Continuum Modelling using the Discrete Element Method. Theory and Implementation in an Object-Oriented Software Platform

M. Santasusana
E. Oñate

Publication CIMNE Nº-381, September 2012

Continuum Modelling using the Discrete Element Method. Theory and Implementation in an Object-Oriented Software Platform

M. Santasusana
E. Oñate

Publication CIMNE Nº-381, September 2012

International Center for Numerical Methods in Engineering
Gran Capitán s/n, 08034 Barcelona, Spain

Abstract

The Discrete Element Method is a relatively new technique that has nowadays and intense research in the field of numerical methods. In its first conception, the method was designed for simulations of dynamic system of particles where each element is considered to be an independent and non-deformable entity that interacts with other particles by the laws of the contact mechanics and moves following the second Newton's law. This first approach for the DEM has obtained excellent results for granular media simulations or another discontinuous-like case. The existing challenge nowadays for the DEM is to be able to simulate the behaviour on a continuous media discretized by a mesh of particles ruled by the equations of the DEM. Although there exist more adequate methods to solve the continuous problem as they are the different variants of the Finite Element Method, the DEM is expected to have a better behaviour when the failure of the media occurs; in terms of tracking the evolution of the fracture locally between the elements of the discretization and also the post-fractural behaviour of the material.

Nowadays, there are several DEM codes that try to solve this problem although there is no one which can assure an accurate solution applicable universally to any case. The objective of the present work is to develop calculation software for the Discrete Element Method included in the platform for numerical methods KRATOS, which is developed in CIMNE. One of the goals of the so-called *DEM-Application* is to be able to reproduce a wide set of engineering problems; not only the discrete ones such as the excavation or agroalimentary applications but also to reproduce the continuous media, simulating compression test for concrete or asphalt samples for instance. In addition it is desired that the application permits the coupling with another methods, particularly the Finite Element Method.

In order to do this, the present work includes the study of all the advances and ideas that, globally in the numerical method field and particularly in CIMNE, have been discussed to give other approaches and to keep improving and developing the to the Discrete Element Method.

Title: *Continuum modelling using the Discrete Element Method. Theory and implementation in an object-oriented software platform.*
Author: *Miquel Santasusana Isach*
Supervisors: *Eugenio Oñate Ibáñez de Navarra, Miguel Ángel Celigueta Jordana.*
Key words: *Discrete Element Method, KRATOS, CIMNE, continuum simulation, C++ programming.*

Resum

El Mètode dels Elements Discrets és un mètode relativament nou el qual avui dia és objecte d'una intensa recerca en el món dels mètodes numèrics. Originalment el mètode fou concebut per a la simulació de sistemes dinàmics de partícules on cada element és considerat com a entitat independent i indeformable que interacciona amb les altres seguint les lleis del contacte mecànic i es mou segons la segona llei de Newton. Aquest primer plantejament sobre el M.E.D. a tingut molts bons resultats per a simulacions de medis granulars o qualsevol assimilable a un medi discontinu. El repte actual per al DEM és ésser capaç de simular també el comportament d'un medi continu discretitzat per una malla de partícules que interaccionin segons les lleis del M.E.D. Tot i existir mètodes molt més adequats a resoldre aquest problema com son les variants del Mètode dels Elements Finits, el M.E.D. promet tenir un millor comportament a l'hora de seguir l'evolució de la fractura a nivell local entre els elements de la discretització i el comportament post fracturat del material.

Actualment, existeixen molts programes de Elements Discrets que intenten resoldre aquest problema sense haver-n'hi cap que asseguri una solució acurada aplicable a nivell universal i amb versatilitat. L'objectiu de la tesina és desenvolupar un programa de càlcul d'Elements Discrets inclòs en la plataforma per mètodes numèrics KRATOS, desenvolupada al CIMNE. Un dels objectius de l'anomenada *DEM-Application* és poder reproduir un ampli conjunt de problemes d'enginyeria; no tan sols els que són merament "discrets" com el medis granulars, tal i com poden ser l'excavació o aplicacions agroalimentàries, sinó també la simulació del medi continu, com ara reproduir provetes de formigó, asfalt, etc. Paral·lelament es desitja que l'aplicació permeti el càlcul acoblat amb altres mètodes, en particular amb el Mètode dels Elements Finits.

Per dur-ho a terme, en la tesina, s'ha estudiat tot el conjunt d'avenços i idees que, en el món dels mètodes numèrics a nivell global i a CIMNE en particular, es plantegen per donar altres punts de vista, millorar i continuar desenvolupant el Mètode dels Elements Discrets.

Títol: *Modelització del medi continu mitjançant el Mètode dels Elements Discrets. Teoria i implementació en una plataforma de programació orientada a objectes.*
Autor: *Miquel Santasusana Isach*
Supervisors: *Eugenio Oñate Ibáñez de Navarra, Miguel Ángel Celigueta Jordana.*
Paraules clau: *Mètode dels Elements Discrets, KRATOS, CIMNE, simulació del continu, programació C++.*

Table of Contents

List of Figures

PART I: DISCRETE ELEMENT METHOD

PART II: KRATOS DEM-APPLICATION

Introduction and objectives

This dissertation is the result of the implementation of a Discrete Element Method code in an open source object-oriented software platform called KRATOS developed in CIMNE (Barcelona). The result of this work is the so-called DEM-Application, which is the program that has been coded for the author forming part of team of engineers in CIMNE.

This document presents all the discussions and the special topics that have been taken into account in order to develop the application. A basic introduction to the Discrete Element Method is presented in the first part of the document with the topics in discussion for the special features and characteristics of the DEM, including the features needed to introduce the simulation of the continuous media with the DEM.

In the second part, the KRATOS framework is introduced and the basic structure of the DEM-application is explained. The implementations of the utilities that differentiate this application from others are highlighted in the second part.

The objective of the DEM-Application is to have a base program for the DEM coded in a very powerful and versatile platform, KRATOS. This permits different researchers extending and improving the code as well as using as a closed package for projects and simulation by advanced users and engineers.

The objective of this document is to guide those users or developers in using the program and understanding the underlying numerical methods implemented as well as introducing them to the theoretical aspects and capacities of the Discrete Element Method when dealing with the continuum modelling problems.

Part I: The Discrete Element Method

1. OVERVIEW OF THE METHOD

1.1. Brief history of the Discrete Element Method

Peter A. Cundall [1] developed a general method to apply in rock mechanics and referenced it as Discrete Element Method for the first time to the scientific literature in 1971. The method, frequently called Distinct Element Method has its theoretical basis on the method that Sir Isaac Newton established in 1697. Its first applications in engineering problems were in geomechanics years later in 1990 described in the book Numerical Modelling in Rock Mechanics, by Pande, G., Beer, G. and Williams, J.R. [2].

Since that point the method has been rapidly spread; an important impulse for the method was the 1st, 2nd and 3rd *International Conferences on Discrete Element Methods,* which have been a common point for researchers to publish advances in the method and its applications. Journal articles reviewing the state of the art have been published by Williams, Bicanic, and Bobetet al. Regarding the DEM-FEM combined method, a comprehensive treatment is contained in the book The Combined Finite-Discrete Element Method by Munjiza [3].

1.2. Introduction and general aspects of the DEM formulation

The Discrete Element Method was firstly introduced by Cundall (1971)) [1] for the analysis of the fracture mechanics problems and, afterwards, it had been applied to solids by Cundall and Strack (1979) [4]. From that time to now, the method has evolved so much and has acquired new perspectives that bring engineers the possibility to study a large type of problems. Some of these new insights will be commented in next sections but the objective of this first introduction is to set the bases of the original method.

The crucial difference between a DEM models and the FEM is that the material is represented by a discontinuous particle structure without any need of a mesh in the strict sense. The infinite number of material points of the continuum is replaced by a finite number of particles of finite extent that interact through collisions with each other.

The DEM shows several apparent similarities compared to the "classical" mesh free methods. However, the key difference between the DEM, on the one side, comparing with traditional mesh-aligned methods (FEM, BEM, FDM) and also comparing with meshless methods (SPH, MLS, PIC, etc.)[1] on the other side, is that the first one describes a discontinuous media, while the last groups describe a continuous media. This means that the methods which describe a continuum are all based on a formal discretization, while methods like the DEM are based on a physical discretization; i.e. elements of the DEM represent physical objects.

The method simulates the mechanical behaviour of a system formed by a set of particles arbitrarily disposed. This method, in its original conception, considers the particles to be discrete elements forming part of a higher more complex system. Each distinct element has an independent movement; they interact each other due to the contacts.
Basically, the Discrete Element Method algorithm, from a computational point of view, is based on three basic steps:

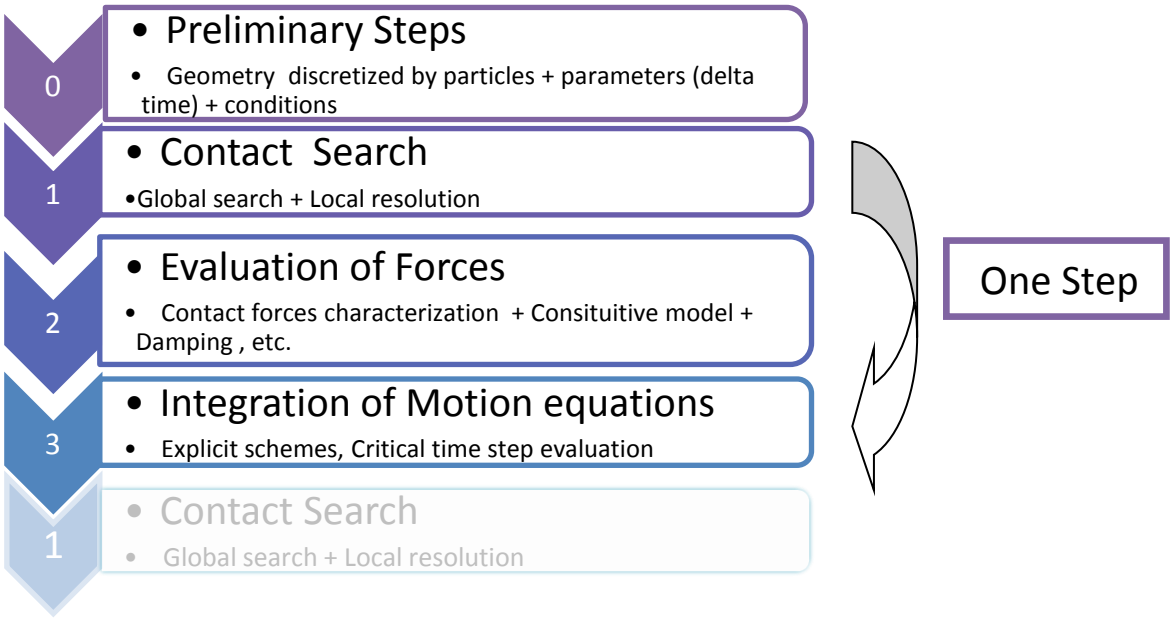


Figure I. 1 Basic computational scheme for the DEM.

[1] *Mesh-Aligned Methods: FEM: Finite Element Method, BEM: Boundary Element Method, FDM: Finite Difference Method.*
Meshless Methods: DEM: Discrete Element Method, SPH: Smooth Particle Hydrodynamics, MLS: Moving Least Squares, PIC: Particle In Cell

1.2.1.Preliminary steps:

The geometry of the problem is formed by different subsets of discrete particles; normally a determined volume is discretized by hundreds, thousands, millions of discrete elements filling the space. This discretization would simulate the distribution of the physical objects that form the domain; the geometry in some classical applications represent bulk materials in silos or tanks, containers and transport of agro-industry good, chemistry and pharmaceutical applications, soil, rock, for landslide, excavation, transport, mining.

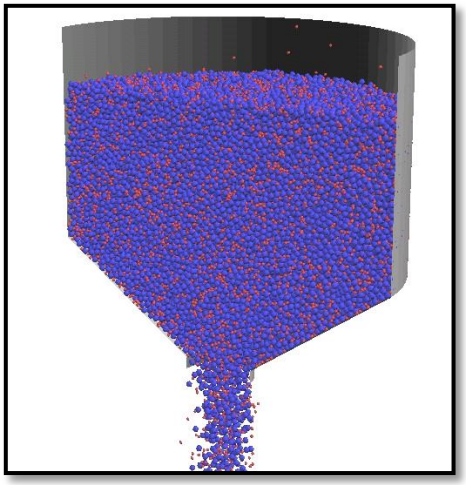


Figure I. 3 Flow of particles in a hopper

Figure I. 2 Particles on a conveyor belt

Once the geometry is represented by discrete elements the characterization of the parameters takes part. Many relations have been established to set the values on the model and all of them require the information from the geometry and the macroscopic properties of the material such as the Young modulus and the Poisson ratio.

The time step is here selected and it has to take in account several aspects:

- Time of the simulation: for long simulations, longer time steps are needed.
- For accurate detailed solutions with little indentations, a very small time step is required.
- The range of time steps available is limited by the chosen integration scheme. In explicit methods, which are the most used ones, the critical time step is a limitation on the time step selection.

The conditions applicable to the system of particles are simply constrains on some degrees of freedom for the movement of particles as well as forces applied or velocities imposed to them.

1.2.2.Contact Search

In contrast with the Finite Element Method-based methods, the DEM, has no connectivities between the nodes by means of the elements. So the transfer of properties such as forces, and in consequence, accelerations, velocities and displacement are possible for the contact criteria between the pair of elements involved. The contact is determined, in the most classical way, when one body belonging to a discrete element intersects with another body that defines another discrete element. The contact search is a very important part of the method in terms of computational cost (range 60%-90% of simulation time) and it is possibly the most difficult part to treat when dealing with particles that have no spherical/circular shape.

1.2.3.Evaluation of Forces

This method applies the solid-rigid mechanics in the particle level and, in principle, the discrete elements are considered to be rigid, non deformable elements. The constitutive model or behaviour of the material is established in the contact areas between particles.
Rheologically it can be described with a set of springs, dashpots and frictional elements. The characterization of the parameters defining these devices is a fundamental issue. There has been a lot of discussion and research trying to determine the correct values for these parameters and there is not a unique universal solution. Contrarily, there are good approaches that parameterize the contacts for specific cases that can differ from the simulations of discrete granular media, interaction between tool and rock in excavation or continuum simulating problems.

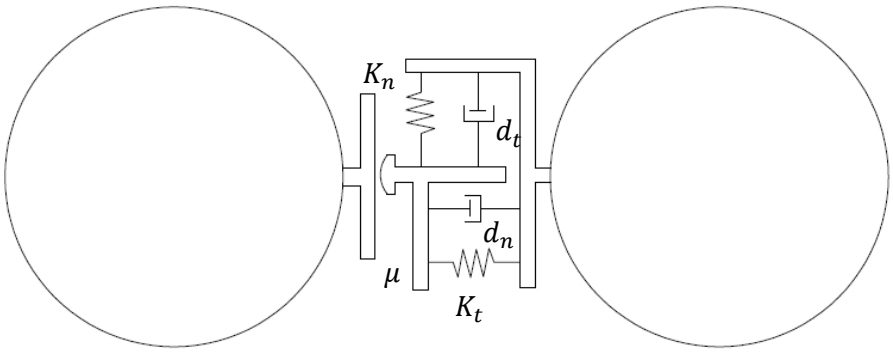


Figure I. 4 Rheological model for the contact

There are more complicated schemes than the presented on *Figure I. 4* and there are also simpler ones that would depend on what kind of problem is being analysed.

1.2.4. Integration of Motion Equations

The only differential equation that needs to be integrated is the second-order time derivative of the position due to the basic Newton's-Law. The translational and rotational motion of rigid spherical or cylindrical particles is described by means of Newton-Euler equations of rigid body dynamics. For every element:

$$\begin{cases} m \cdot \ddot{u} = F \\ I \cdot \dot{\omega} = M \end{cases}$$

Eq. 1 Equations of motion, translation and rotation

Where u is the displacement of the particle centre in a fixed (inertial) coordinate frame X, ω the angular velocity, m the element (particle) mass, I the moment of inertia, F the resultant force, and M the resultant moment about the central axes.

Vectors F and M are sums of all the forces and moments applied to each element:

$$F = \sum_{c=1}^{N_c} F^c + F^{ext} + F^{damp}$$

$$M = \sum_{c=1}^{N_c} (r^c \times F^c + q^c) + M^{ext} + M^{damp}$$

Eq. 2 Expression for the applied Force and Moments

Where F^{ext} and M^{ext} are external applied forces and moments while F^c, M^c are the resultant forces from the interaction with the neighbouring spheres and other entities; finally F^{damp} and M^{damp} are the forces and moments resulting from damping in the system. r^c is the vector connecting the centre of the particles of the target element with the contact point. N_c is the number of particles being in contact and q^c are the torques due to rolling or torsion (not related with tangential forces).

The presented equation for the rotational motion is only valid for spheres and cylinders (in 2D) and is simplified with respect to the general form for an arbitrary rigid body with the rotational inertial properties represented by the second order tensor.

2. DEM THEORY DISCUSSION

Many aspects that have been introduced previously in the introduction section are now extended. In this section also, some specific features are exposed here as part of the "issues in discussion" that are interesting for this work. Furthermore, these issues are relevant of the interest of CIMNE researchers during the study of the method and for the implementation in D.E.M-Application.

2.1. Contact Detection

This represents one of the key issues of the algorithm. Depending on the number of objects involved and the complexity of the shape, it can represent from 60 % up to 80-90 % of the total simulation time. This means basically that the approach must be very efficient and also the method must be adequate specifically to the case; it will not be interesting to execute a complex algorithm that could cover any shape contact detection if our problem contains only spherical particles.

The contact detection basically consists in determining, for every target object, which other objects are in contact with it, and then, judge for the correspondent interaction. Normally, objects move freely and the contact is determined when an overlap occurs, and so, then they must interact. It is usually desired to have a very low overlapping 0.1% ~1% (this is discussed on *2.2.4 Indentation permitted*) to have realistic results, but of course, it depends on the time step selection and the dynamism (velocity) of the particle/objects.

Well, it has already been said that the contact detection is a very expensive part of the algorithm, therefore it's logical to limit the search of neighbours/contacts only when it is necessary[1]. Obviously there is no need to update the contacts at every time step of the calculation (if the time step is considerably small, the neighbours will be the same from several time step calculations) but, if delaying too much the search, it can happen to suddenly find large indentation on a new contacting pair; so the repulsion forces would be too big, therefore there would be a huge amount of "created" energy that would lead some problems as It will be explained in section *2.2.5 Gain of energy*. This can be solved by using the so-called buffer zone, explained in this section.

[1] *DEM-App: in the application we have introduced the possibility to choose the number of time steps between the every contact detection search.*

The different contact detection can be divided in two basic stages:

- **Global Contact Search:** It consists on locating the list of potential contact objects for each given target body. There are two different basic schemes: the Grid/Cell based algorithms or the Tree based ones. There are numerous different methods and variations for each type:

  - ♣ Grid based algorithm: A general rectangular grid is defined in the entire domain, a unified bounding box or sphere is adopted to represent the discrete elements; the potential contacts are determined by selecting the surrounding cells where each target body is centred on.

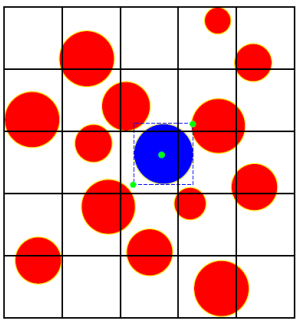


Figure I. 5 Grid/Cell structure

  - ♣ Tree based algorithm: each element is represented by a point. Starting from a centred one, it splits the domain in two sub domains obtaining points that have larger X coordinate in one sub domain and points with smaller values of the X in the other sub domain. The method proceeds for next points alternating every time the splitting dimension and obtaining a tree structure like the one shown in the *Figure I. 6*. Once the tree is constructed, for every particle, the nearest neighbours have to be determined following the tree in upwind direction.

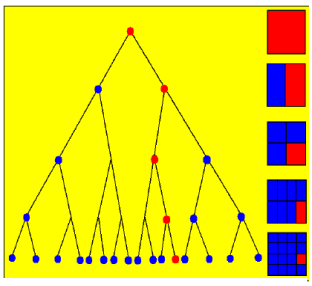


Figure I. 6 Tree structure

- **Local Resolution Check:** The objective is to establish the actual contact configuration. Starting from the potential contacts or areas found in the global contact search, now the contact is analysed in detail. This is the difficult and expensive part of the contact detection; even for simple polygonal shapes the detection criteria is not trivial. The complexity is much higher for 3D cases, which are the most frequent ones. [1]

[1] *In the DEM-Application the local resolution check is not very expensive since it considers only spheres contacts. The team is currently developing basic regular shape contacts; this is introduced on* 2.1.8 Local contact resolutions

2.1.1. Buffer Zone

In order to solve the problem of permitting the neighbours updating every predefined set of time steps and, also assuring that there would not be large indentations, the concept of buffer zone has to be introduced. The buffer zone is a safety zone around the objects used to check whether there are other future neighbours or not. When a simple search is performed for contacting elements, the criteria is usually, as already said, the indentation (overlap) between the target particle and the possible neighbour. Using the concept of the buffer zone, the area around the target element will be enlarged and it will be checked if there are any other elements, which are not currently in contact with the target now, but are situated inside the buffer zone; therefore it can be considered as a potential future neighbour in a very near time step.

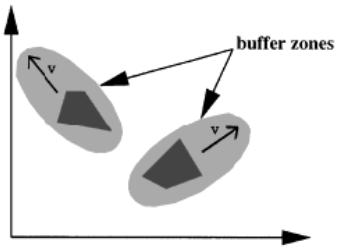


Figure I. 7 particles with Buffer Zone

These buffer zones can be used to change the frequency of neighbour search, i.e. if the search is defined every 10 calculation time but during one of these time steps a possible future neighbour is detected close, the search can be renewed earlier than this predefined 10 time steps in order to capture well the moment of contact.

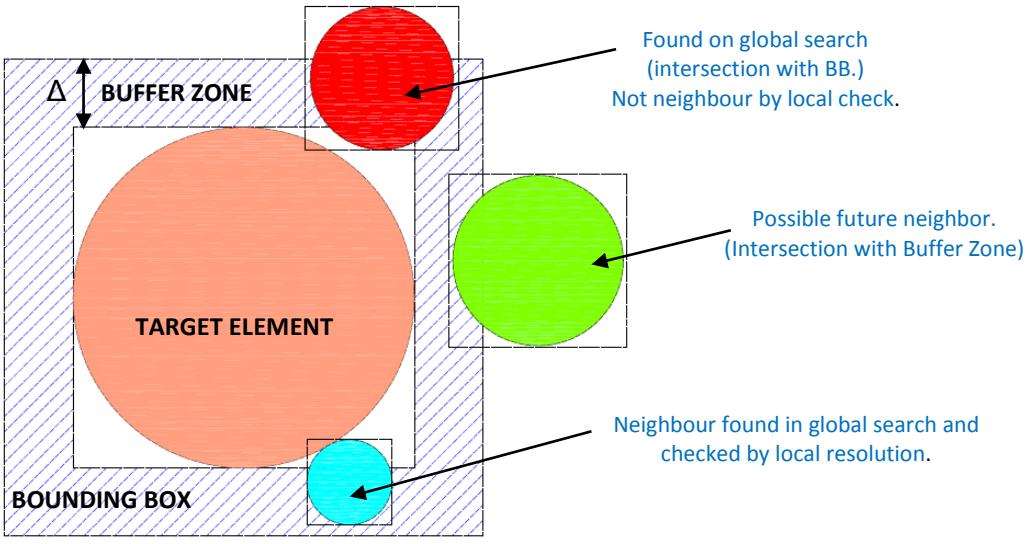


Figure I. 8 Buffer zone example for neighbouring search

There are many criteria to determine the size of the buffer zone. It has to be a function of the time step, the number of time steps between each search and the velocity of the particles.

$$\Delta = f(\Delta t, \mathrm{N}_{\mathrm{ts-s}}, V_t, V_p)$$

As many aspects of the Discrete Element Method, there are almost no parameters that can be determined globally and be universal for all cases; contrarily, they depend on the characteristics of the problem. For a dynamic discrete system the following expression is proposed:

$$\Delta = N_{ts-s} \cdot \Delta t \cdot (V_t + V_{max})$$

Where: N_{ts-s} = Number of time steps between each search.

Δt = Calculation time step.

V_t = Velocity of the target particle (norm of the velocity).

V_{max} = Max. velocity of the particles in the system (norm of the velocity).

Δ = Extension of the bounding box, defining the buffer zone (space units).

This is a very simple expression that assures that there will never be a neighbour able to pass through the buffer zone without being detected. When the neighbour is detected inside the bugger size, the time step can be reduced in order to capture the possible contact. Nevertheless, this proposed expression is too conservative and there are other more efficient expressions.

2.1.2. Bounding Box/Sphere representation

It consists on, as a first approach for the global potential neighbours, inscribing our discrete elements into a box/sphere in 2D or 3D. This coarser representation of the particle will determine the area where potential neighbours can be found.

There are many different schemes to represent an arbitrarily shaped object by a simple bounding geometry entity (volume). In video games there are a lot more sophisticated ones but the OBB hierarchies are used frequently.

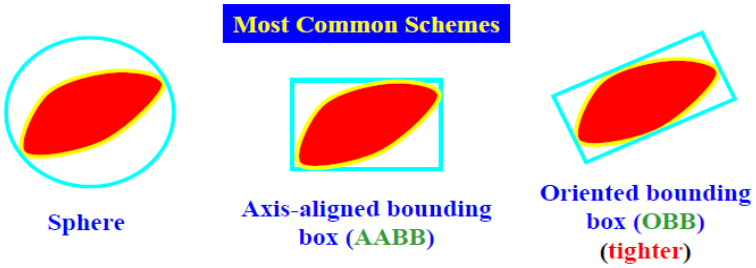


Figure I. 9 Most common types of bounding box representations

This proceeding is used when applying a tree-based or a grid-based search method; the algorithm searches contacts between these simplified coarse representations of the particles which is much easier because the spheres or boxes substitute the complex geometries.

Once the potential neighbour particles are determined by the selected method, a more sophisticated criterion is applied for the neighbouring detection of the actual geometry of the elements that were detected to be possibly in contact.

The buffer zone is applied to enlarge the size of the bounding box or sphere around the

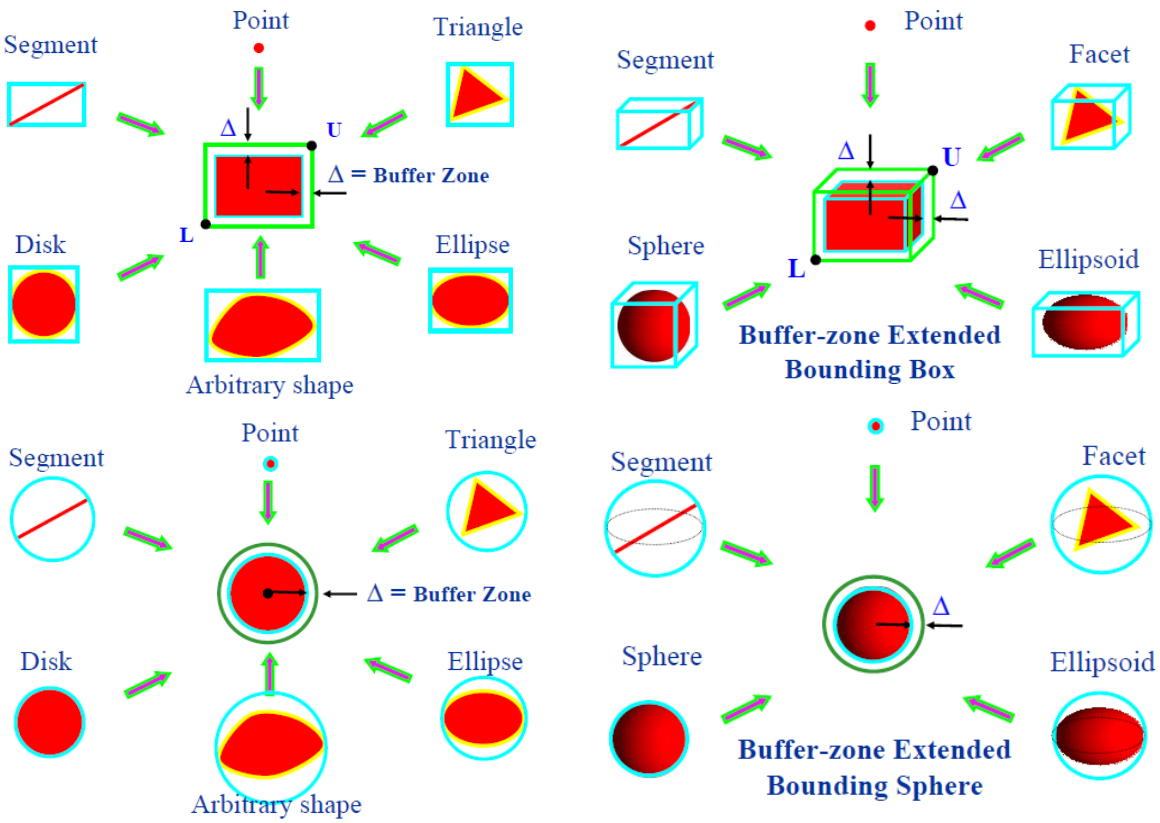


Figure I. 10 Bounding Box/Sphere and buffer zone

More detailed information of these issues can be found on a paper from Walizer, L.E. and J.F. Peters. [5].

2.1.3.Brute Force Search Method

The first method we are going to analyse is the simplest one, the brute search. It will be a reference to compare the reduction of computational cost versus the complexity of the other methods. The name is due to its simple and rudimentary approach. To find which ones are the contacting elements to a given target, the method calculates the minimum distance between each pair, and judges whether there is contact or not.

For every element, the method does a loop for all other elements checking for the contact. Regardless of the complexity of the judgement, the order of the number of operations needed is quadratic: $\mathcal{O}(N^2)$.

2.1.4.Static Cell Search (grid-based method)

The domain is discretized with rectangular cells (2D) or hexahedral (3D) of the same size. The elements are associated to the cells depending on the coordinates. The efficiency of the method depends on the balance between the size of the cells and the number of elements in each cell. This homogeneous the distribution of elements is, the more efficient the method is. There are some variants existing to this method with adaptive meshes for the case of non uniform element distribution.

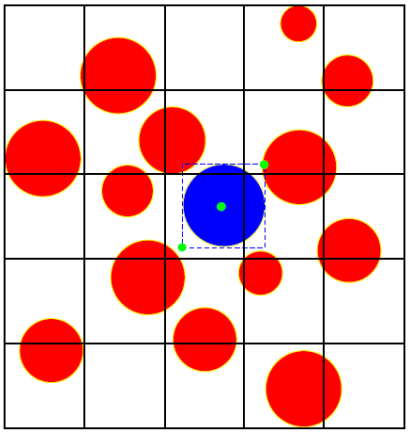


Figure I. 11 Static Cell construction example

The summarized steps of the method are:

- Mapping stage:
  - Find the maximum space occupied by the objects.
  - Determine the maximum size of the objects.
  - Divide the space into cells of the maximum size.
  - Map each object to a cell based on the position of its lower, upper corner or central point.
- Search stage:
  - 1. Check the overlap between the objects within each cell.
  - 2. Check overlap of the objects in the cell with those in the neighbouring cells (8 in 2D).
  - Using the contact symmetry, the number of neighbouring cells to be checked can be halved.
- Features (pros and cons):
  - It is a very simple method.
  - Effective for small, compact problems.
  - The computer costs are usually of the order $\mathcal{O}(N \cdot \log(N))$.
  - Very expensive for large simulations where the spatial distribution of objects is sparse and irregular (i.e. large number of empty cells).
  - The cell size must be no smaller than the maximum size of objects.

2.1.5.Dynamic Cell Search (grid-based method)[1]

This method is also a grid-based method but the approach is quite more sophisticated. The 2D domain case is explained next, it can be extended to 3 dimensions easily.

First of all the domain is divided in rows (or columns) and for each row several cells are created. Each cell is assigned the elements where its boundary box bottom-left corner is situated on.

The proceeding starts now from the first bottom row, going cell after cell checking the contacts between the elements assigned to each cell.

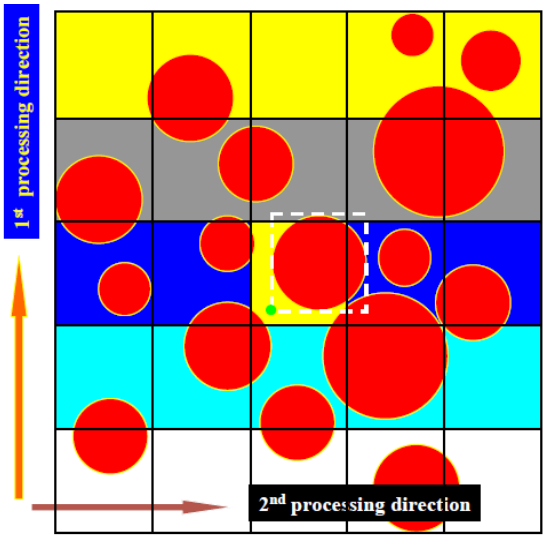


Figure I. 12 Dynamic Cell Search overview

After a cell is completed, there is a check of element migration corresponding to the cells. The same way after a whole row is completed there is the correspondent check for elements that migrate to the next row.

- Types of migration checks:
  - **Row migration:** Elements are migrated to the next row if their upper y-coordinate is greater than the lower y-coordinate of the next row. This is done only if the next row is non-empty.
  - **Cell migration:** Elements are migrated to the next cell if their upper x-coordinate is greater than the lower x-coordinate of the next cell. This is done only if the next cell is non-empty.

[1] *This is the method that currently we use in DEM-Application. It is implemented for the disc/spheres neighboring search. We have developed some extra features like the radius extension and the tolerance in the search that is explained in PART II section* 8.5.7 Neighbour Search utility and Extended Radius Search.

- **Types of elements in each cell:** for a generic cell during the calculation, the situation is presented in the following figure, where 4 types of elements can be differentiated.

  - **Type 0-0:** Element 6; it's an element original from this row and from this current cell. It will be migrated to the next cell and also to the next row.

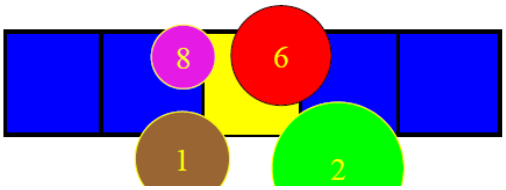


Figure I. 13 Dynamic Cell Search example

  - **Type 0-1:** Element 8; this element is original from this row but is migrated from the previous cell.
  - **Type 1-0:** Element number 2; an element migrated from the previous row but new for the cells of this row.
  - **Type 1-1:** element number 1; it's an element migrated from the previous row and also from previous cell.

- **Local Checks:** the necessary checks for contact proceeding cell by cell are:

  - Determine contacts between the elements among the type 00.
  - Determine contacts of the elements of type 00 against type 01.
  - Elements type 00 against type 11.
  - Elements type 01 against type 10.

- **Main features** of the method are:

  - Dynamic processing of the cells
  - Linear complexity. CPU cost can be optimized to $\mathcal{O}$(N).
  - Very effective for large simulations.
  - Not sensitive to the spatial distribution of objects.
  - No performance degradation for objects with wide range of size distribution.
  - Arbitrarily choice of cell size.
  - Readily extendable to any dimension.

The 4th and 5th listed features of the dynamic method represent a poweful advantage for the usage of this method against the static one. However, the choice of the cell size is crucial to obtain an optimal CPU cost.

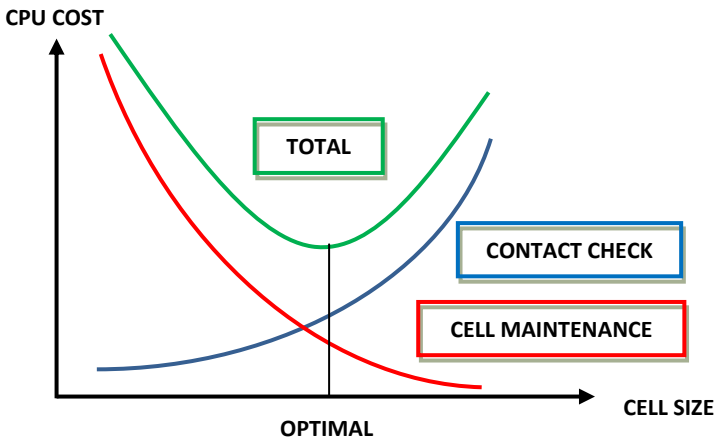


Figure I. 14 CPU cost vs. Cell size on D-Grid methods

Again there is no unique selection of the parameters; the optimal cell size depends on several factors, such as object size distribution, packing density, and some other computer and hardware issues.

Further information on this method can be found on the following reference: Perkins, E. and J.R. Williams [6].

2.1.6.No binary Search Method

The NBS (no binary search) algorithm was proposed by Munjiza in 1998 [7] and it is mostly convenient for problems involving large quantity of bodies with large movements. In optimal conditions, the total detection time is proportional can to the number of particles, $\mathcal{O}$(N). This result is more or less independent from the packing density which affects insignificantly on the memory requirements. The only limitation of the algorithm is its applicability to systems comprising bodies of similar size.

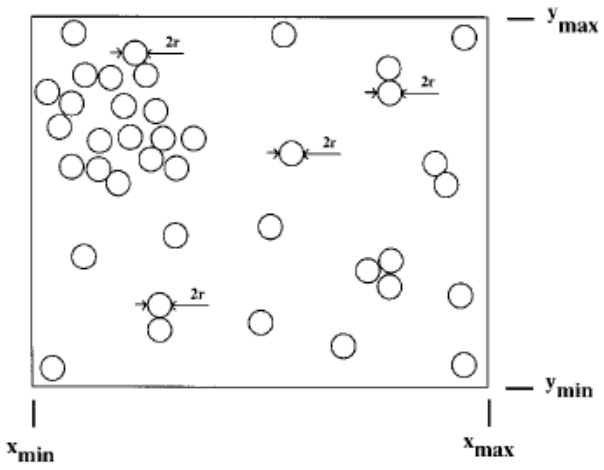


Figure I. 15 Domain and particles representation in NBS

The method starts determining the whole domain and dividing it in regular cells. First of all, every particle will be represented by an evolving sphere or disc which would be the larger contained in the space. The diameter of this representative volume will be the one that de cell size determines; the domain is divided in cells of the same maximum diameter size.

Each particle is mapped to a cell and identified with and integer index for x and y. Then, for every row the linked lists containing all the indexed particles on each row are created. The correspondent loops are effectuated to determine optimally the neighbouring and are explained in the reference [7]. Basically the method proceeds row by row from bottom to top and for every particle labelled (ix, iy) the particles in the following cells have to be checked: (ix,iy), (ix-1, iy), (ix-1, iy-1), (ix, iy-1), (ix +1, iy-1) and (ix+1, iy).

2.1.7.Tree-based algorithms

These are alternative method to the cell-based algorithms. These methods have an average performance of $\mathcal{O}$(N·logN) for the CPU time. The difference, in terms of efficiency, from the cell methods is the strong dependency on the construction of the tree and de order of insertion of the particles to the tree.

K-D-Tree: It consists on inserting one after another the coordinates of the particles into an algorithm which divides the space alternatively in X, Y, Z, etc. It is extendible to any dimension. Every new particle that is inserted is compared in the first stage against the first dimension and it goes to the left if the value is smaller or to the right if its larger, then in the next stage the 2nd dimension is compared and the proceeding is the same; this is being done shifting at each level one dimension and inserting the new particles in new levels that can contain a maximum of two particles.

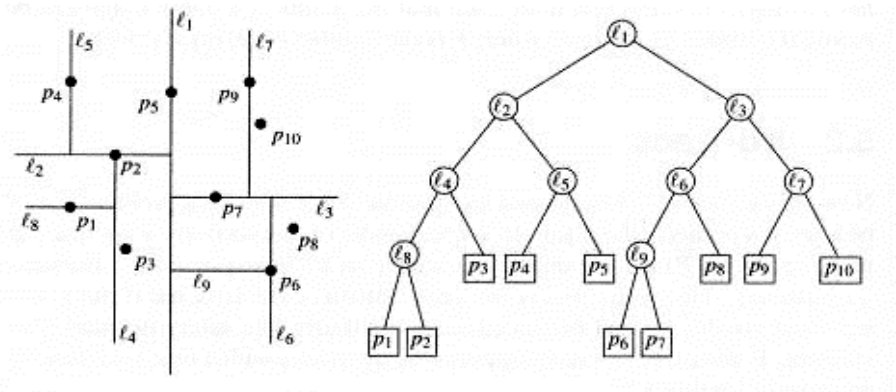


Figure I. 16 K-2 Tree construction

Once the tree is constructed, it is easy to identify which particles are potential neighbours to others. This method is especially sensitive to the particle insertion order, and so the performance is very dependent on it.

