308                temp3+=(U[e][i]-U_old[e][i])*temp1[e][i];
0309            }
0310
0311            for (int i=0; i<n_nodes; i++) {
0312                for (int j=0; j<n_nodes; j++) {
0313                    temp4[e][i][j]=0.;
0314                    if (U_Uold_diff[e][k]>=displacement_tolerance) {
0315                        for (int r=0; r<n_nodes; r++) {
0316                            temp4[e][i][j]+=temp2[e][i][r]*B_1[e][r][j]/temp3;
0317                        }
0318                    }
0319                }
0320            }
0321
0322            for (int i=0; i<n_nodes; i++) {
0323                for (int j=0; j<n_nodes; j++) {
0324                    B_1[e][i][j]+=temp4[e][i][j];
0325                }
0326            }
0327        }
0328
0329        // incrementing the iteration counter and the total iteration counter
0330        iteration++;
0331        tot_iterations++;
0332
0333        // checking if the maximum number of iteration has been reached
0334        if (iteration==max_iterations) {
0335            cout << "Convergency not reached with maximum iterations, shutting down. \n" << endl;
0336            return(0);
0337        }
0338
0339    }
```

```
0340
0341     // calculating the next value of the displacement and its derivative
0342     for (int e=0; e<num_dimensions; e++) {
0343         for (int k=0; k<n_nodes; k++) {
0344             U[e][k]=U_old[e][k];
0345             for (int j=0; j<n_nodes; j++) {
0346                 U[e][k]+=-B_1[e][k][j]*F[e][j];
0347             }
0348         }
0349
0350         // setting to zero the displacement value of the boundary nodes
0351         for (int k=0; k<i_boundarynodes; k++) {
0352             U[e][k]=0.;
0353         }
0354
0355         // calculating the displacement's first derivative
0356         for (int k=i_boundarynodes; k<n_nodes; k++) {
0357             U_dot_m1[e][k]+=delta_t/2.*(U_dotdot[e][k]+U_dotdot_m1[e][k]);
0358         }
0359
0360     }
0361
0362     // saving the displacement of the output nodes
0363     if (n>=output_index*time_steps/num_out_results && num_out_results>=output_index) {
0364
0365         for (int i=0; i<output_nodes.size(); i++) {
0366             displacement_peridynamic[i][output_index]=U[0][output_nodes[i]];
0367         }
0368         time[output_index]=n*delta_t;
0369         output_index++;
0370
0371     }
0372
0373     if (n>=perc_print*time_steps/100) {
0374
0375         // calculating the percentage of progress of time integration
0376         perc_progress=n/(double)time_steps*100;
0377
0378         // showing the time integration progress
0379         cout << fixed << perc_progress << "% completed \n " << endl;
0380         perc_print++;
0381
0382     }
0383
0384 }
```

B.4 C++ backward finite difference code

```
0001 cout << "Starting time integration algorithm \n" << endl;
0002 cout.precision(0);
0003
0004 // initializing auxiliary variables
0005 double perc_print=0., perc_progress, theta_term, max_displacement_difference;
0006 int output_index=0, iteration=0, tot_iterations=0, max_iterations=200;
0007
0008 // core calculation of peridynamic equation
0009 for (int n=1; n<(time_steps+1); n++) {
0010
0011     // initializing the inverse matrix B and the system solution parameters
0012     for (int e=0; e<num_dimensions; e++) {
0013         for (int i=0; i<n_nodes; i++) {
0014             for (int j=0; j<n_nodes; j++) {
0015                 if (i==j) {
0016                     B_1[e][i][j]=alpha;
0017                 } else {
```

