EPFL

SEMESTER PROJECT

Akantu implementation of the INTERNODES method for contact mechanics

Bruno Ploumhans & Fabio Matti

supervised by
Dr Guillaume Anciaux
Raquel Dantas Batista

February 12, 2023

CONTENTS

1 INTRODUCTION

INTERNODES (INTERpolation for NOn-conforming DEcompositionS) is a recently developed method for numerical contact mechanics. It was initially proposed in [1] and further developed for two-body contact problems in [4]. It is believed to be simpler and more robust than alternatives such as the mortar finite element method. Continuing the work of previous students, we fully implemented and tested the INTERNODES method in the framework of Akantu[1], a generic and efficient finite element solver developed by EPFL written in C++, with the main goal of making our implementation reliable for broader usage.

In this paper, we briefly explain how the method works. Then, we go through important details of our implementation, and we dicuss the tests that we wrote to validate it. Finally, we discuss further improvements that can be made to the computational contact mechanics package of Akantu.

[1] https://gitlab.com/akantu/akantu

2 THE INTERNODES METHOD FOR CONTACT MECHANICS

In essence, the INTERNODES method makes use of two different interpolants which are used to transfer information from one interface to another. To this purpose, radial basis function (RBF) interpolants are introduced.

2.1 RADIAL BASIS INTERPOLATION

The engine of radial basis interpolation are the radial basis functions (RBF) $\phi : \mathbb{R}^d \to \mathbb{R}$. These functions take values that are isometric, i.e. only depend on the distance from their corresponding node ξ and are parametrized by the radius parameter r.

The radial basis interpolant of $g : \mathbb{R}^d \to \mathbb{R}$ at interpolation nodes $\xi_1, \ldots, \xi_M$ with associated radius parameters $r_1, \ldots, r_M$ will be defined as

$$\Pi(\mathbf{x}) = \sum_{m=1}^{M} g(\xi_m)\phi(\|\mathbf{x} - \xi_m\|, r_m) \tag{2.1}$$

One can easily see that the interpolant is merely a superposition of rescaled RBFs. This fact, as well as the parameters used to define the radial basis function are depicted in Figure 2.1.

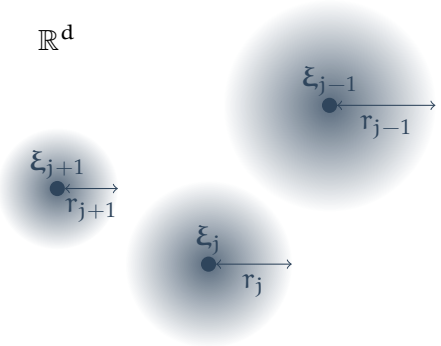


FIGURE 2.1 – Visualization of the parameters which are used to define radial basis functions. Notice that the radius parameters r_j must not necessarily coincide with the radius of the support of the function, as is the case for the Wendland C^2 RBF.

It is established in [4] that the Wendland C^2 RBF possesses particularly desirable properties. The aforementioned RBF is defined as

$$\phi(\delta) = (1-\delta)_+^4(1+4\delta) \tag{2.2}$$

with $\delta = \|\mathbf{x} - \xi_m\| / r$.

Denoting with $\mathbf{g}_\zeta = (g(\zeta_1), \ldots, g(\zeta_N))^\top$ the evaluations of the function g at positions $\zeta_1, \ldots, \zeta_N$ collected in a vector, and with $\mathbf{g}_\xi = (g(\xi_1), \ldots, g(\xi_M))^\top$ the function values at the interpolation nodes $\xi_1, \ldots, \xi_M$, we may write

$$\mathbf{g}_\zeta = \mathbf{D}_{NN}^{-1}\mathbf{\Phi}_{NM}\mathbf{\Phi}_{MM}^{-1}\mathbf{g}_\xi \tag{2.3}$$

where the radial basis matrices are defined as

$$(\mathbf{\Phi}_{MM})_{ij} = \phi(\left\|\xi_i - \xi_j\right\|, r_j) \quad i,j \in \{1,\dots,M\} \tag{2.4}$$
$$(\mathbf{\Phi}_{NM})_{ij} = \phi(\left\|\zeta_i - \xi_j\right\|, r_j) \quad i \in \{1,\dots,N\},\ j \in \{1,\dots,M\} \tag{2.5}$$

Therefore, the interpolation matrix is identified as

$$\mathbf{R}_{NM} = \mathbf{D}_{NN}^{-1}\mathbf{\Phi}_{NM}\mathbf{\Phi}_{MM}^{-1} \tag{2.6}$$

There are two particularities about the above formulation of an interpolation which distinguish it from classical radial basis interpolaion. The two modifications which were proposed in Deparis et. al. [1] are:

- Localized radius parameters for each node r_j, $j \in \{1,\dots,M\}$
- Rescaling with $\mathbf{D}_{NN}^{-1}$ to recover exact interpolation of constant functions, where we define

$$(\mathbf{D}_{NN})_{ij} = \begin{cases} (\mathbf{\Phi}_{NM}\mathbf{\Phi}_{MM}^{-1}\mathbf{1}_\zeta)_i, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{2.7}$$

These two modifications decisively impact the stability and well-posedness of the interpolation. To achieve a "good" interpolation, multiple conditions on the radius parameters r_j need to be satisfied. These will be written down and discussed in the next subsection.

2.2 Radius parameters

In order to ensure the invertibility of both $\mathbf{\Phi}_{MM}$ and $\mathbf{D}_{NN}$ the following conditions derived in [4] should hold:

Conditions on radius parameters

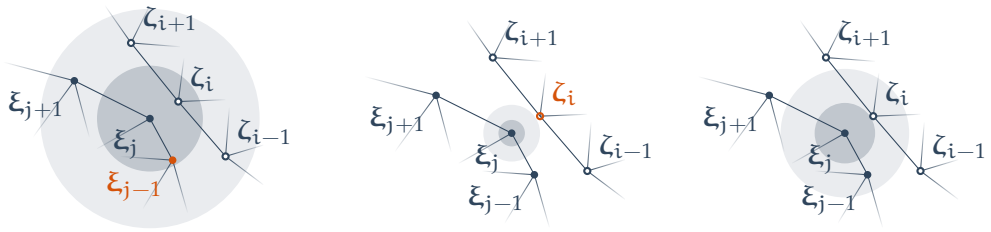
There exist $c \in (0,1)$ and $C \in (c,1)$ such that

$$\forall i : \#\{j \neq i : \left\|\xi_i - \xi_j\right\| < r_i\} < 1/\phi(c) \tag{Condition 1}$$
$$\forall i \neq j :\ \left\|\xi_i - \xi_j\right\| \geqslant c r_j \tag{Condition 2}$$
$$\forall i, \exists j :\ \left\|\zeta_i - \xi_j\right\| \leqslant C r_j \tag{Condition 3}$$

Condition 1 says that there can only be a limited number of other interpolation nodes within the support of the radial basis function sitting at each interpolation nodes. Condition 2 prohibits two interpolation nodes from being chosen too close to each other. Finally, Condition 3 ensures that at each point at which the interpolant will be evaluated is within the support of an interpolation nodes.