Quad-Tree: Working in 2D, the domain is divided in rectangular cells with a maximum number of particles of four. If any resulting cell has more than four particles it is divided again in four cells more. While doing this the domain discretization can be represented in a tree scheme with four branches. This structure allows indentifying easily the objects belonging to the different sub domains tracking downwards onto the. The cost of this search is of the order of $\mathcal{O}(N \cdot \log_4 N)$.

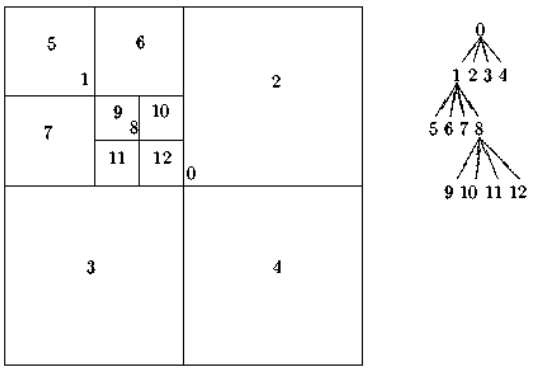


Figure I. 17 Quad-Tree structure

Oct-Tree: The extension of this concept to three dimensions in space it brings an eight element structure. Now, the domain will be divided in cells which will contain a maximum of eight elements. If there is any cell with more than eight elements it will be subdivided in eight inner cells. The search for elements belonging to any sub domain is performed by checking the tree from top to bottom following the branches like the previous cases, this time the cost of the search is about $\mathcal{O}(N \cdot \log_8 N)$. A recommended reference for the oct-tree implementation is Raschdorf [8].

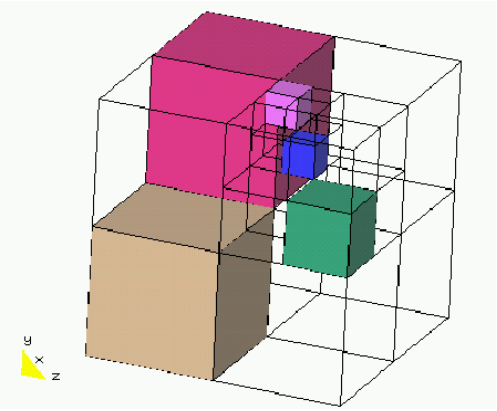


Figure I. 18 Oct-Tree structure

The construction of a new tree structure for each time step would be so expensive. For fine time steps, most of the contacts will be more or less the same than the previous stage. The information of the previous contacting pairs can be very useful to improve the current search. These methods, like the cell-based ones, are used as a first global search stage to determine the potential neighbours and they would need a local resolution check after it.

2.1.8.Local contact resolutions

Given that two discrete objects are potentially in contact, local contact resolution establishes if they are indeed in contact based on their actual geometric shapes.

It is not the objective of this work to enter much in detail for these methods, which can be much more complicated than the global search, especially for irregular shapes. This problem is well-known because it has been studied in many disciplines apart from D.E.M., including video-game programming. Interesting information can be found on [9-10]

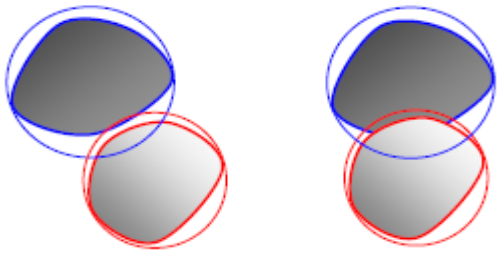


Figure I. 19 Local contact resolution after global search

If the pair is in contact, the normal and tangential contact directions, the contact point and the characteristics of the overlap, such as the penetration, contact width and contact area, may also need to be determined at this stage, depending on the interaction laws to be used. This will be discussed in next section *2.2 Constitutive Modelling of the Contact*

- Contact Directions
- Contact Point/Centre
- Overlap, Curvature,
- Width, Area, etc

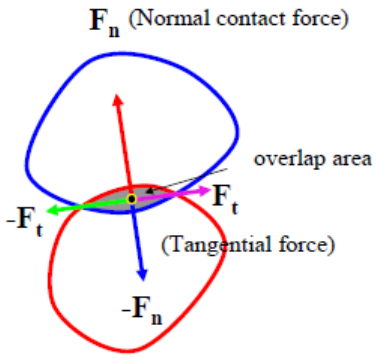


Figure I. 20 Contact directions and area

This is the most time consuming part in contact detection. Every effort should be made to make it computationally as efficient as possible.

The complexity of the detection depends only on the shape of the bodies that are analysed. The basic one is the contact between spheres, which is the easiest, but it can be interesting to use ellipsoids instead or, in a higher level of complexity, polyhedral or even irregular complex shapes can be used.

Spheres/discs

The local resolution check is obviously trivial for spheres where; for every possible pair (detected via global search), the only comparison needed is the distance between their centre coordinates against the sum of the radius. Remember that the particles are supposed to be in contact when an indentation positive or equal to zero is presented. In this case everything is well defined; not only the check is easy but also the normal direction of the contact is simply defined by joining the centres, and therefore the tangential plane is defined; the contact point also well defined the overlap is direct and the contacting are is geometrically easy to determine.

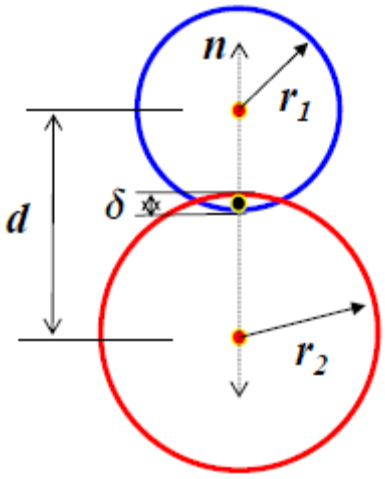


Figure I. 21 Local contact example between discs

Ellipses/Ellipsoids/superquadrics

The general determination would involve solving a two nonlinear (quadratic) system of equations. It normally requires the use of an iterative procedure, such as the Newton-Raphson method which is computationally too expensive to apply in our case. Fortunately, there are some other well defined approximate methods to simplify the contact detection between two ellipses. In general these methods are applicable to other analytically represented non-circular objects.

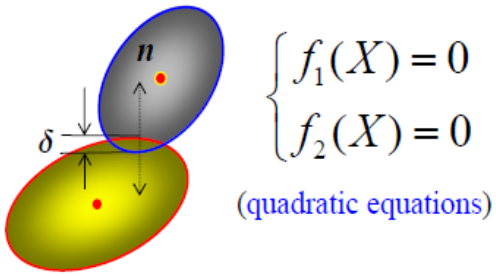


Figure I. 22 Contact between two ellipses

Unfortunately these methods are not extendible to ellipsoid 3D shapes.

It is not the object of this work to enter much in detail about these methods but it is essential to give an insight of how difficult can be entering on these aspects, and moreover, making it efficiently.

In the next figure, just as example, it is presented the ambiguity of determining the centre and the normal direction for the contact between to convex shapes. For a detailed description of the methods refer to Feng, Y. lecture notes [11]

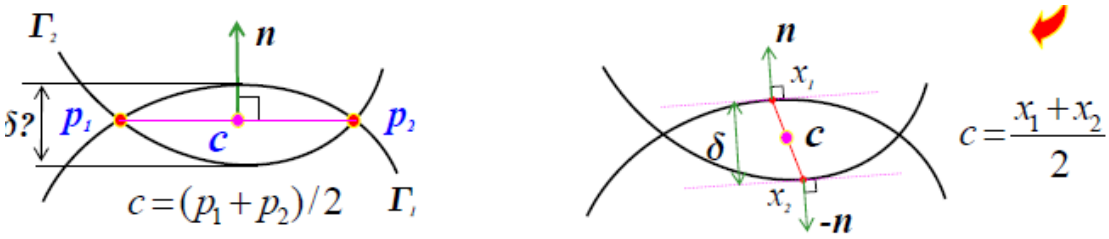


Figure I. 23 Overview of the method for the ellipses

In a similar sense there are methods developed for ellipsoids and superquadric objects which may interest the reader; to learn about that see Owen, [12].

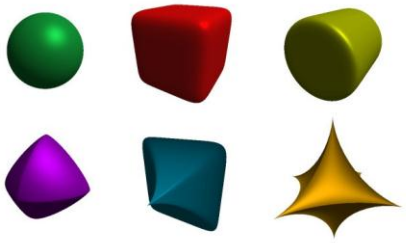


Figure I. 24 Superquadric 3D shapes

Polygon/Polyhedron

These cases are very complicated compared to the sphere contact. The basic questions that have to be solved now are:

- 1. Are the bodies in contact?
- 2. How to define the characteristic parameters?
  - ♣ What is the appropriate definition of the overlap?
  - ♣ How to determine the normal contact direction?
- 3. What interaction laws should be associated with?

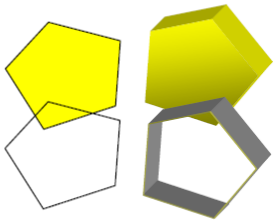


Figure I. 25 Polygon/Polyhedra contact

Local check

There are algorithms than can determine the overlap region between two polygons or between polyhedron, with a linear complexity $\mathcal{O}$(N) (where N is the number of vertices/edges).

These types of contacts are also a common operation in computer games/graphics. Significant research has been conducted and highly efficient public codes are available, e.g. *V-Clip, I-COLLIDE,* in which the temporal coherence (see page 26) plays a key part.

Characterization of the parameters

In 2D there are only three types of contacts: plane to plane (which has no problem with the definition), plane to corner and corner to corner. In 3D this gets quite more involved.

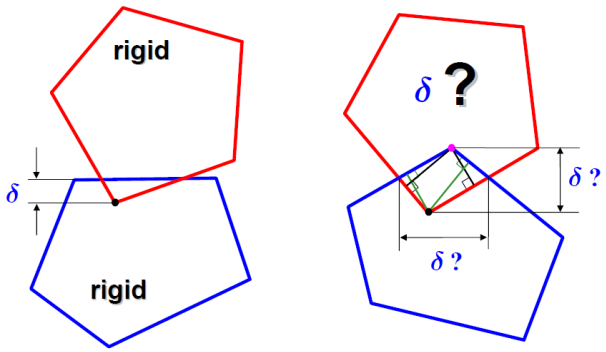


Figure I. 26 Corner to plane and corner to corner contact in polygons

In corner to corner contact, it is also not clear how to properly define the overlap (penetration). The same problem also arises in FEM.

Some methods have been developed to characterize the correct normal directions and the overlap.

- **Rounding the corners:** By rounding the corners with circular arcs, the difficulties associated with the corner/corner contact may be (partially) overcome. However, different circular radius can lead to different results. No method is available to guide the selection of a proper radius. Thus, this treatment is also an artificial numerical procedure.

- **Contact energy based algorithm:** A functional of energy is defined in terms of the overlap area between the two bodies, the normal direction is the variation of the area with respect to the position and the resulting force is proportional to the derivative of the energy with respect to the position. Depending on how the energy is defined in terms of the area we have different models for quantifying the force. See reference Feng, lecture notes [11].These methods have been generalized to 3D polyhedra by the same author.

Methods for irregular shapes

The objective of many studies has been to determine a global algorithm for the contact detection between any shapes; in this section some of the advanced techniques that nowadays are being under research are commented. Particularly a couple of methods that are being studied by a collaborator in KRATOS DEM-Application[1]. The two methods that the KRATOS team are currently working on for future features of the program are the *Discretized finite elements contact* and the *Semi-Spring/Semi-Edge method*

The so-called *discretized finite element contact* method simply consists on a simple approach that would make the contact detection much easier when dealing with the DEM elements – FEM elements contacting problem (see section 4*. DEM-FEM)*. The method is an idea proposed by C. Feng and M. Santasusana to solve the contact detection and characterization with a coupled DEM-FEM problem; the basic idea of the method relies on using the discretized mesh instead of the geometry to define the entities which are susceptible to the contact detection. The idea is that if the entire continuum where the elements apply is discretized with triangles/tetrahedra or quadrilaterals/hexahedra, these complex discrete bodies/particles could also be discretized with the same type of mesh (except the discs/spheres, which don't need to be discretized).
By doing so, the possible contacts would always be triangle against triangle or quadrilateral against quadrilateral, even in 3D. This is because, when analysing the local contact, only the surfaces of the neighbouring bodies have to be taken into account.

[1] *PhD candidate C. Feng, which is a researcher on IMECH, CAS (CHINA), who during the spring 2012 has been a collaborator in KRATOS DEM-Application*

As a first stage, a method that evaluates contacts only between triangles could be coded and then, for the tetrahedra discretization in 3D or triangular meshes in 2D, all the contacts could be checked easily. A next step would be doing the same with quadrilateral, also including the easy detection against spheres and finally mixing up these commonly used geometries in most of the FEM and DEM simulations respectively.

For instance, the contact between hexahedra should involve the different six possible situations:

- Face against Face:
- Edge against Face:
- Vertex against Face:
- Edge against Edge:
- Edge against Vertex:
- Vertex against Vertex:

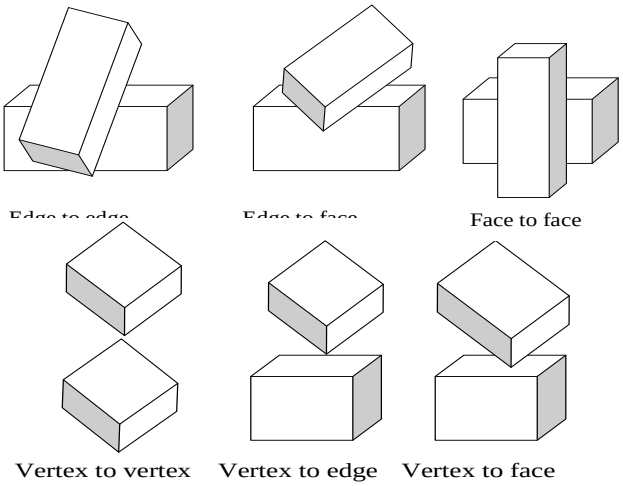


Figure I. 27 Types of contact between hexahedra

The above mentioned *Semi-Spring/Semi-Edge method* is a work original from *Feng Chun* who defines an inner contact control points to simplify the contact detection to only 2 possibilities: contact between semi-springs and contact between semi edges.

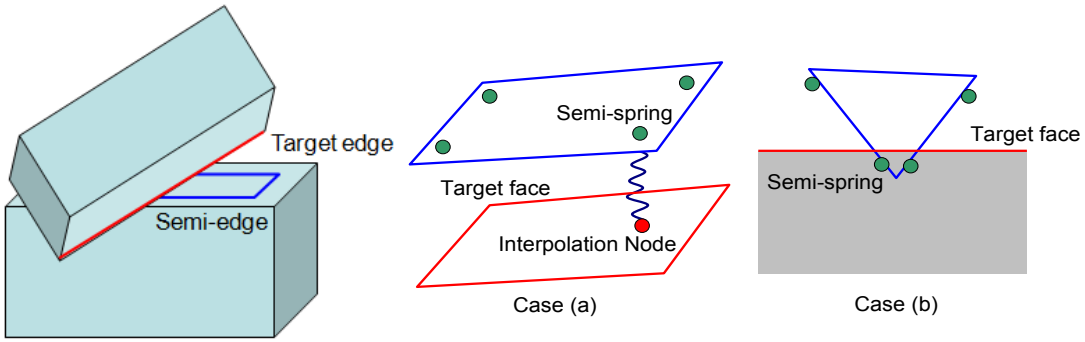


Figure I. 28 Semi-spring/Semi-edge method overview.

Some other conventional methods related with penetrating edges can be found here for tridimensional blocks, Yung-ming, C et altri [13].

Clusters of spheres

Another technique that has been used recurrently is the representation of complex shapes[1] by the assemblage of large number of spheres. This method is permits easy implementation because the contacting zones can be considered independent spheres, whose the contact laws of those are well-known.

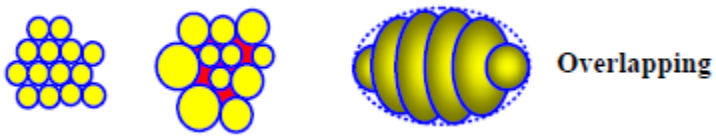


Figure I. 29 Clusters of particles

This is an alternative not only for irregular shapes but also for simple ellipsoids and polyhedra which, with a large number of little spheres surrounding their boundary, can be accurately well defined.

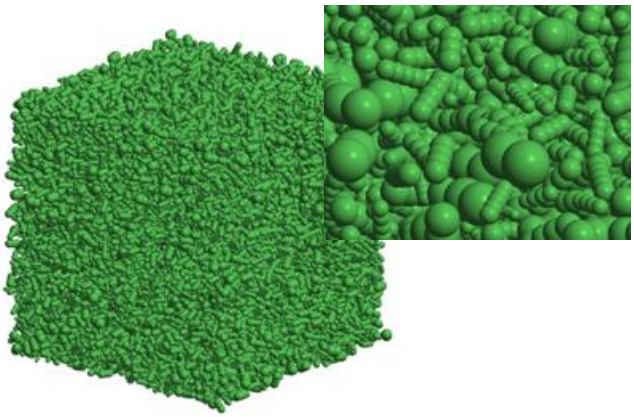


Figure I. 30 Cluster generated in GiD by ULCV CIMNE

For pharmaceutical application it may be of interest this reference that treats about the tablet shape contact, from Song, Y et altri [14]

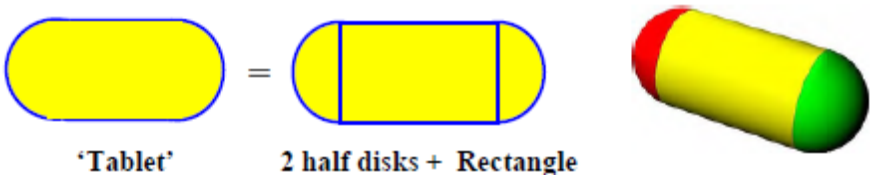


Figure I. 31 Tablet shape particles

[1] *The UCLV CIMNE classroom (CUBA) is currently working on this type of irregular particle representations for bio-medical applications. See section* 6.2 Current development and collaboration *on Part II of this work.*

Temporal coherence

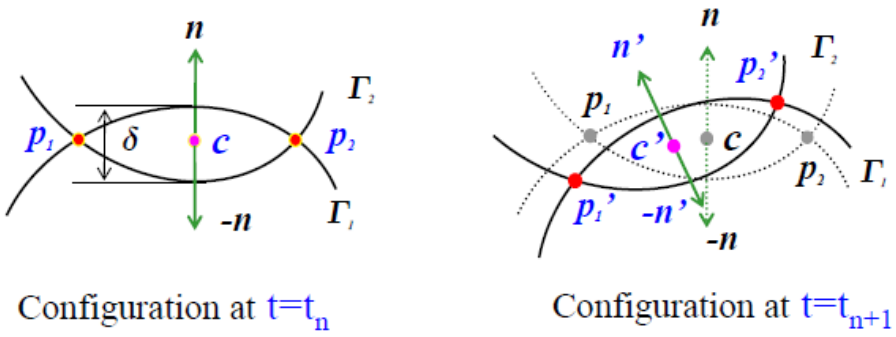


Figure I. 32 Contact between convex surfaces at different time steps

Due to the use of explicit time integrations for dynamic analysis (see section *1.2.4 Integration of Motion Equations*) where the time steps are generally very small (~10^{-6} sec.) the difference between the two contact configurations at two consecutive time instants are normally very small. Thus, some contact characteristics at the current time step can be used as a very good initial guess for the next time step to significantly increase the solution convergence of the contact resolution. This is called *temporal coherence*, and should be exploited wherever possible.

2.2. Constitutive Modelling of the Contact

This section will focus basically on the contact between spheres, which is the easiest and the most used particle in the D.E.M. The characterization of the parameters that rule the physics of the contact are a fundamental issue in this method. Depending on the type of simulation, continuum simulating or discrete dynamic systems, the parameters and the models itself have to take in account different nuances.

In essence, the contact can be described rheologically by a set of simple devices like a spring, a dashpot and frictional or cohesive devices. However, complicated systems can be obtained combining these different devices.

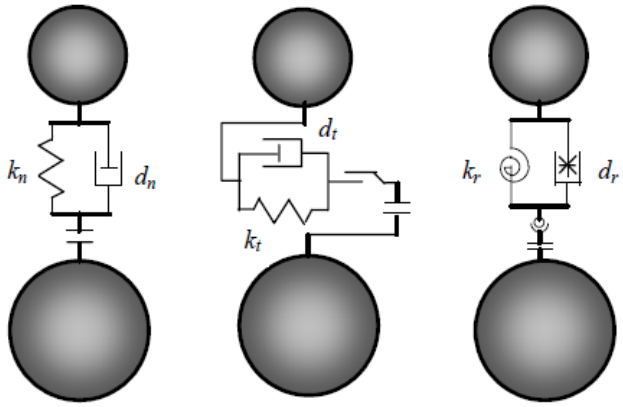


Figure I. 33 Rheological models for the contact between two spheres

These parameters that rule the forces and stresses within a contact are often called *Micro parameters* of the model and have to be defined by the method:

- **The stiffness parameters:** K_n (K_{nc}, K_{nt}), K_t, K_r that relate the forces and moments with the displacements and rotations.
- **Strength parameters:** $\mathcal{F}_n$ $(\mathcal{F}_{nc}, \mathcal{F}_{nt})$, $\mathcal{F}_s$, $\mathcal{M}_r$ that are related with the stress limits and determine the strength values for the normal compressive, tensile, shear and moment stresses.
- **Friction coefficient:** μ, normally Coulomb's friction model.
- **Damping coefficients:** d_n, d_t, d_r for the translational and rotational motion.

There are two philosophies regarding the characterization of these parameters, a global approach for every contact and a locally description that depends on each contact. Next, the derivation of the values for these parameters is presented in a local way is presented; also extra considerations or simplifications are discussed.

2.2.1. Normal interaction forces

The simplest D.E.M. methods must contain at least one device: the normal spring; with just this simple model a dynamic system of frictionless particles interacting can be simulated. Therefore, this is the starting point for any D.E.M simulation. Complementing this, the dashpots, frictional devices, shear springs, etc. can be introduced to represent more complex and realistic contact situations.

In general, the normal interaction force can be described:

$$F(\delta) = -K(\delta) \cdot \delta^{\alpha}$$ Eq. 3 General contact force

For the classical linear spring (Hook's law):

$$F(\delta) = -K \cdot \delta \quad where\ K > 0$$ Eq. 4 Hook's law

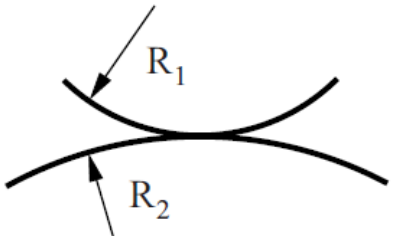


Figure I. 34 Contact between two spheres.

From the Hertzian theory, for compressive contact between two spheres:

$$F(\delta) = -K \cdot \delta^{\frac{3}{2}}$$ Eq. 5 Hertzian general contact

$$where\ K = \frac{4}{3} \cdot \hat{E}^* \cdot \sqrt{R^*},\ \hat{E} = \frac{E}{1-\upsilon^2}$$ Eq. 6 Parameters for two spheres contact (Hertz)

$$\hat{E}^* = \left(\frac{1}{\hat{E}_1} + \frac{1}{\hat{E}_2}\right)^{-1} \quad R^* = \left(\frac{1}{R_1} + \frac{1}{R_2}\right)^{-1}$$ Eq. 7 Parameters for different radius and materials (Hertz)

Here, the generic case for the Hertzian contact between two spheres is presented. Note that the case with one sphere and a plane is also contemplated, i.e. $R_2 \to \infty$.

There are many other expressions similar to this one to evaluate the behaviour in normal direction depending on the desired case. When dealing with continuum simulations some other expression can be used[1]. See section *3.3 The effective contacting volume method* for more details.

[1] *In DEM-App. we can change easily from one law to other and we can apply an incremental method or absolute method. When dealing with continuum simulations we use the expression deduced in the section* 3.3 The effective contacting volume method.

2.2.2. Absolute position method and incremental method

There are two ways of calculating the contact force that each time one body applies to the other due their indentation: the absolute method or the incremental method.

The first case is easier to implement; basically once the force-displacement relation is defined, for both tension and compression cases (may differ from each other), the force can be obtained in any time just by knowing the contact distance in such time. This method simply requires an evaluation of the force for a given indentation for each time, independently of the type of function and the dependency with the indentation.

In the incremental case, the position of every particle at every time is not needed for the calculation. The new normal force is obtained just adding up to the previous force, the current force contribution due to the current incremental displacement at every time step.

For the classical linear spring:

$$F(\delta)_{t+\Delta t} = F(\delta)_t + K \cdot \Delta\delta_{t+\Delta t} \qquad \text{Eq. 8 Algorithm for linear expression}$$

It is easy to see that it will only work if the relation between the force and the displacement is linear. Nonetheless, for non-linear expressions, a linearization of the function can be done in terms of a Taylor expansion:

$$F(\delta)_{t+\Delta t} = F(\delta)_t + F'(\delta)_t \frac{\Delta\delta_t}{1!} + F''(\delta)_t \frac{{\Delta\delta_t}^2}{2!} + F'''(\delta)_t \frac{{\Delta\delta_t}^3}{3!} + \cdots \qquad \text{Eq. 9 Taylor expansion}$$

However, this would complicate things if the derivatives of the function have to be calculated; although numerics can be applied again, it is explained on next section that this operations are usually not worth it.

In contrast with the normal contacts, where the absolute method it's easier and completely accurate, the shear forces (if they are condisderated) have to be treated obligatory with an incremental method. See section *2.2.7 Tangential interaction forces*.

2.2.3. Relative importance of the accuracy on the stiffness value

For anyone who has never coded or dealt with the D.E.M, or simply contact or impact problems, it might sound strange to state that the value for the normal stiffness needn't to be accurate. In fact, good results can be obtained with very different values than the theoretical ones.

Remembering that the normal contact is rheologically represented by a spring, when an impact/contact occurs, it converts the kinematic energy to potential elastic energy and it does the same operation back again. Therefore, regardless of the stiffness value for the spring, the energy would be perfectly conserved.

The only difference of choosing large or small stiffness value is on the deformation observed when a ball gets inside the other (non realistic) and the contact time; rigid springs produce fast impacts instead of soft springs which lead to large indentations and so, larger contacting times. In the figure this is represented by a ball that falls from an initial height, contacts with the fixed one compressing the spring and it is repulsed back again by the elastic force, recovering the initial height.

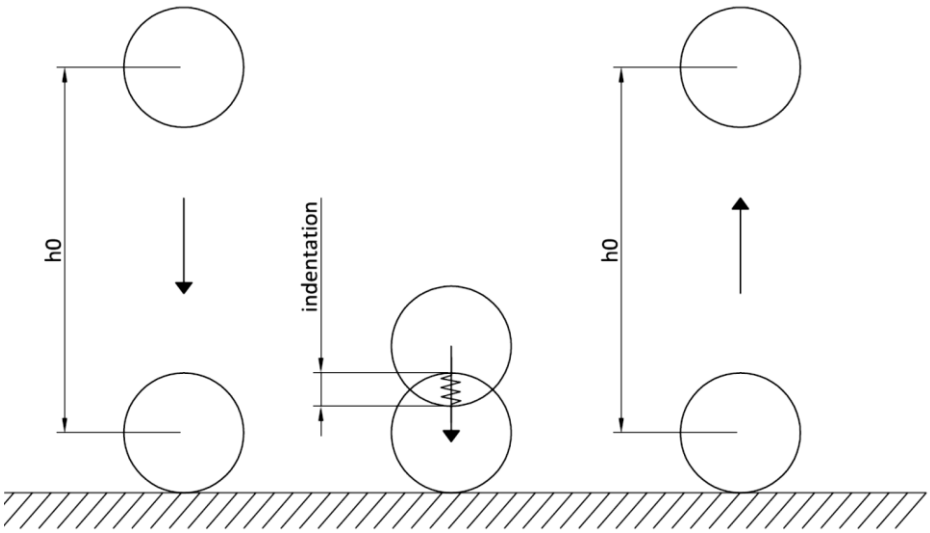


Figure I. 35 Impact between two spheres/discs.

The selection for a normal stiffness value has some other important implications that would be determinant when choosing the correct value: the gaining of numerical energy, the damping, the time step selection, the indentation permitted, etc. Some of these topics are discussed next.

2.2.4.Indentation permitted

Usually when simulating contacts/impacts between discrete media, the spheres represent granular material or steel or concrete, which usually are highly non-deformable materials. A large stiffness value should be introduced in order not to observe large unreal indentations between the spheres, sometimes it can be even bigger than the theoretical one. In realistic simulations, for the indentation between two rigid bodies, the indentation should be in the range of 0.1% to 1%, which implies large stiffness values. This would incur in some "gain of energy" in the system if no further considerations are taken into account.

2.2.5.Gain of energy

The integration methods are discussed on section *2.3 Integration of the motion laws* but it shall be advanced here that the most used methods are the explicit schemes for memory storage reasons. The problem of these schemes is that they "gain" energy during a contact. The reason is because a penalty method is chosen to impose the constrains of impenetrability of the particle. Many explicit methods that can be used like the Fordward-Euler have themselves the problem of energy gaining even in the simple system of a mass suspended on a spring while others can assure no gaining energy for this academic case. However, when we apply the penalty method the problem is different and all of these classical methods would win some energy (that can be more or less negligible).

In a simple forward method example we can show the reasons of this effect: First of all, in general, it is not feasible to capture the exact moment when spheres get in superficial contact. The particles move until some indentation is found by the neighbour search; at that moment, the correspondent spring force is applied with some delay and we do the same for the next steps in a discrete way. So, as it can be seen in the next figure, we are not applying all the correct force and so the kinematic energy is not well dissipated and the particle is able to penetrate more than it should.

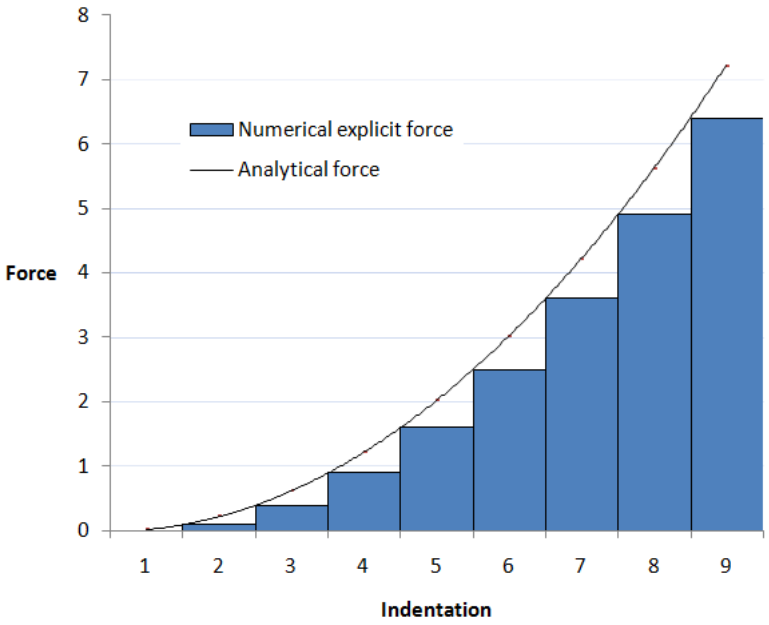


Figure I. 36 Comparison between numerical and analytical force determination

When returning back the delay is also present ant the over rated values of elastic force will act including the last step where the particle will be already outside the contacting entity.

Another effect related with this happens for large (realistic) values of the stiffness. Happens when the selected time step is not sufficiently small .the first time a sphere "enters" inside the other, the first free indentation produces an enormous repulsion force that would have never happened in the analytical case where the sphere couldn't indent so much because the force would have been acting for smaller values of indentation.

For real stiffness in steel or concrete time steps of $\mathcal{O}(10^{-6})$ are needed to avoid this phenomenon, which is sometimes non practical because it makes the simulations too expensive and time consuming. The common way is to use smaller stiffness values letting the particles have a little indentation in order to be able to use larger time steps without gaining so much energy.

2.2.6. Numerical damping and physical damping

The needing of a "non physical" damping in the DEM is due to the numerical error accumulated in the explicit scheme as it has been presented before. An extra damping can be devises to kill this effect in a calibrated way; the determination of the amount of damping needed for balancing a determined choice of delta time and stiffness parameters is an issue interesting to discuss.

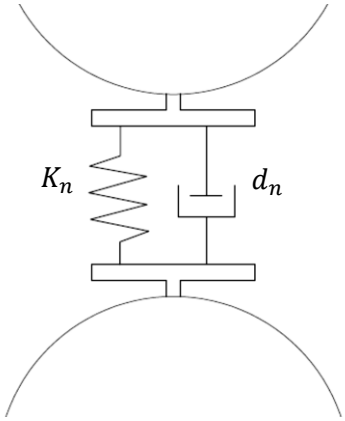


Figure I. 37 Rheological representation of the contact.

Rheologically, the damping on the contacts, is represented by a system formed by a spring and a dashpot like the *Figure I. 37 Rheological representation of the contact.* The characterization of the value for the representative dashpot can be achieved by several ways; basically the most used physical damping types are very frequently used in DEM simulations, the viscous damping and the so-called *background* damping[1]. The first one is more adequate for the dynamic simulations with impacts at a considerable speed while the second one is especially devised for the quasi-static problems, namely the compression tests in continuum simulations.

[1] In the DEM-Application both viscous damping and background damping have been implemented.

Viscous damping

The contact damping force is calculated contact by contact assuming that it is viscous type and given by

$$\mathrm{F_{vd}} = -\mathrm{d_n} \cdot \mathrm{V_{rel}} \qquad \text{Eq. 10 Visco damping force}$$

Where the relative velocity of the centres of the two particles is defined by:

$$\mathrm{V_{rel}} = \left(\dot{\mathrm{u}}_{\mathrm{target}} - \dot{\mathrm{u}}_{\mathrm{c}}\right) \cdot \mathrm{n} \qquad \text{where u is the displacement}$$

Eq. 11 Relative velocity between two particles

The damping coefficient d_n can be defined as a fraction α of the critical damping C_{cr} for the system of two rigid bodies with masses m_i and m_j, connected with a spring of stiffness k_n with:

$$d_n = \propto C_\sigma = 2 \propto \sqrt{m_{ij} k_n} \qquad \text{Eq. 12 Visco damping force}$$

with 0 ≤ α ≤ 1, and where m_{ij} is the reduced mass of the contact:

$$m_{ij} = \frac{m_i m_j}{m_i + m_j} \qquad \text{Eq. 13 Reduced mass at a contact}$$

The fraction α it is related with the coefficient of restitution C_r, which is fractional value representing the ratio of speeds after and before of an impact, through

$$\propto = \frac{-\ln C_r}{\sqrt{\pi^2 + \ln^2 C_r}} \qquad \text{Eq. 14 Expression for the fraction of the critical damping}$$

In the present work, when dealing with continuum simulations, the recommended value for the critical damping is α = 0.9, assuming a quasi-static state for the simulated processes.

Background damping

The contact damping previously described is a function of the relative velocity of the contacting bodies. It is sometimes necessary to apply damping for non-contacting particles to dissipate their energy. However this dissipation will also be effective during a contact, which is interesting for the continuous simulations in order to kill all the dynamic effects. Two types of damping have been considered here, the viscous damping and the non-viscous damping referred here as background damping. In both cases damping terms F_i^{damp} and M_i^{damp} are added to the traditional equations of motion:

$$\begin{cases} m \cdot \ddot{u} = F \\ I \cdot \dot{\omega} = M \end{cases}$$

For the non-viscous damping, the damping force is proportional to the magnitude to the resultant force and resultant moment in the direction of the velocity.

$$F_i^{damp} = -\propto^n ||F'_i|| \frac{\dot{u}_i}{||\dot{u}_i||}$$
$$M_i^{damp} = -\propto^r ||M'_i|| \frac{\omega_i}{||\omega_i||}$$

Eq. 15 Background dampings as a proportion of the magnitude

Where $\propto^n$, $\propto^r$ are damping constants, and F'_i , M'_i are defined as

$$F_i' = \sum_{c=1}^{n_c} F_{ic} + F_i^{ext}$$
$$M_i' = \sum_{c=1}^{n_c} (l_i^c \cdot F_i^{ext} + q_i^c) + M_i^{ext}$$

Eq. 16 Expression for the applied Force and Moments on a contact

As it will be commented on PART II, both damping types are implemented in DEM-Application in a way that the user can activate or disable each one separately.

2.2.7. Tangential interaction forces

In this section the frictional model for original discrete elements (detached contacts) is explained. The tangential behaviour when the contact is cohesive (continuum simulating) can be found on section 3 *CONTINUUM MODELLING WITH DEM.*

In the absence of cohesion (if the particles were not bonded at all or the initial cohesive bond has been broken) the tangential reaction F_i appears by friction opposing the relative motion at the contact point. The relative tangential velocity at the contact point v_{rt} is calculated from the following relationship:

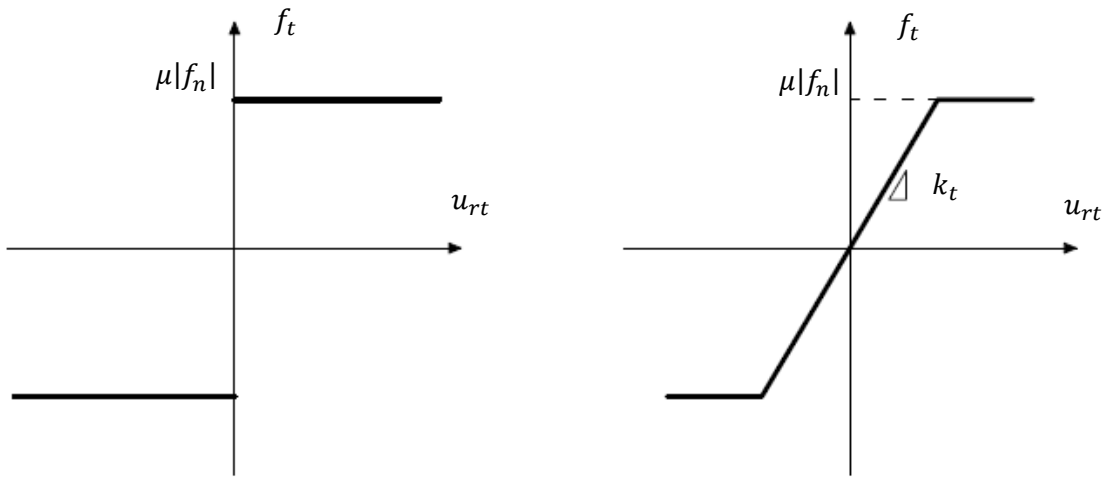
$$\boldsymbol{v}_{rt} = \boldsymbol{v}_{\boldsymbol{r}} - (\boldsymbol{v}_r \cdot \boldsymbol{n})\boldsymbol{n}$$

With

$$\boldsymbol{v}_r = \left(\dot{\boldsymbol{u}}_j + \boldsymbol{\omega}_j \cdot \boldsymbol{r}_{cj}\right) - (\dot{\boldsymbol{u}}_i + \boldsymbol{\omega}_i \cdot \boldsymbol{r}_{ci})$$

Eq. 17 Relative velocity of two particles in the tangential direction

Where $\dot{x}_i$, $\dot{x}_j$ and ω_i , ω_j are the translational and rotational velocities of the particles and r_{ci} and r_{cj} are the vectors connecting particle centres with contact points.



(a) *Classical Coulomb law* (b) *Regularized Coulomb law*

Figure I. 38 Classical Coulomb Law and Regularized Coulomb Law.