```
0018                     B_1[e][i][j]=0.;
0019                 }
0020             }
0021         }
0022     }
0023
0024     // initializing the max displacement difference variable and the number of iterations variable
0025     max_displacement_difference=2.*displacement_tolerance;
0026     iteration=0;
0027
0028
0029     // calculating the initial value of F function
0030
0031     // performing a sweep of all (k) nodes
0032     for (int k=0; k<j_nodes_id.size(); k++) {
0033
0034         // initializing to zero the theta(k) value (if required)
0035         if (fabs(a)>1.E-4) {
0036             theta[k]=0.;
0037         }
0038
0039         // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0040         for (int i=0; i<j_nodes_id[k].size(); i++) {
0041
0042             // saving the ID of the (j) node and the interaction id in two variables for clarity
0043             j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0044
0045             // initializing the norm of the csi+eta vector
0046             norm_csi_eta=0.;
0047
0048             // calculating the eta(k)(j) vector
0049             for (int e=0; e<num_dimensions; e++) {
0050
0051                 eta_vector[e][int_id]=U[e][j]-U[e][k];
0052
0053                 // incrementing the norm of the csi+eta vector
0054                 norm_csi_eta+=pow(csi_vector[e][int_id]+eta_vector[e][int_id],2.);
0055
0056             }
0057
0058             // calculating the norm of the csi+eta vector
0059             norm_csi_eta=sqrt(norm_csi_eta);
0060
0061             // calculating the stretch(k)(j) value
0062             stretch[int_id]=(norm_csi_eta-csi_modulus[int_id])/csi_modulus[int_id];
0063
0064             // calculating the unit vector of the interaction direction
0065             for (int e=0; e<num_dimensions; e++) {
0066                 int_direction_unit[e][int_id]=(csi_vector[e][int_id]+
0067                                                     eta_vector[e][int_id])/norm_csi_eta;
0068             }
0069
0070             // theta calculation (if required)
0071             if (fabs(a)>1.E-4) {
0072
0073                 // initializing to zero the lambda value
0074                 lambda[int_id]=0.;
0075
0076                 // calculating the lambda(k)(j) value
0077                 for (int e=0; e<num_dimensions; e++) {
0078                     lambda[int_id]+=int_direction_unit[e][int_id]*
0079                                         csi_vector[e][int_id]/csi_modulus[int_id];
0080                 }
0081
0082                 // calculating the theta(k) value
0083                 theta[k]+=d*delta*G_d[int_id]*stretch[int_id]*lambda[int_id]*v_c[int_id]*V[j];
0084
```

```
0085            }
0086        }
0087    }
0088
0089    // performing a second sweep of all (k) nodes to calculate the force acting on the node
0090    for (int k=0; k<j_nodes_id.size(); k++) {
0091
0092        // initializing to zero the force(k) value
0093        for (int e=0; e<num_dimensions; e++) {
0094            force[e][k]=0.;
0095        }
0096
0097        // theta term initialization
0098        theta_term=0.;
0099
0100        // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0101        for (int i=0; i<j_nodes_id[k].size(); i++) {
0102
0103            // saving the ID of the (j) node and the interaction id in two variables for clarity
0104            j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0105
0106            // theta term calculation (if required)
0107            if (fabs(a)>1.E-4) {
0108                theta_term=2*a*d*delta*G_d[int_id]*lambda[int_id]/
0109                                csi_modulus[int_id]*(theta[k]+theta[j]);
0110            }
0111
0112            // incrementing the value of the force
0113            for (int e=0; e<num_dimensions; e++) {
0114                force[e][k]+=(theta_term+4*b*delta*G_b[int_id]*stretch[int_id])*
0115                                int_direction_unit[e][int_id]*v_c[int_id]*V[j];
0116            }
0117
0118        }
0119
0120    }
0121
0122    for (int e=0; e<num_dimensions; e++) {
0123        for (int k=i_boundarynodes; k<j_nodes_id.size(); k++) {
0124            // calculating the displacement second derivative
0125            U_dotdot[e][k]=(force[e][k]+body_f[e][k])/ro[k];
0126        }
0127    }
0128
0129    // saving the previous values of displacement and its second derivative
0130    U_m3=U_m2;
0131    U_m2=U_m1;
0132    U_m1=U;
0133
0134    // calculating the value of F
0135    for (int e=0; e<num_dimensions; e++) {
0136        for (int k=0; k<n_nodes; k++) {
0137            F[e][k]=2.*U[e][k]-5.*U_m1[e][k]+4.*U_m2[e][k]-U_m3[e][k]-pow(delta_t,2.)*U_dotdot[e][k];
0138        }
0139    }
0140
0141    // starting the Broyden iterative process
0142    while (max_displacement_difference>displacement_tolerance && iteration<max_iterations) {
0143
0144        // saving the previous values of F and displacement
0145        F_old=F;
0146        U_old=U;
0147
0148        // calculating the next value of the displacement
0149        for (int e=0; e<num_dimensions; e++) {
0150            for (int k=0; k<n_nodes; k++) {
0151                U[e][k]=U_old[e][k];
```