In Figure 2.2 these conditions are visualized using examples which violate them.

Our task is now to find suitable radius parameters which satisfy the above conditions. To first simplify the notation and later use it to reduce the complexity of the algorithm we introduce the distance matrix between two sets of nodes $\{\xi_1, \xi_2, \dots\}$ and $\{\zeta_1, \zeta_2, \dots\}$ which we define as



(a) Violates Condition 2. (b) Violates Condition 3. (c) Valid parameter choice.

FIGURE 2.2 – Visualization of the process of finding suitable radius parameters. The inner circle has radius cr_j while the outer one has radius Cr_j. Marked in orange are nodes which in each configuration cause one of the conditions to be violated.

$$\mathbf{D}^{\xi,\zeta}(i,j) = \left\|\xi_i - \zeta_j\right\| \tag{2.8}$$

Making use of this definition, we now propose a procedure, which, given an arbitrary set of interpolation nodes $\{\xi_1, \xi_2, \dots\}$ finds a suitable set of radius parameters $\{r_1, r_2, \dots\}$. The procedure is detailed in Algorithm 1.

Algorithm 1 Computation of radius parameters

Require: Positions of interpolation nodes $\{\xi_1, \xi_2, \dots\}$
Require: Radial basis function $\phi : \mathbb{R} \to \mathbb{R}_{\geqslant 0}$
Require: Constant $c \in (0, C)$
Compute distance matrix $\mathbf{D}^{\xi,\xi}$ defined in (2.8)
For each node i, compute distance to closest neighbor $d_i = \min_{j \neq i} \mathbf{D}^{\xi,\xi}(i,j)$
while $c \leqslant C$ **do**
    For each node i, let $r_i \leftarrow d_i/c$ ▷ Condition 2
    For each node i, count $n_i = \#\{j \neq i : \mathbf{D}^{\xi,\xi}(i,j) < r_i\}$
    if For all nodes i, $n_i < 1/\phi(c)$ **then** ▷ Condition 1
        break
    end if
    Increase c
end while
return Radius parameters $\{r_1, r_2, \dots\}$

In the setting of finite element approximations for contact mechanics, the interpolation nodes are usually taken to be the mesh points corresponding to the interface between two bodies that are in contact. Consequently, $\{\xi_1, \xi_2, \dots\}$ are the mesh points which correspond to one body, named the "primary", and $\{\zeta_1, \zeta_2, \dots\}$ correspond to the other body, named the "secondary". Based on Algorithm 1, we now propose a second procedure which ensures that for a given contact problem Condition 3 is satisfied. In multiple iterations an index set $\mathcal{I}$ of primary interface nodes and an index set $\mathcal{J}$ of secondary interface nodes are determined by removing nodes from the original set of nodes which are isolated, i.e. not clearly within the

support of an interpolation node from the opposite interface. The procedure is given in Algorithm 2.

Algorithm 2 Search for interpolation nodes

Require: Positions of primary nodes $\{\xi_1, \xi_2, \dots\}$
Require: Positions of secondary nodes $\{\zeta_1, \zeta_2, \dots\}$
Require: Radial basis function $\phi : \mathbb{R}^d \to \mathbb{R}_{\geqslant 0}$
Require: Constant $C \in (c, 1)$
1: Let $\mathcal{I} = \{1, 2, \dots\}$ and $\mathcal{J} = \{1, 2, \dots\}$ denote an index set of active nodes
2: Compute distance matrix $\mathbf{D}^{\xi,\zeta}$ defined in (2.8)
3: **while** $\mathcal{I}$ or $\mathcal{J}$ were modified in the previous iteration **do**
4: Obtain radial basis parameters r_i^ξ, $i \in \mathcal{I}$ and r_j^ζ, $j \in \mathcal{J}$ using Algorithm 1
5: Remove isolated nodes $i \in \mathcal{I}$ with $\min_{j\in\mathcal{J}} \mathbf{D}^{\xi,\zeta}(i,j) \geqslant C r_j^\zeta$ ▷ Condition 3
6: Remove isolated nodes $j \in \mathcal{J}$ with $\min_{i\in\mathcal{I}} \mathbf{D}^{\xi,\zeta}(i,j) \geqslant C r_i^\xi$ ▷ Condition 3
7: **end while**
8: **return** Sets of active nodes $\mathcal{I}$ and $\mathcal{J}$ with radius parameters $r_i^\xi, i \in \mathcal{I}$ and $r_j^\zeta, j \in \mathcal{J}$

These two algorithms (Algorithm 1 and Algorithm 2) hold three main benefits over the implementation found in [4]:

- For uniform meshes (constant element size) all radial basis parameters will be the exact same, i.e. $r_1^\xi = r_2^\xi = \cdots = r_1^\zeta = r_2^\zeta = \cdots$.
- The computation of the distance matrix is done outside the `while`-loop, and hence reduces the compuational complexity by a factor which corresponds to the number of iterations required to find the suitable sets of interpolation nodes
- The algorithm will find interface nodes for a wider class of examples.

This procedure can further be optimized by introducing a `SpatialGrid`, which we have done in our implementations in Akantu.

2.3 Strong form

A stereotypical example of a contact problem which we aim to solve is sketched in Figure 2.3.