The relationship between the friction force f_t and the relative tangential displacement u_{rt} for the classical Coulomb model (for a constant normal force f_n) is shown in *Figure I. 38 (a)*. This relationship would produce non physical oscillations of the friction force in the numerical solution due to possible changes of the direction of sliding velocity. To prevent this, the Coulomb friction model must be regularized. The regularization procedure chosen involves decomposition of the tangential relative velocity into reversible and irreversible parts v_r^{rt} and v_{rt}^{ir} , respectively as:

$$v_{rt} = v_r^{rt} + v_{rt}^{ir}$$

Eq. 18 Tangential velocity decomposition

This is equivalent to formulating the frictional contact as a problem analogous to that of elastoplasticity, which can be seen clearly from the friction force-tangential displacement in the relationship in *Figure I. 38 (b)*. This analogy allows us to calculate the friction force employing the standard radial return algorithm. First a trial state is calculated.

$$F_t^{trial} = F_t^{n-1} - k_t v_{rt} \Delta t$$

Eq. 19 Trial force expression for the tangential case

And then the slip condition is checked

$$\emptyset^{trial} = || F_t^{trial} || - \mu |f_n|$$

Eq. 20 Slip condition checking

If $\emptyset^{trial} \leq 0$, a stick contact occurs and the friction force is assigned the trial value

$$F_t^n = F_t^{trial}$$

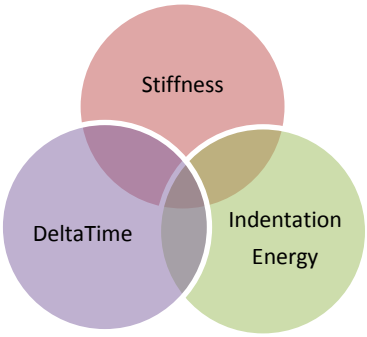
Otherwise (slip contact) a return mapping is performed giving

$$F_t^n = \mu |f_n| \frac{F_t^{trial}}{|| F_t^{trial} ||}$$

Eq. 21 Classical return mapping for the tangential force

2.2.8. Final remark on the constitutive modelling of the contact.

As a conclusion of this sub section it is important to remark the key factors that have to be taken into account when (also in users' stage) simulating either a purely discrete DEM or a continuum simulating problem. There exist one remarkable triad of parameters that have to be combined in an equilibrated form to end with a convention for the solution: Normal Spring Stiffness – Delta Time – Indentation Permitted/Energy gained. The problem is obviously more critical when dealing with dynamic simulations.



$$K \uparrow \implies indent \downarrow, \qquad E \uparrow \ (linear\ relation)$$

$$\Delta T \uparrow \implies indent \sim \uparrow\uparrow, \qquad E \uparrow\uparrow \ (non\ linear)$$

It would be desired to choose the correct stiffness derived theoretically from the contact mechanics and also a suitable time step coarse enough to have fast calculations. After the calculation a realistic solution with neither energy gaining nor large unrealistic indentations it would be also expected. Unfortunately this is not always achieved and a convection should be done assuming either some unreal indentation and avoiding energy gaining or, the other way round, performing an extremely accurate simulation with very little time steps and consequently long simulation times.

A further analysis should be realised to calibrate the correct choice of these parameters and analyse the dependence and the sensitivity of these parameters with respect to the energy gaining. Introducing some numerical damping is often the solution that permits to use the desired realistic parameters and kill the energy gaining effect.

2.3. Integration of the motion laws

2.3.1. Explicit integration schemes.

In the context of large simulation problems, the implicit schemes are not suitable because of massive memory requirements. The discontinuous-based simulation methods like DEM use explicit integration scheme. For this purpose a whole range of explicit schemes has been developed, namely the Central Difference Scheme, Leap Frog Scheme, Newmark-Beta Method or Runge-Kutta. Between these schemes, we can found second order, third order or even fourth order. Higher order schemes are possible, but involve repeated force evaluations. Since intensive CPU-time is required, higher-order schemes may not be as efficient in terms of computational cost, in comparison with lower-order schemes. In the literature, there are a lot of reports about comparisons between the different schemes where the stability, accuracy and computational cost are analysed. Some details of the comparisons can be found in [15].

As an illustrative case, the Central Difference Scheme is used for the integration on the equations of motion. It is a second-order time integration scheme originally developed in the context of structural dynamics while in some applications it is also referred to as the *Velocity Verlet* algorithm. This scheme presents a good ratio between accuracy and computational cost. Time integration operator for the translational motion at the $n-th$ time step is as follow:

$$\ddot{u}_i^n = \frac{F_i^n}{m_i}$$
$$\dot{u}_i^{n+1/2} = \dot{u}_i^{n-1/2} + \ddot{u}_i^n \Delta t$$
$$u_i^{n+1} = u_i^n + \dot{u}_i^{n+1/2} \Delta t$$

Eq. 22 Central Difference Integration Scheme

The first two steps in the integration scheme for rotational motion are identical to those given by the previous equations:

$$\dot{\omega}_i = \frac{T_i^n}{I_i}$$

$$\omega_i^{n+1/2} = \omega_i^{n-1/2} + \dot{\omega}_i^n \Delta t$$

Eq. 23 Iterative algorithm for the rotational velocity

For rotational plane (2D) motion the rotation angle θ_i can be obtained similarly as the displacement vector x_i:

$$\theta_i^{n+1} = \theta_i^n + \omega_i^{n+1/2} \Delta t$$

Eq. 24 Calculation of the step rotation

In three-dimensional motion, rotational position cannot be defined by just one vector. The rotational velocity ω cannot be integrated. The vector of incremental rotation is obtained as

$$\Delta\theta_i = \omega_i^{n+1/2} \Delta t$$

It must be remarked that knowledge of the rotational configuration is not always necessary. If tangential forces are calculated incrementally, then knowledge of the vector of incremental rotation $\Delta\theta$ is sufficient. This saves considerable computational cost of the time integration scheme.

2.3.2. Numerical stability of the method and critical time step

Explicit integration in time yields high computational efficiency and it enables the solution of large models. The known disadvantage of the explicit integration scheme is its conditional numerical stability imposing the limitation on the time step Δt, i.e.

$$\Delta t \leq \Delta t_\sigma$$

Where Δt_σ is a critical time step determined by the highest natural frequency of the system ω_{max}.

$$\Delta t_\sigma = \frac{2}{\omega_{max}}$$

If damping exists, the critical time increment is given by

$$\Delta t_\sigma = \frac{2}{\omega_{max}} \left(\sqrt{1 + \varepsilon^2} - \varepsilon \right)$$ Eq. 25 Critical time increment (with damping)

Where ε is the fraction of the critical damping corresponding to the highest frequency ω_{max}. Exact calculation of the highest frequency ω_{max} would require solution of the eigenvalue problem defined for the whole system of connected rigid particles.

In the algorithm implemented an approximate solution procedure is employed. An eigenvalue problem can be defined separately for every rigid particle. The maximum frequency is estimated as the maximum of natural frequencies of mass-spring systems defined for all the particles with one translational and one rotational degree of freedom

$$\omega_{max} = \max \omega_i$$

And the natural frequency for each mass-spring system (contact) is defined as

$$\omega_i = \sqrt{\frac{k}{m_i}}$$ Eq. 26 Natural frequency for the classical mass-spring system

With k the spring stiffness and m_i the mass of particle i. Now it is possible to rewrite the critical time step as

$$\Delta t_\sigma = min\, 2\sqrt{\frac{m_i}{k}}$$

The effective time step is considered as a fraction of the critical time step

$$\Delta t = \beta \Delta t_\sigma$$

With

$$0 \leq \beta \leq 1$$

The value of β has been studied by different authors. A good review can be found in [16] where the author recommend values close to $\beta = 0.17$ for 3D simulation, and $\beta = 0.17$ in the 2D case.

3. CONTINUUM MODELLING WITH DEM

The Discrete Element Method has been presented in this work and in many books and papers as a good numerical method to simulate the discontinuous media as a system of independent particles in dynamic motion. However, when dealing with the continuum, nowadays, results are not completely satisfactory even though a lot of research has been done. There have been, indeed, a vast number of different approaches for this question: How shall the contact models be characterized (micro scale parameters) in order to get the macro scale continuum behaviour?

The challenge in all DEM models is finding an objective and accurate relationship between the DEM parameters and the standard constitutive parameters of a continuum mechanics model (hereafter called "continuum macro parameters"), namely the Young modulus *E*, the Poisson ration ν and the tension and shear stresses σ_t^f and τ^f, respectively.

Two different approaches can be followed for determining the DEM constitutive parameters namely the *global approach* and the *local approach*. In the global approach *uniform global* DEM properties are assumed in the whole discrete element assembly. The values of the global DEM parameters can be found using different procedures. Some authors have used numerical experiments for determining the relationships between DEM and continuum parameters expressed in dimensionless form [17]. This method has been used by the authors in previous works and is described in the next sections. Other procedures for defining the global DEM parameters are based on the definition of average particle size measures for the whole discrete particle assembly and then relating the global DEM and continuum parameters via laboratory tests.

A second approach, followed in this work, is to assume that the DEM parameters depend on the *local properties* of the interaction particles, namely their radii and the continuum parameters at each interaction point. Many alternatives for defining the DEM parameters via a "local approach" have been reported by different authors [18].

This relation should yield a characterization of the media such that the model behaves in terms of stress and strains like the continuum media does. Also the D.E.M is expected to be an effective and powerful numerical technique for reproducing multifracture and failure of geomaterials (soils, rocks, concrete), masonry and ceramic material, among others. There has been a lot of research done in this field and some codes presume of obtaining good approaches to this problem using the Discrete Element Method. Nevertheless, there is not an absolute universal method that assures generality to the continuum modelling problem.

3.1. Global derivation of DEM micro parameters using dimensionless relationships

Global DEM mechanical parameters can be determined using the methodology developed by Huang [19], based on the combination of dimensional analysis with numerical simulation of standard laboratory test for rocks, namely the unconfined compression and the Brazilian tests.

The challenge in global constitutive models is finding the relationship between the continuum material parameters: Young modulus *E*, Poisson's ratio ν, compressive strength σ_c and tensile strength σ_t in terms of global DEM parameters: K_n, K_t, $\mathcal{F}_n$, $\mathcal{F}_s$, μ, d_n d_t, d_r[1]. DEM material properties also depend on other parameters related with the particle assembly characterization, such as the average particle radius *r*, the material density ρ and the porosity of the particle assembly *n*. All these parameters are strongly related to the assembly generation algorithm. The set of global DEM parameters can be completed with geometrical parameters represented by the specimen size *L* (due to possible scale effect) and loading velocity v_e. Thus, the number of relevant parameters *N* is 12. We have three primary dimensions involved: mass, length, time (*p*=3). Typically it is assumed that there are 9 independent parameters.

The global DEM parameters are not unique and can be modified by taking into account some other parameters that can influence macroscopic properties. In the minimum and maximum element radii, r_{min} and r_{max}, respectively, were included to the relevant parameters in orden to better consider the influence of the element size distribution on macroscopic properties. This influence can be taken into account indirectly through the use of the porosity n which depends on the size distribution. The wider size distribution the lower porosity in the discrete element model can be achieved.

[1] *The micro parameters of the model are considered to be the same for all the contacts and are derived in a general way from the macro parameters using adimensional relationships.*

A general definition of the dimensionless DEM parameters includes the following set of nine independent parameters: $\left\{\frac{K_n r}{R_n}, \frac{R_s}{R_s}, \frac{K_s}{K_n}, n, \frac{r}{L},\ \mu,\ d_n, d_t, \frac{v_e}{\sqrt{\frac{K_n}{\rho}}}\right\}$. The set of parameters can be reduced by neglecting dynamic effects and removing $\frac{v_e}{\sqrt{\frac{K_n}{\rho}}}$, d_n and d_t. Further on, assuming that the element size *r* is small compared to macroscopic dimension L, $(r \ll L)$, the influence of the parameter $\frac{r}{L}$ can be negletec. The friction coefficient μ has influence mainly on the post-failure material behaviour, so is generally omitted it in the relation-ships for elastic constants and strength parameters. After these simplifications set of relevant dimensionless parameters is reduced to the following four: $\left\{\frac{K_n r}{R_n}, \frac{R_s}{R_s}, \frac{K_s}{K_n}, n\right\}$. Assuming that the elastic stiffness parameters are determined in the range in which the failure is not initiated yet, only two dimensionless parameters should be considered: $\left\{\frac{K_s}{K_n}, n\right\}$. Thus, the folowing dimensionless functional relationships linking continuum and global DEM parameters can be postulated as:

$$\frac{El}{K_s} = \Phi_E\left(\frac{K_s}{K_n}, n\right), \qquad \nu = \Phi_\nu\left(\frac{K_s}{K_n}, n\right)$$
$$\frac{\sigma_c A}{\mathcal{F}_n} = \Phi_c\left(\frac{\mathcal{F}_s}{\mathcal{F}_n}, \frac{K_s}{K_n}, n\right), \qquad \frac{\sigma_t A}{\mathcal{F}_n} = \Phi_c\left(\frac{\mathcal{F}_s}{\mathcal{F}_n}, \frac{K_s}{K_n}, n\right)$$

Eq. 27 Dimensionless relationships global parameters

l is a certain length parameter and A is a characteristic area related to the discrete element model. The characteristic length l and area A are defined in different way for 2D and 3D problems. For 2D problems, where cylindrical particles are used, it is convenient to take l as equal to the length (height) of the particles, with a unit value. For more information, important references in global-derived methods are Huang [20] [19] and also Oñate and Labra [21] and Rojek [22].

3.2. Local definition of DEM elastic constitutive parameters

This alternative philosophy is to assume that the DEM parameters depend on the *local properties* of the interaction particles. Instead of defining a global set of parameters $\{K_n, K_t, \mathcal{F}_n, \mathcal{F}_s, \mu, d_n\, d_t, d_r\}$,[1] each contact would have different characterization depending on the physics of the contact and the properties of the interacting particles, namely their radii and the continuum parameters. The method seeks the adequate parameterization of the contacts between particles in order to obtain a model that behaves in terms of stress and strains like the continuum media does.

This is the path that CIMNE follows; indeed, it is the most paradigmatic example of the aspects in discussion about the continuum simulating. The next sections of this work will be specifically related with this local definition philosophy.

First of all, it is important to clarify that the problem is not the same as the discontinuous case in terms of dynamism and elastic deformation behaviour. In the discontinuous media simulations it has been seen that the normal stiffness value is not such important a parameter as it will be here, in the continuum simulating case, where we want to capture realistic strains and stresses. Then, would the high values for the stiffness yield now the same problems that occurred with the dynamic systems?

On one hand, the problems induced by the large repulsive forces that occurred when a particle got inside the other in a time step won't be present now; the particle now will move much slowly compared to a dynamic system and so, the indentations that can occur in a given time step are much smaller. On the other hand, most of the particles will be confined by a large number of contacting neighbours and, when a particle is excited by an external force, it would impact to others and then do the same in the opposite direction. This can easily lead to a system that has an energy feedback and it increases uncontrolled. That's why here the global damping (see section *2.2.6 Numerical damping and physical damping*) takes an important role taking the system to a quasi-static state and killing the dynamic effects.

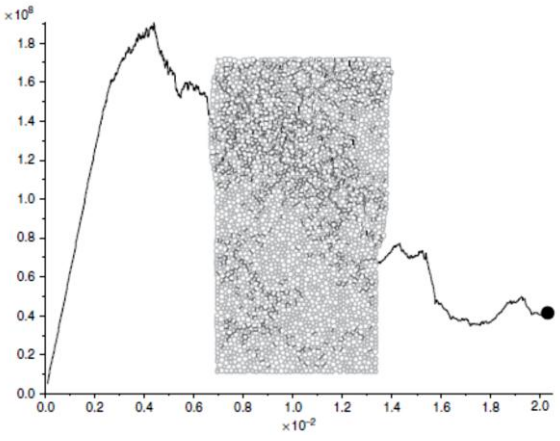


Figure I. 39 Compression test simulation with DEM

[1] *The same micro parameters derived (from macro parameters) locally for each contact.*

3.3. The effective contacting volume method

This is an original idea from Professors E. Oñate J. Miquel and F. Zárate, from CIMNE, and it is currently in research. This proposal theory will be implemented in DEM-Application and will be tested once the program completely allows continuum based simulations.

It has been already mentioned that now the coefficients for the normal and tangential spring stiffness are very important. The objective is to correctly characterise the continuum properties to expect the same results in a global behaviour.

The method proposes to get the elastic characteristic values for linear springs in normal direction and for the transversal one from the equivalent axial stiffness and shear stiffness respectively that the correspondent truncated conical volume would yield.

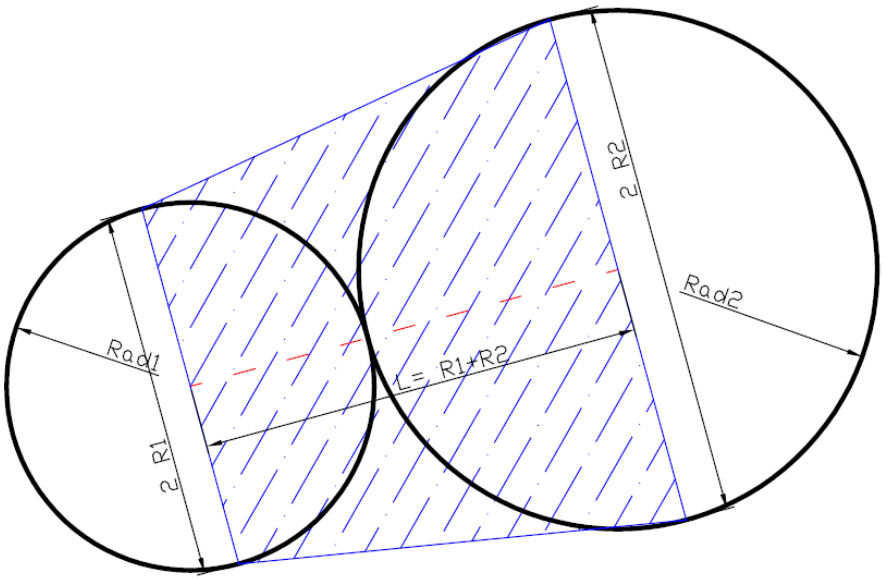


Figure I. 40 Equivalent volume corresponding to by the contact

Axial stress:

$$\delta_x = u_2 - u_1 = \int_0^L \varepsilon dx = \int_0^L \frac{F_x}{EA} dx = \frac{F_x}{E} \int_0^L \frac{dx}{A(x)}$$

Eq. 28 Axial strain-stress approach

Linear variation of the radius:

$$R = R_1(1 + \beta x) \quad where \ \beta = \frac{R_2 - R_1}{R_1 \cdot R_2}$$

Yields:

$$\delta_x = u_2 - u_1 = \frac{F_x}{E}\int_0^L \frac{dx}{\pi(1+\beta x)^2} = \frac{Fx}{\pi E}\cdot\frac{R_1+R_2}{R_1\cdot R_2}$$

$$F_x = k_N \cdot \delta_x \quad k_N = \pi E \cdot \frac{R_1\cdot R_2}{R_1+R_2} = \frac{\pi E}{2}\cdot R_{eq}$$

Eq. 19: Consistent formulation for the normal stiffness.

Where $R_{eq} = 2\frac{R_1\cdot R_2}{R_1+R_2}$ Eq. 2 Equivalent radius for two spheres in contact.

Shear stress:

$$\delta_y = v_2 - v_1 = \int_0^L \gamma dx = \int_0^L \frac{Q}{G\Omega}dx = \frac{Q}{G}\int_0^L \frac{dx}{\Omega(x)}$$

Eq. 30: Shear strain-stress approach.

Where $\Omega(x) = \pi R(x)^2 \cdot \mu$ Eq. 3: Reduced shear area.

Linear variation of the radius:

$$R = R_1(1+\beta x) \quad where\ \beta = \frac{R_2-R_1}{R_1\cdot R_2}$$

$$\delta_y = v_2 - v_1 = \frac{Q}{G}\cdot\frac{1}{\pi\mu}\cdot\frac{R_1+R_2}{R_1\cdot R_2}$$

$$Q = k_T \cdot \delta_y \quad k_T = \mu\pi G \cdot \frac{R_1\cdot R_2}{R_1+R_2} = \frac{\mu\pi E}{4(1+v)}\cdot R_{eq}$$

Eq. 31: Consistent formulation for the shear stiffness.

Relationship between k_T and k_N:

$$k_T = k_N \cdot \frac{\mu}{2(1+\nu)}$$

Eq. 32: Resulting relation between normal and shear stiffness.

A formal relationship between the main parameters υ, E, R and the elastic stiffness coefficients of the equivalent springs on the contact has been proposed to be used in continuum simulating problems. Obviously there exist many others in the literature but the DEM-Application has chosen the family of locally derived micro parameters methods. The DEM-Application team has clearly have in mind that these stiffness parameters for the springs have to be deduced contact by contact in order to obtain a method that could be used in generic cases without much particular calibration; in that sense, the algorithm implemented in our application takes the macro values of υ, E, R for the contacting pair and easily applies the presented formula or any desired one[1].

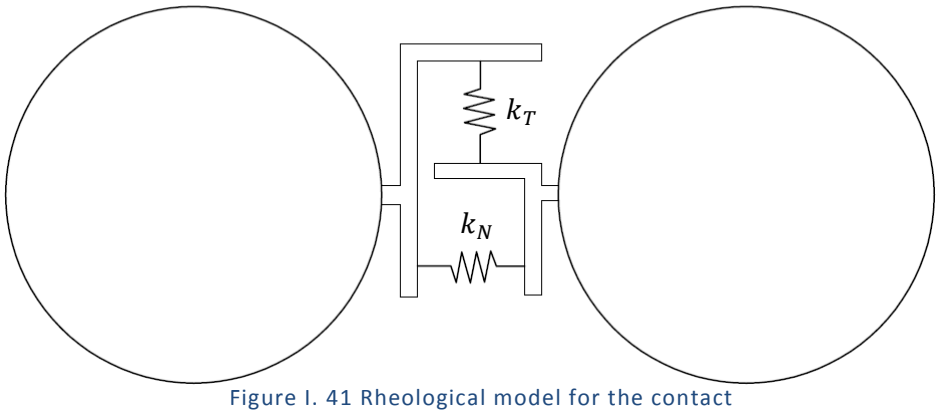


Figure I. 41 Rheological model for the contact

Until now two new concepts have been introduced regarding the characterization of the contact with respect to the original DEM for discrete systems: In first place, the normal spring has also a tensile strength and so can resist tensile forces; in second place, there exist a classical tangential spring that resists the α tangential shear forces until a certain limit. When this tangential spring reaches the failure limit the contact becomes frictional. Afterwards, the regularized Coulomb law is recovered; it can be interpreted as another little spring system also with a slipping limit.

In the new section a completely innovative devise special for the continuum simulating case is introduced: the rotational spring.

[1] *The DEM-Application, and usually in many others, is quite easy to tune the values of these parameters in order to get better results; in this sense there is no loose of generality when proceeding with this approach since afterwards, the characterization can change if comes out not to be the proper one.*

3.4. Rotational Spring

3.4.1. Justification of the rotational spring

Some authors recommend the use of an additional spring that acts opposing the relative rotation that a contacting pair suffers in continuum simulations. The necessity of it appears when the rotation is applied on the problem; the particles are given a rotational inertia and also we calculate the moments that come from the forces applied on the contacts with respect to the mass centre of these particles. When a moment applies to a certain particle that has a rotational inertial it begins to rotate due to the angular acceleration and in principle is not resisted by any mechanism.

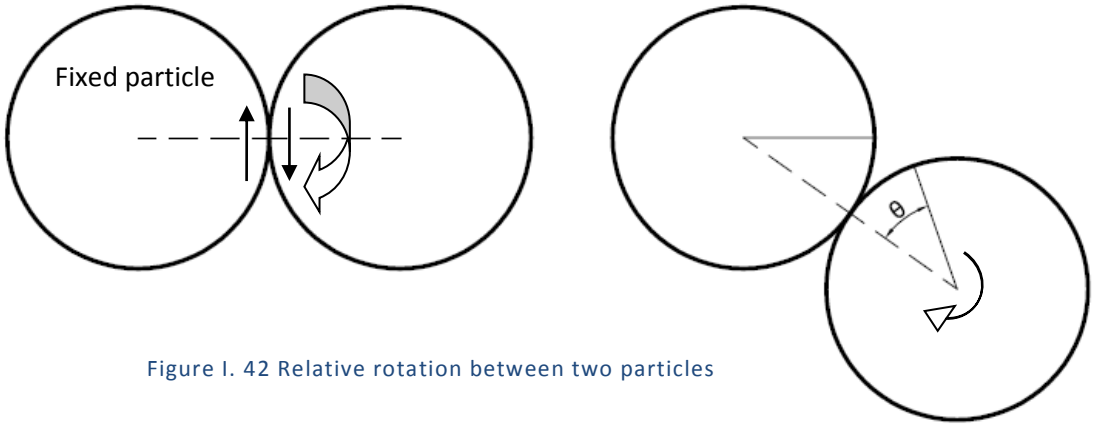


Figure I. 42 Relative rotation between two particles

Note that this rotation is not resisted by the tangential spring because there is no relative tangential displacement between the contacting points. Therefore, the rotational spring shall be introduced to opposite the relative rotation.

3.4.2. Proposed stiffness for the rotational spring

The characterization of the stiffness value for the spring can be easily done relating the tangential displacement with the rotational one.

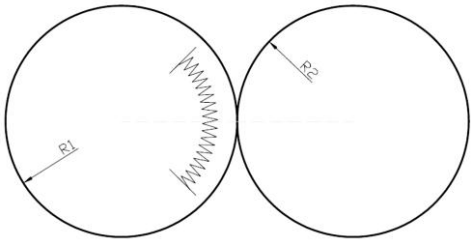


Figure I. 43 Rheology for the rotational spring

Applying the hypothesis that for a little rotation the normal component of the movement is zero and all the movement is in the tangential direction and proposing an equation for the rotational spring we get that:

$$u^t = \theta \cdot R_1$$
$$M^r = k_R \cdot \theta$$

Eq. 33 Expression for the resistant moment in the rotational spring

Also we get the ordinary expression for the resistant shear in the tangential displacement and we calculate the equivalent moment that it would create:

$$T = k_T \cdot u^t = k_T \cdot \theta \cdot R_1$$
$$M^r = T \cdot R_1$$

Eq. 34 Moment produced by the tangential force opposing the rotation

Finally comparing the produced moment and the resistant one we can obtain a relationship for between the rotational spring stiffness and the shear spring stiffness.

$$k_R = k_T \cdot {R_1}^2$$

Eq. 35 Proposed expression for the rotational spring stiffness

3.4.3. Remarks on the rotational spring

While some authors insist in the necessity of a rotational spring others don't use it. In fact when some entity is discretized by more than one row (in 2D) of particles (spherical or not) the combination of the normal and tangential springs act to oppose the global or local bending of the particles; this is due to the fact that this rotation implies displacement in the normal and tangential directions of the neighbouring particles with respect to the contacts and so, the tangential and normal forces act. It is not clear if this spring which is indispensable in a "1 row" case shall contribute to resist the bending mechanism or not.

The rotational spring, if it is enabled, has the same treatment than any other elastic devise; we will apply a damping on it and it will also have a limit strength which would lead to the fracture of the contact if its value is exceeded.

3.4.4. Example of application

In order to exemplify the functionality of this spring, a simple bending beam is presented in a very illustrative case.

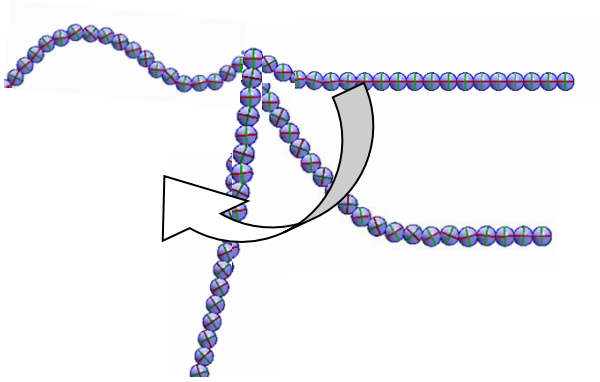


Figure I. 45 Ball chain without rotational spring

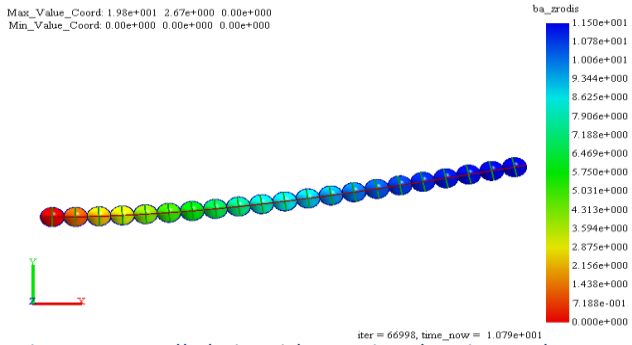


Figure I. 44 Ball chain with rotational spring under an ascendant load

In the first figure a chain of balls is created without any rotational spring, the first ball is fixed and the problem is calculated under gravity; the behaviour is like a chain of balls which can have free relative rotation. On the other hand, in the second example, the rotational spring is applied forming then a beam that resist bending moments like the one illustrated in the figure due to an ascendant force applied on the extreme. It shall be commented that the results are very accurate comparing the solution with the theoretical ones.

The previous examples have been done by researcher Feng Chun using the CDEM software (see section 6.2 Current development and collaboration); The DEM-Application takes the implementation methodology of the rotational spring from that program.

3.5. Failure of the contacts, plasticity and damage

The method permits applying easily any micromechanical constitutive model with cohesion. Given a contact defined by the properties of the contacting spheres and the forces in every direction: normal and tangential, the fracture criterion can be established following the classical laws of the mechanics of solids. The implemented method in the DEM-Application is the perfect brittle elasticity, which is the simplest one, but nowadays is being extended to other more sophisticated ones.

The elastic perfectly brittle model is characterized by linear elastic behaviour when cohesive bonds are active. An instantaneous breakage of these bonds occurs when the interface strength is exceeded. When two particles are bonded the contact forces in both normal and tangential directions are calculated from the linear constitutive relationships:

$$\sigma = k_n \cdot u_n$$
$$\tau = k_t \cdot u_t$$

Eq. 11: Linear elastic force displacement relationship.

Where σ and τ are the normal and tangential contact forces, respectively, k_n and k_t are n the interface stiffness in the normal and tangential directions u_n and u_t, the normal ad tangential relative displacements, respectively.

Cohesive bonds are broken instantaneously when the interface strength is exceeded in the tangential direction by the tangential contact force or in the normal direction by the tensile contact force. The failure (decohesion) criterion is written (for 2D) as:

$$\sigma \leq \mathcal{F}_n$$
$$\tau \leq \mathcal{F}_s$$

Where R_n and R_t are the interface strengths in the normal and tangential directions, respectively. In the absence of cohesion the normal contact force can be only compressible, i.e.

$$\sigma \leq 0$$

And the (positive) tangential contact force is given by the Coulomb friction law, with μ being the Coulomb friction coefficient:

$$\tau = \mu|\sigma| \qquad \text{Eq. 12: Coulomb's friction law}$$

Contact laws for the normal and tangential directions for the elastic perfectly brittle model are shown in next figures:

$\boldsymbol{\tau}$

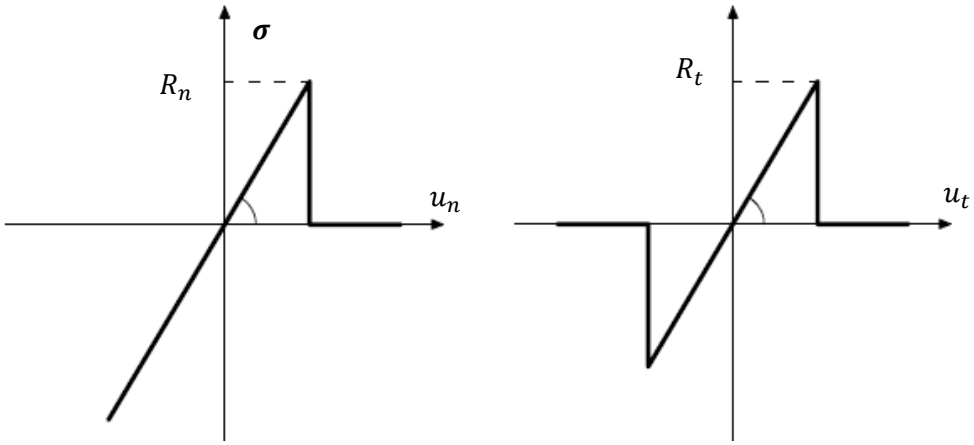


Figure I. 46 Normal and tangential contact force in perfectly brittle model.

Other more sophisticated models can be applied to the DEM for the failure of the contacts: elastoplastic contact with linear softening, hardening, exponential plasticity laws, contacts with elastic damage, etc.

3.6. Generation: modelling the structure of the continuum

Given a geometric definition of the media, the discrete element mesh has to be generated. A fundamental aspect when simulating the continuum is to obtain a good packaging, i.e. the minimum voids inside the domain. Several techniques have been developed to perform a generation that fills a given volume with a good packaging; originally the way to obtain these meshes was to fill a volume by gravitational deposition of a pack of particles.

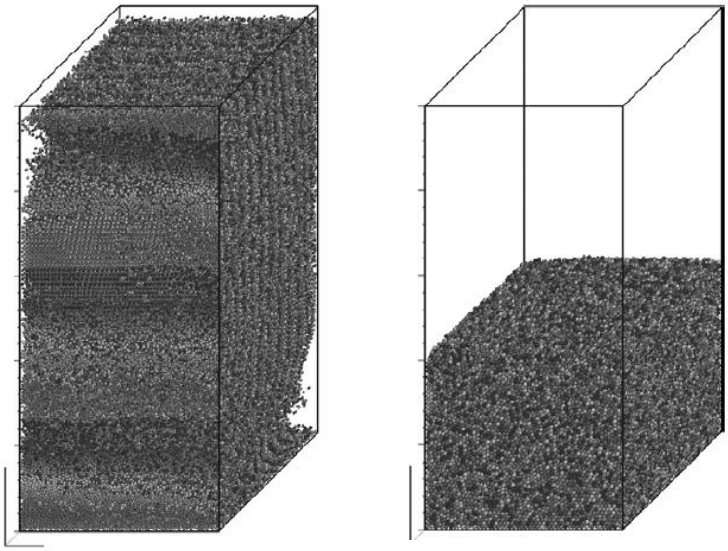


Figure I. 47 Gravitational deposition test by Munjiza

This method gives a good packaging but it is computationally inefficient since we have to do a DEM discontinuum simulation with a large number of particles. Find more information about this on [3] *Munjiza*.

Other methods for the generation of the DEM mesh are algorithms of generation of particles that try to fill the volume with a determined distribution of sizes or assuring a determined volume of voids in the domain. There exist a large number of generators of this type in the DEM software; GiD has its own generator and is the one used in the DEM-Application. It will be explained in further detail in Part II *8.5.7 Neighbour Search utility and Extended Radius Search* that this generator, like many others, introduce some gaps between particles; in that section a special utility of the DEM-Application is presented that helps solving this. On the other hand there are also other generators that produce a more dense packing and don't have these problems; however this kind of generators often present initial indentations between particles. Another utility has been devised for the DEM-Application to solve this problem and be able to use these dense packing generators.

Finally, some new advanced geometry definition and sphere generator methods are being used nowadays for the DEM. By the use of tomography scans a detailed geometry of parts of the body such as bones, organs and vessels can be represented with thousands of particles. This is a topic in discussion that is especially interesting for the KRATOS research group in order to give the DEM new applications.

Figure I. 48 Fine particle mesh generated on a skull by UCLV-CIMNE

4. DEM-FEM

This is the current challenge of the Discrete Element Method. From the experience taken from the study of the continuous and the discontinuous media with DEM It can be announced that nowadays there is no a unique effective methodology to describe the continuum media although there are several "well-posed" for the discrete particle physics. Further research has to be done in the continuum simulation via DEM but nowadays this method is proficient with the discrete media. What comes next is the idea of combining to methods that are efficient in their respective fields to simulate coupled problems where both phenomena can be presented.

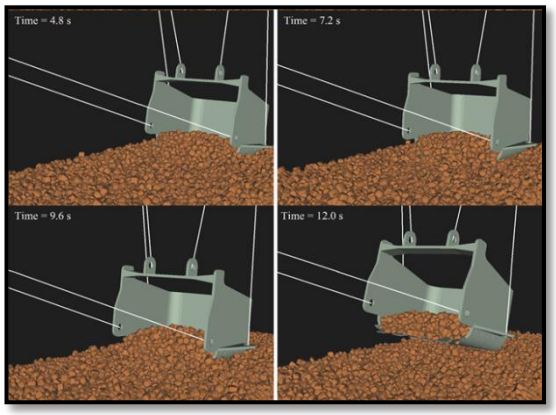


Figure I. 49 Interaction between a tool (FEM) and rocks (DEM)

This has recently been properly studied but there is nowadays an increasing interest on this subject. One of the most important applications of the DEM-FEM applications is on the interaction between granular materials and excavation tools. A recommended reference on this issue is: [20] by Huang.

There are different possible points of view of the DEM-DEM coupled problem:

- Interaction between FEM-discretized bodies and DEM-discretized domain: For simulations of excavation problems generally to analyse the stresses on the tools used in the excavation. This is a useful method to test the strength and the wear or the mechanical devices.

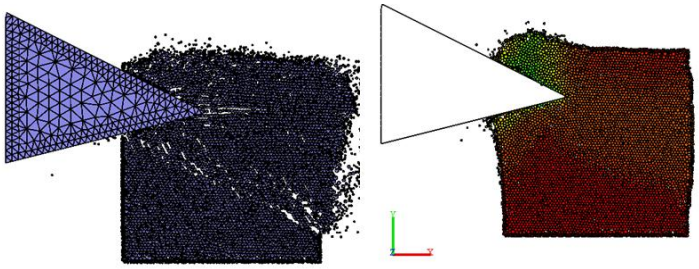


Figure I. 50 FEM wedge introduced in DEM domain

- Creation of DEM particles from fractured FEM elements: Transition from continuum to discontinuum in the combined finite-discrete element method occurs through fracture and fragmentation processes. A typical combined finite-discrete element method based simulation, such as rock blasting, may start with a few discrete elements and finish with a very large number of discrete elements. Fracture in general occurs through alteration, damage, yielding or failure of microstructural elements of the material. To describe this complex, material-dependent phenomenon, the alteration of stress and strain fields due to the presence of microstructural defects and stress concentrations must be taken into account. In order to simulate the friction and the interaction that is generated in the continuum when a crack appears on the FEM domain it is possible to analyse it generating DEM particles in that zone. A recommended reference is Munjiza, A: The Combined Finite Discrete Element Method [3].

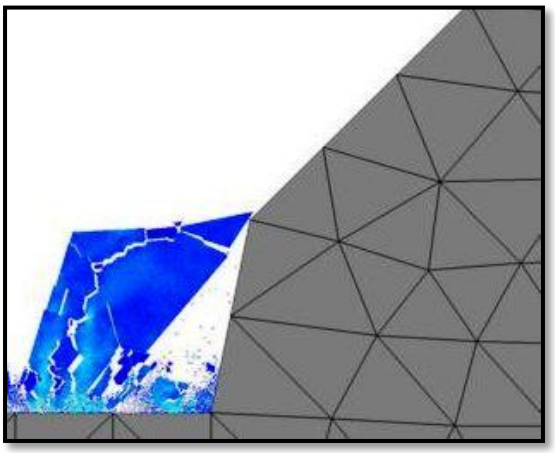


Figure I. 51 Fracture of finite element discretized media

- FEM discretization of the DEM particles: This is a way to introduce the deformation of the particles and the detailed stress field on the DEM particles. Also it can be possible to track the fractures in the DEM particle. This method is not very much used because of the high cost that it leads.

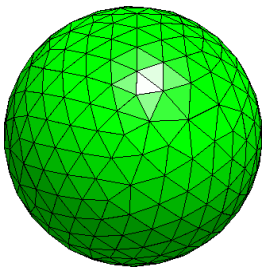


Figure I. 52 FEM discretization of a DEM particle

- DEM discretization of the FEM elements: when a finite element reaches the limit stresses it is substituted by a set of discrete elements in order to capture the fracture and the frictional behaviour of the fractured parts of the continua.

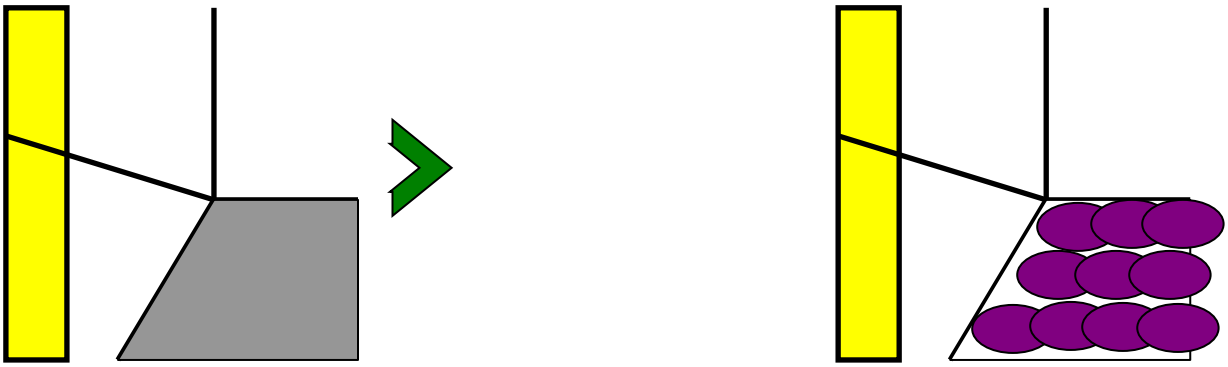


Figure I. 53 DEM discretization of a FEM element

Without entering on these methods, a few ideas shall be kept in mind when coupling the two problems. First of all, and very important, the search becomes much more complicated. Even if all the media is discretized with spheres, there would be numerous contacts between spherical entities and non spherical entities; therefore it would be necessary to recover the methods presented on section *2.1 Contact Detection.* Also, the treatment of the contacts between the FEM elements and DEM elements it is a complicated issue; how to characterize the force that a DEM element introduces to each node of the FEM element when these entities intersect, and afterwards determine the stresses, is not an easy problem.