```
0152                for (int j=0; j<n_nodes; j++) {
0153                    U[e][k]+=-B_1[e][k][j]*F[e][j];
0154                }
0155            }
0156
0157            // setting to zero the displacement value of the boundary nodes
0158            for (int k=0; k<i_boundarynodes; k++) {
0159                U[e][k]=0.;
0160            }
0161        }
0162
0163        // performing a sweep of all (k) nodes
0164        for (int k=0; k<j_nodes_id.size(); k++) {
0165
0166            // initializing to zero the theta(k) value (if required)
0167            if (fabs(a)>1.E-4) {
0168                theta[k]=0.;
0169            }
0170
0171            // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0172            for (int i=0; i<j_nodes_id[k].size(); i++) {
0173
0174                // saving the ID of the (j) node and the interaction id in two variables for clarity
0175                j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0176
0177                // initializing the norm of the csi+eta vector
0178                norm_csi_eta=0.;
0179
0180                // calculating the eta(k)(j) vector
0181                for (int e=0; e<num_dimensions; e++) {
0182
0183                    eta_vector[e][int_id]=U[e][j]-U[e][k];
0184
0185                    // incrementing the norm of the csi+eta vector
0186                    norm_csi_eta+=pow(csi_vector[e][int_id]+eta_vector[e][int_id],2.);
0187
0188                }
0189
0190                // calculating the norm of the csi+eta vector
0191                norm_csi_eta=sqrt(norm_csi_eta);
0192
0193                // calculating the stretch(k)(j) value
0194                stretch[int_id]=(norm_csi_eta-csi_modulus[int_id])/csi_modulus[int_id];
0195
0196                // calculating the unit vector of the interaction direction
0197                for (int e=0; e<num_dimensions; e++) {
0198                    int_direction_unit[e][int_id]=(csi_vector[e][int_id]+
0199                                                    eta_vector[e][int_id])/norm_csi_eta;
0200                }
0201
0202                // theta calculation (if required)
0203                if (fabs(a)>1.E-4) {
0204
0205                    // initializing to zero the lambda value
0206                    lambda[int_id]=0.;
0207
0208                    // calculating the lambda(k)(j) value
0209                    for (int e=0; e<num_dimensions; e++) {
0210                        lambda[int_id]+=int_direction_unit[e][int_id]*
0211                                        csi_vector[e][int_id]/csi_modulus[int_id];
0212                    }
0213
0214                    // calculating the theta(k) value
0215                    theta[k]+=d*delta*G_d[int_id]*stretch[int_id]*lambda[int_id]*v_c[int_id]*V[j];
0216
0217                }
0218            }
```

```
0219        }
0220
0221        // performing a second sweep of all (k) nodes to calculate the force acting on the node
0222        for (int k=0; k<j_nodes_id.size(); k++) {
0223
0224            // initializing to zero the force(k) value
0225            for (int e=0; e<num_dimensions; e++) {
0226                force[e][k]=0.;
0227            }
0228
0229            // theta term initialization
0230            theta_term=0.;
0231
0232            // performing a sweep of all (j) nodes in the neighbourhood of the (k) node
0233            for (int i=0; i<j_nodes_id[k].size(); i++) {
0234
0235                // saving the ID of the (j) node and the interaction id in two variables for clarity
0236                j=j_nodes_id[k][i]; int_id=interaction_id[k][i];
0237
0238                // theta term calculation (if required)
0239                if (fabs(a)>1.E-4) {
0240                    theta_term=2*a*d*delta*G_d[int_id]*lambda[int_id]/
0241                                    csi_modulus[int_id]*(theta[k]+theta[j]);
0242                }
0243
0244                // incrementing the value of the force
0245                for (int e=0; e<num_dimensions; e++) {
0246                    force[e][k]+=(theta_term+4*b*delta*G_b[int_id]*stretch[int_id])*
0247                                    int_direction_unit[e][int_id]*v_c[int_id]*V[j];
0248                }
0249
0250            }
0251
0252        }
0253
0254
0255        for (int k=i_boundarynodes; k<j_nodes_id.size(); k++) {
0256            for (int e=0; e<num_dimensions; e++) {
0257                // calculating the displacement second derivative
0258                U_dotdot[e][k]=(force[e][k]+body_f[e][k])/ro[k];
0259            }
0260        }
0261
0262        // calculating the value of F
0263        for (int e=0; e<num_dimensions; e++) {
0264            for (int k=0; k<n_nodes; k++) {
0265                F[e][k]=2.*U[e][k]-5.*U_m1[e][k]+4.*U_m2[e][k]-U_m3[e][k]-
0266                            pow(delta_t,2.)*U_dotdot[e][k];
0267            }
0268        }
0269
0270        // calculating the U-U_old difference
0271        for (int e=0; e<num_dimensions; e++) {
0272            for (int k=0; k<n_nodes; k++) {
0273                U_Uold_diff[e][k]=fabs(U[e][k]-U_old[e][k]);
0274            }
0275        }
0276
0277        // calculating the maximum difference between the displacement
0278        // at the current iteration step and at the previous one
0279        max_displacement_difference=0.;
0280        int k=0;
0281        int e=0;
0282        while (max_displacement_difference<displacement_tolerance && e<num_dimensions && k<n_nodes) {
0283            max_displacement_difference=U_Uold_diff[e][k];
0284            k++;
0285            if (k==n_nodes) {
```