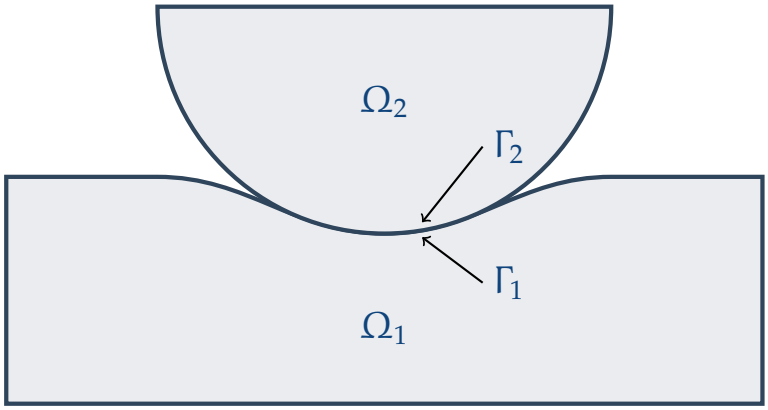


FIGURE 2.3 – Contact problem where a semisphere Ω_2 interfaces with a half-space Ω_1.

The strong form of the problem is formulated for the displacement field $\mathbf{u} : \Omega_{1,2} \to \mathbb{R}^d$. Each body participating in the contact needs to individually satisfy the following equilibrium equations [4]:

Equilibrium equations

A solid subject to an external force $\mathbf{f}$ satisfies the following boundary problem

$$\begin{aligned}
\operatorname{div}(\boldsymbol{\sigma}(\mathbf{u})) &= \mathbf{f} && \text{(Differential equation with Cauchy stress tensor } \boldsymbol{\sigma}\text{)} \\
\mathbf{u} &= \mathbf{g} && \text{(Dirichlet boundary conditions with displacement field } \mathbf{g}\text{)} \\
\boldsymbol{\sigma}(\mathbf{u})\mathbf{n} &= \mathbf{t} && \text{(Neumann boundary conditions with surface traction } \mathbf{t}\text{)} \\
\boldsymbol{\sigma}(\mathbf{u})\mathbf{n} &= \boldsymbol{\lambda} && \text{(Lagrange multipliers } \boldsymbol{\lambda} \text{ defined along interface } \Gamma\text{)} \\
\boldsymbol{\lambda} \cdot \mathbf{n} &\leqslant 0 && \text{(Hertz-Signorini-Moreau condition along interface } \Gamma\text{)}
\end{aligned}$$

In order to numerically approximate solutions for these problems, a finite element discretization of the domain needs to be computed, and the strong form converted into a weak form of the problem. This will be done in the next section.

2.4 WEAK FORM

For the weak formulation, the Hertz-Signorini-Moreau inequality constraint is relaxed, which results in a linear system for the displacements $\mathbf{u}$ and Lagrange multipliers $\boldsymbol{\lambda}$ [4]. The linear system of equations takes the form of a saddle point problem.

INTERNODES system of equations

For a contact problem between two bodies Ω_1 and Ω_2 with interfaces Γ_1 and Γ_2 the following system of equations is solved:

$$\underbrace{\left[\begin{array}{cc|cc|c} \mathbf{K}_{\Omega_1\Omega_1} & \mathbf{K}_{\Omega_1\Gamma_1} & & & \\ \mathbf{K}_{\Gamma_1\Omega_1} & \mathbf{K}_{\Gamma_1\Gamma_1} & & & -\mathbf{M}_{\Gamma_1} \\ \hline & & \mathbf{K}_{\Omega_2\Omega_2} & \mathbf{K}_{\Omega_2\Gamma_2} & \\ & & \mathbf{K}_{\Gamma_2\Omega_2} & \mathbf{K}_{\Gamma_2\Gamma_2} & -\mathbf{M}_{\Gamma_2}\mathbf{R}_{\Gamma_2\Gamma_1} \\ \hline & & & -\mathbf{R}_{\Gamma_1\Gamma_2} & \end{array}\right]}_{=\mathbf{A}} \left[\begin{array}{c} \mathbf{u}_{\Omega_1} \\ \mathbf{u}_{\Gamma_1} \\ \hline \mathbf{u}_{\Omega_2} \\ \mathbf{u}_{\Gamma_2} \\ \hline \boldsymbol{\lambda} \end{array}\right] = \underbrace{\left[\begin{array}{c} \mathbf{f}_{\Omega_1} \\ \mathbf{f}_{\Gamma_1} \\ \hline \mathbf{f}_{\Omega_2} \\ \mathbf{f}_{\Gamma_2} \\ \hline \mathbf{d} \end{array}\right]}_{\mathbf{b}} \tag{2.9}$$

The subscripts indicate which degrees of freedom are collected in the objects. $\mathbf{M}$ and $\mathbf{K}$ denote the finite element mass and stiffness matrices respectively. The interpolation matrices $\mathbf{R}$ are defined in (2.6). The external force is $\mathbf{f}$ and the nodal gaps between the interfaces is $\mathbf{d}$.

2.5 INTERNODES CONTACT ALGORITHM

In [4] an algorithm is suggested for solving problems in contact mechanics. For an initial guess of the interface Γ, the algorithm solves the system (2.9) and in each iteration updates the interface by removing interface nodes that are in tension or adding nodes that are interpenetrating to the interface. In all available reference implementations, these two operations of adding and removing interface nodes were treated as mutually exclusive in each iteration [3, 4, 5]. In order to speed up the convergence, we have modified the procedure to be able to simultaneously remove nodes from and add back nodes to the interface.