In CIMNE the DEM-FEM application has been developed in parallel with the DEM application and all the DEM discretized elements that appear in DEM-FEM are characterized in the same way as the original DEM particles from the DEM-Application. The basic concepts of the implementation of the DEM-Application are explained in the next part of this work.

Part II: KRATOS DEM-Application

5. KRATOS-MULTIPHYSICS PLATFORM

Free Multi-physics F.E.M.-Based C++ Open Source Code

5.1. What is KRATOS?

KRATOS was born a framework for building multi-disciplinary finite element programs and has been extended more generally to other engineering application. It provides several tools for easy implementation of FEM-like engineering applications and a common platform for natural interaction of the same in different ways.

KRATOS is an innovative variable base interface designed and implemented to be used at different levels of abstraction and to be very clear and extendible. A very efficient and flexible data structure can be used to store any type of data in a type-safe manner. An extendible IO is also present to overcome a bottleneck in dealing with multi-disciplinary problems and the major interpreting task is given to the Python interpreter.

The kernel and application approach is used to reduce the possible conflicts arising between developers of different fields and layers are designed to reflect the working space of different people also considering their programming knowledge. It permits to create your new application starting from a template for every basic generic part of your program. The application connects to the main KRATOS general structure and it benefits of its data base common utilities for general FEM-like engineering programs ready to be used, the IO structure to interact with graphical interfaces and the powerful and optimized usage of the combined C++ and python languages.

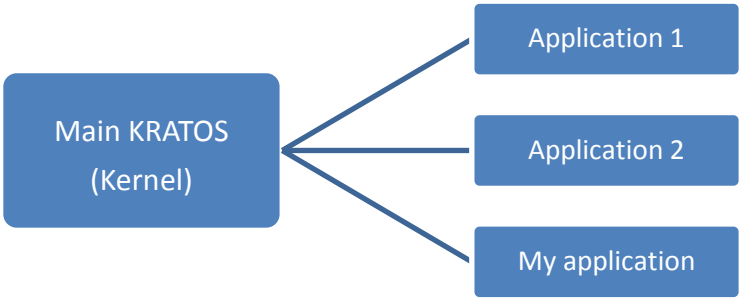


Figure II. 1 KRATOS basic scheme

5.2. Who may use KRATOS?

Some potential users of KRATOS are:

- Research engineers: These developers are considered to be more expert in numerical methods and engineering calculation methods, from the physical and mathematical points of view, than in C++ programming. For this reason, KRATOS provides their requirements without involving them in advanced programming concepts.

- Application Developers: These users are less interested in finite element programming and their programming knowledge may vary from very expert to higher than basic. They may use not only KRATOS itself but also any other applications provided by finite element developers, or other application developers. Developers of optimization programs or design tools are the typical users of this kind.

- Package Users Engineers: and designers are other users of KRATOS. They use the complete package of KRATOS and its applications to model and solve their problem without getting involved in internal programming of this package. For these users KRATOS has to provide a flexible external interface to enable them use different features of KRATOS without changing its implementation.

5.3. Who is KRATOS?

The KRATOS structure, due to its multi disciplinary nature, has to support the wide variety of algorithms involved in different areas. That's the principal reason that explains the variety of people, mostly engineers, composing the KRATOS Community.

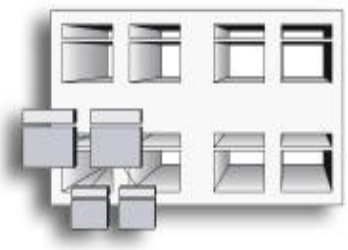
I encourage you to visit the website http://KRATOS-wiki.cimne.upc.edu/index.php/Main_Page to learn more about KRATOS.

5.4. What makes KRATOS useful?

KRATOS is MULTI-PHYSICS. One of the main topics in engineering nowadays is the combination of different analysis (thermal, fluid dynamic, structural) with optimising methods in one global software package with just one user interface and, even more, the possibility to extend the implemented solution to new problems.

KRATOS is FINITE ELEMENT METHOD (FEM) based. Many problems in engineering and applied science are governed by Partial Differential Equations (PDE), easily handled by computer thanks to numerical methods. The FEM is one of the most powerful, flexible and versatile existing methods.

KRATOS is OBJECT ORIENTED. An integration of disciplines, in the physical as well as in the mathematical sense, suggests the use of the modern object oriented philosophy from the computational point of view. The modular design, hierarchy and abstraction of these approaches fits to the generality, flexibility and reusability required for the current and future challenges in numerical methods.



KRATOS is OPEN SOURCE. The main code and program structure is available and aimed to grow with the need of any user willing to expand it. The GNU Lesser General Public License allows using and distributing the existing code without any restriction, but with the possibility to develop new parts of the code on an open or close basis depending on the developers.

KRATOS is FREE because is devoted mainly to developers, researchers and students and, therefore, is the most fruitful way to share knowledge and built a robust numerical methods laboratory adapted to their users' needs. Free because you have the freedom to modify and distribute the software. The one thing you're not able to do with free software is take away other people's freedom. Read the license for more detailed information in KRATOS webpage.

5.5. KRATOS structure

An *object-oriented* structure has been designed to maximize the reusability and extensibility of the code. This structure is based on finite element methodology and many objects are designed to represent the basic finite element concepts. In this way the structure becomes easily understandable for developers with a finite element method background. In this design, Vector, Matrix, and Quadrature represent the basic numerical concepts. Node, Element, Condition, and DoF are defined directly from finite element concepts. Model, Mesh, and Properties are from the practical methodology used in finite element modelling complemented by ModelPart, and SpatialContainer, for organizing better all data necessary for analysis. IO, LinearSolver, Process, and Strategy represent the different steps of a finite element program flow. Finally Kernel and Application are defined for library management and its interface definition.

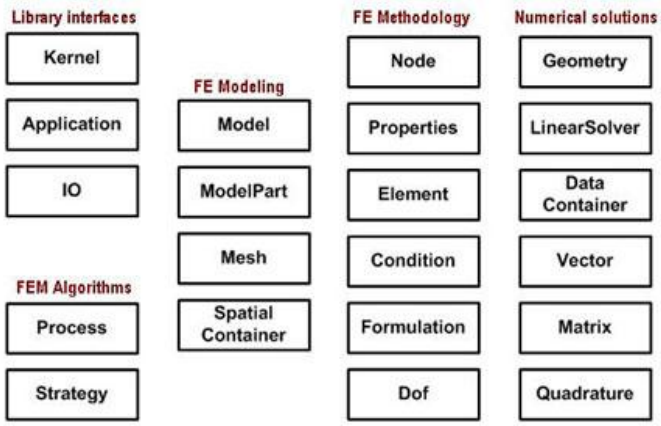


Figure II. 2 KRATOS framework

5.6. Basic tools

Different reusable tools have been implemented to help developers in writing their applications in KRATOS. Several geometries and different quadrature methods are provided and their performances are optimized. Their flexible design and general interface make them suitable for use in different applications. Their optimized performance makes them appropriate not only for academic applications but also for real industrial simulations.

An extensible structure for linear solvers has been designed and different common solvers have been implemented. In this design the solver encapsulates only the solving algorithms and all operations over vectors and matrices are encapsulated in space classes. In this way solvers become independent of the type of mathematical containers and can be used to solve completely different types of equations systems like symmetric, skyline, etc. This structure also allows highly optimized solvers (for just one type of matrices or vectors) to be implemented without any problem.

5.7. Versioning system (SVN)

Apache Subversion, SVN, is a software versioning and revision control system distributed under an open source license. KRATOS main developers are attached into a subversion sharing network in order to up-load their developments and up-date the current and historical versions of files from the basic code or from new application parts being developed by others.
This way, a new integration method, for instance, can be developed by anyone and included in KRATOS database; after that, any other user or developer can update the modified parts of their code and they get instantaneously the integration method.
In order to avoid conflicts between implementations from different people, there is also a benchmarking system checking for the correct functioning and compilation of any new implementation.

5.8. Benchmarking system

Every night, the cluster of CIMNE automatically updates KRATOS using the versioning system, after that, it compiles everything and runs different preset cases for each application.
These cases are tests that have been specially designed for each application. They consist on a simple application usage to calculate a predefined problem that has a predetermined known solution.
If the cluster doesn't get the expected solutions when running the case or, moreover, if the cluster is not able to compile the code after a new contribution from a developer onto the versioning system, everyone gets a warning reporting the problem. If this is the case, the last uploads have to be revised for the good functioning of every application.

6. KRATOS DEM-APPLICATION

6.1. Born of DEM-Application

In year 2011 CIMNE's director, Eugenio Oñate, considered interesting to start developing a Discrete Element Method in the KRATOS environment as many other Finite Element Methods had been implemented.

The seed for the code was taken from the CIMNE's current DEM program called *DEM-PACK;* this precedent program had been developed in Fortran language by some CIMNE doctoral researchers who had built their own code from scratch. DEM-PACK is nowadays the program that CIMNE uses for its projects as it is in fact a complete program that permits elaborating studies for numerous case simulations.

However as the technique and research advances, new methods and approaches has been introduced in the Discrete Element Method and also in computer science:

- Concerning to the DEM theory, it has been already explained some of these advances in the first part of this document; this refers to the new ideas about **continuum-simulating**, clusters of particles and arbitrary shape contacting just to name a few.
- The advances in computer science that shall be considered are, for instance, the power of the **parallelization** techniques.
- KRATOS is prepared to connect different applications, in this sense it will be easier to create a **DEM-FEM application.**

Also, a more versatile developing interface like KRATOS was needed in programs like DEM-Pack. Being part of KRATOS helps in the code improvement and makes the cooperation between different developers easier. Many of the technical problems that could appear during the implementation can be solved by the help from the KRATOS community that may have faced similar problems performing similar solutions.

The philosophy of CIMNE nowadays is to gradually transcribe or rewrite every code used in the centre onto the KRATOS framework due to its numerous advantages. The past years every developer or group had been creating their own code with their preferred programming language and structure. This way the final result was a good program from a user level but not a code ready to be improved, extended or revised for others in a developer stage.

KRATOS DEM-Application is the result of this concern, to rewrite and improve a more ambitious code for the Discrete Element Method to substitute in long-term the currently used DEM-Pack.

6.2. Current development and collaboration

As KRATOS is an open-source platform the collaboration between different institutions and particular developers it's one of its benefits.

Currently the DEM-Application is being developed by the author of this work, *Miquel Santasusana Isach*, and other doctoral and post-doctoral researchers in CIMNE: *Miguel Ángel Celigueta Jordana*, *Nelson Lafontaine*, etc.

Fortunately, the power of KRATOS has awakened interest in many other research institutions and some others have joined the KRATOS discipline and so the DEM-Application team has been increased. This is the case of the Cuban CIMNE Classroom[1] *UCLV-CIMNE,* and also The Institute of Mechanics, Chinese Academy of Sciences *IMECH,CAS.*

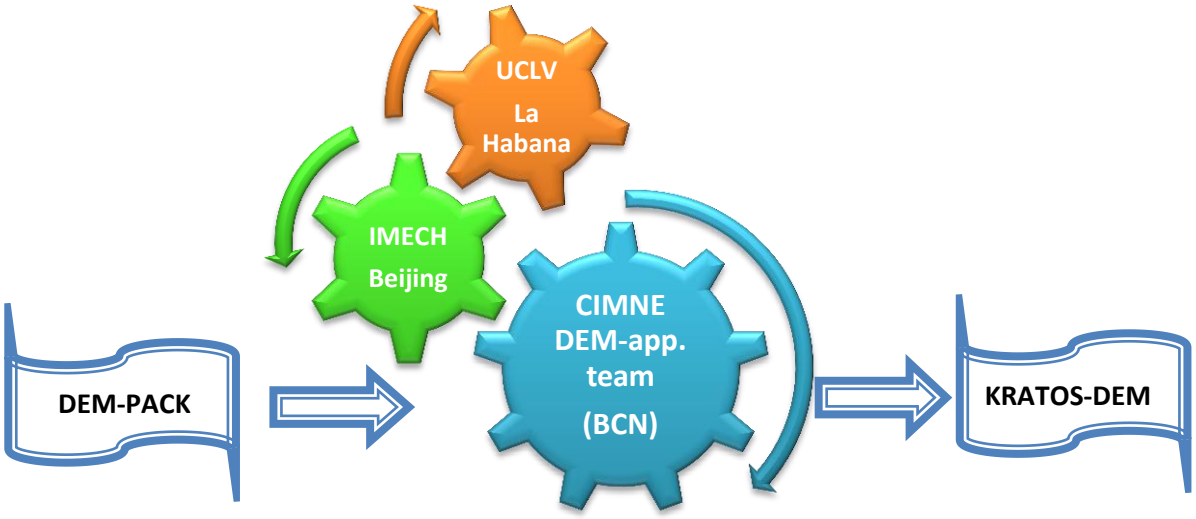


Figure II. 3 KRATOS DEM partnership

[1] *The CIMNE Classrooms are physical spaces jointly created by CIMNE and a University for the development of training, research and technology transfer activities.*

SEED: The first implementation was done by an ex-doctoral researcher on CIMNE who started, in 2011, building the code in KRATOS environment taking the idea from the existing CIMNE's DEM code: DEM-PACK. The new team has started working in it since February 2012.

CIMNE DEM-APP. TEAM (CATALUNYA): Several works are being done by the core team. First of all, a step-by-step development of the very basics of the method has been done; also the implementation of new different utilities and case possibilities. Secondly, the validation of each one, taking the ideas from the best current programs abroad while being validated with the most serious theoretical studies; in parallel, our own new formulations are being deduced. Also some research is being done in new ideas and usages of DEM, particularly in terms of continuum simulations. Finally the Barcelona team manages all the collaborations and tries to merge the incoming implementations and does the corresponding feedback.

UCLV (CUBA): This CIMNE classroom has developed its own DEM code and they have sophisticated integration schemes, search contacts routines, particle type implementation, etc. They help the CIMNE team in improving the application's abovementioned utilities. Currently the UCLV is developing bio-medical applications for the DEM with advanced cluster grouping techniques.

Figure II. 4 Discrete elements from tomographies, UCLV Cuba

IMECH, CAS (CHINA): IMECH has a powerful DEM code called CDEM that performs the simulation of not only discrete problems but also DEM-FEM coupled problems. The research centre helps KRATOS-DEM implementing basic functions for contact force calculation and integration algorithms. The objective is to develop also a DEM-FEM method to KRATOS-DEM as it is explained on part I of this document.

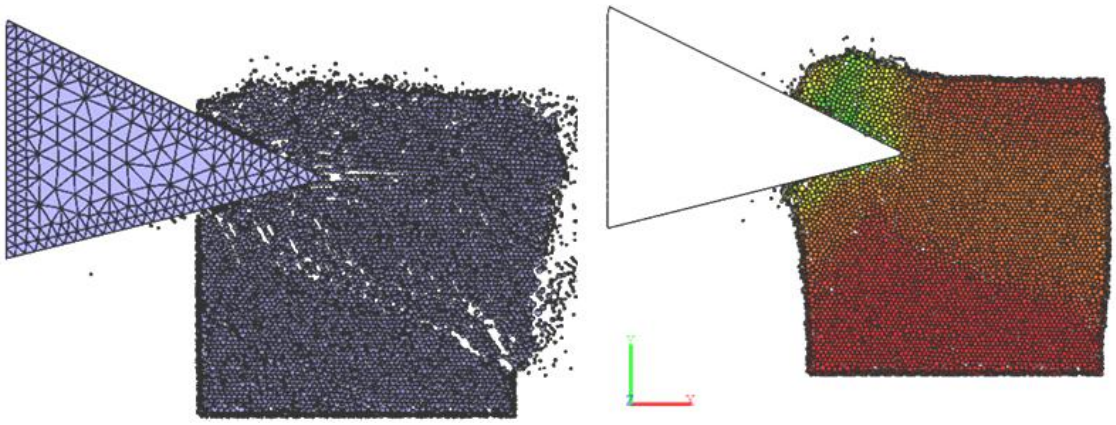
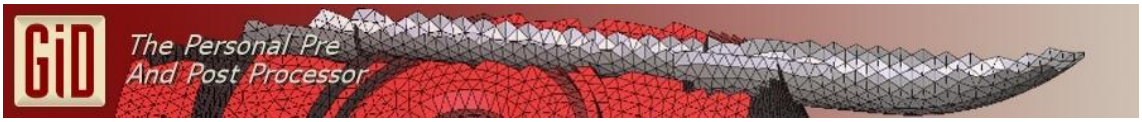


Figure II. 5 Extract from IMECH works

7. GRAPHIC INTERFACE

7.1. GiD Pre and Post Processor



KRATOS applications are highly compatible with graphical interfaces. The one that CIMNE has been using for KRATOS and many other *ProblemTypes* is GiD interface which has its origins on CIMNE itself. In parallel a new specific graphical interface is being developed for the KRATOS package in order to improve the problem definition of the different applications that KRATOS supports. In the present section the GiD interface will be introduced as it is the currently used one and in next sections the specific implementations and the usage of this software for the development and testing of DEM-Application is explained in detail.
GiD is a versatile multipurpose software that provides a graphical support to the pre-process and the post-process stage.

7.1.1. Pre-Process:

This stage consists on setting the geometry and the data for the problem definition (forces, movements, properties...) as well as imposing boundary conditions and the calculation options. After the problem definition GiD also dispose of different mesh generators for the FEM (or others) problem calculation.

Geometry: GiD Pre Process is a CAD system that features the widely used NURBS surfaces (trimmed or not) for the geometry definition. Typically geometrical operations can be used as transformations (translations, rotations, etc.), Boolean operations in surfaces and volumes. A complete set of tools are provided for quick geometry definition.

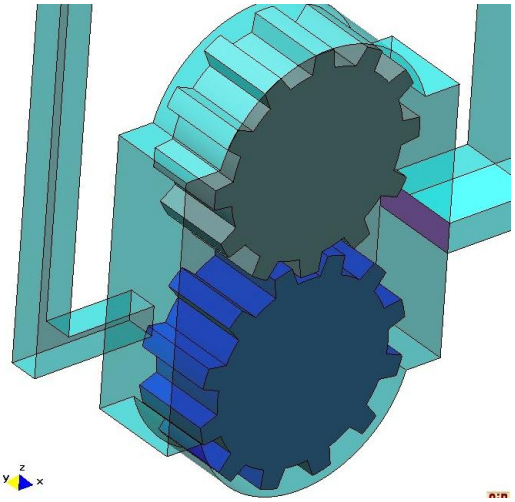


Figure II. 6 GiD Geometry editing example

ProblemType:

The user can load to GiD a specific *ProblemType*. When a *ProblemType* is loaded, some specific options from the particular case appear, such as conditions or calculation parameters. The geometry will be used by the *ProblemType* to effectuate the physical calculation.

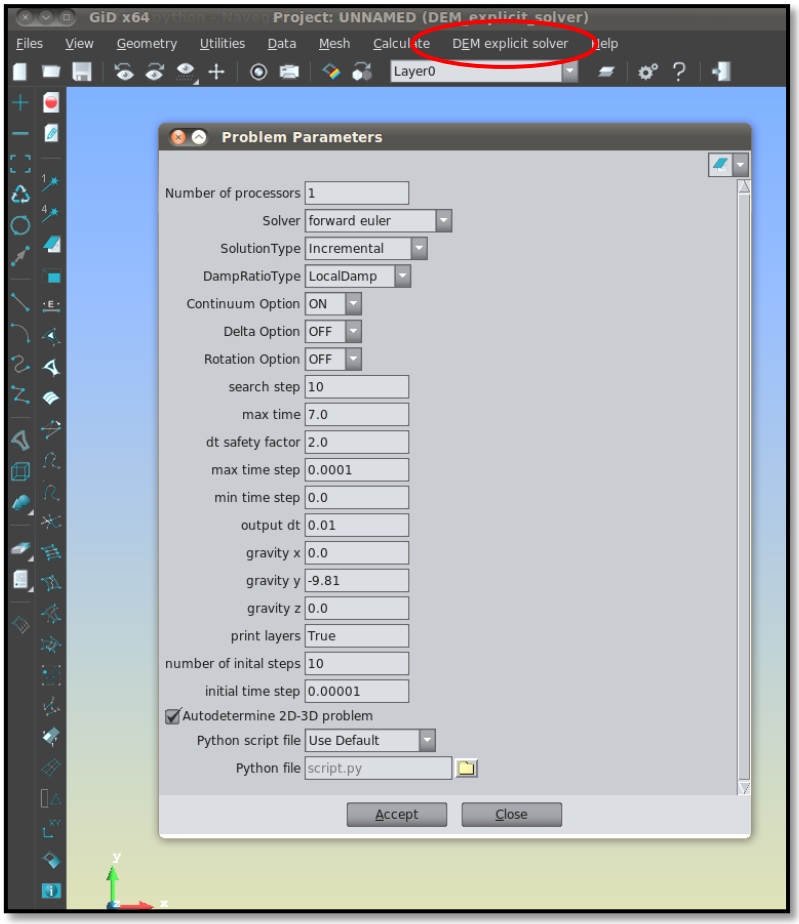


Figure II. 7 DEM_explicit_solver ProblemType Options

7.1.2.Calculation Process:

Once the geometry is drawn and the conditions, loads and parameters that each *ProblemType* requires are defined, the calculation shall be done. When the Calculate button of GiD is used, the *ProblemType* reads the geometry, applies the conditions and so one and calculates the problem. It has to be remarked that GiD doesn't calculate, it triggers the *ProblemType* inner calculation.

7.1.3.Post-Process

GiD passes from the Pre to the Post with a simple button click. The options that the user has in each part of the program are different; In the Post there is numerous utilities aiming to plot and analyse the results.

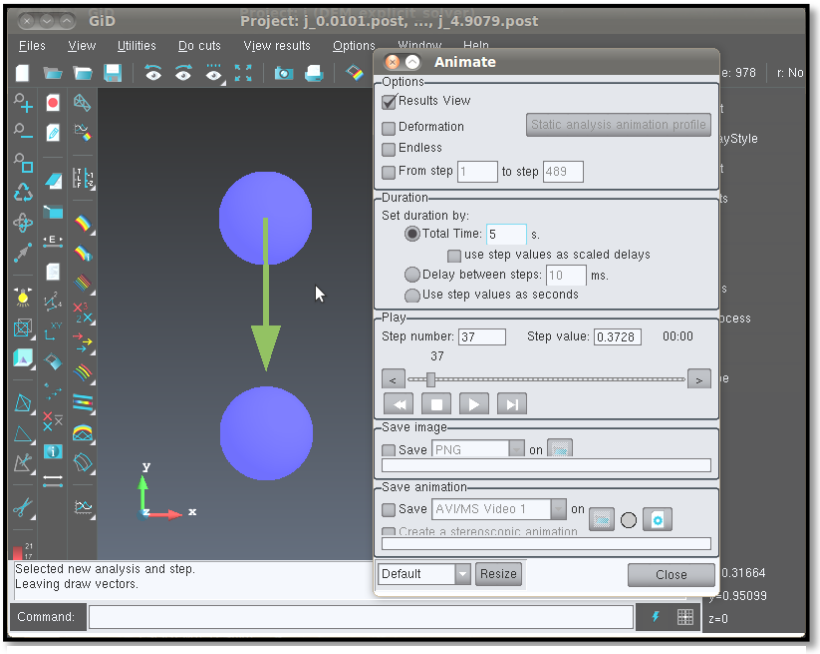


Figure II. 8 Post process screenshot. Animation on results.

In the View Results tab many types of visualization are available to represent the accessible output data. The *View Results & Deformation* window permits to choose the representation of the results either on the original mesh or on the deformed one, selecting a suitable scale. On the *View results* is possible to select first of all a type of representation; the available representations are Display Vectors, Contour Fills, Contour Lines, etc depending on which one the user considers that is the best for every different type of result. It's frequent to represent the vector magnitudes such as a force, velocity or displacement in a vector display and the scalar magnitudes like von misses stress or strains, energy, etc in colour scales.

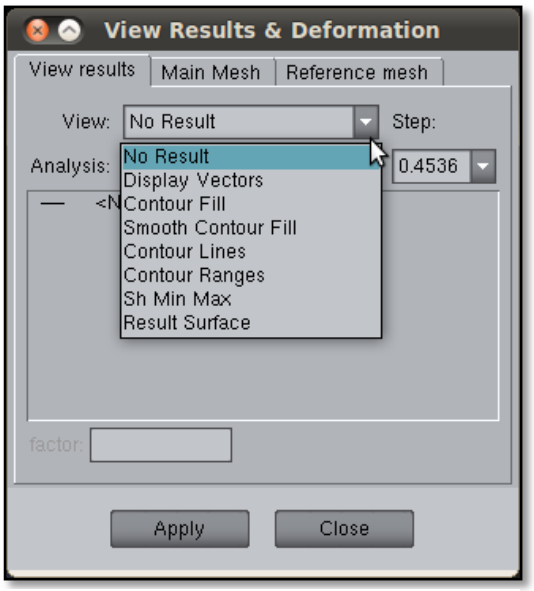


Figure II. 9 Type of visualization selection

Once the type of visualization method is chosen, the analysis variable to display is selected. In the next figure there are three different magnitudes, let's say displacement, force and velocity available for the visualization. In general the analysis program is the one who stores and prepares the output values for the GiD PostProcessor to display. In the particular case of the KRATOS DEM-Application it's easy to select which nodal variable (the ones which have information in every output time step) has to be loaded in the output.

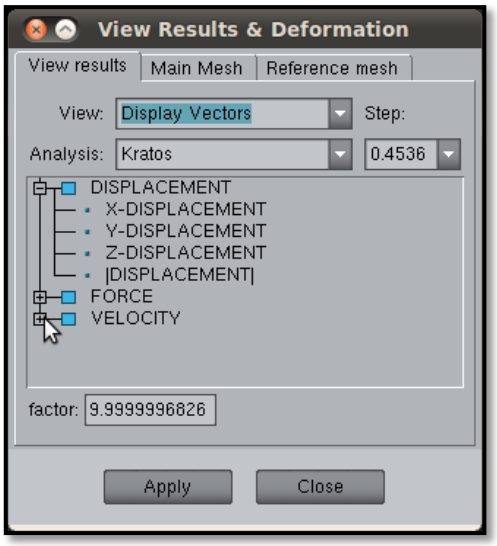


Figure II. 10 Selection of the magnitude to be shown

As the reader can see from the figure, vector magnitudes such as the displacement can be plotted in any component and also its scalar modulus.

7.2. Implementations done in the Pre-Processor for DEM-App.

One of the first meetings that the author of this work, M. Santasusana, and its supervisor and partner of the DEM-Application, M.A. Celigueta , had when the project began, served to set the first priorities for the new implementations that had to be done in the DEM-Application. The resulting lists noted down was a pretty large list indeed, nevertheless the team realised that the first action to be done was getting a quick and user-friendly Pre-Processor in GiD.

Why a improving the Pre-Processor first? A useful and comfortable Pre-Processor is necessary in order to test more quickly any implementation that is incorporated to the application. For every new damping, every new contact criteria, every material parameter, any new implementation, several test have to be created with a pre-processor to analyse the results

and check for errors or malfunctioning. So, the starting point was improving the Pre-Processor interface generated by the DEM-Application *ProblemType* in GiD. Next, the original state of the Pre-Processor is presented and the improvements done within the framework of this work.

7.2.1. Inherited Pre-Process

The Pre-Process interface that was available when the DEM-Application began was very simple. It had a very few options and what is more important; it was very slow and tedious to create a simple example. In the inherited Pre, points were used to represent the particles; All the assignations, included the radius, were needed to simulate the sphere or circle. In the Post, this had to be taken in account in order to represent the spheres instead of points.
The properties assignment to the spheres or circles had to be done property by property and the material concept was not defined. This made easier to forget the assignment of some condition or properties and so yield problems or errors during the calculation. That is the main reason to create a new useful and user-friendly *ProblemType,* to ease and accelerate the problem definition.

7.2.2. New DEM-Application Pre-Process

There have been changes done in the following aspects of the pre-process:

- Geometry- Mesh definition.
- Condition assignment –"Nodal Values"
- Problem Parameters
- Material assignment.

First of all, the user has to dispose of the DEM_explicit_solver *ProblemType* (the current name for the DEM-Application), which can be loaded in GiD by the Data -> ProblemType tab.
Once the problem is loaded, a new drop down tab appears in GiD with five different options.

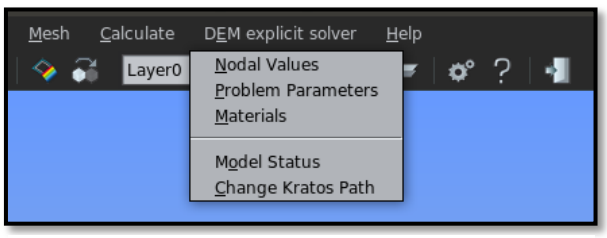


Figure II. 11 DEM_explicit_solver menu

Geometry and Mesh definition:

The geometry edition has been improved in order to make the problem definition easier and versatile. In the new *ProblemType* spheres/circles can be generated on any geometry: points, lines, regular and irregular surfaces and volumes. Some processes in *TCL* language has been coded in order to create sphere or circle elements on these geometries.

The development of the DEM-FEM application was in parallel with DEM application; in that sense the *ProblemType* was designed to be applicable to DEM-FEM also. That would require having geometries meshed with finite elements and geometries meshed with discrete elements, i.e. spheres and circles. What GiD really does, when meshing a geometry defined by the user, is meshing it with the chosen criteria: regular or irregular finite element meshes. Afterwards it automatically applies some operations on the mesh defined by the developer, following the instructions of a code written in *TLC* programming language. Here is where it has been specified to create spheres or circle on those finite element meshes where the user had applied a radius condition. The sphere or circle is created in the centre of the finite element created by the mesh.

Example with a line (geometry) meshed with 4 regular linear elements. The second one has a radius condition applied on it.

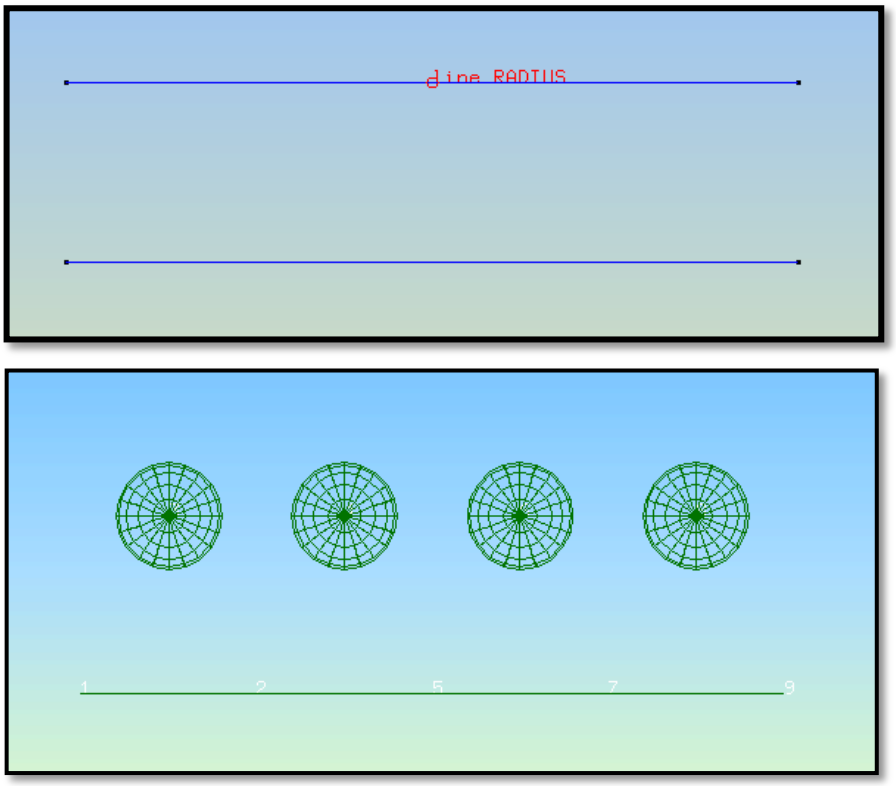


Figure II. 12 Example geometry – mesh in line entities

Apart from the commented regular and irregular mesh criteria that permit choosing different type of elements: tetrahedron, quadrilateral, triangles, rectangles, etc. there is a sphere and circle mesher available that had been created especially for the *DEM-Pack* programme. This mesh criterion directly creates spheres on a volume or circles on a surface with some predefined options in the mesh options menu. In this case that geometry needn't to have a radius condition applied on it; the mesher creates different sizes of spheres on the volume (or circles in the surface) depending on the options of the mesher.

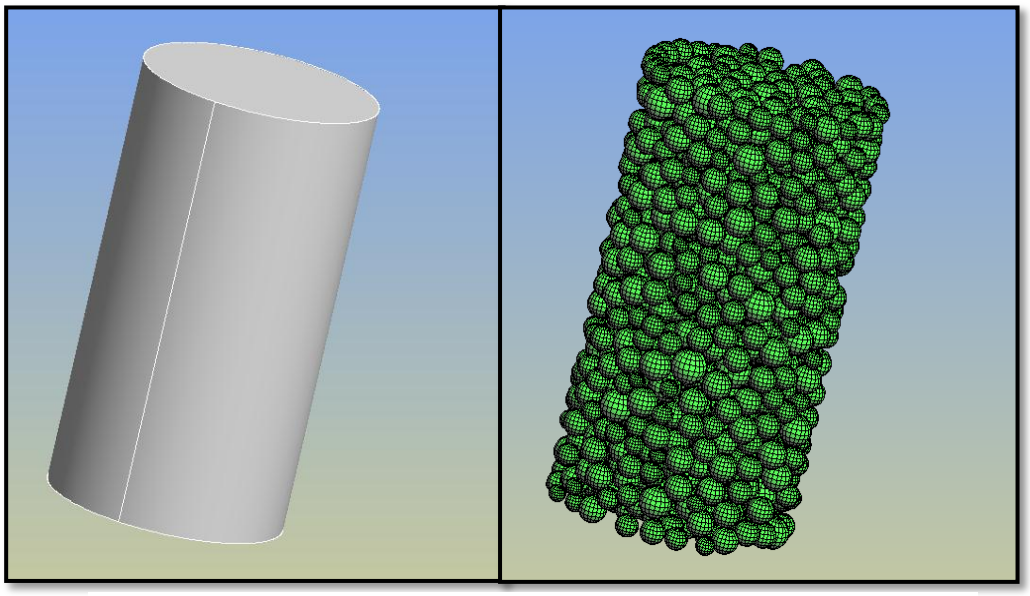


Figure II. 13 Cylinder meshed with GiD sphere mesh generator

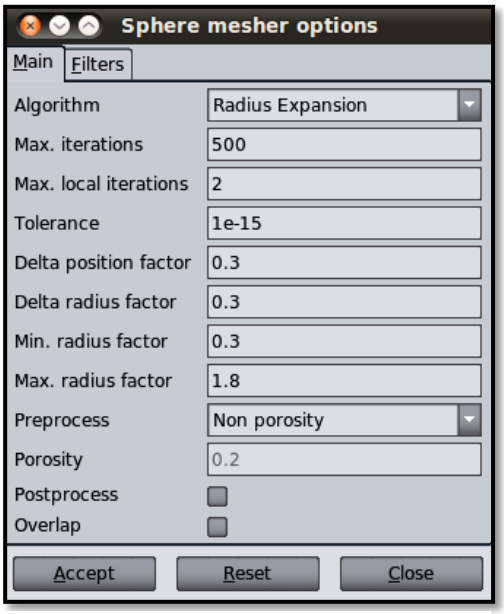


Figure II. 14 GiD Sphere mesher options

Next, an example is presented where different geometries in 2D and 3D are meshed, some of them have a radius condition applied while others not. Also here the three types of mesh criteria commented above are applied on the geometries to mesh.

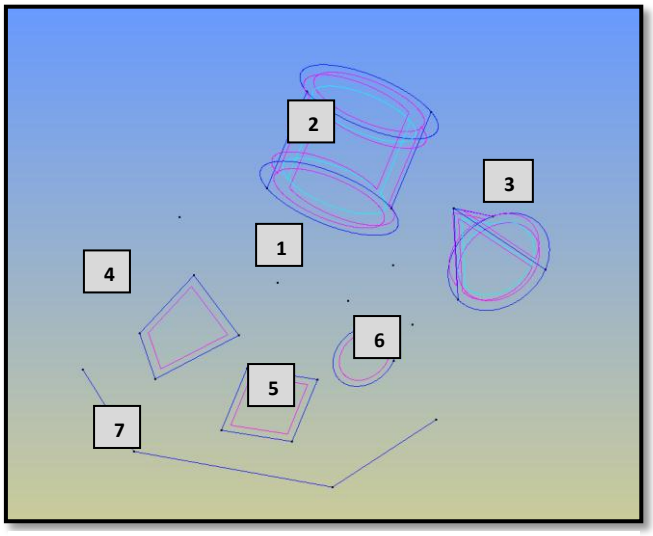


Figure II. 16 Miscellaneous geometry definition

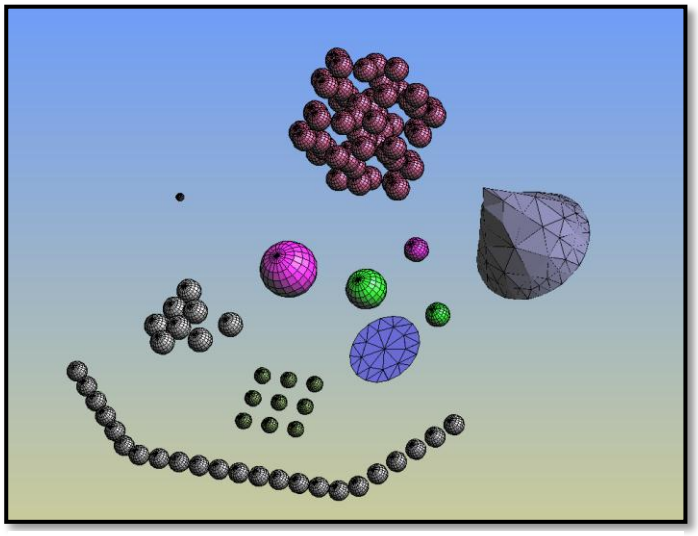


Figure II. 15 Mesh resulting from miscellaneous geometry

Id	Geometry	Radius condition	Mesh
1	Point	ON	Regular
2	Volume	ON	Irregular
3	Volume	OFF	Irregular
4	Surface	ON	Irregular
5	Surface	ON	Regular
6	Surface	OFF	Irregular
7	Line	ON	Regular

Condition assignment – Nodal data.

Currently there are only two conditions that can be applied to the geometry elements: An imposed velocity and an assigned radius. The assignment can be done to any geometry, point, line surface or volume and the condition will be transferred to the discrete elements that will be created by the mesh on the selected entities. For example, applying the radius condition to a line and meshing it, several spheres of the selected radius will be created in the line.

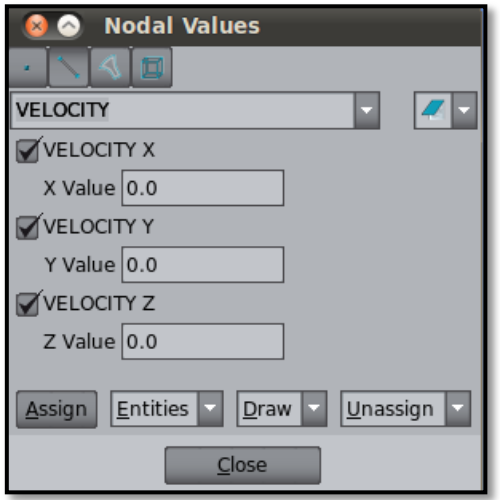


Figure II. 17 Conditions assignment – Nodal Values

The velocity condition permits establishing an initial value for the velocity in the three components (X, Y, Z) on a discrete element. It also can be defined as a fixed value if the body has to have, on any of the three components of velocity, an unchangeable value.
In newer versions of the *ProblemType*, there will be the applied force and applied moment condition implemented (it is currently being implemented).

Problem Parameters:

Here, general parameters of the problem are defined.