```
0286                e++;
0287                k=0;
0288            }
0289
0290        }
0291
0292        // calculating the value of B_1 for the next iteration
0293        for (int e=0; e<num_dimensions; e++) {
0294
0295            temp3=0.;
0296            for (int i=0; i<n_nodes; i++) {
0297                temp1[e][i]=0.;
0298                for (int j=0; j<n_nodes; j++) {
0299                    temp1[e][i]+=B_1[e][i][j]*(F[e][j]-F_old[e][j]);
0300                }
0301
0302                if (U_Uold_diff[e][k]>=displacement_tolerance) {
0303                    for (int j=0; j<n_nodes; j++) {
0304                        temp2[e][i][j]=(U[e][i]-U_old[e][i]-temp1[e][i])*(U[e][j]-U_old[e][j]);
0305                    }
0306                }
0307
0308                temp3+=(U[e][i]-U_old[e][i])*temp1[e][i];
0309            }
0310
0311            for (int i=0; i<n_nodes; i++) {
0312                for (int j=0; j<n_nodes; j++) {
0313                    temp4[e][i][j]=0.;
0314                    if (U_Uold_diff[e][k]>=displacement_tolerance) {
0315                        for (int r=0; r<n_nodes; r++) {
0316                            temp4[e][i][j]+=temp2[e][i][r]*B_1[e][r][j]/temp3;
0317                        }
0318                    }
0319                }
0320            }
0321
0322            for (int i=0; i<n_nodes; i++) {
0323                for (int j=0; j<n_nodes; j++) {
0324                    B_1[e][i][j]+=temp4[e][i][j];
0325                }
0326            }
0327        }
0328
0329        // incrementing the iteration counter and the total iteration counter
0330        iteration++;
0331        tot_iterations++;
0332
0333        // checking if the maximum number of iteration has been reached
0334        if (iteration==max_iterations) {
0335            cout << "Convergency not reached with maximum iterations, shutting down. \n" << endl;
0336            return(0);
0337        }
0338
0339    }
0340
0341    // calculating the next value of the displacement and its derivative
0342    for (int e=0; e<num_dimensions; e++) {
0343        for (int k=0; k<n_nodes; k++) {
0344            U[e][k]=U_old[e][k];
0345            for (int j=0; j<n_nodes; j++) {
0346                U[e][k]+=-B_1[e][k][j]*F[e][j];
0347            }
0348        }
0349
0350        // setting to zero the displacement value of the boundary nodes
0351        for (int k=0; k<i_boundarynodes; k++) {
0352            U[e][k]=0.;
```

```
0353         }
0354
0355     }
0356
0357     // saving the displacement of the output nodes
0358     if (n>=output_index*time_steps/num_out_results && num_out_results>=output_index) {
0359
0360         for (int i=0; i<output_nodes.size(); i++) {
0361             displacement_peridynamic[i][output_index]=U[0][output_nodes[i]];
0362         }
0363         time[output_index]=n*delta_t;
0364         output_index++;
0365
0366     }
0367
0368     if (n>=perc_print*time_steps/100) {
0369
0370         // calculating the percentage of progress of time integration
0371         perc_progress=n/(double)time_steps*100;
0372
0373         // showing the time integration progress
0374         cout << fixed << perc_progress << "% completed \n " << endl;
0375         perc_print++;
0376
0377     }
0378
0379 }
```

Appendix C

Viscoelastic simulation codes

Hereinafter will be provided the portion of the MATLAB peridynamic code that performs the explicit time integration using the viscoelastic material formulation. Every step of the code is commented to provide a better understanding of the operations performed.

C.1 MATLAB code