The algorithm is said to have converged, i.e. a solution to the problem has been found, if

1. No nodes belonging to the interface are in tension, i.e.

$$\boldsymbol{\lambda} \cdot \mathbf{n} \leqslant 0 \tag{2.10}$$

2. No interpenetrating nodes exist, i.e. the nodal gaps $\mathbf{d}$ after solving the system all satisfy

$$\mathbf{d} \cdot \mathbf{n} \geqslant 0 \tag{2.11}$$

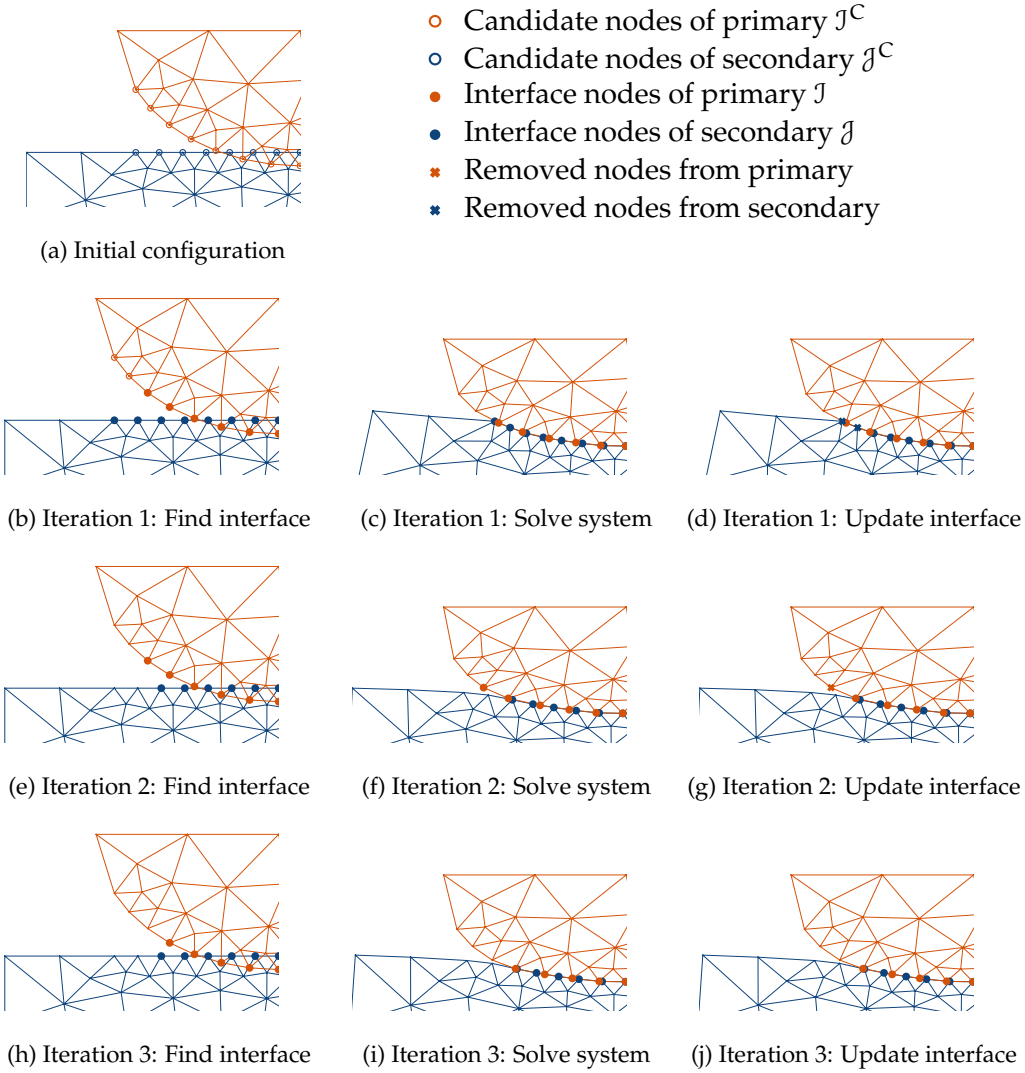
This procedure is summarized in Algorithm 3.

In order to understand this algorithm more thoroughly, we run the algorithm on an example problem and visualize the different steps in Figure 2.4. Notice that for producing this figure we use a simpler version of the algorithm where the iterations are only used to define the interface nodes. The Python implementation for prototyping uses this version for simplicity. The alternative is to use the "unphysical" intermediary solutions as the starting points for the next iteration in the algorithm. This is done in the Akantu implementation.

Algorithm 3 Contact algorithm for interrnodes method

Require: Positions of primary nodes $\{\xi_1, \xi_2, \dots\}$
Require: Positions of secondary nodes $\{\zeta_1, \zeta_2, \dots\}$
Require: Interface candidate index sets $\mathcal{I}^C$ and $\mathcal{J}^C$

1: **while** $\mathcal{I}^C$ or $\mathcal{J}^C$ were modified in the previous iteration **do**
2: Determine interface nodes $\mathcal{I}$ and $\mathcal{J}$ with radius parameters $r_i^\xi, i \in \mathcal{I}$ and $r_j^\zeta, j \in \mathcal{J}$ using Algorithm 2 on the candidate index sets $\mathcal{I}^C$ and $\mathcal{J}^C$
3: Assemble the matrix $\mathbf{A}$ and right-hand side $\mathbf{b}$ of (2.9)
4: Solve (2.9) to obtain displacements $\mathbf{u}$ and Lagarnge multipliers $\boldsymbol{\lambda}$
5: Update the interface candidate nodes $\mathcal{I}^C$ and $\mathcal{J}^C$ with $\mathcal{I}$ and $\mathcal{J}$, respectively, by removing all nodes in tension and adding interpenetrating nodes
6: **end while**



(a) Initial configuration

(b) Iteration 1: Find interface

(c) Iteration 1: Solve system

(d) Iteration 1: Update interface

(e) Iteration 2: Find interface

(f) Iteration 2: Solve system

(g) Iteration 2: Update interface

(h) Iteration 3: Find interface

(i) Iteration 3: Solve system

(j) Iteration 3: Update interface

FIGURE 2.4 – Example of the INTERNODES algorithm for contact mechanics applied to a two-dimensional problem where a circle interfaces with a rectangular structure. Convergence is obtained after three iterations.

3 IMPLEMENTATION OF THE INTERNODES METHOD

Our main objective with this project was to fully implement the INTERNODES method and integrate it with the Akantu finite-element library. For a few years, Akantu has had one contact mechanics model implementation, that we will usually call the *penalty method*. Our main objective was to provide a viable alternative for Akantu users. As such, we had multiple goals beyond the mere implementation of the INTERNODES method:

- To make future developments easier and to keep the code neatly organized, the INTERNODES and penalty implementations should follow a similar structure and share as much code as possible.
- To allow users to switch between the two methods easily, the Python interface for INTERNODES and penalty contact should be as close as possible.
- Last but not least, the INTERNODES method implementation should be robustly implemented and well-tested to be useful in a variety of scenarios.