- Number of Processors: In a multi-core computer it may be interesting to calculate the problem in parallel to save time. Many of the loops that are implemented in the DEM-Application have been parallelized with OpenMP (see *section 8.5.1 Parallelization*). This option is equivalent to define the environment variable: OMP_NUM_THREADS = X. At a certain number of particles it is worth it to parallelize the code.
- Solver selection: Here, the integration scheme is selected for obtaining the displacements of the particles from his accelerations, given a time step. There are currently the Forward Euler, the mid-point rule and the constant average acceleration (Newmark beta-method) schemes. Some more sophisticated like Runge-Kutta and so on are being developed. See Part I: Section *2.3* Integration *of the motion laws.*

- Solution Type: It has also been commented in the first part that the normal forces at a given time step can be calculated with an *absolute* way or in an *incremental* way. It might have more interest for the developers than the common users of the application.

- Damping Ratio Type: There are two possible damping types that can be applied. The viscodamping that acts when a contact force acts upon a sphere; the damping magnitude is proportional to the velocity up to a constant defined here. The local damping is a damping that acts lowering the unbalanced forces proportionally to their magnitude. It is especially interesting to apply this damping in quasi-static problems, i.e. continuum simulating problems.

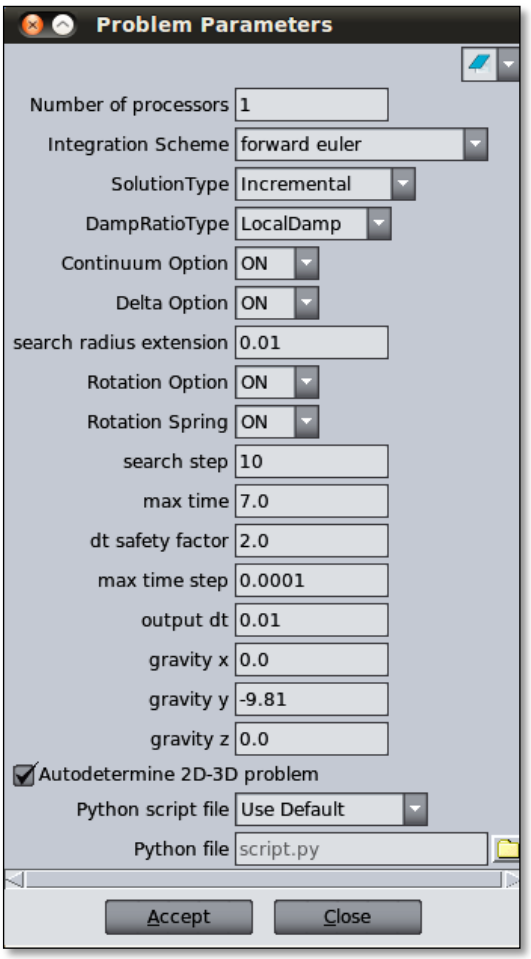


Figure II. 18 Problem Parameters menu

- Continuum Option: These specific options like the Delta Option and Rotation Option have a switch that activates or deactivates the set of functions needed to take in account these problems. When simulating the continuum with the DEM-Application the program becomes more expensive because the problem is more complex; that is the reason why is recommendable to disable these options if is not being used.

  Delta Option: Following the same idea than for the Continuum Option, disabling this option will save some time of the calculation; otherwise enabling it will permits to define a geometry with a mesh of spheres where the initial indentations given between the sphere are considered as passive indentations, producing no repulsive contact force. The opposite also holds; for a little separation between particles, they will be considered neighbours and can represent an unfractured continuum if the previous option is activated. When the Delta Option is activated, a new option appears for the radius extension. In This extension of the radius for the neighbouring search is defined as a percentage of the particle's radius.

- Rotation Option: This works similar to the previous options, activating or disabling the rotation to our problem and also requiring more time in the calculation. Once this option is activated the user can define whether the rotational spring is activated.
- Search step: Here the user selects how many steps are desired to wait until a new neighbour search is computed. Remember that the neighbouring search is usually the most expensive operation in terms of computational cost. For quasi-static problems this value can be considerably higher than in dynamic problems.
- Time parameters: Here the following can be defined: the total time of the calculation, the safety factor to apply on the critical time step calculation, the maximum desired time step (the program will take the maximum one if it is lower than the critical one) and the output time step that defines the time between the exportation of results that will be printed in GiD for instance.
- Dimensions: GiD automatically detects if the problem is 2D or 3D depending on the definition of the geometry. The option to choose manually between 2D and 3D calculation has been implemented in order to have the possibility to choose if the calculation is performed with a 2d cylinder or 3d sphere in a 2D domain.

Material Assignment:

This is completely new for any DEM *ProblemType* in CIMNE. The inherited *ProblemType* and the DEM-Pack *ProblemType* needed to assign each property, one by one to the different entities or group of entities.

Now, the user can define its own materials that include all the necessary physical properties and assign with only one action all these properties at once.

- Continuum Group: This is one of the innovations of this application. A group identifier (0, 1, 2, 3...) can be assigned to any entity as a label. Entities with the same material properties can belong to different group in order to have a “non-cohesive” interface or otherwise two different materials can compose the same continuum “cohesive” body. The group zero is reserved for the materials or particles that won’t resist any tension between particles. See 8.5.6 Continuum Simulating Option.

- Particle density: This will determine the mass of the discrete element.
- Young modulus: This is the parameter that has more influence in the stiffness of the contact springs. In an ideal case it would take the value of the representing material but in order to save calculation time, a smaller value can be adopted as it is explained on Part I: Section 0
- *Relative importance of the accuracy on the stiffness* value
- Poisson ratio: This also affects the determination of the stiffness for the springs. The correct value shall be used.
- Cohesion: This value determines the ultimate admissible shear stress (in terms of force). If this value is exceeded, the contact becomes frictional.

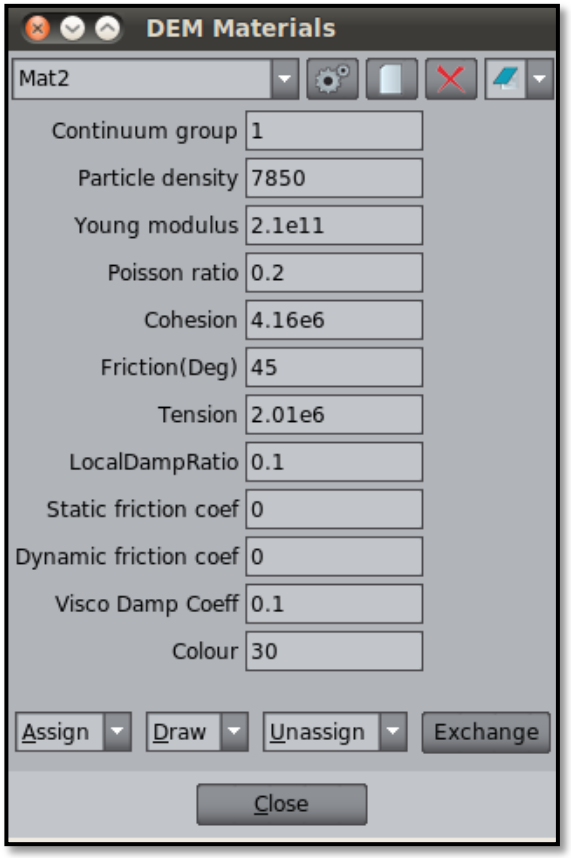


Figure II. 19 DEM Materials selection

- Friction: This value corresponds to the angle of internal friction of the material; the calculation of the friction forces is obtained by means of its tangent.
- Tension: Determines the ultimate tensile strength that the contact can hold in terms of forces (Newtons).
- LocalDampRatio: Affects on the persistent damping that acts whenever an unbalanced force is present onto a body.
- Static friction coefficient: To calculate the static friction force opposing the movement.
- Dynamic friction coefficient: For the friction forces when slipping exists.
- Visco Damp Coeff: This applies to the coefficient for the viscous damping.

7.3. Implementations done in the Post-Processor for DEM-App.

As it has been commented in previous sections, GiD Post-Process performs the visualization of the results that KRATOS obtains during the calculation.
There have been only a few modifications done in the Post-Process but there is still some work to do in improving this aspect as it will be commented later on.

7.3.1. Inherited Post-Process

First of all, as it has been exposed in the previous section, the point entity has been used to define the particles' geometry. This was a problem for the representation in the Post where just points are represented instead of the real spheres with the corresponding radius. One feature that GiD offers to the user in order to simulate the sphere is to enlarge point by point the representation size until it has the required radius. This has to be done manually and can incur in some problems in the animation.

As it has been already commented, the Post permits visualizing determined output variables that KRATOS stores during the calculation. In the inherited code the three components of the displacement, the velocity and the unbalanced force were exported for the viusalitzation; GiD is able to represent with different techniques, as it has been said, any component of these variables and also the resultant vector norm.

7.3.2. New DEM-Application Post-Process

Due to the introduction of the spheres and circles entities to the mesh of the Pre-Process, one of the implementations needed was already satisfied; now, the visualization does not depend anymore on the enlarged size of the represented point; it captures the sphere or circle real assigned size.

Advanced representation features:

- Fracture type output: It has been commented in Part I (see *3.5 Failure of the contacts, plasticity and damage*) that it can be interesting to store the dominating fracture type when a particle is detached from the continuum. KRATOS permits representing any

variable that can be stored on the nodes; as it is explained on section *7.1.3 Post-Process, GiD* can represents in many ways, for example setting a range of colours.

- Utility for the rotation visualization: When the rotation option is applied the results may vary for the groups of particles that now will be affected by the moments; also the single particles will rotate due to the moments and the results will vary indeed, though, from the original visualization it is not possible to determine whether it is a rotating or a not rotating particle. This has been solved in a very illustrative way in the CDEM program where we visualize a radial stroke in the particle that rotates accordingly to the rotation of the particle. This is an original solution for representing the 2D effect. For the 3D effect, the DEM team is developing an X, Y, Z local initial axis that will rotate following the motion of the particle.

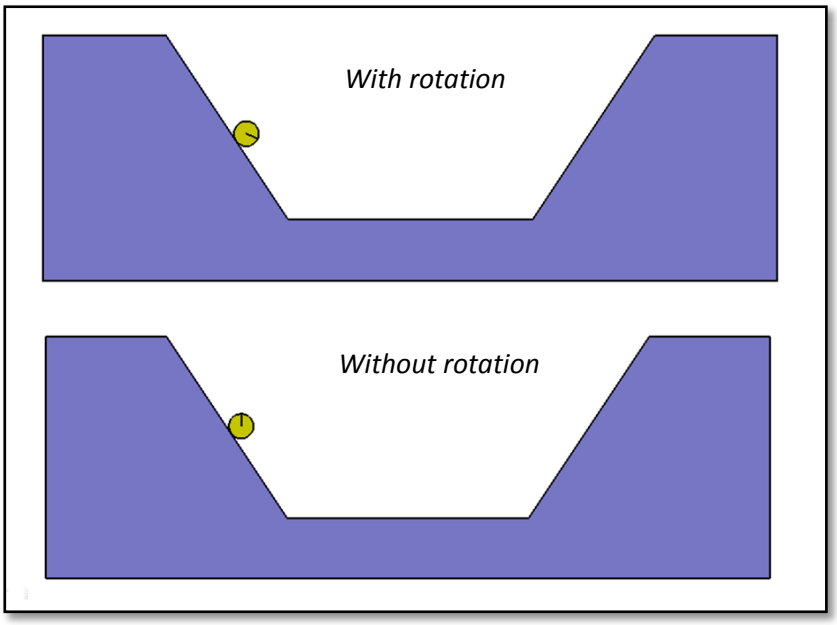


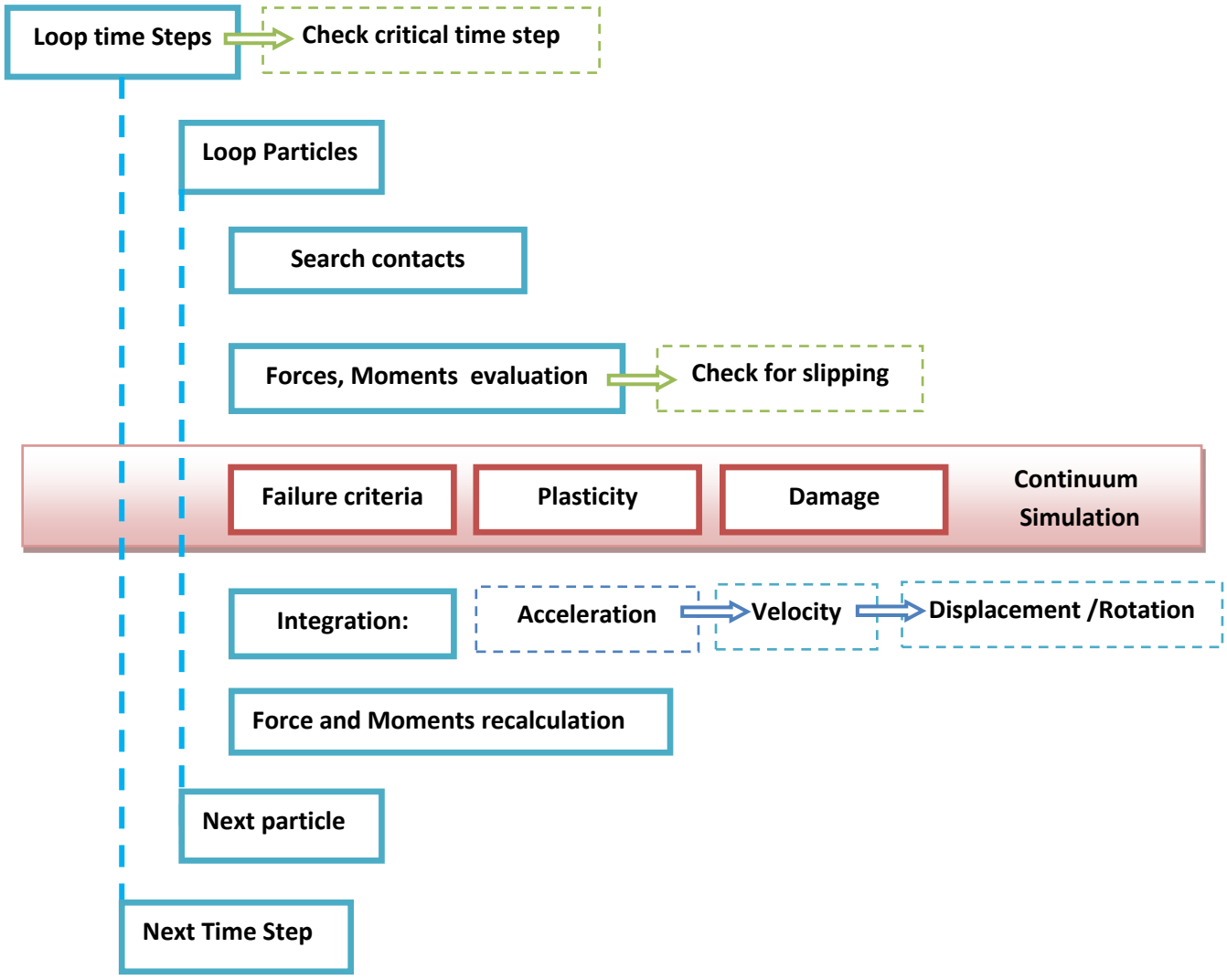
Figure II. 20 Example of rotation visualization option in 2D (CDEM)

8. IMPLEMENTATION IN KRATOS

The objective of this section is to introduce the interested lector to the basic structure of the DEM-Application that has been implemented in the KRATOS platform during the course of this work. Also a briefly explanation of the different files can be found for any developer interested in the functioning of KRATOS and the approach for the DEM-Application

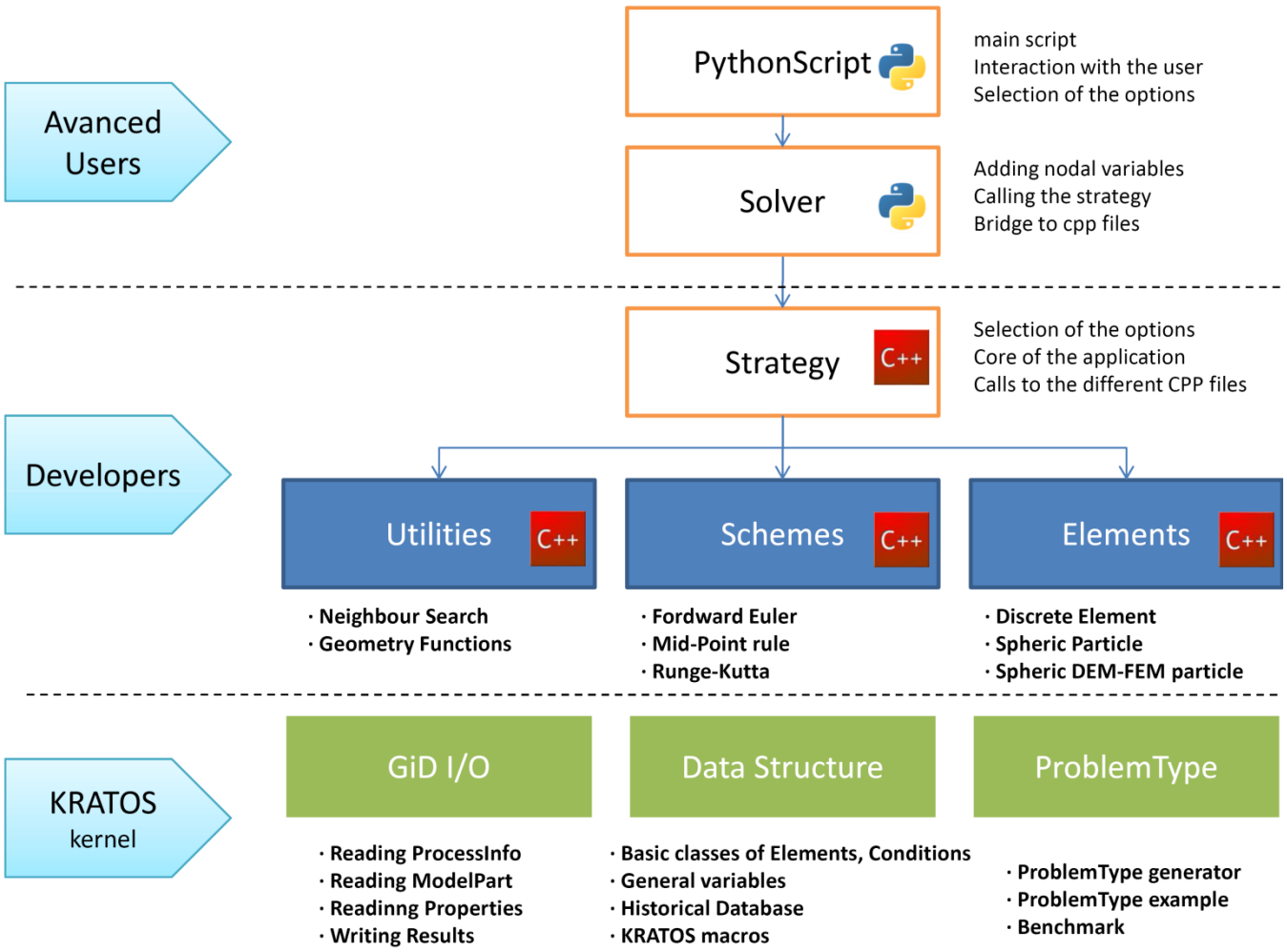
8.1. Basic computational sequence for a discrete element code

The first algorithm was proposed by *Cundall* [4] and it doesn't differ so much from the DEM-Applciation one . The code developed as most of the commercial codes that use the Discrete Element Methods, has a basic sequence calculation that is roughly based in the same steps.

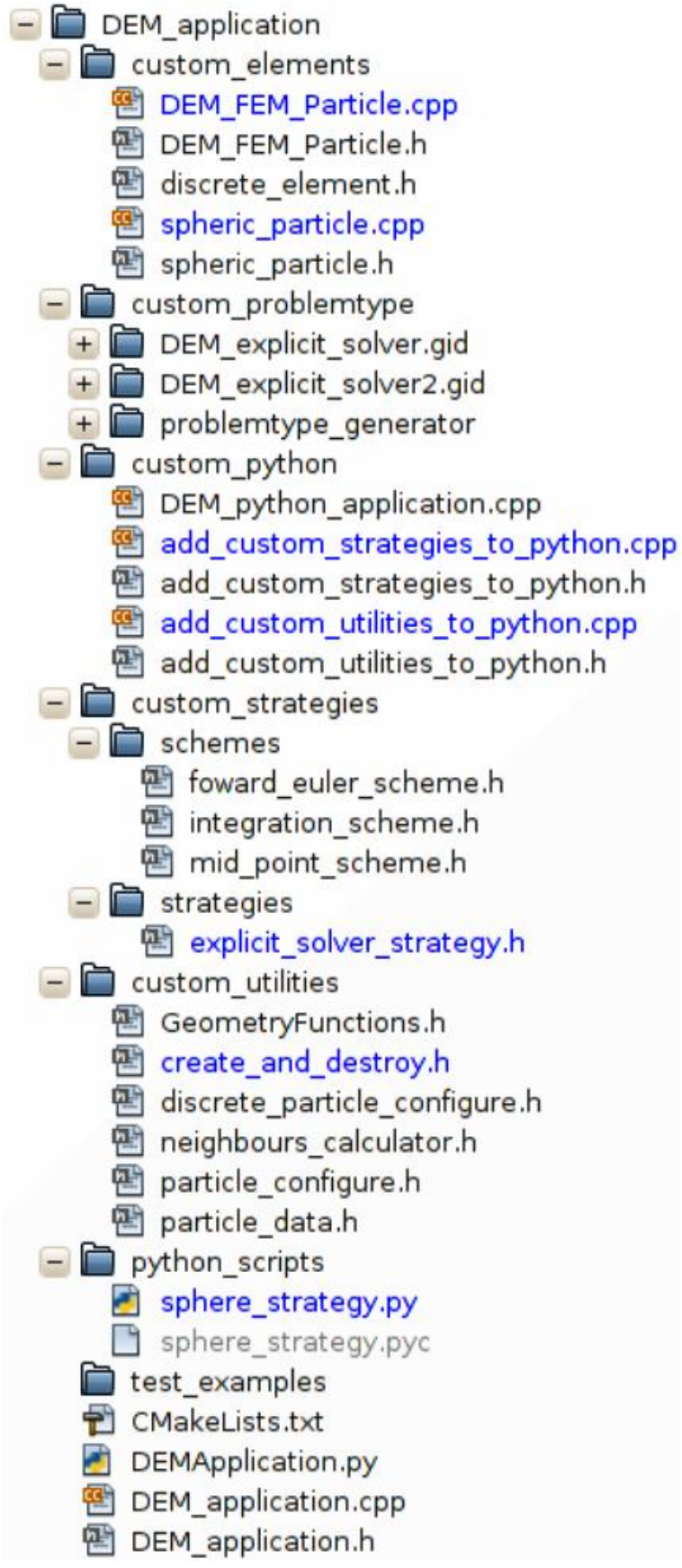


8.2. Basic structure of the DEM-Application

It has been commented in the section 5. *KRATOS-multiphysics platform* that the platform is oriented to implement FEM-based applications; some of the applications that are already implemented are: *Incompressible fluid App., Structural App., PFEM App., Meshing App., ThermoMechanical App, kElectrostatic App., e*tc. However, a Discrete Element-based method has been also implemented, without problems, in the KRATOS platform. The differences are minor comparing structure of our application and the one from other applications because one of the principal recommendations when implementing in KRATOS is trying to keep the same format in order to be more accessible to the other developers. The other fundamental reason is to be able then to couple easily two different applications. In that sense, a complete restructuration of the DEM-Application was made when the DEM-Application project started; there have been an intense work done on it to match the application to the dictated structure that would make it much more versatile



8.3. Folders and files in the application:



Elements available

Spheric_DEM-FEM_particle derives from Spheric_particle and has the extra features for the DEM_FEM problem.

ProblemType

A folder ready to copy to GiD and a problemtype generator that creates these packages with the new implementations

CustomPython

These files translate the strategies and the utilities defined in C++ language to be able to call them from python scripts.

Schemes

Two explicit integration schemes available, this clases derive from a base class *integration_scheme.h*

Strategy

The main script that calls subsequencially the shcemes and the functions on the elements during the loop.

Utilities

Geometric functions predefined, the neighbouring search function, the configuration of the particle, etc.

Python Solver

Python interface where the main function of the strategy such as Initialize or Solve are called. Adding the nodal variables.

DEM_application

Here the variables that will be used are created and registered to python and KRATOS.

8.4. Explanation of the main files:

Here, a brief explanation of the files and the main functions of the DEM-Application are presented with the objective to give a general overview of the capacities and functioning of the program. For the reader interested in the details of the code, developers and advanced users, in the annex the code for the DEM-application is attached.

8.4.1. Advanced users:

The advanced users of the KRATOS applications won't enter inside the C++ encoding of the application but they would use the python files to code, with this basic language, some instructions that permit a rapid interaction and modifications with no needing to compile.

When an engineer is running a case, frequently would have the necessity to apply an extra force to some particles of the domain, to change the material of several bodies, to extend the simulation time to eliminate, create or modify some existent discrete elements, etc. Most of these useful operations can be done in the following python scripts avoiding the needing to modify the created example with the graphical interface.

Very often the user needs to modify punctually the coordinates of an element, its properties, eliminate one or create a new one. This can be done quickly modifying another file with the extension "mdpa"; this file is auto-generated by the graphical interface.

***.mdpa:** mdpa is the extension of the file that contains all the information of the ModelPart that GiD creates from the geometry, properties and conditions definition. This file is translated automatically by means of the GiD I/O module which will create the different elements and assign the corresponding properties and values of the elemental and nodal variables interpreting the information on this file.

Nodes List: listing of every node with its three coordinates.

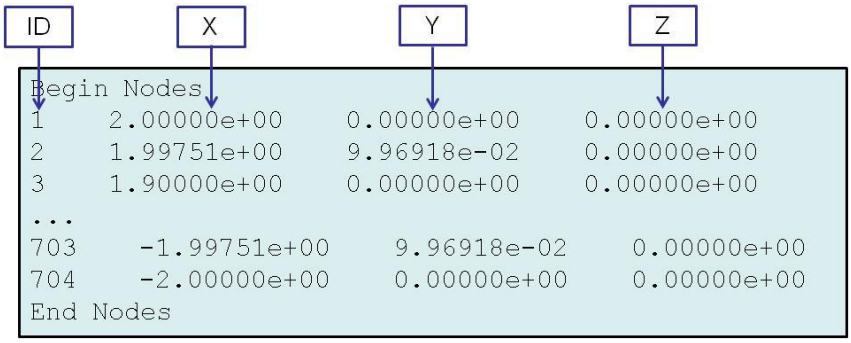


Figure II. 21 MDPA example, nodes list.

Elements List: listing of every element with its connectivities (nodes assigned).

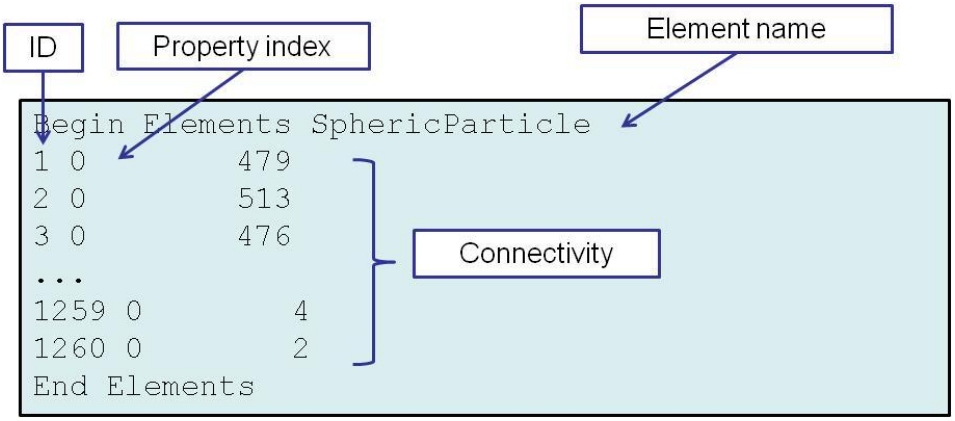


Figure II. 22 MDPA example, list of elements.

Nodal Data: for every variable predefined in GiD, the list of the nodes, the condition of fixity on the variable and the value is presented.

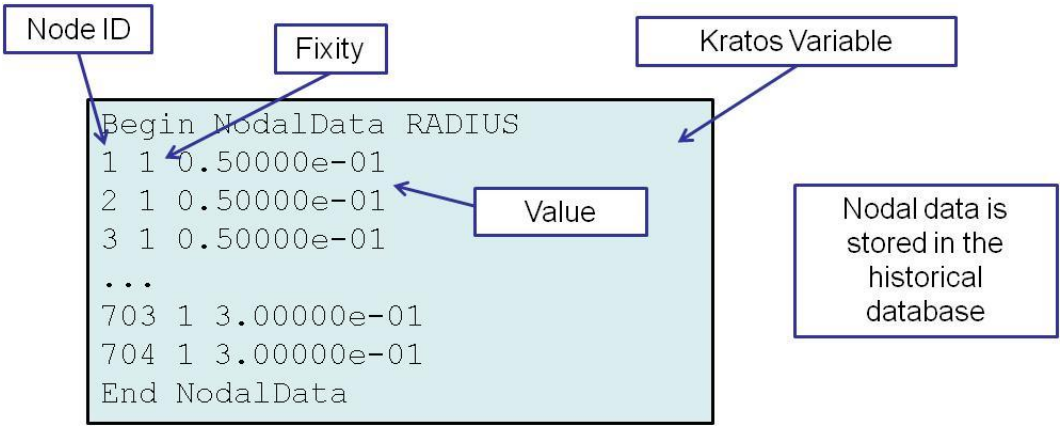


Figure II. 23 MDPA example, example of nodal data.

The advanced users of the application can find it useful to edit this file to change some options and determine special conditions manually without needing to modify the geometry or the properties using a graphical interface, such as GiD.

PythonScript: This file, usually named as *Script.py* written in Python language is the main script that the users launch to trigger the calculation of the problem. This file is included in the ProblemType folder of the application that any user of the application (basic or advanced) would have linked with the Pre/Postprocessor software, for example GiD.

```
import DEM_explicit_solver_var
import time as timer

from KratosMultiphysics import *
from KratosMultiphysics.DEMApplication import *

#defining a model part for the solid part
my_timer=Timer();
solid_model_part = ModelPart("SolidPart");
#########################################

#introducing input file name
input_file_name = DEM_explicit_solver_var.problem_name
#reading the solid part

gid_mode = GiDPostMode.GiD_PostBinary
multifile = MultiFileFlag.MultipleFiles
deformed_mesh_flag = WriteDeformedMeshFlag.WriteDeformed
write_conditions = WriteConditionsFlag.WriteConditions
##selecting output format

import sphere_strategy as SolverStrategy
SolverStrategy.AddVariables(solid_model_part)

gid_io = GidIO(input_file_name, gid_mode, multifile, deformed_
model_part_io_solid = ModelPartIO(input_file_name)
model_part_io_solid.ReadModelPart(solid_model_part)

solid_model_part.SetBufferSize(2)

##adding dofs
SolverStrategy.AddDofs(solid_model_part)

#creating a solver object
solver = SolverStrategy.ExplicitStrategy(solid_model_part, 3);

solver_strategy = DEM_explicit_solver_var.Solver
if (solver_strategy == 'forward_euler'):
    solver_id = 1
else :
    solver_id = 2

solution_type = DEM_explicit_solver_var.SolutionType
if(solution_type == "Absolutal"):
    type_id = 2
else:
    type_id = 1

damp_ratio_type = DEM_explicit_solver_var.DampRatioType
if(damp_ratio_type == "ViscDamp"):
    damp_id = 2
else:
    damp_id = 1

gravity = Vector(3)
gravity[0] = DEM_explicit_solver_var.gravity_x
gravity[1] = DEM_explicit_solver_var.gravity_y
gravity[2] = DEM_explicit_solver_var.gravity_z
final_time = DEM_explicit_solver_var.max_time
output_dt = DEM_explicit_solver_var.output_dt
safety_factor = DEM_explicit_solver_var.dt_safety_factor
number_of_inital_steps = DEM_explicit_solver_var.number_of_inital_steps
initial_time_step = DEM_explicit_solver_var.initial_time_step
introduced_max_dt =  DEM_explicit_solver_var.max_time_step
min_dt = DEM_explicit_solver_var.min_time_step
dt = DEM_explicit_solver_var.max_time_step
```

COMMENTS

The first part defines the "includes" necessaries to link the KRATOS folders and the DEM-App. The model part is created and the input file name is captured.

Here the solver strategy is chosen, the variables are added to the solver.

Setting the buffer size: This determines the historical database of the problem. By setting the value of 0,1,2,3, etc. there will be access to the current value of the nodal variables, to the current and the previous time step value, to the current and the two preceding values, and so on.

The values for the different options and variables that KRATOS would use in the application can be imported here in a very intuitive way. These values have been exported by GiD to the *_var* file: the integration scheme, the type of solution, type of damping, the time step, the output time step for the results, etc.

```
#settings to be changed
n_step_estimation = 1
n_step_destroy_distant = DEM_explicit_solver_var.search_step      # how many steps between elimination
of distant particles?
n_step_search = DEM_explicit_solver_var.search_step
bounding_box_enlargement_factor = 2.0    # by what factor do we want to enlarge the strict bounding box
min_risk_factor = 1.0 / safety_factor

solver.search_radius_extension=DEM_explicit_solver_var.search_radius
solver.delta_time = dt
solver.gravity = gravity

solver.Initialize()

nsteps = 26500;

#initializations
time = 0.0
step = 0
time_old_print = 0.0

final_time = nsteps*dt

current_pr_time = timer.clock()
current_real_time = timer.time()

print 'Calculation starts at instant: ' + str(current_pr_time)
while(time < final_time):

    time = time + dt
    solid_model_part.CloneTimeStep(time)
    solid_model_part.ProcessInfo[TIME_STEPS] = step

    solver.Solve()

############     GiD IO
#################################################################################
    time_to_print = time - time_old_print
    print str(time)

    if(time_to_print >= DEM_explicit_solver_var.output_dt):
        gid_io.InitializeMesh(time);
        gid_io.WriteSphereMesh(solid_model_part.GetMesh());
        gid_io.FinalizeMesh();
        gid_io.InitializeResults(time, solid_model_part.GetMesh());
        gid_io.WriteNodalResults(VELOCITY, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(DISPLACEMENT, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(RHS, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(RADIUS, solid_model_part.Nodes, time, 0)
        gid_io.FinalizeResults()
        time_old_print = time

    step += 1

print 'Calculation ends at instant: ' + str(timer.time())
elapsed_pr_time = timer.clock() - current_pr_time
elapsed_real_time = timer.time() - current_real_time
print 'Calculation ends at processing time instant: ' + str(timer.clock())
print 'Elapsed processing time: ' + str(elapsed_pr_time)
print 'Elapsed real time: ' + str(elapsed_real_time)
print (my_timer)
print "COMPLETED ANALYSIS"
```

COMMENTS

After the definition of several options and parameters the function *Initialize()* of the solver is called.

Here also the time step is determined and the main loop is devised by a *while* structure. For every time step the function *Solve()* of the solver is executed.

At some time steps the different results are exported to GiD, this way, they are available for the visualizaton them in the Postprocessor.

This code is totally manageable and easy to tune the different parameters. The changes on this file don't require recompilation. The reason of the usage of these two languages is that until a certain level of usage is interesting to interact with the application with easiness and, very important, without recompiling. After this level, C++ is a more powerful language with much more possibilities with a more efficient use of the memory and better in terms of computing time.

There are two ways to run this file:

From Terminal: From the folder where the GiD case has been created, the so-called *script.py* can be triggered with the preceding instruction *python* (in Linux).

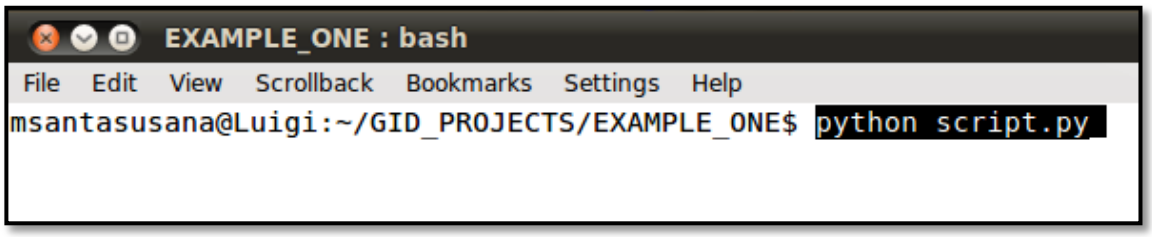


Figure II. 1: Snapshot of the terminal in Linux.

From GiD : (or other graphical interface) with the calculate button:

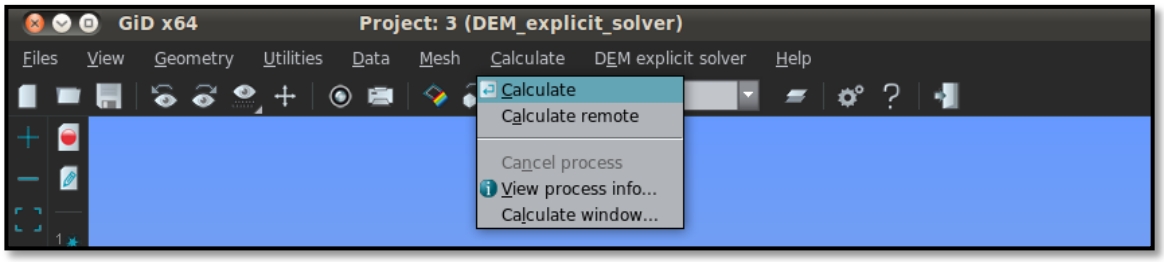


Figure II. 2: Snapshot of the GiD interface when running a case.

Solver: This file, named as ***Sphere_strategy.py*** in our DEM-App., is written also in Python language and it's the last door to the core of the program written in C++ language.

AddVariables (model_part): In this file the first definition is about which variables will be stored in the nodes for every time step and also the value of these variables would be available for different times depending on the buffer size selected.

```
def AddVariables(model_part):
    #model_part.AddNodalSolutionStepVariable(NUMBER_OF_NEIGHBOURS)
    model_part.AddNodalSolutionStepVariable(DISPLACEMENT)
    model_part.AddNodalSolutionStepVariable(VELOCITY)
    model_part.AddNodalSolutionStepVariable(RHS)
    model_part.AddNodalSolutionStepVariable(APPLIED_FORCE)
        ⋮
```

Sphere_strategy.py

AddDofs (model_part): Also here the degrees of freedom are set here. These variables would accept initial conditions defined on GiD.

```
def AddDofs(model_part):

    for node in model_part.Nodes:
    #adding dofs
        node.AddDof(DISPLACEMENT_X,REACTION_X);
        node.AddDof(DISPLACEMENT_Y,REACTION_Y);
        node.AddDof(DISPLACEMENT_Z,REACTION_Z);
        node.AddDof(VELOCITY_X,REACTION_X);
              ⋮
```

Sphere_strategy.py

The constructor of the class of the solver is defined then and some variables and options are defined here with the default values that the user sets.

```
class ExplicitStrategy:

    def __init__(self,model_part,domain_size):

        self.model_part                     = model_part
        self.domain_size                    = domain_size
        self.damping_ratio                  = 0.00;
        self.penalty_factor                 = 10.00
              ⋮
   def Initialize(self):

       #definir les variables del ProcessInfo:
       self.model_part.ProcessInfo.SetValue(GRAVITY, self.gravity)
       self.model_part.ProcessInfo.SetValue(DELTA_TIME, self.delta_time)
              ⋮
       self.solver.Initialize()

  ########################################################################
  def Solve(self):
      (self.solver).Solve()
```

Sphere_strategy.py

The *Initialize (self):* function is called by the *script.py* script as it has been seen. In that script the parameters for the calculation should be imported from GiD or set automatically. Now when initializing these values are stored into variables that are accessible in the different files of the application by means of the ProcessInfo (a data container in KRATOS). Also the Initialize function calls the Initialize function of the solver.

After that the script stores the variables, usually related with options, defined here or in the *Script.py* to the ProcessInfo container. This container is accessible in the C++ files where value set by the user for these variables may be used in the calculation.

8.4.2. Developers stage

In a higher level, the developers that want to implement or modify some functions of the application are required to have a certain level of knowledge in C++ programming language. In addition, some introduction or practice in KRATOS framework is necessary.

Strategy: This file is called *explicit_solver_strategy.h* in our application; it is the principal framework of the application, it calls the different functions and utilities.