```
0001 fprintf('Starting time integration algorithm \n');
0002
0003 % initializing the variables for output saving
0004 perc_print=0;
0005 output_index=0;
0006
0007 % core calculation of peridynamic equation
0008 for n=1:time_steps
0009
0010     body_f(1,n_nodes)=1/volume*sin(2*pi()/(5.e-4)*n*delta_t);
0011
0012     % saving the old stretch value
0013     stretch_old=stretch;
0014
0015     % performing a sweep of all (k) nodes
0016     for k=1:n_nodes
0017
0018         % performing a sweep of all (j) nodes in the neighbourhood of the
0019         % (k) node
0020         for i=1:max_interactions_per_node
0021
0022             % checking if the ID of the (j) node is valid
0023             if  j_nodes_id(i,k) ~= 0
0024
0025                 % saving the ID of the (j) node in the j variable
0026                 j=j_nodes_id(i,k);
0027
0028                 % saving the ID of the interaction in the ind_id variable
0029                 int_id=interaction_id(i,k);
0030
0031                 % calculating the eta(k)(j) vector
0032                 eta_vector(:,int_id)=U(:,j)-U(:,k);
0033
0034                 % calculating the norm of the csi+eta vector
0035                 norm_csi_eta=norm( csi_vector(:,int_id)+eta_vector(:,int_id) );
0036
0037                 % calculating the stretch(k)(j) value
0038                 stretch(1,int_id)=( norm_csi_eta - csi_modulus(1,int_id) )/csi_modulus(1,int_id);
```

```
0039
0040                 % calculating the unit vector of the interaction direction
0041                 int_direction_unit(:,int_id)=( csi_vector(:,int_id)+ ...
0042                                                 eta_vector(:,int_id) )/norm_csi_eta;
0043
0044             end
0045         end
0046     end
0047
0048
0049     % performing a second sweep of all (k) nodes to calculate the force
0050     % acting on the node
0051     for k=1:n_nodes
0052
0053         % initializing to zero the force(k) value
0054         force(:,k)=0;
0055
0056         % performing a sweep of all (j) nodes in the neighbourhood of the
0057         % (k) node
0058         for i=1:max_interactions_per_node
0059
0060             % checking if the ID of the (j) node is valid
0061             if  j_nodes_id(i,k) ~= 0
0062
0063                 % saving the ID of the (j) node in the j variable
0064                 j=j_nodes_id(i,k);
0065
0066                 % saving the ID of the interaction in the ind_id variable
0067                 int_id=interaction_id(i,k);
0068
0069                 % incrementing the value of the force
0070                 force(:,k)=force(:,k) + ( 4*b*delta*G_b(1,int_id)/(1-lambda)*...
0071                     (stretch(1,int_id)-lambda/csi_modulus(1,int_id)*e_db(1,int_id)))*...
0072                     int_direction_unit(:,int_id)*(v_c(1,int_id)*V(1,j));
0073
0074                 % calculating the back extension value at next time step
0075                 e_db(1,int_id)=stretch(1,int_id)*csi_modulus(1,int_id)*...
0076                     (1-exp(-delta_t/tau))+e_db(1,int_id)*exp(-delta_t/tau)+...
0077                     (1-tau/delta_t*(1-exp(-delta_t/tau)))*(stretch(1,int_id)-...
0078                     stretch_old(1,int_id))*csi_modulus(1,int_id);
0079
0080             end
0081         end
0082
0083         % calculating the acceleration vector
0084         U_dotdot(:,k)=( force(:,k)+body_f(:,k) )/ro(1,k);
0085
0086     end
0087
0088     % calculating the velocity vectors at the next time step
0089     U_dot=U_dotdot*delta_t+U_dot;
0090
0091     % calculating the displacement vectors at the next time step
0092     U=U_dot*delta_t+U;
0093
0094
0095     % setting to zero the displacement value of the boundary nodes
0096     for k=1:i_boundarynodes
0097         U(:,k)=0;
0098     end
0099
0100
0101     % saving the displacement of the output nodes
0102     if n >= output_index*time_steps/num_out_results && num_out_results>=output_index+1
0103         output_index=output_index+1;
0104         for i=1:size(output_nodes,2)
0105             displacement_peridynamic(i,output_index)=U(1,output_nodes(i));
```