At this point, it is important to note that our work is the continuation of a Merge Request by Moritz Waldleben [5] adding a partially complete Akantu C++ implementation of INTERNODES. We also had access to a Python implementation by Moritz Waldleben and an earlier MATLAB implementation by Yannis Voet [4] as references.

We first cover how we finished the implementation of the INTERNODES method and how it fits within Akantu. We also briefly discuss the Python interface. Then, we discuss the tests that we wrote to validate and verify our implementation.

3.1 IMPLEMENTATION DISCUSSION

3.1.1 EXTENDED PYTHON PROTOTYPE

One of the first things we did was to refactor and extend Mortz Waldleben's Python reference implementation. This allowed us to better understand the method, and made prototyping easier. The implementation is available on c4science[2].

Thanks to this first step, we came up with validation tests that we later ported to C++ (see Section 3.2).

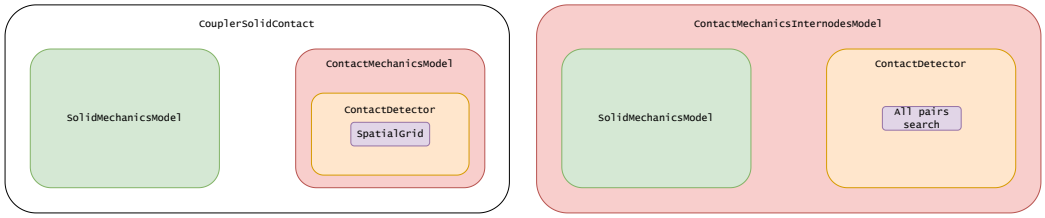
It is also thanks to this extended Python prototype that we noticed flaws in the previous implementations, particularly around the convergence check (i.e. equations (2.10) and (2.10)). Once we got the convergence check working in Python, we were then able to port it to C++. In general, the extended prototype was the testing ground for various improvements that made it into the C++ version.

3.1.2 C++ ADDITIONS

Let us first look at the code organization of the two contact mechanics models before our changes, which can be seen in Figure 3.1.

The `ContactMechanicsModel` (i.e. the penalty method) was initially implemented as follows:

[2] https://c4science.ch/source/INTERNODES-CM/



(a) Penalty implementation diagram (b) INTERNODES implementation diagram

FIGURE 3.1 – Contact mechanics architectures.

- `ContactMechanicsModel` is the main class of that model.
- `ContactDetector` handles the contact detection phase, i.e. finding out which elements of the two bodies are in contact.
  - To avoid searching over all pairs, the contact detector uses a `SpatialGrid` to skip checking elements that are too far.
- `CouplerSolidContactTemplate` takes care of the coupling between a solid mechanics model, and the penalty contact detector model.

The `ContactMechanicsInternodesModel` (i.e. the INTERNODES method) was initially implemented as follows:

- `ContactMechanicsInternodesModel` is the main class of that model.
- `ContactDetectorInternodes` handles the contact detection phase.
  - No `SpatialGrid` is being used, so all pairs need to be checked.
- The `ContactMechanicsInternodesModel` contains a `SolidMechanicsModel`. There is no separate coupler.

As a performance optimization and a first step toward shared abstraction, we changed `ContactDetectorInternodes` to use a `SpatialGrid`. With this we introduced `AbstractContactDetector`: a base class for contact detectors, containing mostly helper functions to build the spatial grids, and holding the updated positions vector used by both contact detectors. Unfortunately, we were not able to further refactor the contact detectors nor the contact models due to their largely different internal workings.

The main architectural difference between penalty and INTERNODES is that in the former a coupler model manages all the interaction between the solid and the contact mechanics models, whereas in the latter the contact model directly references the solid model. We attempted to make INTERNODES use a coupler model too, but we found it too complicated. As a compromise, we ensured that the Python interfaces were very similar, so that a user could easily swap from one to the other, as seen in Figure 3.2. In Section 4.2, we discuss this in more detail and propose a comprehensive solution.

```
  mesh = aka.Mesh(spatial_dimension)
  mesh.read(mesh_file)
- model = aka.CouplerSolidContact(mesh)
+ model = aka.ContactMechanicsInternodesModel(mesh)
  model.applyBC(...)
  # and so on...
```

FIGURE 3.2 – Moving from penalty to INTERNODES contact mechanics in Python.

Beyond architecture concerns, a crucial missing feature in the C++ implementation was the convergence check. We implemented it in C++ as we have explained in Section 2.5. Even though they were loosely ported from the Python prototype, the functions `getInterfaceNormalAtNode`, `findPenetratingNodes`, and `updateAfterStep` are of particular interest.

We also extended the INTERNODES method to work with 3D problems. The code is essentially the same, except for a difference in the computation of the interface normal.

It should be noted that the number of active nodes, hence of DOFs (degrees of freedom), can change between two iterations. We were unable to resize an allocated DOFs vector with Akantu. As a workaround, we allocate enough DOFs for all possibly active nodes (i.e. the initial candidate sets). To preserve numerical stability for the system resolution, we use an identity submatrix for the inactive nodes.

While we were implementing the above, we made a few important changes to the core of Akantu. We

- fixed multiplication of a non-square matrix by a vector throwing an exception;
- expanded `NodeGroup::applyNodeFilter` to accept lambdas and return how many elements were erased;
- fixed undefined behavior in `NodeGroup::applyNodeFilter` and `Array::erase`;
- expanded `SpatialGrid` with helper methods to list values neighboring a specific cell and point;
- added a few missing `const` markers.

3.2 EXPERIMENTS AND TESTS

In order to verify the correctness and the implementation and ensure the integrity of the code during development, a set of test cases were implemented. Each is a problem which can be solved analytically using the theory of Hertzian mechanics [2]. This allows a comparison of certain quantities of interest with the approximation obtained with the INTERNODES method.

3.2.1 CONTACT BETWEEN A SEMISPHERES AND A HALF-SPACE

The first test case involves a semisphere being vertically pushed into a half-space both in two and three dimensions. The experimental setup is shown in Figure 3.3.