Basically the *Initialize()* function realizes the first neighbouring search and initializes the elements. In addition, if the case has the indentation option activated (delta_option) or it is trying to simulate the continuum (continuum_simulation_option) the program will call the function that stores the results of this first neighbouring search to each particle. See section 8.5.5 *Initial Delta Option* and *8.5.6 Continuum Simulating Option*.

```
void Initialize()
{
ModelPart& r_model_part           = BaseType::GetModelPart();
bool extension_option = true;
SearchNeighbours(r_model_part,extension_option);

///Initializing elements
  if(mElementsAreInitialized == false)
     InitializeElements();
  mInitializeWasPerformed   = true;

  if(mdelta_option || mcontinuum_simulating_option){
      Set_Initial_Contacts(mdelta_option, mcontinuum_simulating_option);
  }
}
```

explicit_solver_strategy.h

Afterwards, the function *Solve()* is called from the *Script.py*, through the *sphere_strategy.h.*

```
double Solve()
{
 //1. Get and Calculate the forces
 GetForce();

     //1.1. Calculate Local Dampings
     ApplyLocalDampings();

 //2. Motion Integration
 ComputeIntermedialVelocityAndNewDisplacement();
 //Calls the scheme where the time integration is done.

//3. Búsqueda de veins
 SearchNeighbours(r_model_part,false); //extension option false;
```

explicit_solver_strategy.h

Each one of these functions is a sub function defined in the same Strategy file that they subsequently call the corresponding script that handles each operation. For example, in the strategy, the function *SearchNeighbours ( )* is simply a recalling to the *Neighbour_Calculator ( )* search function that is stored on the Utilities folder.

```
void SearchNeighbours(ModelPart r_model_part,bool extension_option)
{

  /*...*/

  if (mdimension == 2)
      Neighbours_Calculator<2, ParticleType>::Search_Neighbours(r_model_part,extension_option);
  else if (mdimension == 3)
      Neighbours_Calculator<3, ParticleType>::Search_Neighbours(r_model_part,extension_option);

}//SearchNeighbours
```

explicit_solver_strategy.h

After doing the operation we would have found all the neighbours for every particle and we can proceed with the Get Forces module. The Get Forces function initializes the parallelization for every particle defined in the model part. Then inside the loop over the particle, the function *ComputeForces ( )*, which is included in the element, is called.

Parallelization: (see 8.5.1 Parallelization)

Also the parallelization has to be defined here when a loop over the particles is performed. The parallelization implemented here is the OpenMP[1]. To parallelize or not is a decision on the user and it is reflected as an option that can be activated or disabled from GiD interface, also there is a variable to set the number of threads desired for the parallelization.

As an example the function *ApplyLocalDampings ( )*, is presented here to see what the aspect of a loop is over all the particles parallelized by means of OpenMP.

```
void ApplyLocalDampings()
{
    ModelPart& r_model_part           = BaseType::GetModelPart();
    ProcessInfo& rCurrentProcessInfo  = r_model_part.GetProcessInfo();
    ElementsArrayType& pElements      = r_model_part.Elements();

    #ifdef _OPENMP
    int number_of_threads = omp_get_max_threads();
    #else
    int number_of_threads = 1;
    #endif
```

explicit_solver_strategy.h

1 OpenMP (Open Multiprocessing) is a application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.
OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

```
    vector<unsigned int> element_partition;
    OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

    unsigned int index = 0;
    #pragma omp parallel for private(index)
    for(int k=0; k<number_of_threads; k++)
    {
        typename ElementsArrayType::iterator it_begin=pElements.ptr_begin()+element_partition[k];
        typename ElementsArrayType::iterator it_end=pElements.ptr_begin()+element_partition[k+1];
        double dummy = 0.0;

        for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
        {
            it->Calculate(PARTICLE_LOCAL_DAMP_RATIO, dummy, rCurrentProcessInfo);
        } //loop over particles
    }// loop threads OpenMP
}//Apply local damps
```

explicit_solver_strategy.h

All these lines of code what are simply doing is to divide the number of elements in the model part where we have to apply some Damping in different sets that will be send separately to the selected number of threads. For every particle the specified function is calculated normally. Without loose of generality, in this script different elements can be used. When the model part is read, every particle is assigned to one of the elements defined in the Elements folder. The strategy will call the *Calculate ( )*, and other functions that are public and bridge to the private functions of the elements no matter which one it is. In this sense we can define several types of elements that should have the same entry functions but they can be specially defined in a different way in each element.

Utilities folder: In our applications there are not a lot of utilities defined yet but they can be introduced separately to complete new features for the DEM application. Currently there is the *neighbour_calculator.h* utility which is a fundamental function for the DEM and will be detailed next. The *GeometryFunctions.h* is a file that would make easier the operations like transformation of coordinates, the vector and scalar product of vectors, the calculation of the norms in different spaces, etc. Also the *particle_configure.h* is present, which defines a set of inline functions necessaries in other parts of the code, such as distances between points, intersection of the particles, etc. Finally the create_and_destroy.h utility is basically used currently for the destruction of the particles that get out from a determined bounding box.
These functions are utilities in the literal sense and will be explained in the section *8.5 Utilities for the DEM* application.

Schemes folder: From the strategy, the *ComputeIntermedialVelocityAndNewDisplacement ( )* function is called. An object of the class scheme is created; it would solve the integration in time of the movement. The created different schemes, which can be called from the strategies, solve the differential equations in time using an explicit method. Currently the Mid Point Rule and the Forward Euler are implemented and some high order schemes are being developed by collaborators in UCLV CIMNE classroom (CUBA), see section *6.2 Current development and collaboration.*

Elements folder: Currently the application disposes of three elements, the *Discrete_Element,* the *Spheric_particle* and the *Sheric_DEM-FEM_particle*. The *Discrete_Element.cpp/.h* is directly derived from the KRATOS base class *Element.cpp/.h*, the main element for this application were the different elements of the applications derive from.

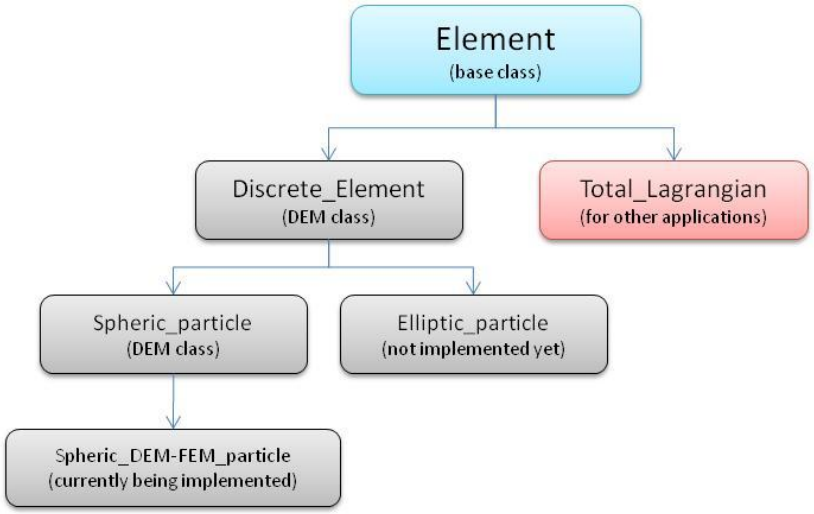


Figure II. 3. Base element class and derivate classes

***Discrete_Element.cpp/.h*:** It has been derived to differentiate from the basic class. As a derived class it inherits all the functions and members defined in the base class. Here nothing new has been implemented yet but it is devised to have the characteristic definitions that define differently this element with respect to the basic one. Also this is the base of all the elements that the application would include.

The functions that this element has are the same as the ones that base *Element.h* has. These functions are needed to be generic and very basic; in the strategy these are the only functions available to call for a generic element; once we are inside these functions in our derived element we can implement differently the functions to redirect to the new private functions that will be implemented new in our elements.

General functions (used in DEM App.) to access the elements:

```
Initialize(), CalculateRightHandSide(), GetDofList(),
InitializeSolutionStep(), Calculate().
```

Currently there are only two derived elements that are commented next: the *Spheric_particle.cpp/.h* and the *Spheric_DEM-FEM_particle.cpp/.h* which are elements designed to represent obviously spheres as a discrete body of analysis. In the future possibly the application will have other elements like *Ellipsoid_particle* or *Tetrahedron_particle*, etc. which will derive from the *Discrete_Element.h* but they probably would need a different definition of the functions to take in account the differences in geometry.

Spheric_Particle.cpp/.h: This is the file of the DEM Application that has had more work done on it. The functions contained on it were completely written new with the help and the experience of other researchers and one of the collaborators, Feng Chun. (See section *6.2.Current development and collaboration* and section 9.*Future of dem-application*). Next, a list of the more important functions is presented. A brief explanation of them is also included:

Entry functions:

- `void SphericParticle::Initialize()`:This function is an "entry" function, in the sense that is a function inherited from the base class; these functions are used to call subsequently private functions that are specific for the derived element.
- `void SphericParticle::InitializeSolutionStep()`: This function will be called once, during the first iteration and will recall the *`SetInitialContacts()`* function.
- `void SphericParticle::CalculateRightHandSide()`: This function calls different force and moment calculation functions.
- `void SphericParticle::Calculate()`: The calculate function permits passing the name of a variable and depending on the value of this name we can implement different calculations or callings to other functions. It represents a useful way to enter to more complex functions of the elements by means of a simple generic one. In the DEM-Application, the function calculates either the critical time step or recalls the damping functions.

Private functions:

- `void SphericParticle::SetInitialContacts()`:This function is called from *`InitializeSolutionStep()`*if the *delta_option* or *continuum_simulation_option* are activated. It stores for the first search the neighbour that every particle has. This is needed for the continuum simulation and the indentation permitted in the generation as it will be explained in detail in section: *8.5 Utilities for the DEM application.*

- `void SphericParticle::ComputeParticleContactForce()` [1] :This function is possibly the most complex function in the application. It will be called from the strategy by means of the *`GetForce();`* this one constructs a loop over the particles (parallelized) and for each one, it gets into this private function by means of *`CalculateRightHandSide()`*. It has the following parts:

  Initial operations:

  - ♣ Reading of the conditions and options activated.
  - ♣ Getting particle properties (including force and moment vectors)

  Loop over the neighbours:

  - ♣ Getting the neighbour properties
  - ♣ Evaluation of the equivalent parameters.
  - ♣ Calculating the relative distance and displacements with respect to the previous step.
  - ♣ Add the contributions of the rotational motion.
  - ♣ Evaluate the forces in local coordinates
  - ♣ Check for the failures
  - ♣ Apply viscous damping contact by contact
  - ♣ Transforming to global coordinates and adding up forces and moments.
  - ♣ Returning back the rotational moments

[1] *The detailed implementation of the function can be found on the annex.*

- `void SphericParticle::ApplyLocalForcesDamping()`:This function is called from *`ApplyLocalDampings()`* in the strategy just after the *`GetForce()`* function. It opens a parallelized loop over the particle and enters, via the entry function *`Calculate(PARTICLE_LOCAL_DAMP_RATIO,...)`*, which calls this function in the ***Spheric_Particle.cpp*** element class. The function applies a global damping to the forces.
- `void SphericParticle::ApplyLocalMomentsDamping()`:This function is called in the same way than the previous one; after applying the damping to the forces, if the rotation option is activated, the damping is applied to the moments.
- `void SphericParticle::ComputeParticleRotationSpring()`: A similar proceeding to *`ComputeParticleContactForce()`* function is performed here. Given a relative rotation between two particles upon a contact, the rotational spring acts opposing this effect. The function is called by the same way than *`ComputeParticleContactForce()`*; if the rotation option is activated the function is calculated after the force calculations.

Spheric_DEM_FEM_Particle.cpp/.h: As it has been commented repeatedly, in parallel with the implementation of the DEM Application, a DEM-FEM Application has been developed combining elements from the Structural Application and the elements from the DEM Application. The application combines these two elements in the same program but unfortunately when these elements have to interact between them they need some extra functions that the *Spheric_Particle.cpp/.h* doesn't need to include. That is why in this case is useful to *derive* a class that would inherit all the basic functions of his base class and also the new ones exclusively for the interaction with the elements from the structural application. The functions are not detailed here as it is part of another application development and it has no further interest for our application by now.

8.5. Utilities for the DEM application

In this section, the topics discussed are basically the troubleshooting of the problems and the approaches that have been done during the implementation of the DEM-Application. Since the main structure has already been explained roughly and it doesn't differ so much from the classical DEM algorithms, the next points deal with special features that have been devised in order to have a versatile and efficient code with a strong robust core.

- **Efficiency:** Parallelization of the code, Critical time and Virtual Mass method.
- **Visualization:** Bounding Box, Create & Destroy utility, Fracture and Rotation Plotting.
- **Versatility:** Initial Delta option, Continuum option, Extended Radius Search.

The utilities implemented for efficiency and visualization can be considered as simple utilities introduced to the code while the three utilities for versatility implied a global restructuration of the code. After the explanation of these utilities a framework is attached where the logical process of the algorithm is exposed.

8.5.1. Parallelization

A Discrete Element Method code without parallelization has a very limited use in practice; the reality is that for considerably large amount of particles (common simulations) the code needs to be parallelized to be competitive against other methods. The good thing of DEM is that the parallelization is quite easy to achieve; the method in its original concept is based on calculating each particle independently, i.e. from the forces that we obtain on a target particle, it evolves in an explicit time step scheme, independently from the other particles. In this sense the main processes in the computational scheme: force calculation, evolve motion, search neighbours can be parallelized.

There exist two types of remarkable architectures for computers, the Shared Memory Machines and the Distributed Memory Machines. In computer science, Distributed Memory refers to a multiple-processor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In

contrast, a Shared Memory multi processor offers a single memory space used by all processors.

There are two widespread techniques of parallelization suitable for C++ language, OpenMP and MPI, which can be implemented in KRATOS and are currently being introduced to DEM-Application. The suitable technique for SMM is Open MP (Open Multiprocessing); it permits parallelizing the loops of the process by using compilation directives so the code runs in serial until the loop, runs the loop on parallel and then reverts back to serial. This can be done by splitting the loop and calculating each part by the different CPU of the same computer; OpenMP runs on a shared memory system so most part of the personal computers would permit parallelizing the calculation and saving time. OpenMP works fine if every unit step of the loop is independent from the others so can be split without problems; the DEM permits doing so. Next, an example of parallelization by OpenMP for the DEM-Application is presented. It is a partitioning of the loop over the particles for the different threads of the computer.

```
#ifdef _OPENMP
int number_of_threads = omp_get_max_threads();
#else
int number_of_threads = 1;
#endif

vector<unsigned int> element_partition;
OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

unsigned int index = 0;

#pragma omp parallel for private(index)
for(int k=0; k<number_of_threads; k++)

{

    typename ElementsArrayType::iterator it_begin=pElements.ptr_begin()+element_partition[k];
    typename ElementsArrayType::iterator it_end=pElements.ptr_begin()+element_partition[k+1];

    double dummy = 0.0;

    for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
    {

        it->Calculate(PARTICLE_LOCAL_DAMP_RATIO, dummy, rCurrentProcessInfo);

    } //loop over particles

}// loop threads OpenMP
```

explicit_solver_strategy.h

For DMM architecture the suitable technology is the MPI (Message Passing Interface); this would permit running a case, usually with large number of particles in a computer cluster where hundreds, thousands or more CPUs intervene in the calculation. With MPI the entire code is launched on each node which would store the data in its own memory. The passing of information and the synchronization of the calculation can be controlled. It is also possible to combine MPI with OpenMP to get the best of every technology.

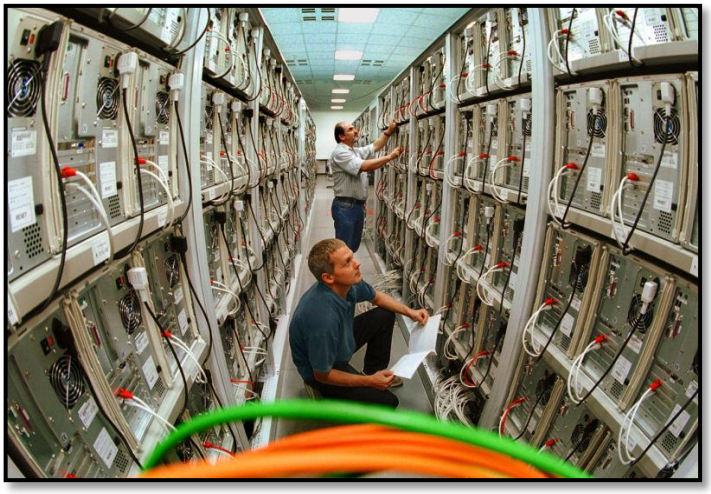


Figure II. 24 Cluster of Distributed Memory Machines

8.5.2. Compute Critical Time + Virtual Mass

Critical time

This operation is done at the first time step, a loop over all the particles is done and for each one, we calculate it critical time step depending on the normal spring stiffness value. This function is very simple and it neither considers the tangential spring nor the rotational one. However there is an "experimentally determined" factor for reducing the critical time step when we introduce the rotation to the particles.

This simple formula for the normal spring is always on the safety side because the K taken for the critical times step will be the maximum one, corresponding to the biggest particle. This K will appear only if the biggest sphere contacts another one with the same radius. The implemented expressions for the values of K can be found on the code on the annex.

$$K_{contact}(R_1, R_2) \leq K_{contact}(R_1, R_1) \text{ With } R_1 > R_2$$
$$K_{critical} \leq K_{contact}(R_{Max}, R_{Max})$$

Next, the simple calculation performed on the DEM-Application is presented:

Calculate (DEM_DELTA_TIME, ...)

```
void SphericParticle::Calculate(const Variable<double>& rVariable, double& Output, const Proces
{

    if (rVariable == DEM_DELTA_TIME)
    {
        double E = this->GetGeometry()(0)->FastGetSolutionStepValue(YOUNG_MODULUS);
        double K = E * M_PI * this->GetGeometry()(0)->FastGetSolutionStepValue(RADIUS);
        Output = sqrt( mRealMass / K);

        if(rCurrentProcessInfo[ROTATION_OPTION] == 1)
        {
            Output = Output * 0.5; //factor for critical time step when rotation is allowed.
        }
    } //CRITICAL DELTA CALCULATION
```

spheric_particle.cpp

Please refer to section *2.3.2 Numerical stability of the method and critical time step* in PART I, where the critical time step calculation is discussed.

Virtual mass method

The virtual mass method is a method implemented by Feng Chun (see section *6.2 Current development and collaboration*) especially for the DEM-FEM_Application. It will be adapted also to the DEM-Application. It is a method appropriated for the quasi-static problems such as compression test, where the dynamic effects are no longer important.

In DEM_FEM_Application, explicit method is adopted, so the stable time step of numerical system depends on the smallest time step among the critical time steps of FEM elements and DEM contacts. In an optimized case, the calculation of the critical time step should be done at several time steps as the contacts renew; In heterogeneous meshes the critical time step may vary seriously from one time step to the other and therefore it will increase the computation time dramatically. To solve the problem above mentioned the virtual mass method is adopted. By adjusting the mass of FEM elements and DEM particles, the critical time step of each contact entity, becomes the same (equal 1, for example). In that sense the critical time step is determined for all the simulation and won't be a limitation of the problem calculation.

During evolvement calculation, accelerations of particles are obtained according to virtual mass (real unbalanced forces divide virtual mass); in that sense the dynamic motion wouldn't be correctly calculated and that's why this method should only be used in static or quasi-static problems.

8.5.3.Bounding Box + Create and Destroy

This utility is devised for dynamic system simulation where the movement of the particles in the domain it is dominant. The bounding box concept (differing from the one view on Part I section *2.1.2 Bounding Box/Sphere representation*) it's a surrounding box that includes all the particles present in the domain at the initial state. Whenever a particle gets out of that domain it is destroyed. In a dynamic problem is easy to happen that a particle goes far of the initial domain or moreover fades away indefinitely by the effect of the gravity; that would lead to an enlargement of the visualization domain and consequently problems in the display of the results. If this option is activated the domain will remain more or less fixed on the initial place and the particles that go far away are no longer of our interest.

8.5.4.Plotting the different fractures[1]

In the DEM-Application it is possible to simulate the continuum as it has been explained and in the previous section where the details for the delta option, the continuum simulation and the extended search are exposed. These utilities configure the instrumentation that the code need for the simulation of continuous medium problems. In these simulations the particles will pass from a "cohesive" state to a detached configuration when some failure criterion is reached. The objective of this utility for visualization is to identify particle by particle the cause of the detachment.

In general, it may be difficult (or simply impracticable) to plot the fracture type (shear, tensile, etc) for every contact pair; however, in a test with a large number of particles we may find useful to plot the dominant type of fracture that the detached particle suffered in its change from continuous to discontinuous state. The idea consists in determining an integer for the different failure types that could occur in a contact (can be defined by the user), namely the shear failure, tensile, Von misses criterion, etc.

[1] *This utility is still in development and it's not present on the DEM-Application yet. In the DEM-Application the Contact Failure Id is active and used to distinguish whether a contact is "cohesive" or detached during the calculation but the exportation to the global failure mode of the particle depending on its contacts and the visualization is not implemented yet.*

First of all clarify that depending on the fracture criterion chosen for the there would be different classifications of the fracture type. For example, we need only two categories if we simply consider independently the tensile stress limit and shear stress limit. On the other hand if we set models like Von Misses or Rankine, which evaluate the global stress state or simply the principal one, we may not have distinction for type of failure.

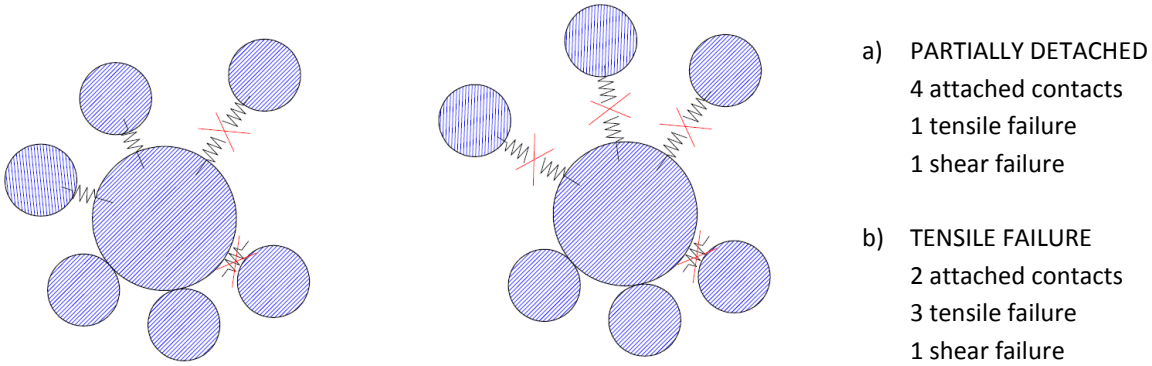
This is the proposed criteria for the DEM-App. (Falure of the contacts):

Contact Failure Id = 0	Still attached, tensile and shear strength.
Contact Failure Id = 1	Generally detached: Not initial neighbours, not same continuum group
Contact Failure Id = 3	Tension failure, neither tensile strength applicable nor shear.
Contact Failure Id = 4	Shear failure, neither tensile strength applicable nor shear.
Contact Failure Id = 5	Von Misses failure criteria, etc. (can be defined by the user)

Note that the Contact Failure Id = 2 is not defined. The idea is to reserve this identifier for the partially detached particles, the ones that neither the shear nor the tensile are dominant (for example). We will distinguish now between *Contact Failure Id*. and *Particle Failure Id*.

Particle Failure Id = 0	All contacts are still attached.
Contact Failure Id = 1	Generally detached: From a discontinuous group or surrounded by particles from other groups
Contact Failure Id = 2	Partially detached. Some contacts detached. Not a dominant case.
Contact Failure Id = 3	Tension failure dominant in the contacts.
Contact Failure Id = 4	Shear failure dominant in the contacts.
Contact Failure Id = 5	Other used-defined criteria.

Some criterion has to be defined to determine when, in a target particle, it will be considered that the dominating fracture type is one of the occurring contact failures.



(a) *Partially Detached* (b) *Tensile failure dominant*

a) PARTIALLY DETACHED
4 attached contacts
1 tensile failure
1 shear failure

b) TENSILE FAILURE
2 attached contacts
3 tensile failure
1 shear failure

Figure II. 25 Example of application for different failure types

8.5.5.Initial Delta Option

The geometry - mesh options have been exposed in section 7.2.2 New DEM-Application Pre-Process where the GiD Sphere mesher has been explained. It is very common to mesh real case problems with meshers like this in order to fill a determined volume with some set of spheres with different radius (or not) depending on the meshing options. The GiD Sphere mesher is the one that is available in GiD but there are others developed for other groups that should be used for DEM-Application; in particular, UCLV (CUBA) CIMNE classroom is developing a sophisticated generator that will probably be linkable to GiD.

Some of these generators, including the one from UCLV, may produce some indentation between particles created to fill a determined geometry. In the original DEM conception this would lead to considerable problems because these indentations would result into large forces and "explosions" due to the repulsive tendency of the particles. In our application, this issue has been taken into account and (if the option is activated) we let the particles have an initial passive indentation that doesn't produce any force.

The complexity of the implemented algorithm that allows doing so is due to the fact that we want this indentation to be remembered for the next eventual collisions with these two particles. This avoids gaining volume and energy from nowhere but involves pretty much the algorithm. A framework of this implementation is on *8.5.8 Framework for the Versatility utilities*.

.

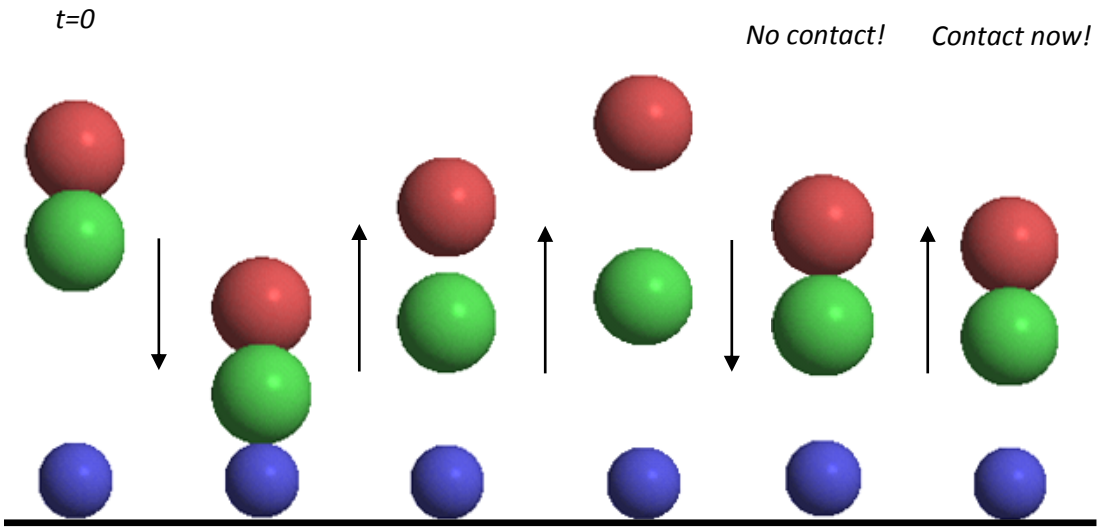


Figure II. 26 Initial Delta remembered in a contact

8.5.6.Continuum Simulating Option

It has been commented in the part one that the tendency nowadays in DEM research is to find suitable ways to reproduce the behaviour of the continuum by outfitting the discrete particles with some resistant mechanisms. There have been many theories and approaches to this problem and there is not a unique accurate solution for the characterization of the properties that would differentiate the discrete element as a continuum simulating one. Regardless of the characterization we do, when we deal in a program with: non-cohesive discrete particles, continuum simulating particles (cohesive) and the possibility to change from one state to the other by means of the fracture, the algorithm that has to be devised become involved in a similar way that the Delta option does. This is also present in *8.5.8 Framework for the Versatility utilities*.

8.5.7.Neighbour Search utility and Extended Radius Search

Neighbour Search.h: this is a basic function of our application and the implementation of it has been complex. The original function was much more basic; it consisted simply in a calling to other functions that are in the KRATOS libraries that can be used for any application. In KRATOS, many neighbouring search functions like: static bins, dynamic bins, Oct-tree, etc. are already implemented ready to be used in different applications. What was implemented new in this file is the generalization to a neighbouring search that permits including the options of some initial indentation between the particles and the continuum simulating option. Summarizing, the utility included in a C++ header file is the coupling of a basic search function with a set of loops and operations that permit the treatment of the Continuum Simulating problem and the Initial Delta utility.

Apart from these utilities, it was interesting to give an extra feature to the basic search that is the possibility to consider a particle to be a neighbour of a target one even if there is a little separation between them. This is very frequent to happen with the sphere mesh generators; including the GiD mesher whose example is shown in *Figure II. 27 Gap left by the mesher.*

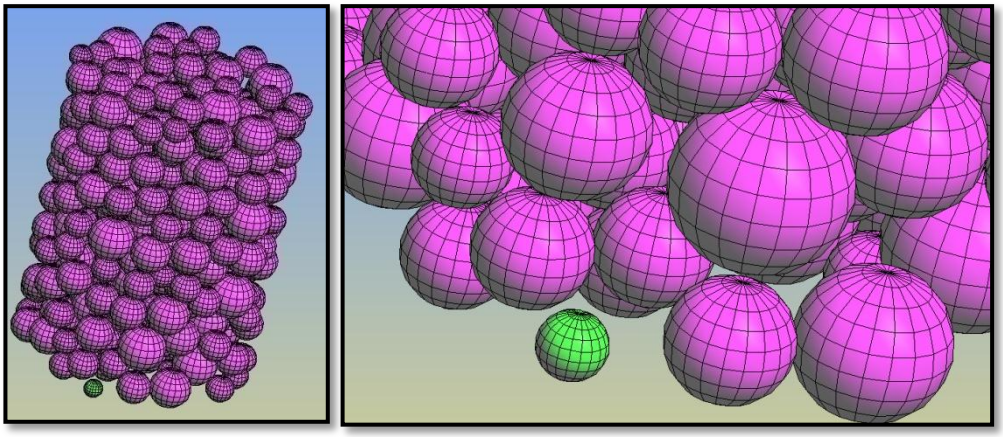


Figure II. 27 Gap left by the mesher.

Developers from KRATOS have implemented a modification to the bins function that searches for the contacts introducing a new parameter that is the radius of search. The original function took the radius of the target particle as the radius of search for neighbours and it used to find only the particles that where at a distance $d \leq R_{target}$ (tangent or indented). Now, with this utility, the R_{target} becomes enlarged by some percentage. $R_{Extended} = (1 + \alpha)R_{target}$ Normally the extension is $\alpha \approx 5\%$. If the value needed is larger, it should be considered to revise the generation. This initial separation is stored and treated in the same way like the indentation in the Initial Delta option.

8.5.8.Framework for the Versatility utilities

Initialize function:

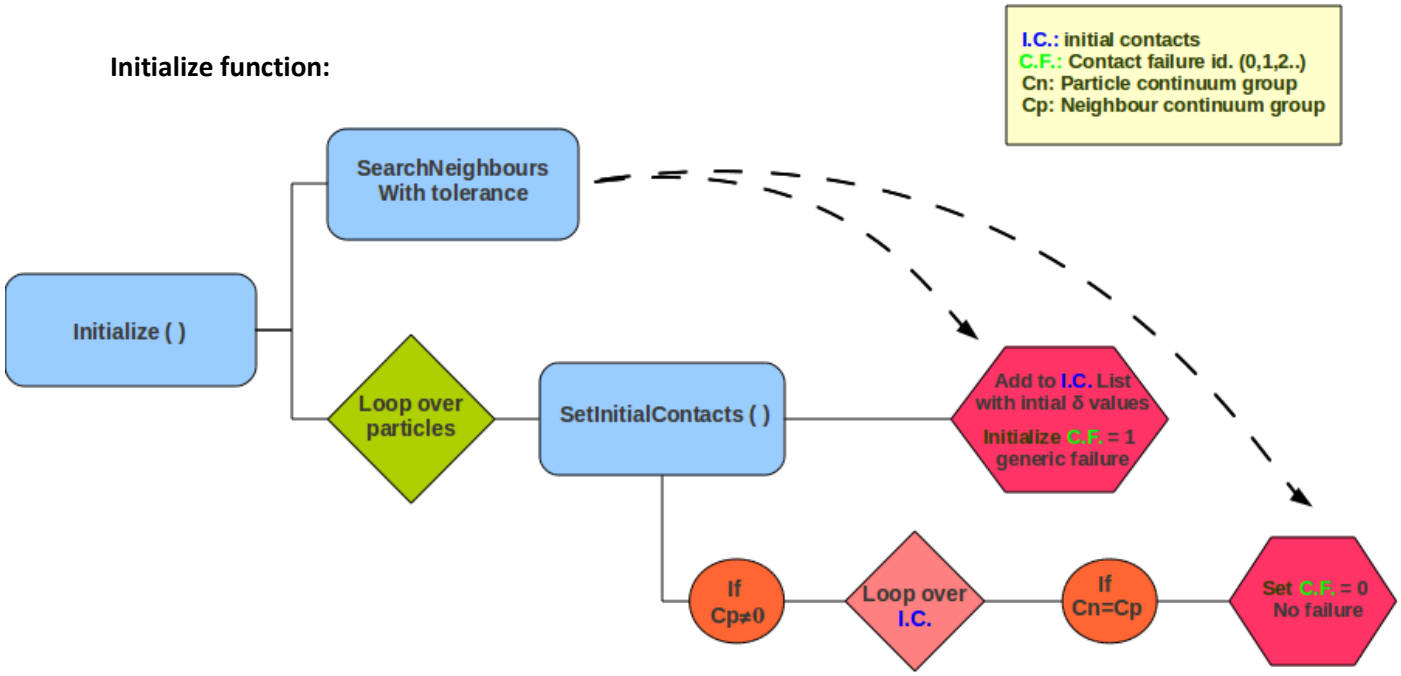


Figure II. 28 Framework of the *Initialize* algorithm implemented

This function is called only in the first iteration of the process. Here the neighbouring search is done with the extension over the radius. After the neighbours are set, this search is stored as the Initial Neighbours for every particle. After doing this, the values of the failure and the initial delta will be characterized.

If two contacting particles belong to the same continuum group they are attached and they will have tensile strength so, the failure is set to 0. The particles that have different group or the ones that belong to the zero group are considered to have a failure type=1 (generally detached). The distance between the centres of these particles is compared against the sum of their radius and also the Initial Delta is stored for every Initial Neighbour for every target particle.

Main Solve function:

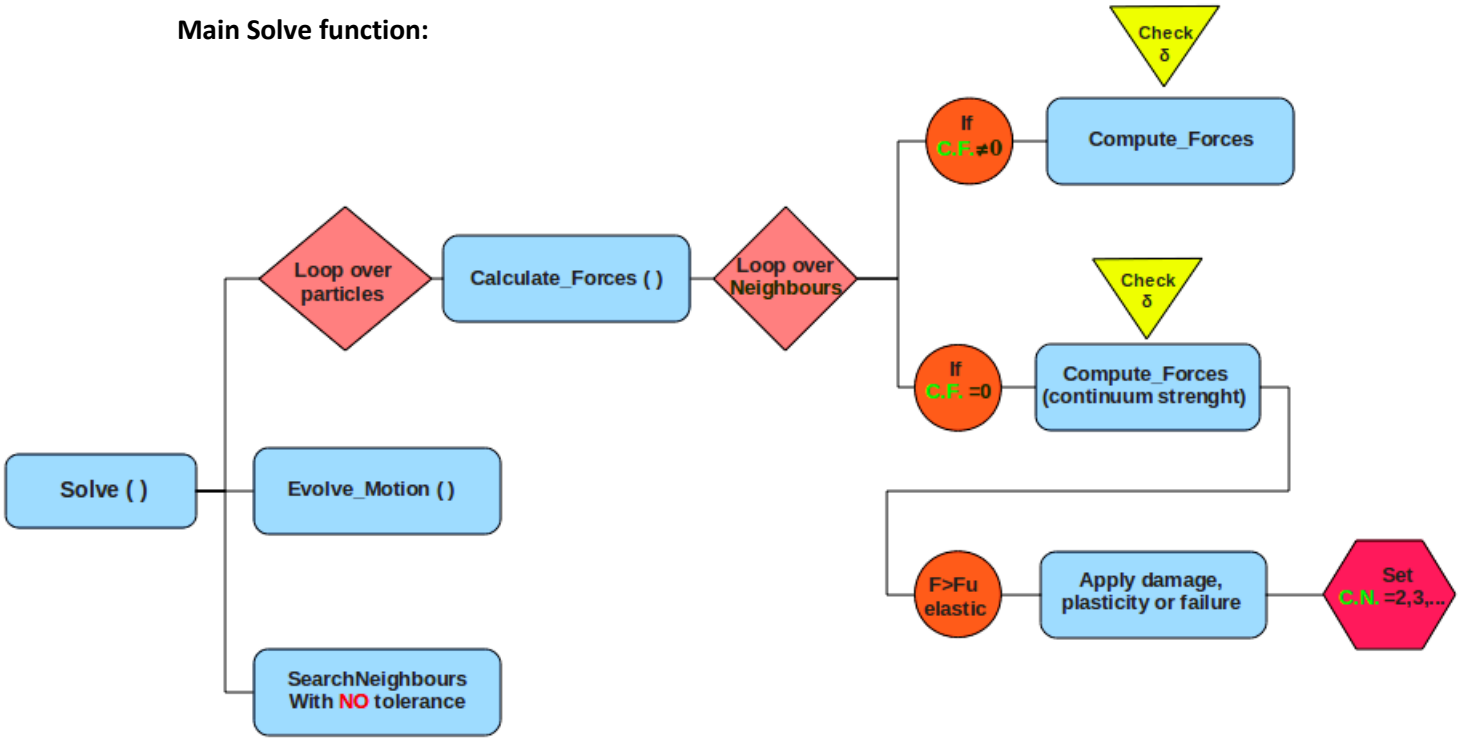


Figure II. 29 Framework of the *Solve* algorithm implemented

These are the basic steps of the DEM: Calculate Forces – Evolve Motion – Search Neighbours. When calculating the forces we have to check the *Failure Id* in order to apply tensile forces or not, shear forces or friction, etc... The *Contact Failure Id* is the flag (see section *8.5.4 Plotting the different fractures)* used in order to distinguish between a continuum simulating treatment of the contact or an original DEM treatment.

The forces that result from the calculation can overpass the limits established for the elastic regime; depending on the failure criterion defined, some contact can be considered detached. The fracture criteria implemented can be various and also more sophisticated codes can be implemented like damage or plasticity; most of these models have not been implemented yet. Regardless of the criterion, when we overpass the fracture limit (being tensile, shear or whatever stress) the contact has to change from a continuum simulating to a detached one. For every contact calculation we also need to recover the Initial Delta for the contacting pairs if it is applicable.

The evolve motion process consists only in integrating the accelerations to get the displacements, it has been explained in Part I: *1.2.4 Integration of Motion Equations*.

The neighbouring search without any tolerance or extension is simply the search for the particles that are in a distance $d \leq R_{target}$. The extended radius shall be used only during the first step. The next framework explains the functioning of the ***neighbour_calculator.h*** file.

Neighbour search algorithm:

This utility file has been developed in a very robust and general way to permit the neighbouring search for different type of element; as an example, the DEM-FEM strategy calls the same function with another type of element without any modification. It is a C++ template class that permits the calling with several type definitions and different elements and dimensions (2D or 3D) for the dynamic bins search. As it has been commented it also permits a parameter to determine an extension for the radius of search.

In order to deal with the Continuum option and Initial Delta utility the algorithm complicates.