```
0106         end
0107         time(1,output_index)=n*delta_t;
0108     end
0109
0110
0111     if n >= perc_print*time_steps/100
0112
0113         % calculating the percentage of progress of time integration
0114         perc_progress=n/time_steps*100;
0115
0116         % showing the time integration progress
0117         fprintf('%.0f%% completed \n',perc_progress);
0118         perc_print=perc_print+1;
0119
0120     end
0121
0122 end
```

Bibliography

[1] Askari E., Xu J., Silling S. A. *Peridynamic analysis of damage and failure in composites.* Paper presented at the 44th AIAA aerospace sciences meeting and exhibit. Grand Sierra Resort Hotel, Reno (2006)

[2] Broyden, C. G. *A Class of Methods for Solving Nonlinear Simultaneous Equations.* Mathematics of Computation (American Mathematical Society) (1965) vol. 19, pp. 577-593

[3] Foster J. T., Silling S. A., Chen W. *Viscoplasticity using peridynamics.* International journal for numerical methods in engineering (2010) vol. 81, pp. 1242-1258

[4] Kelley C. T. *Iterative Methods for Linear and Nonlinear Equations.* Society for Industrial and Applied Mathematics, Philadelphia (1995)

[5] Kilic B. *Peridynamic theory for progressive failure prediction in homogeneous and heterogeneous materials.* Dissertation, University of Arizona (2008)

[6] KilicB., Madenci E. *Peridynamic Theory for Thermomechanical Analysis.* Advanced Packaging, IEEE Transactions (2010) vol. 33, pp. 97-105

[7] Kilic B., Madenci E. *An adaptive dynamic relaxation method for quasi-static simulations using the peridynamic theory.* Theoretical and Applied Fracture Mechanics (2010) vol. 53, pp. 194-204

[8] Kilic B., Madenci E. *Coupling of peridynamic theory and the finite element method.* Journal of Mechanics of Materials and Structures (2010) vol. 5, pp. 707-735

[9] Mitchell J. A. *A Nonlocal, Ordinary, State-Based Plasticity Model for Peridynamics.* Sandia National Laboratories (2011) SAND2011-3166

[10] Mitchell J. A. *A Non-local, Ordinary-State-Based Viscoelasticity Model for Peridynamics.* Sandia National Laboratories (2011) SAND2011-8064

[11] Newmark N. M. *A method of computation for structural dynamics.* Journal of Engineering Mechanics (1959) vol. 85, pp. 67-94

[12] Oterkus E., Madenci E., Weckner O., Silling S. A., Bogert P., Tessler A. *Combined finite element and peridynamic analyses for predicting failure in a stiffened composite curved panel with a central slot.* Composite Structures (2012) vol. 94, pp. 839-850

[13] Oterkus E., Madenci E. *Peridynamic Theory and Its Applications.* Springer (2014) ISBN 978-1-4614-8464-6

[14] Silling S. A. *Reformulation of elasticity theory for discontinuities and long-range forces.* Journal of Mechanics and Physics of Solids (2000) vol. 48, pp. 175-209.

[15] Silling S. A., Epton M., Weckner O., Xu J., Askari E. *Peridynamic States and Constitutive Modeling.* Journal of Elasticity (2007) vol. 88, pp. 151-184

[16] Silling S. A., Bobaru F. *Peridynamic modeling of membranes and fibers.* International Journal of Non-Linear Mechanics (2005) vol. 40, pp. 395-409

[17] Silling S. A., Zimmermann M., Abeyaratne R. *Deformation of a peridynamic bar.* Journal of Elasticity (2003) vol. 73, pp. 173-190

[18] Silling S. A. *Linearized Theory of Peridynamic States.* Journal of Elasticity (2010) vol. 99, pp. 85-111

[19] Taylor M. J. *Numerical simulation of thermo-elasticity, inelasticity and rupture in membrane theory.* Dissertation, University of California, Berkley (2008)

[20] Weckner O., Abdullah N., Mohamed N. *Viscoelastic material models in peridynamics.* Applied Mathematics and Computation (2013) vol. 219, pp. 6039-6043