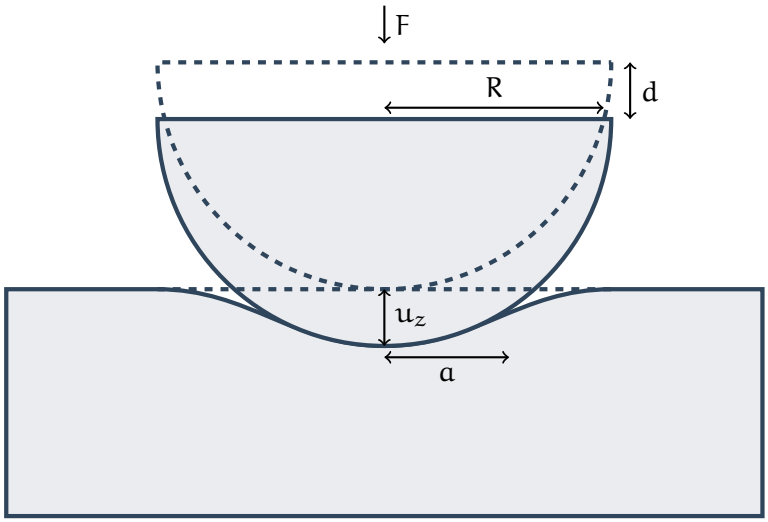


FIGURE 3.3 – Sketch of the first experimental setup used for testing.

Below, the analytical quantities which are used to verify the implementation are defined. The symbols listed in Table 3.1 are used.

TABLE 3.1 – Explanation of symbols

Symbol	Description
E	Young's modulus
ν	Poisson's ratio
R	Radius of semisphere/semicircle
d	Applied vertical displacement
a	Radius of contact area
p_0	Contact pressure amplitude
u_z	Penetration depth

The radius of the contact area is given by

$$a = \sqrt{Rd} \tag{3.1}$$

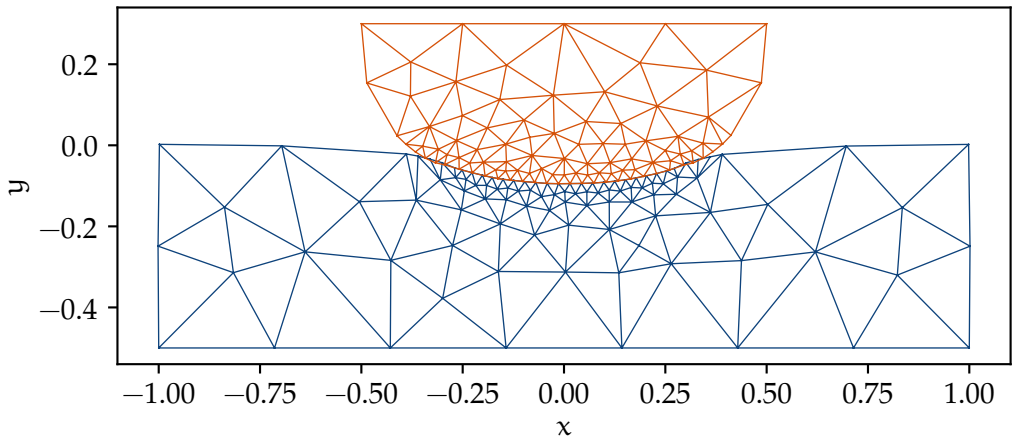
The amplitude of the contact pressure between the two interfaces is

$$p_0 = \frac{E}{\pi(1-\nu^2)}\sqrt{\frac{d}{R}} \tag{3.2}$$

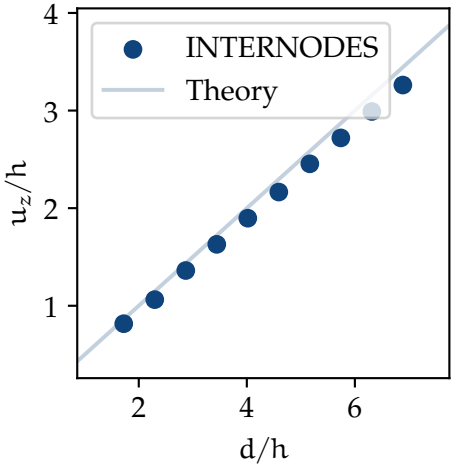
The normal displacement (i.e. the penetration depth of the semisphere into the half-space) is

$$u_z = \frac{d}{2} \tag{3.3}$$

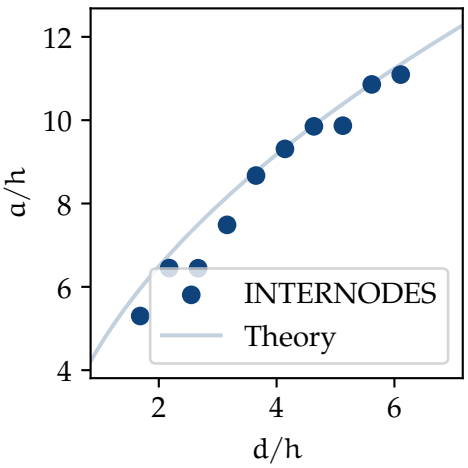
We now run the Akantu implementation on one instance of this problem. For the two-dimensional case, Figure 3.4a shows the obtained solution for a single applied vertical displacement d. In Figure 3.4b and Figure 3.4c we solve the same problem with a fixed mesh for multiple different displacements d and plot the penetration depth u_z and the radius of the contact are a, respectively. All quantities are normalized by the mesh size h at the interface.



(a) The solution obtained from solving the problem. The mesh is refined around the interface in order to increase the precision.



(b) Penetration depths u_z for multiple vertical displacements d scattered against the exact values obtained with (3.3). The quantities are normalized by the mesh size h at the interface.

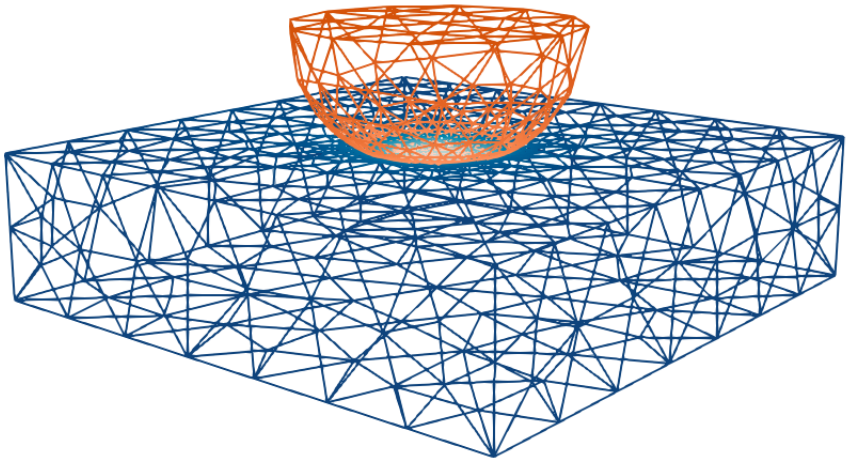


(c) Radii of contact areas a for multiple vertical displacements d scattered against the exact values obtained with (3.1). The quantities are normalized by the mesh size h at the interface.

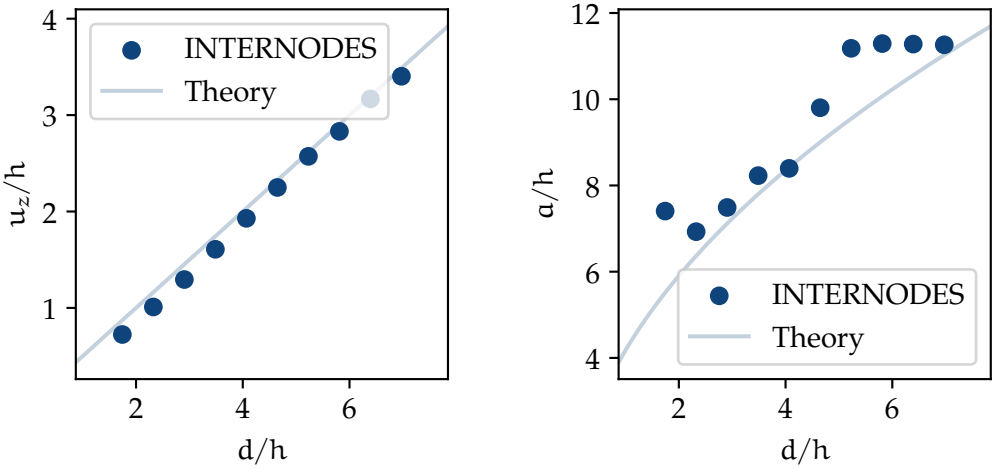
FIGURE 3.4 – Results obtained for a problem of the type sketched in Figure 3.3. For a fixed mesh of mesh size h at the interface, multiple displacements d are applied.

A similar analysis is now done for the three dimensional case. Here, a sphere is pushed into a half-space, analogously to Figure 3.3, and the results are visualized

in Figure 3.5. Notice that in three dimensions the computation of the radius of the contact area is not as robust as in two dimensions, since we have more degrees of freedom. The error for said quantity, however, is still within two mesh sizes h, which is to be expected.



(a) Solution obtained from solving the problem. The color corresponds to the total displacement of the respective nodes. Orange is associated with a large displacement, while blue means the node was invariant.



(b) Penetration depths u_z for multiple vertical displacements d scattered against the exact values obtained with (3.3). The quantities are normalized by the mesh size h at the interface.

(c) Radii of contact areas a for multiple vertical displacements d scattered against the exact values obtained with (3.1). The quantities are normalized by the mesh size h at the interface.

FIGURE 3.5 – Results for the three dimensional version of the problem sketched in Figure 3.3, i.e. a semisphere is pushed into a half-space. For a fixed mesh of mesh size h at the interface, multiple displacements d are applied.

3.2.2 CONTACT BETWEEN TWO SEMISPHERES

The second test case models the contact between two semispheres (Figure 3.6).

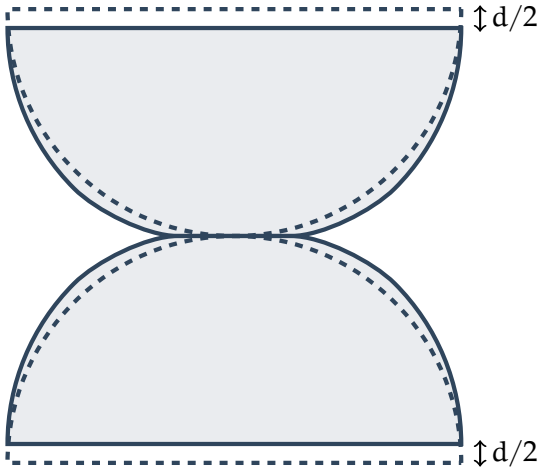
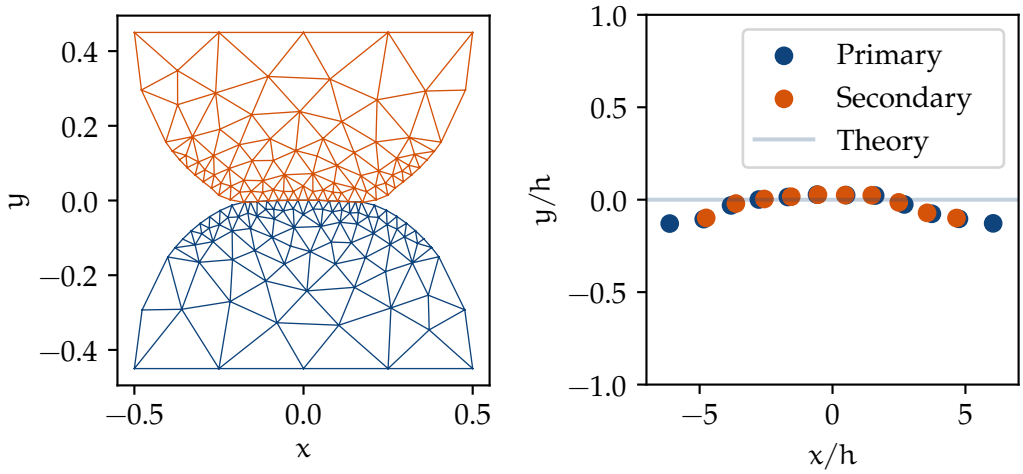


FIGURE 3.6 – Sketch of the second experimental setup where two semispheres are put into contact.

In theory, the resulting interface is planar. This fact is used to verify the implementation by testing whether the deviation of the interface nodes from the theoretical contact plane is within a reasonable upper bound.



(a) Plot of the obtained solution.

(b) Zoomed-in view of the interface nodes of both the primary and the secondary body. From theory we would expect them to describe a line. The axes are normalized by the mesh size h at the interface.

FIGURE 3.7 – Results for a two dimensional problem with two semicircles in contact.

4 FURTHER WORK

4.1 CONTACT MECHANICS REFACTORS

For a long time, Akantu only had a single implementation of contact mechanics, which is why its classes use generic names such as `ContactMechanicsModel` or `ContactDetector`. Now that there is a second contact mechanics implementation, we could simply rename the penalty method classes to more appropriate names, for example `ContactMechanicsPenaltyModel` or `ContactDetectorPenalty`. However, these renames should come with a larger refactor of the architecture of the contact mechanics package, that we will discuss in the next section.

4.2 SHARED INTERFACE FOR BOTH CONTACT MODELS

As discussed in Section 3.1.2, we attempted to refactor `CouplerSolidContact` to use it for the INTERNODES method. The attempt is accessible on the Akantu repository[3]. While we were not successful, this allowed us to find an issue with the current architecture, and leads us to propose a solution...

A key problem is that the `ContactMechanicsModel` (penalty method) extends `Model` even though it is not usable as a standalone model, and needs to be used through the `CouplerSolidContact`. Additionally, the `CouplerSolidContact` does not act as a standalone `Model`-`Model` coupler; it is quite tailored to the implementation details of `ContactMechanicsModel`. Hence, we should simply combine the two classes and make the resulting contact model contain the solid model.

As such, we propose the following refactor:

1. Move all the penalty contact mechanics code from `ContactMechanicsModel` to `CouplerSolidContact`, under a new name. To express that it's the penalty contact model, we suggest `ContactMechanicsPenaltyModel`.

2. Introduce a shared base class for `ContactMechanicsPenaltyModel` and `ContactMechanicsInternodesModel` that contains the solid model, and all the code that is common to all contact mechanics implementations.

3. Once that is done, to ensure that both methods can be used in all situations, it will be useful to go through the two contact mechanics model implementations, and validate that they work correctly with all the solver types.

The final class diagram is on Figure 4.1. Here is a brief reminder of what each class is supposed to do in the final state:

1. `Model`: base class for all models in Akantu, unchanged.

2. `ContactMechanicsModel`: base class for all contact mechanics implementations. Contains a `SolidMechanicsModel` instance, and all the methods that are relevant to all the contact models.

[3] https://gitlab.com/akantu/akantu/-/commit/87d81a453d57f97546d166989c9a4dfa27cf19f6

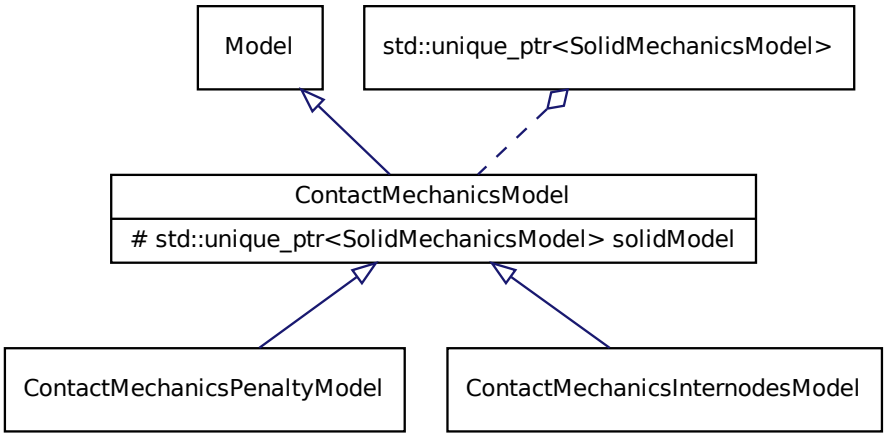


FIGURE 4.1 – Proposed contact mechanics package refactor.

3. `ContactMechanicsPenaltyModel`: penalty contact mechanics model, fully handles both the contact and solid parts of the problem when `solveStep` is called.

4. `ContactMechanicsInternodesModel`: INTERNODES contact mechanics model, fully handles the contact and solid parts of the problem when `solveStep` is called. This will be quite similar to the current state of INTERNODES, with some of the methods moved to the superclass.

5 Conclusion

In conclusion, we detailed how the INTERNODES method works, identified many shortcomings of the previous implementations, and improved upon them to deliver a robust and well-tested implementation in Akantu, in the hope that its users will find it suitable for further research in numerical contact mechanics.

REFERENCES

[1] Simone Deparis, Davide Forti, and Alfio Quarteroni. A rescaled localized radial basis function interpolation on non-cartesian and nonconforming grids. *SIAM Journal on Scientific Computing*, 36(6):A2745–A2762, 2014. doi: 10.1137/130947179.

[2] Kenneth Langstreth Johnson. *Contact mechanics*. Cambridge University Press, 1985.

[3] Yannis Voet. On the preconditioning of the internodes matrix for applications in contact mechanics. Master's thesis, EPFL, 2021.

[4] Yannis Voet, Guillaume Anciaux, Simone Deparis, and Paola Gervasio. The internodes method for applications in contact mechanics and dedicated preconditioning techniques. *Computers & Mathematics with Applications*, 127:48–64, 2022. doi: 10.1016/j.camwa.2022.09.019.

[5] Moritz Waldleben. Implementation of the internodes method for contact mechanics, 2022.

APPENDIX

REFACTOR OF PENALTY RESOLUTION CLASSES

While working on the contact mechanics code written for Akantu, we have identified a lot (800+ lines) of shared code in the `Resolution` family of classes. These classes we then refactored according to the diagram sketched in Figure 5.1. In order not to pay the cost for virtual calls, the refactor is done by creating a templated `ResolutionPenalty` class. This class is now easily extendable with other penalty function beyond the linear and quadratic one. The implementation can be found on a feature branch of Akantu[4].

(a) Initial class diagram of resolutions

(b) Refactored class diagram of resolutions

FIGURE 5.1 – Combine linear and quadratic penalty resolution by templating the classes with a penalty function.

[4]https://gitlab.com/akantu/akantu/-/tree/features/modularize-resolution-penalty-classes