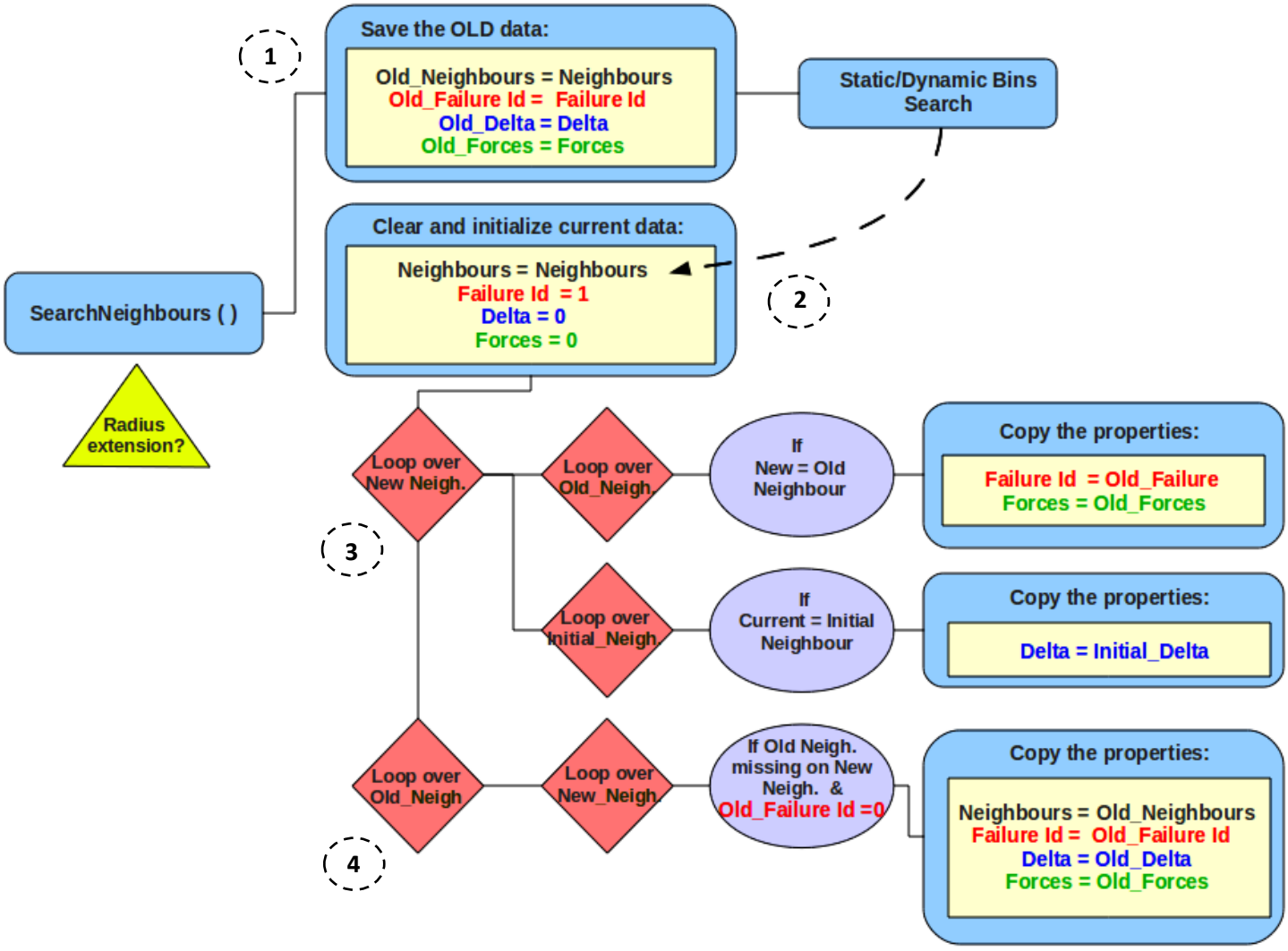


Figure II. 30 Framework of the *neighbour calculator* utility implemented.

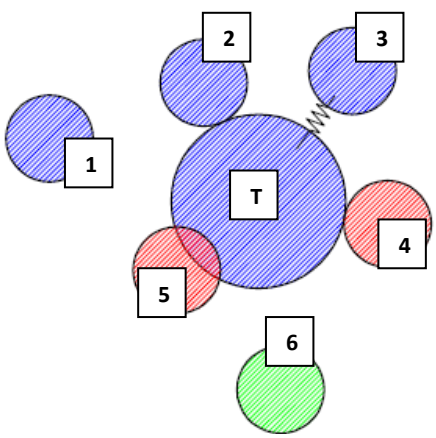
1. *Old Neighbours* storing: Before the *New* search is done, all the necessary information is stored on the *Old_neighbours*, *Old_forces*, *Old_Failure*_Id, and *Old_Delta* arrays.

2. Initializations for the new search: The *New* search is performed and the *New* array parameters are initialized as zero for the forces and the deltas and equal to one for the failure id.

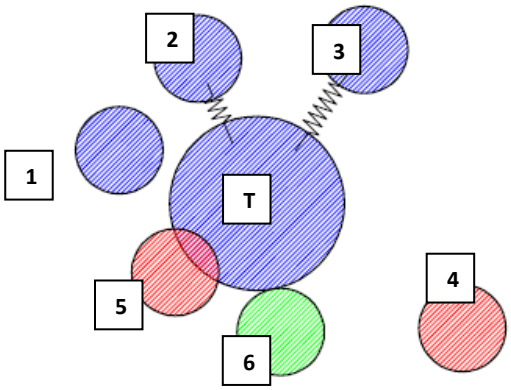
3. Recovering data from *Old* and *Initial Neighbours*: For the *New_Neighbours* we want to recover the information that we already got if they were previous neighbours (*Old_Neighbours*), namely the forces and the failure id. From the *Initial_Neighbours* we will copy the Initial Delta data those *New_Neighbours* that coincide with an *Initial Neighbour*.

4. Missing *Old Neighbours:* Finally we have to check for those *Old_Neighbours* that couldn't be found by the ordinary search because they formed part from a continuum simulating contact and they were separated from the target particle due to a tensile strain. (Remember that the neighbour search for the main loop is without any radius extension). We need to do a push-back on the *New_ Neighbours* including these not detected neighbours with the corresponding *Old* information. This is explained is an illustrative figure after the framework.

Example for the search algorithm:



Initial state t=0



Time t1 >0

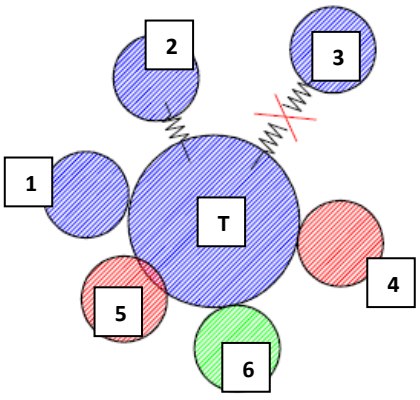
Characteristics of the particles at time T=0:

Id	Continuum group	Neighbour?	Observations
T	1 (blue)	-	The target particle has 4 neighbours: 2,3,4,5
1	1 (blue)	NO	Not in extended radius.
2	1 (blue)	YES	Same group – Cohesive.
3	1 (blue)	YES	In extended radius. Same group – Cohesive.
4	2 (red)	YES	Not same group – non cohesive.
5	2 (red)	YES	Delta will be stored. Not same group – non cohesive.
6	3 (green)	NO	-

Characteristics of the particles at time T=T1:

Id	Continuum group	Neighbour?	Observations
T	1 (blue)	-	The target particle has 4 neighbours: 2,3,5,6
1	1 (blue)	NO	In extended radius. But search without extension on T>0
2	1 (blue)	YES	Separated but remembered from previous neighbours.
3	1 (blue)	YES	Remembered still attached by tensile strength.
4	2 (red)	NO	Lost neighbour.
5	2 (red)	YES	-
6	3 (green)	NO	New neighbour.

Take special attention to the particle number 2: The new search performed in T1 does not find particle 2 as a neighbour because these searches are without any extension of the radius. However, recalling (4) we recover the neighbours 2 and also 3 which was previously in the same condition. Particle number 1 for example is now closer than particle number 3 but it is not a new neighbour because it will only be if the contact is on the surface; although it is from the same continuum group it won't be a cohesive neighbour because it is not an initial neighbour.



Characteristics of the particles at time T=T2:

Id	Continuum group	Neighbour?	Observations
T	1 (blue)	-	The target particle has 6 neighbours: 1, 2, 3*, 4, 5, 6**
1	1 (blue)	YES	New contact. But non cohesive despite same group.
2	1 (blue)	YES	Separated but remembered from previous neighbours.
3	1 (blue)	YES	Tensile failure. *Not neighbour for the next step.
4	2 (red)	NO	Neighbour again, contact on the surface
5	2 (red)	YES	Fake neighbour**. Not contact force applicable.
6	3 (green)	NO	Nothing new with respect to the previous step.

The particle 1 is now a new neighbour but not with continuum properties (tensile or shear strength) because it doesn't belong to the initial neighbour list.

The particle number three was found as a recovered neighbour but during the check for the forces the tensile strength was exceeded and at this time the *Failure Id* changes for this contact being detached for the next steps. Although it belongs to the initial neighbour list, whenever this particle contacts again the target, the contact won't be cohesive anymore due to the fracture Id stored.

The particle number 5 is found in the common search because the particles intersect; however, when the force is calculated the program will read the value of the delta stored and would determine that there is no contact force. This particle will transmit a compressive force only when it comes back and hits the Target particle in a distance closer or equal the value of the delta. Contrarily the neighbour 4 collides normally when the surfaces intersect.

Remark on versatility utilities:

As a final remark on the versatility utilities it shall be commented that the use of these utilities should be restricted to the quasi-static problems where the contacts doesn't change so much and the continuum simulating problem is present. However the application is designed in a general way so we can combine discrete elements with continuum simulating particles in dynamic or static conditions. Obviously if we make use of all these utilities the calculation increases the cost in CPU time and memory.

Disabling the utilities with a simple switch ON/OFF is crucial in order to skip complex calculations in the cases that they are not necessary.

9. FUTURE OF DEM-APPLICATION

9.1. Further development of the DEM-Application

Fortunately the Discrete Element Method has a remarkable interest nowadays in CIMNE as well as in many research institutions. A lot of research is being done in the different fields where the DEM is applicable. One of the fields which the DEM has to tend now and is of our interest in CIMNE is to simulate correctly the continuum behaviour in elasticity using a local rigorous characterization of the contacts and get good results for the tracking of the fracture and the post-fracture behaviour.

Regarding the DEM-Application, the team have a considerable list of new features and modifications to introduce to the application which is in fact just the foundation of an ambitious project. To name some examples:

- Pre/Post: Applied forces condition should be introduced to the *ProblemType*, also some tracking method for the rotations in 3D.
- Constitutive modelling: models of plasticity and elastic damage should be introduced.
- Elements: New elements, ellipsoids, tablet type, polyhedral shapes such as cubes, prisms, tetrahedron, clusters of spheres and DEM blocks concept.
- Non-DEM bodies: rigid boundaries and objects
- Integration: high order scheme for the integration of the motion laws.
- Neighbouring search: Advanced search schemes with linear complexity. Neighbouring search feasible for difficult shapes.
- Parallelization: MPI and OpenMP optimized for the principle functions as the force calculation, the neighbour search and the evolving motion.
- Generation: a high-quality generator is needed to mesh complex geometries with high order of compacity. Also links for the mesh to the advanced technology such as tomography.
- Revision: Checking of the implemented functionalities, cleaning the code, improving the efficiency of the programming.

9.2. Parallel research wit DEM/DEM-Application in CIMNE

In this section I am pleased to introduce the work that other researchers in CIMNE are doing in the field of the Discrete Element Methods. Some of them will use the DEM-Application as a base; they also collaborate on the development of the DEM itself or customize some utilities for their particular application.

PhD candidates Javiera Valdivia, Nerea Mangado

Biomedical application for DEM: bone regeneration in prosthesis interface

The main specific goal is the development of new computational models using Discrete Element Methods for the analysis of the bone-implant-living-interfaces and prostheses mechanics. This modelling is of high concern in order to get a realistic response of the bone integration with the prostheses stems. The objective of these simulations is to help medical doctors to predict the long-term evolution of bones and to detect eventual pathologies.

DEM-Application will develop a module that would Include some biological reaction functions depending on time and some spurs to represent the behaviours of the bone regeneration cells on an interface bone-prosthesis that would be also simulated with discrete elements as a porous media.

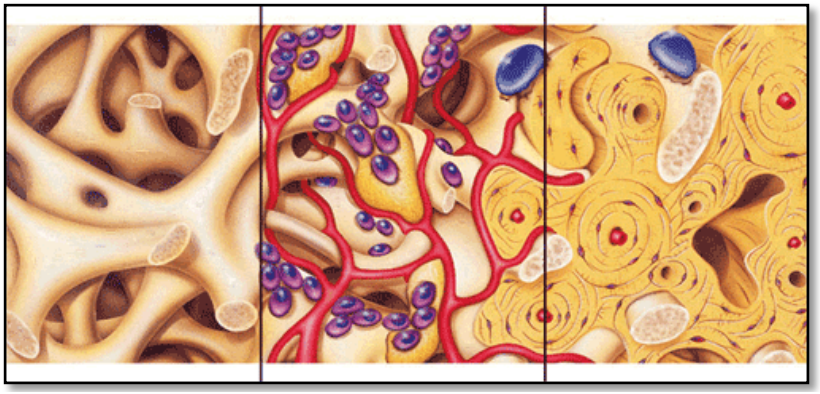


Figure II. 31 Bone regeneration in the bone-prosthesis interface

Keywords: Bone, Prostheses, Biomedics.

PhD candidate Chun Feng

DEM & FEM coupling: landslide simulation

Landslide is one of the most serious geological disasters in the world. To simulate the evolvement of landslide well, the DEM & FEM coupled method should be introduced into KRATOS. For the sliding body, the DEM should be used, while for the bedding rock, the FEM should be adopted. My work in CIMNE is to create the DEM & FEM Application, program the coupling strategy, and make it possible to simulate the DEM & FEM coupled problems. So collaborating with DEM KRATOS team is crucial.

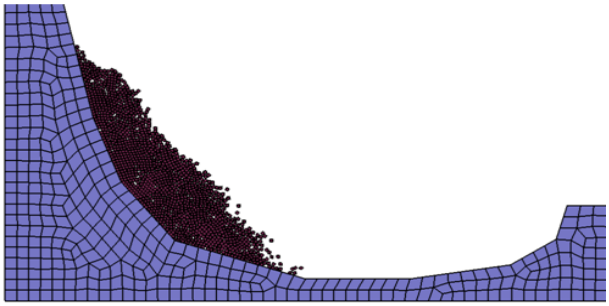


Figure II. 32 Rockfall simulation. DEM particles on a FEM domain

Keywords: Landslide, DEM-FEM coupling, bedding rock.

CIMNE researcher Ferran Arrufat

DEM, drilling simulations

As global demand for energy pushes oil operators into increasingly challenging environments, the need of a high level assurance to drill at minimum risk is a must. Discrete element models had been rarely used to simulate drill bits, but with a good definition of the considered material properties, it is possible to simulate the rotation of the bottom hole assembly through the fractured ground.

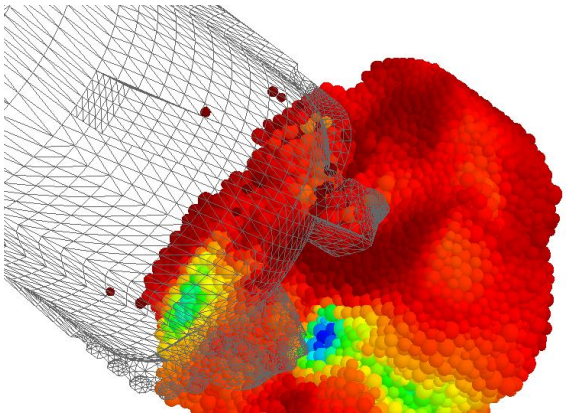


Figure II. 33 Interaction of drilling tool with DEM-discretized rocky media.

Keywords: Drilling, Tool-Rock Interaction, DEM applications.

PhD Candidate Victor Eduardo

DEM Explosion

To estimate the structural damage caused by the detonation, multiple numerical simulations have been performed varying the quantity of explosive load and the distance between the structure and the detonation for each scenario. Afterwards, compression tests were performed in the resulting structure, getting its maximum load capacity and stiffness after the detonation in order to quantify the damage.

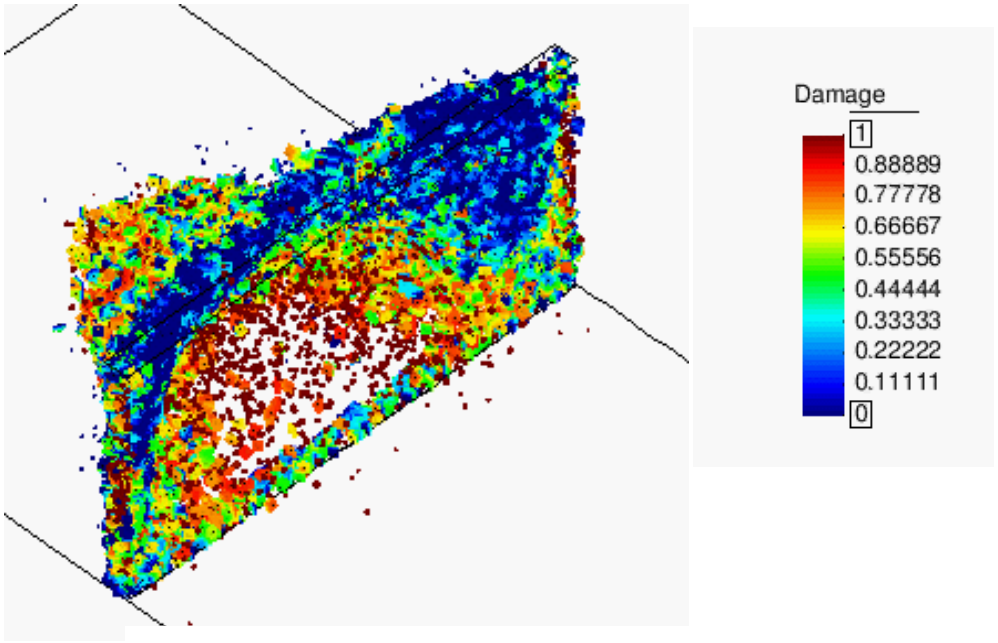


Figure II. 34 Simulation of an explosion on a wall

Keywords: Explosion, Structural damage, DEM applications.

Apart from these parallel works, the DEM-Application team has the collaboration of many researchers and institutions that have been commented. In addition, some students from the Civil Engineering school of Barcelona have recently joined us, expanding the DEM-Application team of KRATOS.

Conclusions

Regarding to the Discrete Element Method it has to be mentioned that it is an excellent method when facing the simulation of the discontinuous media problems. Not only for its theory conception which is so adequate for the dynamic problems but also from a numerical and computational point of view; the use of explicit methods combined with essence of the DEM that permits considering each particle independent from the others, allows implementing a powerful code that is almost completely parallelizable and makes the method have no competitor in this field.

Contrarily its usage when dealing with simulations of the continuum is still doubtful. The great expectations that this method has in this field is the great capacity to track the fracture and simulate the frictional behaviour of the post fractured areas where the discrete particles generated would be well described with the DEM. However the characterization of the parameters for the correct behaviour of the discrete particles when simulate the continuum is neither unique nor universally known; simply there are good approaches for both the elastic behaviour and for the plastic or failure stage in particular cases.

Concerning to KRATOS, say that it is a magnificent platform for any numerical method application. It has been conceived as a framework for FEM-base codes but it has represented an unsurpassable platform for the DEM-Application. A part from the utilities and the libraries that KRATOS provide, the possibility of coupling different applications is one of its greatest pros. The DEM-FEM application is an example of this versatility; the application combines the DEM-Application with the Structural Application (also from KRATOS) without any compatibility problem and with high facility. Another excellent feature of KRATOS is that it is an open source free platform that can be used by anyone and anywhere; this will definitely lead this complete platform to a great success.

DEM-Application is still on development; what is now available is a basic program that can be incorporated to a graphic interface such as GiD that permits doing basic simulations of DEM with spherical and circular elements for continuous and discontinuous media problems. This work has represented the beginning of the development for this ambitious program that have been founded on a very wide and versatile structure to permit a large set of possibilities that can be implemented for anyone interested in joining the DEM-Application team.

Final Personal Comments

Justification of the document presented

This work has claims to be a guide for engineers and developers interested in the implementation of a code for the Discrete Element Method, specially the DEM-Application developed in CIMNE. The code is explained only to a certain detail to understand the basic structure and the capabilities and reasons to be of the utilities and parts of the program; however, a detailed description of the code is not given for two basically reason. This first one is because the objective of the work is not to be a manual of usage of the code (which is not closed yet) but a guide as it is already said. Secondly, mention that the code is in a constant development stage; from the first week of the redaction of this work and also in the very last day, changes have been done daily in the program. This has introduced also difficulties to the redaction of the document since a lot of figures and information has had to be updated.

What's next?

Although the application presents a good base for new implementations in a developer's stage and also a basic program for the DEM continuous and discontinuous simulations, it is not a "*long term release*" version. This means that now the program needs to enter a stage where it has to be done loads of validations of accuracy of the program, tests for the robustness of the code, solving of bugs that would appear improvement of the efficiency and the use of programming languages, etc. There is still a hard work to do to consolidate what it has been implemented in this first stage.

Not only a final undergraduate thesis

This work has represented the final thesis for the author's undergraduate course in the E.T.S.E.C.C.P.B, school of civil engineering. Fortunately the DEM-Application and the present document itself are of the interest of many other students and researchers.

The DEM is a method that is trendy nowadays and there are a lot of research institutions interested on it and also students and PhD candidates. During the implementation of the program the KRATOS team received the collaboration from many other researchers, not only from CIMNE but also, as it has been commented, from Cuba and China. This work has been used for all of them for their respective thesis or works, combining different methods with the DEM from KRATOS or sharing technology to improve the applications. Also, recently new undergraduate students have joined the DEM-Application team and they have started helping in the development and validation of this program; they would probably extend the contents of this work and code new features to the program as a part of the respective final thesis of their courses.

Personal challenges

The Implementation of a DEM program requires a lot of knowledge that has been completely new for me. First of all, regarding the theory of DEM I had just a rough idea of the method so it has been a learning process while reviewing the state of art of the method and studying the different approaches and the last advances in the field.

Apart from the theory of DEM, it has been a challenge for me to learn from scratch the programming languages that have permitted me implementing the method: C++ and Python. Moreover, the KRATOS framework itself is a work tool that requires some training and practise, especially if the user background is not computer science. Finally, although I knew how to manage the GiD Pre-Processor in a user stage, the development of a new *Problem Type* gave me the possibility to use GiD as an advanced user.

I consider that this learning process has been so satisfactory for me at the same time that I've realised that the C++ or an equivalent programming language are useful tools that any engineer should know.

Personal satisfaction

It only remains to the author to express his personal satisfaction for the work completed during this period being part of the KRATOS team in CIMNE. The DEM-Application is just the beginning of an ambitious project that the current developers will be pleased to keep forming part of it, completing and improving the current program.

Annex: Code Implemented

As an annex part of the code implemented for the DEM-Application is attached. The files needed to compile and run the code are organized in folders and subfolders in the general KRATOS directory that can be found online in the following repository:
https://svn.cimne.upc.edu/p/kratos/kratos

The KRATOS files are updated daily whenever a developer uploads any modification. The code is completely free and open-source; therefore you can feel free to download it, use the available *ProblemTypes* and customize your own application. Please visit http://kratos-wiki.cimne.upc.edu for further information.

Since the DEM-Application itself has a lot of files and they depend also on many others from the KRATOS *Kernel* (main common files), in this annex just a few of the most important files from the DEM-Application are presented; they have been specially coded for the DEM-Application and can include all the utilities, the strategies and the important functionalities that have been commented during this work. It can serve as an auxiliary material for the lecture of the Part II for the readers interested in the implementation and the possible developers or advanced users.

The attached files are:

FOLDER	FILES
Custom Elements	shperic_particle.cpp
Custom ProblemType	script.py
Custom Strategies	explicit_solver_strategy.h, constant_average_acc_scheme.h
Custom Utilities	neighbours_calculator.h, particle_configure.h
Python Scrips	sphere_strattegy.py

```cpp
// System includes
#include <string>
#include <iostream>


// External includes


// Project includes
#include "includes/define.h"
#include "spheric_particle.h"
#include "custom_utilities/GeometryFunctions.h"
#include "DEM_application.h"


namespace Kratos
{

      SphericParticle::SphericParticle( IndexType NewId, GeometryType::Pointer pGeometry) :
      DiscreteElement(NewId, pGeometry) {}

      SphericParticle::SphericParticle( IndexType NewId, GeometryType::Pointer pGeometry,
      PropertiesType::Pointer pProperties)
      : DiscreteElement(NewId, pGeometry, pProperties)
      {}

      SphericParticle::SphericParticle(IndexType NewId, NodesArrayType const& ThisNodes)
      : DiscreteElement(NewId, ThisNodes)
      {}

      Element::Pointer SphericParticle::Create(IndexType NewId, NodesArrayType const& ThisNodes,
      PropertiesType::Pointer pProperties) const
      {
         return DiscreteElement::Pointer(new SphericParticle(NewId, GetGeometry().Create( ThisNodes ),
         pProperties) );
      }

      /// Destructor.
      SphericParticle::~SphericParticle(){}

      void SphericParticle::Initialize(){

        KRATOS_TRY

        mDimension = this->GetGeometry().WorkingSpaceDimension();

        double density      = GetGeometry()(0)->FastGetSolutionStepValue(PARTICLE_DENSITY);
        double radius       = GetGeometry()(0)->FastGetSolutionStepValue(RADIUS);
        double& mass        = GetGeometry()(0)->FastGetSolutionStepValue(NODAL_MASS);

        double & Inertia         = GetGeometry()(0)->FastGetSolutionStepValue(PARTICLE_INERTIA);
        double & MomentOfInertia = GetGeometry()(0)-
        >FastGetSolutionStepValue(PARTICLE_MOMENT_OF_INERTIA);



        mContinuumGroup     = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_CONTINUUM);
        mFailureId          = !(mContinuumGroup);

        if(mDimension ==2)
        {
            mass     = M_PI * radius * radius * density;

            mRealMass = mass;

            Inertia = 0.25 * M_PI * radius * radius * radius  * radius ;

            MomentOfInertia = 0.5 * radius * radius;
```

```
  }
  else
  {
      mass     = 4.0 / 3.0 * M_PI * radius * radius * radius * density;

      mRealMass = mass;

      Inertia = 0.25 * M_PI * radius * radius * radius  * radius ;

      MomentOfInertia = 0.4 * radius * radius;
  }


  KRATOS_CATCH( "" )

}


void SphericParticle::CalculateRightHandSide(VectorType& rRightHandSideVector,ProcessInfo&
rCurrentProcessInfo){

  ComputeParticleContactForce(rCurrentProcessInfo);


  if( (rCurrentProcessInfo[ROTATION_OPTION] != 0) && (rCurrentProcessInfo[ROTATION_SPRING_OPTION]
!= 0) )
  {
        ComputeParticleRotationSpring(rCurrentProcessInfo);
  }

}
void SphericParticle::EquationIdVector(EquationIdVectorType& rResult, ProcessInfo&
rCurrentProcessInfo){}
void SphericParticle::MassMatrix(MatrixType& rMassMatrix, ProcessInfo& rCurrentProcessInfo)
{

    double radius = GetGeometry()(0)->GetSolutionStepValue(RADIUS);
    double volume =   1.333333333333333*M_PI*radius*radius*radius;
    double density = GetGeometry()(0)->GetSolutionStepValue(PARTICLE_DENSITY);
    rMassMatrix.resize(1,1);
    rMassMatrix(0,0) = volume*density;

}

void SphericParticle::SetInitialContacts(int case_opt) //vull ficar que sigui zero si no son veins
cohesius.
{

    // DEFINING THE REFERENCES TO THE MAIN PARAMETERS


     ParticleWeakVectorType& r_neighbours             = this->GetValue(NEIGHBOUR_ELEMENTS);

     this->GetValue(PARTICLE_INITIAL_DELTA).resize(r_neighbours.size());

     ParticleWeakVectorType& r_initial_neighbours     = this-
>GetValue(INITIAL_NEIGHBOUR_ELEMENTS);

     unsigned int i=0;
```

```
        //SAVING THE INITIAL NEIGHBOURS, THE DELTAS AND THE FAILURE ID

        for(ParticleWeakIteratorType_ptr ineighbour = r_neighbours.ptr_begin();
        //loop over the neighbours and store into a initial_neighbours vector.
         ineighbour != r_neighbours.ptr_end(); ineighbour++){

           if (this->Id() != ((*ineighbour).lock())->Id() ){

                array_1d<double,3> other_to_me_vect = this->GetGeometry()(0)->Coordinates() -
                ((*ineighbour).lock())->GetGeometry()(0)->Coordinates();
                double distance                     = sqrt(other_to_me_vect[0] * other_to_me_vect[0]
                +
                                                      other_to_me_vect[1] * other_to_me_vect[1] +
                                                      other_to_me_vect[2] * other_to_me_vect[2]);

                 double radius_sum                  = this->GetGeometry()(0)-
                 >GetSolutionStepValue(RADIUS) + ((*ineighbour).lock())->GetGeometry()(0)-
                 >GetSolutionStepValue(RADIUS);
                 double initial_delta               = radius_sum - distance;


                 int r_other_continuum_group = ((*ineighbour).lock())->GetGeometry()(0)-
                 >GetSolutionStepValue(PARTICLE_CONTINUUM);

                    /* this loop will set only the 0 (contunuum simulating case) to the initial
                    neighbours. The force calculator will change this
                     * values depending of the type of failure as it is describre here:
                     *
                     *   mContactFailureId values:
                     *      0 := Still a continuum simulating contact
                     *      1 := General detachment (no initial continuum case: non continuum
                    simulating particles or particles from diferent continuum group.)
                     *      2 := Partially detached
                     *      3 := tensile case
                     *      4 := shear case
                     *      5 :=von Misses.....M: define new cases...
                     */

                    if( (r_other_continuum_group == mContinuumGroup) || ( fabs(initial_delta)>1.0e-6
                    ) )
                    //THESE ARE THE CASES THAT NEED TO STORE THE INITIAL NEIGHBOURS
                    {

                        r_initial_neighbours.push_back(*ineighbour);

                        this->GetValue(PARTICLE_INITIAL_DELTA)[i]  =   initial_delta;
                        this->GetValue(PARTICLE_CONTACT_DELTA)[i]  =   initial_delta;

                        if (r_other_continuum_group == mContinuumGroup && (mContinuumGroup != 0) )
                        {this->GetValue(PARTICLE_CONTACT_FAILURE_ID)[i]=0; }
                        else                                              this-
                        >GetValue(PARTICLE_CONTACT_FAILURE_ID)[i]=1;


                    } // FOR THE CASES THAT NEED STORING INITIAL NEIGHBOURS

                    else mFailureId=1;

                    i++;

             }//if I found myself.

          } //end for: ParticleWeakIteratorType ineighbour
    }//SET INITIAL CONTACTS.
```

```
void SphericParticle::ComputeParticleContactForce(const ProcessInfo& rCurrentProcessInfo )

{

    KRATOS_TRY

    ParticleWeakVectorType& r_neighbours             = this->GetValue(NEIGHBOUR_ELEMENTS);

    vector<double>& r_VectorContactInitialDelta         = this-
>GetValue(PARTICLE_CONTACT_DELTA);

    // PROCESS INFO

    const array_1d<double,3>& gravity   = rCurrentProcessInfo[GRAVITY];

    double dt                           = rCurrentProcessInfo[DEM_DELTA_TIME];
    int damp_id                         = rCurrentProcessInfo[DAMP_TYPE];
    int type_id                         = rCurrentProcessInfo[FORCE_CALCULATION_TYPE];
    int rotation_OPTION                 = rCurrentProcessInfo[ROTATION_OPTION];

    int case_OPTION                     = rCurrentProcessInfo[CASE_OPTION];
    bool delta_OPTION;
    bool continuum_simulation_OPTION;

        switch (case_OPTION) {
            case 0:
                delta_OPTION = false;
                continuum_simulation_OPTION = false;
                break;
            case 1:
                delta_OPTION = true;
                continuum_simulation_OPTION = false;
                break;
            case 2:
                delta_OPTION = true;
                continuum_simulation_OPTION = true;
                break;
            case 3:
                delta_OPTION = false;
                continuum_simulation_OPTION = true;
                break;
            default:
                delta_OPTION = false;
                continuum_simulation_OPTION = false;
        }

    // GETTING PARTICLE PROPERTIES

    int continuum_group     = mContinuumGroup;

    double Tension          = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_TENSION);
    double Cohesion         = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_COHESION);
    double FriAngle         = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_FRICTION);
    double Friction         = tan( FriAngle / 180.0 * M_PI);

    double radius               = this->GetGeometry()[0].GetSolutionStepValue(RADIUS);
    double critic_damp_fraction = this->GetGeometry()[0].GetSolutionStepValue(VISCO_DAMP_COEFF);
    double mass                 = mRealMass;

    double young                = this->GetGeometry()[0].GetSolutionStepValue(YOUNG_MODULUS);
    double poisson              = this->GetGeometry()[0].GetSolutionStepValue(POISSON_RATIO);

    array_1d<double,3>& force           = this->GetGeometry()[0].GetSolutionStepValue(RHS);.


    array_1d<double,3> applied_force    = this->GetGeometry()
[0].GetSolutionStepValue(APPLIED_FORCE);

    force  = mass*gravity + applied_force;
```

```
        array_1d<double, 3 > & mRota_Moment = this->GetGeometry()
        [0].GetSolutionStepValue(PARTICLE_MOMENT);


        size_t iContactForce = 0;


        for(ParticleWeakIteratorType neighbour_iterator = r_neighbours.begin();
        neighbour_iterator != r_neighbours.end(); neighbour_iterator++)
        {
         // GETTING NEIGHBOUR PROPERTIES

            double other_radius                 = neighbour_iterator->GetGeometry()(0)-
            >GetSolutionStepValue(RADIUS);
            double other_critic_damp_fraction   = neighbour_iterator->GetGeometry()(0)-
            >GetSolutionStepValue(VISCO_DAMP_COEFF);
            double equiv_visc_damp_ratio        = (critic_damp_fraction +
            other_critic_damp_fraction) / 2.0;
            double other_young                  = neighbour_iterator->GetGeometry()
            [0].GetSolutionStepValue(YOUNG_MODULUS);
            double other_poisson                = neighbour_iterator->GetGeometry()
            [0].GetSolutionStepValue(POISSON_RATIO);
            double other_tension                = neighbour_iterator->GetGeometry()
            [0].GetSolutionStepValue(PARTICLE_TENSION);
            double other_cohesion               = neighbour_iterator->GetGeometry()
            [0].GetSolutionStepValue(PARTICLE_COHESION);
            double other_FriAngle               = neighbour_iterator->GetGeometry()
            [0].GetSolutionStepValue(PARTICLE_FRICTION);

            // CONTINUUM SIMULATING PARAMETERS:

            double initial_delta = 0.0;
            double CTension = 0.0;
            double CCohesion = 0.0;


            array_1d<double,3>& mContactForces  = this->GetValue(PARTICLE_CONTACT_FORCES)
            [iContactForce];


            if (continuum_simulation_OPTION && (continuum_group!=0) && (this-
            >GetValue(PARTICLE_CONTACT_FAILURE_ID)[iContactForce]==0))
            {

                CTension  = (Tension + other_tension)   * 0.5;
                CCohesion = (Cohesion + other_cohesion) * 0.5;
            }


            if( delta_OPTION && (iContactForce < r_VectorContactInitialDelta.size()) )
            {
                initial_delta = r_VectorContactInitialDelta[iContactForce];
            }

            // BASIC CALCULATIONS

            array_1d<double,3> other_to_me_vect = this->GetGeometry()(0)->Coordinates() -
            neighbour_iterator->GetGeometry()(0)->Coordinates();
            double distance                     = sqrt(other_to_me_vect[0] * other_to_me_vect[0] +
                                                  other_to_me_vect[1] * other_to_me_vect[1] +
                                                  other_to_me_vect[2] * other_to_me_vect[2]);

            double radius_sum                   = radius + other_radius;

            double indentation                  = radius_sum - distance - initial_delta;
            double equiv_radius     = 2* radius * other_radius / (radius + other_radius);
            double equiv_area       = M_PI * equiv_radius * equiv_radius;
            double equiv_poisson    = 2* poisson * other_poisson / (poisson + other_poisson);
            double equiv_young      = 2 * young * other_young / (young + other_young);
```

```
Friction                = tan( (FriAngle + other_FriAngle) * 0.5  / 180.0 * M_PI);

double kn               = M_PI * 0.5 * equiv_young * equiv_radius;
double ks               = kn / (2.0 * (1.0 + equiv_poisson));

// FORMING LOCAL CORDINATES

double NormalDir[3]           = {0.0};
double LocalCoordSystem[3][3] = {{0.0}, {0.0}, {0.0}};
NormalDir[0] = other_to_me_vect[0];
NormalDir[1] = other_to_me_vect[1];
NormalDir[2] = other_to_me_vect[2];
GeometryFunctions::ComputeContactLocalCoordSystem(NormalDir, LocalCoordSystem);

// VELOCITIES AND DISPLACEMENTS

array_1d<double, 3 > vel            = this->GetGeometry()(0)->GetSolutionStepValue(VELOCITY);
array_1d<double, 3 > other_vel      = neighbour_iterator->GetGeometry()(0)->GetSolutionStepValue(VELOCITY);

double DeltDisp[3] = {0.0};
double DeltVel [3] = {0.0};

DeltVel[0] = (vel[0] - other_vel[0]);
DeltVel[1] = (vel[1] - other_vel[1]);
DeltVel[2] = (vel[2] - other_vel[2]);

//DeltDisp in global cordinates

DeltDisp[0] = DeltVel[0] * dt;
DeltDisp[1] = DeltVel[1] * dt;
DeltDisp[2] = DeltVel[2] * dt;

    if ( rotation_OPTION == 1 )
    {

        double velA[3]   = {0.0};
        double velB[3]   = {0.0};
        double dRotaDisp[3] = {0.0};

        array_1d<double, 3 > AngularVel       = this->GetGeometry()(0)->FastGetSolutionStepValue(ANGULAR_VELOCITY);
        array_1d<double, 3 > Other_AngularVel = neighbour_iterator->GetGeometry()(0)->FastGetSolutionStepValue(ANGULAR_VELOCITY);

        double Vel_Temp[3]       = {      AngularVel[0],       AngularVel[1], AngularVel[2]};
        double Other_Vel_Temp[3] = {Other_AngularVel[0], Other_AngularVel[1], Other_AngularVel[2]};
        GeometryFunctions::CrossProduct(Vel_Temp,             LocalCoordSystem[2], velA);
        GeometryFunctions::CrossProduct(Other_Vel_Temp, LocalCoordSystem[2], velB);

        dRotaDisp[0] = -velA[0] * radius - velB[0] * other_radius;
        dRotaDisp[1] = -velA[1] * radius - velB[1] * other_radius;
        dRotaDisp[2] = -velA[2] * radius - velB[2] * other_radius;
        //////contribution of the rotation vel
        DeltDisp[0] += dRotaDisp[0] * dt;
        DeltDisp[1] += dRotaDisp[1] * dt;
        DeltDisp[2] += dRotaDisp[2] * dt;

    }//if rotation_OPTION

    double LocalDeltDisp[3] = {0.0};
    double LocalContactForce[3]  = {0.0};
    double GlobalContactForce[3] = {0.0};
    //double GlobalContactForceOld[3] = {0.0};
```

```
            GlobalContactForce[0] = mContactForces[0];
            GlobalContactForce[1] = mContactForces[1];
            GlobalContactForce[2] = mContactForces[2];

            GeometryFunctions::VectorGlobal2Local(LocalCoordSystem, DeltDisp, LocalDeltDisp);
            GeometryFunctions::VectorGlobal2Local(LocalCoordSystem, GlobalContactForce,
            LocalContactForce);

        // FORCES

            if ( (indentation > 0.0) || (this->GetValue(PARTICLE_CONTACT_FAILURE_ID)[iContactForce]
            == 0) )
            // This conditions take in acount the fact that the particles must remember their
            initial delta's between initial neighbours.

            {
                LocalContactForce[0] += - ks * LocalDeltDisp[0];  // 0: first tangential
                LocalContactForce[1] += - ks * LocalDeltDisp[1];  // 1: second tangential
                LocalContactForce[2] += - kn * LocalDeltDisp[2];  // 2: normal force
            }

            //ABSOLUTE METHOD FOR NORMAL FORCE (Allows non-linearity)

            if(type_id == 2)
                //  1--- incremental; 2 --- absolut i amb el cas hertzià
            {
                if(indentation > 0.0)
                {
                    LocalContactForce[2] = kn * pow(indentation, 1.5);
                }
                else
                {
                    LocalContactForce[2] = kn * indentation;
                }
            }

        // TENSION FAILURE

                if (-LocalContactForce[2] > (CTension * equiv_area))
                {
                    LocalContactForce[0]  = 0.0;
                    LocalContactForce[1]  = 0.0;
                    LocalContactForce[2]  = 0.0;

                    this->GetValue(PARTICLE_CONTACT_FAILURE_ID)[iContactForce] = 3.0;  //tensile
                    failure case.
                }

                // SHEAR FAILURE

                else
                {
                    double ShearForceMax = LocalContactForce[2] * Friction + CCohesion *
                    equiv_area;  // MOHR COULOMB MODEL.
                    double ShearForceNow = sqrt(LocalContactForce[0] * LocalContactForce[0]
                                           +      LocalContactForce[1] * LocalContactForce[1]);

                    //Not normal contribution for the tensile case

                    if(LocalContactForce[2] < 0.0)
                    {
                        ShearForceMax = CCohesion * equiv_area;
                    }

                    //No cohesion or friction, no shear resistance

                    if(ShearForceMax == 0.0)
                    {
                        LocalContactForce[0] = 0.0;
                        LocalContactForce[1] = 0.0;
```

```
                        }

                        else if(ShearForceNow > ShearForceMax)
                        {
                            LocalContactForce[0] = ShearForceMax / ShearForceNow *
                        LocalContactForce[0];
                            LocalContactForce[1] = ShearForceMax / ShearForceNow *
                        LocalContactForce[1];

                           this->GetValue(PARTICLE_CONTACT_FAILURE_ID)[iContactForce] = 4.0;
                        }
                    }

            // VISCODAMPING (applyied locally)

                    if (damp_id == 2 || damp_id == 3 )
                    {
                        double visco_damping[3] = {0,0,0};

                        if( abs(equiv_visc_damp_ratio * DeltVel[2]) > abs(LocalContactForce[2]) )
                        {visco_damping[2]= LocalContactForce[2]; }
                        else { visco_damping[2]= equiv_visc_damp_ratio * DeltVel[2]; }

                        LocalContactForce[0] = LocalContactForce[0] - visco_damping[0];
                        LocalContactForce[1] = LocalContactForce[1] - visco_damping[1];
                        LocalContactForce[2] = LocalContactForce[2] - visco_damping[2];
                    }

            // TRANSFORMING TO GLOBAL FORCES AND ADDING UP

                GeometryFunctions::VectorLocal2Global(LocalCoordSystem, LocalContactForce,
                GlobalContactForce);

                force[0] += GlobalContactForce[0];
                force[1] += GlobalContactForce[1];
                force[2] += GlobalContactForce[2];

            // SAVING INTO THE LOCAL SYSTEM ARRAYS FOR NEXT STEPS

                mContactForces[0] = GlobalContactForce[0];
                mContactForces[1] = GlobalContactForce[1];
                mContactForces[2] = GlobalContactForce[2];

                if ( rotation_OPTION == 1 )
                {

                double MA[3] = {0.0};
                GeometryFunctions::CrossProduct(LocalCoordSystem[2], GlobalContactForce, MA);
                mRota_Moment[0] -= MA[0] * radius;
                mRota_Moment[1] -= MA[1] * radius;
                mRota_Moment[2] -= MA[2] * radius;

                }

                iContactForce++;

        }//for each neaighbour

        KRATOS_CATCH("")

    }//ComputeParticleContactForce


    void SphericParticle::ApplyLocalForcesDamping(const ProcessInfo& rCurrentProcessInfo )

    {
        array_1d<double,3>& force           = this->GetGeometry()[0].GetSolutionStepValue(RHS);
        double LocalDampRatio   = this->GetGeometry()
        [0].GetSolutionStepValue(PARTICLE_LOCAL_DAMP_RATIO);
```

```
        // LOCAL DAMPING OPTION FOR THE UNBALANCED FORCES (IN GLOBAL CORDINATES).

          for (int iDof = 0; iDof < 3; iDof++)
          {
              if (this->GetGeometry()(0)->GetSolutionStepValue(VELOCITY)[iDof] > 0.0)
              {
                  force[iDof] = force[iDof] - LocalDampRatio * fabs(force[iDof]);
              }
              else
              {
                  force[iDof] = force[iDof] + LocalDampRatio * fabs(force[iDof]);
              }
          }

    } //ApplyLocalForcesDamping

  void SphericParticle::ApplyLocalMomentsDamping(const ProcessInfo& rCurrentProcessInfo )

  {
      array_1d<double, 3 > & RotaMoment       = this->GetGeometry()
  [0].GetSolutionStepValue(PARTICLE_MOMENT);
      double LocalDampRatio                   = this->GetGeometry()
  [0].GetSolutionStepValue(PARTICLE_LOCAL_DAMP_RATIO);

      // LOCAL DAMPING OPTION FOR THE UNBALANCED FORCES (IN GLOBAL CORDINATES).

      for (int iDof = 0; iDof < 3; iDof++)
      {
          if (this->GetGeometry()(0)->GetSolutionStepValue(ANGULAR_VELOCITY)[iDof] > 0.0)
          {
               RotaMoment[iDof] = RotaMoment[iDof] - LocalDampRatio * fabs(RotaMoment[iDof]);

          }
          else
          {
               RotaMoment[iDof] = RotaMoment[iDof] + LocalDampRatio * fabs(RotaMoment[iDof]);
          }
      }

  } //ApplyLocalMomentsDamping

      void SphericParticle::ComputeParticleRotationSpring(const ProcessInfo& rCurrentProcessInfo)
      {

      double dt                           = rCurrentProcessInfo[DEM_DELTA_TIME];


      double Tension        = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_TENSION);
      double Cohesion       = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_COHESION);
      double young          = this->GetGeometry()[0].GetSolutionStepValue(YOUNG_MODULUS);
      double poisson        = this->GetGeometry()[0].GetSolutionStepValue(POISSON_RATIO);
      double radius         = this->GetGeometry()[0].GetSolutionStepValue(RADIUS);
      double inertia        = this->GetGeometry()[0].GetSolutionStepValue(PARTICLE_INERTIA);

      array_1d<double, 3 > & mRota_Moment = GetGeometry()(0)-
      >FastGetSolutionStepValue(PARTICLE_MOMENT);

      ParticleWeakVectorType& rE             = this->GetValue(NEIGHBOUR_ELEMENTS);

      Vector & mRotaSpringFailureType  = this->GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE);

      size_t iContactForce = 0;

      for(ParticleWeakIteratorType ineighbour = rE.begin(); ineighbour != rE.end(); ineighbour++)
      {

          {

     array_1d<double, 3 > & mRotaSpringMoment  = this->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)[
      iContactForce ];
```

```
            double other_radius    = ineighbour->GetGeometry()(0)->FastGetSolutionStepValue(RADIUS);
            double other_young     = ineighbour->GetGeometry()
            [0].GetSolutionStepValue(YOUNG_MODULUS);
            double other_poisson   = ineighbour->GetGeometry()
            [0].GetSolutionStepValue(POISSON_RATIO);
            double other_tension   = ineighbour->GetGeometry()
            [0].GetSolutionStepValue(PARTICLE_TENSION);
            double other_cohesion  = ineighbour->GetGeometry()
            [0].GetSolutionStepValue(PARTICLE_COHESION);
            double other_inertia   = ineighbour->GetGeometry()(0)-
            >FastGetSolutionStepValue(PARTICLE_INERTIA);

            Tension  = (Tension  + other_tension ) * 0.5;
            Cohesion = (Cohesion + other_cohesion) * 0.5;

            double equiv_radius     = (radius + other_radius) * 0.5 ;
            double equiv_area       = M_PI * equiv_radius * equiv_radius;
            double equiv_poisson    = (poisson + other_poisson) * 0.5 ;
            double equiv_young      = (young  + other_young)  * 0.5;

            double kn               = equiv_young * equiv_area / (2.0 * equiv_radius);
            double ks               = kn / (2.0 * (1.0 + equiv_poisson));

            array_1d<double,3>& mContactForces = this->GetValue(PARTICLE_CONTACT_FORCES)[
            iContactForce ];

            array_1d<double,3> other_to_me_vect = GetGeometry()(0)->Coordinates() - ineighbour-
            >GetGeometry()(0)->Coordinates();

           /////Cfeng: Forming the Local Contact Coordinate system
            double NormalDir[3]           = {0.0};
            double LocalCoordSystem[3][3] = {{0.0}, {0.0}, {0.0}};
            NormalDir[0] = other_to_me_vect[0];
            NormalDir[1] = other_to_me_vect[1];
            NormalDir[2] = other_to_me_vect[2];
            GeometryFunctions::ComputeContactLocalCoordSystem(NormalDir, LocalCoordSystem);

            double LocalRotaSpringMoment[3]     = {0.0};
            double GlobalRotaSpringMoment[3]    = {0.0};
            double GlobalRotaSpringMomentOld[3] = {0.0};

            array_1d<double, 3 > AngularVel       = GetGeometry()(0)-
            >FastGetSolutionStepValue(ANGULAR_VELOCITY);
            array_1d<double, 3 > Other_AngularVel = ineighbour->GetGeometry()(0)-
            >FastGetSolutionStepValue(ANGULAR_VELOCITY);
            double DeltRotaDisp[3] = {0.0};
            DeltRotaDisp[0] = -(AngularVel[0] - Other_AngularVel[0]) * dt;
            DeltRotaDisp[1] = -(AngularVel[1] - Other_AngularVel[1]) * dt;
            DeltRotaDisp[2] = -(AngularVel[2] - Other_AngularVel[2]) * dt;

            double LocalDeltRotaDisp[3] = {0.0};
            GeometryFunctions::VectorGlobal2Local(LocalCoordSystem, DeltRotaDisp,
            LocalDeltRotaDisp);

            GlobalRotaSpringMomentOld[0] = mRotaSpringMoment[ 0 ];
            GlobalRotaSpringMomentOld[1] = mRotaSpringMoment[ 1 ];
            GlobalRotaSpringMomentOld[2] = mRotaSpringMoment[ 2 ];

            GeometryFunctions::VectorGlobal2Local(LocalCoordSystem, GlobalRotaSpringMomentOld,
            LocalRotaSpringMoment);

            double Inertia_I = (inertia + other_inertia) * 0.5;
            double Inertia_J = Inertia_I * 2.0;


            LocalRotaSpringMoment[0] +=  - Inertia_I * LocalDeltRotaDisp[0] * kn / equiv_area;
            LocalRotaSpringMoment[1] +=  - Inertia_I * LocalDeltRotaDisp[1] * kn / equiv_area;
            LocalRotaSpringMoment[2] +=  - Inertia_J * LocalDeltRotaDisp[2] * ks / equiv_area;
```

```
            ////Judge if the rotate spring is broken or not
            double GlobalContactForce[3]  = {0.0};
            double LocalContactForce [3]  = {0.0};

            GlobalContactForce[0] = mContactForces[ 0 ];
            GlobalContactForce[1] = mContactForces[ 1 ];
            GlobalContactForce[2] = mContactForces[ 2 ];
            GeometryFunctions::VectorGlobal2Local(LocalCoordSystem, GlobalContactForce,
            LocalContactForce);

            double ForceN  = LocalContactForce[2];
            double ForceS  = sqrt( LocalContactForce[0] * LocalContactForce[0] +
            LocalContactForce[1] * LocalContactForce[1]);
            double MomentS = sqrt(LocalRotaSpringMoment[0] * LocalRotaSpringMoment[0] +
            LocalRotaSpringMoment[1] * LocalRotaSpringMoment[1]);
            double MomentN = LocalRotaSpringMoment[2];

            //////bending stress and axial stress add together, use edge of the bar will failure
            first
            double TensiMax = -ForceN / equiv_area + MomentS        / Inertia_I * equiv_radius;
            double ShearMax = ForceS  / equiv_area + fabs(MomentN)  / Inertia_J * equiv_radius;

            if(TensiMax > Tension || ShearMax > Cohesion)
            {
                mRotaSpringFailureType[iContactForce] = 1;

                LocalRotaSpringMoment[0] = 0.0;
                LocalRotaSpringMoment[1] = 0.0;
                LocalRotaSpringMoment[2] = 0.0;
            }

            GeometryFunctions::VectorLocal2Global(LocalCoordSystem, LocalRotaSpringMoment,
            GlobalRotaSpringMoment);

            mRotaSpringMoment[ 0 ] = GlobalRotaSpringMoment[0];
            mRotaSpringMoment[ 1 ] = GlobalRotaSpringMoment[1];
            mRotaSpringMoment[ 2 ] = GlobalRotaSpringMoment[2];

            ////feedback, contact moment----induce by rotation spring
            mRota_Moment[0] -= GlobalRotaSpringMoment[0];
            mRota_Moment[1] -= GlobalRotaSpringMoment[1];
            mRota_Moment[2] -= GlobalRotaSpringMoment[2];
        }

        iContactForce++;
        }


    }//ComputeParticleRotationSpring


    void SphericParticle::DampMatrix(MatrixType& rDampMatrix, ProcessInfo& rCurrentProcessInfo){}

    void SphericParticle::GetDofList(DofsVectorType& ElementalDofList, ProcessInfo&
    CurrentProcessInfo){

        ElementalDofList.resize( 0 );

        for ( unsigned int i = 0; i < GetGeometry().size(); i++ )
        {
            ElementalDofList.push_back( GetGeometry()[i].pGetDof( DISPLACEMENT_X ) );
            ElementalDofList.push_back( GetGeometry()[i].pGetDof( DISPLACEMENT_Y ) );

            if ( GetGeometry().WorkingSpaceDimension() == 3 )
            {
                ElementalDofList.push_back( GetGeometry()[i].pGetDof( DISPLACEMENT_Z ) );
            }
        }
    }
```

```
        void SphericParticle::InitializeSolutionStep(ProcessInfo& rCurrentProcessInfo)
        {

          int case_opt         = rCurrentProcessInfo[CASE_OPTION];
          int mSwitch          = rCurrentProcessInfo[DUMMY_SWITCH];

          if( (mSwitch==0) && (case_opt!=0) )
          {
                SetInitialContacts(case_opt);
          }

          array_1d<double,3>& force           = this->GetGeometry()[0].GetSolutionStepValue(RHS);
          noalias(force)                      = ZeroVector(3);

        }
        void SphericParticle::FinalizeSolutionStep(ProcessInfo& CurrentProcessInfo){}

        void SphericParticle::Calculate(const Variable<double>& rVariable, double& Output, const
        ProcessInfo& rCurrentProcessInfo)
        {

            if (rVariable == DEM_DELTA_TIME)
            {
                double E = this->GetGeometry()(0)->FastGetSolutionStepValue(YOUNG_MODULUS);
                double K = E * M_PI * this->GetGeometry()(0)->FastGetSolutionStepValue(RADIUS);
                Output = sqrt( mRealMass / K);

                if(rCurrentProcessInfo[ROTATION_OPTION] == 1)
                {
                    Output = Output * 0.5; /
                }
            } //CRITICAL DELTA CALCULATION

            if (rVariable == PARTICLE_LOCAL_DAMP_RATIO)
            {
                int damp_id             = rCurrentProcessInfo[DAMP_TYPE];
                int rotation_OPTION     = rCurrentProcessInfo[ROTATION_OPTION];

                if (damp_id == 1 || damp_id == 3 )
                {
                   ApplyLocalForcesDamping( rCurrentProcessInfo );

                   if ( rotation_OPTION != 0 )
                   {
                       ApplyLocalMomentsDamping( rCurrentProcessInfo );
                   }
                }
            } //DAMPING
        }

        void SphericParticle::Calculate(const Variable<array_1d<double, 3 > >& rVariable,
        array_1d<double, 3 > & Output, const ProcessInfo& rCurrentProcessInfo){}
        void SphericParticle::Calculate(const Variable<Vector >& rVariable, Vector& Output, const
        ProcessInfo& rCurrentProcessInfo){}
        void SphericParticle::Calculate(const Variable<Matrix >& rVariable, Matrix& Output, const
        ProcessInfo& rCurrentProcessInfo){}

}  // namespace Kratos.
```

```
import DEM_explicit_solver_var
import time as timer

from KratosMultiphysics import *
from KratosMultiphysics.DEMApplication import *

#defining a model part for the solid part
my_timer=Timer();
solid_model_part = ModelPart("SolidPart");
#########################################

#introducing input file name
input_file_name = DEM_explicit_solver_var.problem_name

import sphere_strategy as SolverStrategy
SolverStrategy.AddVariables(solid_model_part)

#reading the solid part
gid_mode = GiDPostMode.GiD_PostBinary
multifile = MultiFileFlag.MultipleFiles
deformed_mesh_flag = WriteDeformedMeshFlag.WriteDeformed
write_conditions = WriteConditionsFlag.WriteConditions

gid_io = GidIO(input_file_name, gid_mode, multifile, deformed_mesh_flag, write_conditions)
model_part_io_solid = ModelPartIO(input_file_name)
model_part_io_solid.ReadModelPart(solid_model_part)

#setting up the buffer size: SHOULD BE DONE AFTER READING!!!
solid_model_part.SetBufferSize(2)

##adding dofs
SolverStrategy.AddDofs(solid_model_part)

#creating a solver object
dimension=DEM_explicit_solver_var.domain_size;
solver = SolverStrategy.ExplicitStrategy(solid_model_part, dimension);

##Obtaning options and values
integration_scheme = DEM_explicit_solver_var.Integration_Scheme
if (integration_scheme == 'forward_euler'):
    time_scheme = FowardEulerScheme()
elif (integration_scheme == 'mid_point_rule'):
    time_scheme = MidPointScheme()
elif (integration_scheme == 'const_average_acc'):
    time_scheme = ConstAverageAccelerationScheme()
else:
    print('scheme not defined')

solution_type = DEM_explicit_solver_var.SolutionType

if(solution_type == "Absolutal"):
    type_id = 2
else:
    type_id = 1

damp_ratio_type = DEM_explicit_solver_var.DampRatioType
if(damp_ratio_type == "ViscDamp"):
    damp_id = 2
elif(damp_ratio_type == "LocalDamp"):
    damp_id = 1
else:
    damp_id = 3

gravity = Vector(3)
gravity[0] = DEM_explicit_solver_var.gravity_x
gravity[1] = DEM_explicit_solver_var.gravity_y
gravity[2] = DEM_explicit_solver_var.gravity_z
solver.gravity=gravity
```

```
#options for the solver

continuum_option = DEM_explicit_solver_var.ContinuumOption
delta_option = DEM_explicit_solver_var.DeltaOption
search_radius_extension=DEM_explicit_solver_var.search_radius_extension

rotation_option =DEM_explicit_solver_var.RotationOption
rotation_spring_option=DEM_explicit_solver_var.RotationalSpringOption

if(delta_option=="OFF"):
  search_radius_extension=0.0;

solver.time_scheme=time_scheme
solver.type_id=type_id


if(continuum_option =="ON"):
  solver.continuum_simulating_OPTION=True

solver.search_radius_extension=search_radius_extension

if(delta_option =="ON"):
  solver.delta_OPTION=True

solver.search_radius_extension=search_radius_extension

if(rotation_option =="ON"):
  solver.rotation_OPTION=1
if(rotation_spring_option =="ON"):
  solver.rotation_spring_OPTION=1

#for critical time step calculation
solver.safety_factor = DEM_explicit_solver_var.dt_safety_factor

# time settings

final_time = DEM_explicit_solver_var.max_time
output_dt  = DEM_explicit_solver_var.output_dt
dt = DEM_explicit_solver_var.max_time_step

# bounding box

n_step_destroy_distant = DEM_explicit_solver_var.search_step
n_step_search = DEM_explicit_solver_var.search_step
solver.n_step_search = n_step_search
bounding_box_enlargement_factor = 2.0

extra_radius = 0.0
max_radius = 0.0
min_radius = 0.0
first_it = True

#calculation of search radius
for node in solid_model_part.Nodes:

  rad = node.GetSolutionStepValue(RADIUS)
  if rad > max_radius:
      max_radius = rad
  if first_it == True:
      min_radius = rad
      first_it = False
  if rad < min_radius:
      min_radius = rad

extra_radius = 2.5 * max_radius
prox_tol = 0.000001 * min_radius  #currently not in use.
bounding_box_enlargement_factor = max(1.0 + extra_radius, bounding_box_enlargement_factor)

solver.enlargement_factor = bounding_box_enlargement_factor
```

```
#Initialize the problem.

solver.Initialize()

#initializations
time = 0.0
step = 0
time_old_print = 0.0

current_pr_time = timer.clock()
current_real_time = timer.time()

print 'Calculation starts at instant: ' + str(current_pr_time)
while(time < final_time):

    print "TIME STEP = ", step

    time = time + dt

    #if ((step + 1) % n_step_destroy_distant == 0):

        #solver.Destroy_Particles(list_of_particles_pointers, solid_model_part)


    solid_model_part.CloneTimeStep(time)
    solid_model_part.ProcessInfo[TIME_STEPS] = step

    solver.Solve()

###########     GiD IO        ############

    time_to_print = time - time_old_print
    print str(time)

    if(time_to_print >= DEM_explicit_solver_var.output_dt):
        gid_io.InitializeMesh(time);
        gid_io.WriteSphereMesh(solid_model_part.GetMesh());
        gid_io.FinalizeMesh();
        gid_io.InitializeResults(time, solid_model_part.GetMesh());
        gid_io.WriteNodalResults(VELOCITY, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(DISPLACEMENT, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(RHS, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(RADIUS, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(PARTICLE_COHESION, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(PARTICLE_TENSION, solid_model_part.Nodes, time, 0)
        gid_io.WriteNodalResults(PARTICLE_FAILURE_ID, solid_model_part.Nodes, time, 0)

        if (rotation_option == 1):
            gid_io.WriteNodalResults(ANGULAR_VELOCITY, solid_model_part.Nodes, time, 0)
            gid_io.WriteNodalResults(MOMENT, solid_model_part.Nodes, time, 0)
        #gid_io.Flush()
        gid_io.FinalizeResults()
        time_old_print = time

    step += 1

print 'Calculation ends at instant: ' + str(timer.time())
elapsed_pr_time = timer.clock() - current_pr_time
elapsed_real_time = timer.time() - current_real_time
print 'Calculation ends at processing time instant: ' + str(timer.clock())
print 'Elapsed processing time: ' + str(elapsed_pr_time)
print 'Elapsed real time: ' + str(elapsed_real_time)
print (my_timer)
print "COMPLETED ANALYSIS"
```

```
#if !defined(KRATOS_EXPLICIT_SOLVER_STRATEGY)
#define  KRATOS_EXPLICIT_SOLVER_STRATEGY

#include "utilities/timer.h"

/* System includes */
#include <limits>
#include<iostream>
#include<iomanip>
#include <iostream>

/* External includes */
#ifdef _OPENMP
#include <omp.h>
#endif

#include "boost/smart_ptr.hpp"

/* Project includes */
#include "includes/define.h"
#include "utilities/openmp_utils.h"
#include "includes/model_part.h"
#include "solving_strategies/strategies/solving_strategy.h"
#include "solving_strategies/schemes/scheme.h"
#include "custom_elements/spheric_particle.h"
#include "includes/variables.h"

#include "custom_utilities/neighbours_calculator.h"
#include "custom_strategies/schemes/integration_scheme.h"

namespace Kratos
{
  template<
  class TSparseSpace,
  class TDenseSpace,
  class TLinearSolver>
  class ExplicitSolverStrategy : public  SolvingStrategy<TSparseSpace,TDenseSpace,TLinearSolver>
     {
      public:

          typedef SolvingStrategy<TSparseSpace,TDenseSpace,TLinearSolver> BaseType;
          typedef typename BaseType::TDataType TDataType;
          typedef typename BaseType::TBuilderAndSolverType TBuilderAndSolverType;
          typedef typename BaseType::TSchemeType TSchemeType;
          typedef typename BaseType::DofsArrayType DofsArrayType;
          typedef typename Element::DofsVectorType DofsVectorType;
          typedef ModelPart::NodesContainerType NodesArrayType;
          typedef ModelPart::ElementsContainerType ElementsArrayType;
          typedef ModelPart::ConditionsContainerType ConditionsArrayType;
          typedef ModelPart::NodesContainerType::ContainerType      NodesContainerType;
          typedef ModelPart::ElementsContainerType::ContainerType   ElementsContainerType;
          typedef ModelPart::ConditionsContainerType::ContainerType ConditionsContainerType;

      /// Pointer definition of ExplicitSolverStrategy
      KRATOS_CLASS_POINTER_DEFINITION(ExplicitSolverStrategy);

      /// Default constructor.
       ExplicitSolverStrategy(){}

       ExplicitSolverStrategy(ModelPart& model_part,
                              const int        dimension,
                              const double     damping_ratio,       masa
                              const double     fraction_delta_time,
                              const double     max_delta_time,
                              const double     n_step_search,
                              const double     safety_factor,
                              const bool       MoveMeshFlag,
                              const bool       delta_option,
                              const bool       continuum_simulating_option,
```

```
                    typename        IntegrationScheme::Pointer pScheme
                )
                : SolvingStrategy<TSparseSpace,TDenseSpace,TLinearSolver>(model_part,
                MoveMeshFlag),

                    mdimension(dimension)
                {

                   mdelta_option                = delta_option;
                   mcontinuum_simulating_option = continuum_simulating_option;
                   mvirtual_mass                = false;
                   mElementsAreInitialized      = false;
                   mConditionsAreInitialized    = false;
                   mCalculateOldTime            = false;
                   mSolutionStepIsInitialized   = false;
                   mInitializeWasPerformed      = false;
                   mComputeTime                 = false;
                   mInitialConditions           = false;
                   mdamping_ratio               = damping_ratio;
                   mfraction_delta_time         = fraction_delta_time;
                   mmax_delta_time              = max_delta_time;
                   molddelta_time               = 0.00;
                   mtimestep                    = 0.00;
                   mpScheme                     = pScheme;

                   mtimestep                    = max_delta_time;
                   mnstepsearch                 = n_step_search;
                   msafety_factor               = safety_factor;

                }
    /// Destructor.
    virtual ~ExplicitSolverStrategy(){}

    double Solve()
    {
      KRATOS_TRY

      std::cout<<std::fixed<<std::setw(15)<<std::scientific<<std::setprecision(5);
      ModelPart& r_model_part              = BaseType::GetModelPart();
      ProcessInfo& rCurrentProcessInfo      = r_model_part.GetProcessInfo();

      int time_step = rCurrentProcessInfo[TIME_STEPS];

      std::cout<<"-----------------------------------------------------------------------------"<<std::endl
      ;
      std::cout<<"                KRATOS DEM APPLICATION. TIME STEPS = "           <<  time_step
      <<std::endl;

      std::cout<<"-----------------------------------------------------------------------------"<<std::endl
      ;

      //STRATEGY:

      //0.PREVIOUS OPERATIONS

      if(mComputeTime==false){
          ComputeCriticalTime();
          mComputeTime = true;
      }
      //1.0
      InitializeSolutionStep();

      //1. Get and Calculate the forces
      GetForce();

          //1.1. Calculate Local Dampings
          ApplyLocalDampings();

      //2. Motion Integration
      ComputeIntermedialVelocityAndNewDisplacement();
```

```
      //3. Neighbouring search. Every N times.

      if ( (time_step + 1)%mnstepsearch == 0 )
      {
          SearchNeighbours(r_model_part,false); //extension option false;
      }

      std::cout <<"FINISHED SOLVE"<<std::endl;
      return 0.00;
      KRATOS_CATCH("")
    }

       void CalculateVirtualMass()
      {
          KRATOS_TRY

          if(mvirtual_mass == true)
          {
            ModelPart& r_model_part          = BaseType::GetModelPart();
            ElementsArrayType& pElements     = r_model_part.Elements();

            ProcessInfo& rCurrentProcessInfo  = r_model_part.GetProcessInfo();
;
            typename NodesArrayType::iterator inode;
            for(inode = r_model_part.NodesBegin(); inode != r_model_part.NodesEnd(); inode++)
            {
                inode->FastGetSolutionStepValue(NODAL_MASS) = 0.0;
            }

              typename ElementsArrayType::iterator it_begin=pElements.ptr_begin();
              typename ElementsArrayType::iterator it_end=pElements.ptr_end();
              for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
              {
                 double Young  = it->GetProperties()[YOUNG_MODULUS];
                 double Length = it->GetGeometry().Length();
                 double Volume = 0.0;
                 double VirtualMass = 0.0;

                 Element::GeometryType& geom = it->GetGeometry();

                 if (geom.size() == 1)
                 {
                    VirtualMass = Young * M_PI * it->GetGeometry()(0)-
                    >FastGetSolutionStepValue(RADIUS);
                    if(rCurrentProcessInfo[PARTICLE_IF_CAL_ROTATE] == 1)
                    {
                        VirtualMass = VirtualMass * 2.5;
                    }
                 }
                 else if (it->GetGeometry().Dimension() == 2 && geom.size() > 2)
                 {
                     Volume = it->GetGeometry().Area();

                     VirtualMass = Young / (Length * Length) * Volume;
                 }
                 else if (it->GetGeometry().Dimension() == 3 && geom.size() > 3 )
                 {
                     Volume = it->GetGeometry().Volume();

                     VirtualMass = Young / (Length * Length) * Volume;
                 }

                  for (unsigned int i = 0; i <geom.size(); i++)
                   {
                      double& mass = geom(i)->FastGetSolutionStepValue(NODAL_MASS);
                      mass  = mass + VirtualMass / (double)geom.size();
                   }
              }
          }
```

```
    mIfHaveCalVirtualMass = true;

    KRATOS_CATCH("")
}

void Initialize()
{
    KRATOS_TRY

ModelPart& r_model_part           = BaseType::GetModelPart();

//1. Search Neighbours with tolerance

bool extension_option = true;
SearchNeighbours(r_model_part,extension_option);

//2. Initializing elements
  if(mElementsAreInitialized == false)
     InitializeElements();
  mInitializeWasPerformed   = true;

// 3. Set Initial Contacts
  if(mdelta_option || mcontinuum_simulating_option){
      Set_Initial_Contacts(mdelta_option, mcontinuum_simulating_option);
  }

  KRATOS_CATCH("")
}

void GetForce()
{
  KRATOS_TRY

  Vector rhs_cond;

  ModelPart& r_model_part           = BaseType::GetModelPart();
  ProcessInfo& rCurrentProcessInfo  = r_model_part.GetProcessInfo();
  ElementsArrayType& pElements      = r_model_part.Elements();

  #ifdef _OPENMP
  int number_of_threads = omp_get_max_threads();
  #else
  int number_of_threads = 1;
   #endif

  vector<unsigned int> element_partition;
  OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

  unsigned int index = 0;

  #pragma omp parallel for private(index)
  for(int k=0; k<number_of_threads; k++)

  {

    typename ElementsArrayType::iterator it_begin=pElements.ptr_begin()+element_partition[k];
    typename ElementsArrayType::iterator it_end=pElements.ptr_begin()+element_partition[k+1];
    for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
      {

            (it)->CalculateRightHandSide(rhs_cond, rCurrentProcessInfo);

     } //loop over particles

  }// loop threads OpenMP

KRATOS_CATCH("")

}
```

```
        void ComputeIntermedialVelocityAndNewDisplacement()
        {
            ModelPart& r_model_part = BaseType::GetModelPart();
            mpScheme->Calculate(r_model_part);

        }

        void ComputeCriticalTime()
        {
            KRATOS_TRY

            ModelPart& r_model_part             = BaseType::GetModelPart();
            ProcessInfo& rCurrentProcessInfo    = r_model_part.GetProcessInfo();

            if(mvirtual_mass == true)
            {
                if(mtimestep > 0.9)
                {
                    mtimestep = 0.9;
                }

                std::cout<<"*****************Virtual Mass TimeStep is Used*****************"
                <<std::endl;
            }
            else
            {
                  double TimeStepTemp = 0.0;
                  ElementsArrayType& pElements     = r_model_part.Elements();

                  typename ElementsArrayType::iterator it_begin = pElements.ptr_begin();
                  typename ElementsArrayType::iterator it_end   = pElements.ptr_end();

                  for(ElementsArrayType::iterator it = it_begin; it!= it_end; it++)
                  {
                      it->Calculate(DEM_DELTA_TIME, TimeStepTemp, rCurrentProcessInfo);
                      KRATOS_WATCH(TimeStepTemp)
                      if(mtimestep > TimeStepTemp)
                      {
                          mtimestep = TimeStepTemp;
                      }
                  }

                  mtimestep = msafety_factor * mtimestep;

                  std::cout<<"*****************Real Mass TimeStep is Used*****************"
                  <<std::endl;
            }

            rCurrentProcessInfo[DEM_DELTA_TIME] = mtimestep;

            std::cout<<"*****************Calculating TimeStep Is "<<mtimestep<<  "*****************"
            <<std::endl;

            KRATOS_CATCH("")
        }

        void ApplyLocalDampings()
        {

            KRATOS_TRY

            ModelPart& r_model_part           = BaseType::GetModelPart();
            ProcessInfo& rCurrentProcessInfo  = r_model_part.GetProcessInfo();
            ElementsArrayType& pElements      = r_model_part.Elements();

            #ifdef _OPENMP
            int number_of_threads = omp_get_max_threads();
            #else
            int number_of_threads = 1;
```

```
      #endif

      vector<unsigned int> element_partition;
      OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

      unsigned int index = 0;

      #pragma omp parallel for private(index)
      for(int k=0; k<number_of_threads; k++)

      {

          typename ElementsArrayType::iterator
          it_begin=pElements.ptr_begin()+element_partition[k];
          typename ElementsArrayType::iterator
          it_end=pElements.ptr_begin()+element_partition[k+1];

          double dummy = 0.0;

          for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
          {

             it->Calculate(PARTICLE_LOCAL_DAMP_RATIO, dummy, rCurrentProcessInfo);

          } //loop over particles
      }// loop threads OpenMP

      KRATOS_CATCH("")

  }//Apply local damps

  void InitializeSolutionStep()
  {

    KRATOS_TRY

    ModelPart& r_model_part          = BaseType::GetModelPart();
    ProcessInfo& rCurrentProcessInfo  = r_model_part.GetProcessInfo();
    ElementsArrayType& pElements     = r_model_part.Elements();

    #ifdef _OPENMP
    int number_of_threads = omp_get_max_threads();
    #else
    int number_of_threads = 1;
     #endif

    vector<unsigned int> element_partition;
    OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

    unsigned int index = 0;

    #pragma omp parallel for private(index)
    for(int k=0; k<number_of_threads; k++)

    {

      typename ElementsArrayType::iterator it_begin=pElements.ptr_begin()+element_partition[k];
      typename ElementsArrayType::iterator it_end=pElements.ptr_begin()+element_partition[k+1];
      for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
        {

          (it)->InitializeSolutionStep(rCurrentProcessInfo); //we use this function to call the
          set initial contacts and the add continuum contacts.

       } //loop over particles

    }// loop threads OpenMP

  KRATOS_CATCH("")
```

```
    }

    void BoundingBoxUtility(double enlargement_factor)
    {

    KRATOS_TRY

    ModelPart& r_model_part              = BaseType::GetModelPart();
    ProcessInfo& rCurrentProcessInfo      = r_model_part.GetProcessInfo();

    Calculate_Surrounding_Bounding_Box(r_model_part,enlargement_factor);

    KRATOS_CATCH("")

    } //BoundingBoxUtility()

    void MoveMesh()
    {
    }

    void FinalizeSolutionStep()
    {
    }

    void CalculateEnergies()
    {
    }

protected:

private:

const unsigned int    mdimension;
unsigned int    minitial_conditions_size;
unsigned int    mcontact_conditions_size;
bool   mInitialCalculations;
bool   mElementsAreInitialized;
bool   mConditionsAreInitialized;
bool   mCalculateOldTime;
bool   mSolutionStepIsInitialized;
bool   mComputeTime;
bool   mInitializeWasPerformed;
bool   mInitialConditions;
bool   mdelta_option;
bool   mcontinuum_simulating_option;

bool   mvirtual_mass;

double mdamping_ratio;
double malpha_damp;
double mbeta_damp;
double mfraction_delta_time;
double mmax_delta_time;
double molddelta_time;
double mtimestep;
int    mnstepsearch;
double msafety_factor;

typename IntegrationScheme::Pointer mpScheme;

  void InitializeElements()
  {
      KRATOS_TRY
      ModelPart& r_model_part          = BaseType::GetModelPart();
      ElementsArrayType& pElements     = r_model_part.Elements();

      //Matrix MassMatrix;
      #ifdef _OPENMP
      int number_of_threads = omp_get_max_threads();
      #else
```

```
        int number_of_threads = 1;
         #endif

        vector<unsigned int> element_partition;
        OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

        #pragma omp parallel for
        for(int k=0; k<number_of_threads; k++)
        {
          typename ElementsArrayType::iterator it_begin=pElements.ptr_begin()+element_partition[k];
          typename ElementsArrayType::iterator it_end=pElements.ptr_begin()+element_partition[k+1];
          for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
            {

              (it)->Initialize();
           }
        }

       mElementsAreInitialized   = true;
       KRATOS_CATCH("")
      }


    void Set_Initial_Contacts(const bool& delta_OPTION, const bool& continuum_simulating_OPTION)
    {

        KRATOS_TRY

ModelPart& r_model_part          = BaseType::GetModelPart();
        ProcessInfo& rCurrentProcessInfo  = r_model_part.GetProcessInfo();
        ElementsArrayType& pElements     = r_model_part.Elements();

        #ifdef _OPENMP
        int number_of_threads = omp_get_max_threads();
        #else
        int number_of_threads = 1;
         #endif

        vector<unsigned int> element_partition;
        OpenMPUtils::CreatePartition(number_of_threads, pElements.size(), element_partition);

        unsigned int index = 0;

        #pragma omp parallel for private(index)
        for(int k=0; k<number_of_threads; k++)

        {
          typename ElementsArrayType::iterator it_begin=pElements.ptr_begin()+element_partition[k];
          typename ElementsArrayType::iterator it_end=pElements.ptr_begin()+element_partition[k+1];
          for (ElementsArrayType::iterator it= it_begin; it!=it_end; ++it)
            {

              (it)->InitializeSolutionStep(rCurrentProcessInfo); //we use this function to call the
set initial contacts and the add continuum contacts.

           } //loop over particles

        }// loop threads OpenMP

      KRATOS_CATCH("")
    }  //Set_Initial_Contacts

    void SearchNeighbours(ModelPart r_model_part,bool extension_option)
    {

      typedef DiscreteElement                                                    ParticleType;
      typedef ParticleType::Pointer                                              ParticlePointerType;
      typedef ElementsContainerType                                              ParticleContainerType;
      typedef WeakPointerVector<Element>                                         ParticleWeakVectorType;
      typedef typename std::vector<ParticlePointerType>
```

```
        ParticlePointerVectorType;
        typedef WeakPointerVector<Element>::iterator
        ParticleWeakIteratorType;
        typedef typename std::vector<ParticleType>::iterator                    ParticleIteratorType;
        typedef typename std::vector<ParticlePointerType>::iterator
        ParticlePointerIteratorType;
        typedef std::vector<double>                                             DistanceVectorType;
        typedef std::vector<double>::iterator                                   DistanceIteratorType;

        ProcessInfo& rCurrentProcessInfo = r_model_part.GetProcessInfo();
        ParticleContainerType& pElements = r_model_part.ElementsArray();

        if (mdimension == 2)

             Neighbours_Calculator<2, ParticleType>::Search_Neighbours(pElements, rCurrentProcessInfo,
        extension_option);

        else if (mdimension == 3)

             Neighbours_Calculator<3, ParticleType>::Search_Neighbours(pElements,  rCurrentProcessInfo,
        extension_option);

      }//SearchNeighbours

  }; // Class ExplicitSolverStrategy

}  // namespace Kratos.

#endif // KRATOS_FILENAME_H_INCLUDED  defined
```

```
#if !defined(KRATOS_CONSTANT_ACERAGE_ACCELERATION_SCHEME_H_INCLUDED )
#define  KRATOS_CONSTANT_ACERAGE_ACCELERATION_SCHEME_H_INCLUDED

// System includes
#include <string>
#include <iostream>

// External includes

// Project includes
#include "integration_scheme.h"
#include "includes/define.h"
#include "utilities/openmp_utils.h"
#include "includes/model_part.h"
#include "utilities/openmp_utils.h"

namespace Kratos
{

  class ConstAverageAccelerationScheme :  public IntegrationScheme
    {
    public:

      typedef ModelPart::NodesContainerType NodesArrayType;

      /// Pointer definition of ConstAverageAccelerationScheme
      KRATOS_CLASS_POINTER_DEFINITION(ConstAverageAccelerationScheme);

      /// Default constructor.
      ConstAverageAccelerationScheme(){}

      /// Destructor.
      virtual ~ConstAverageAccelerationScheme(){}

     void Calculate(ModelPart& model_part)
     {
        KRATOS_TRY
        KRATOS_WATCH("hola wwi")
        ProcessInfo& CurrentProcessInfo  = model_part.GetProcessInfo();
        NodesArrayType& pNodes           = model_part.Nodes();

        double aux          = 0;
        array_1d<double, 3 >  new_accel;
        array_1d<double, 3 >  prev_accel;
        double delta_t      =  CurrentProcessInfo[DELTA_TIME];

        vector<unsigned int> node_partition;
        NodesArrayType::iterator it_begin = pNodes.ptr_begin();
        NodesArrayType::iterator it_end   = pNodes.ptr_end();
        int number_of_threads             = 1; //OpenMPUtils::GetNumThreads();
        OpenMPUtils::CreatePartition(number_of_threads, pNodes.size(), node_partition);

        #pragma omp parallel for firstprivate(aux) shared(delta_t)
        for(int k=0; k<number_of_threads; k++)
        {
          NodesArrayType::iterator i_begin=pNodes.ptr_begin()+node_partition[k];
          NodesArrayType::iterator i_end=pNodes.ptr_begin()+node_partition[k+1];
          for(ModelPart::NodeIterator i=i_begin; i!= i_end; ++i)
          {

             array_1d<double, 3 > & vel            = i->FastGetSolutionStepValue(VELOCITY);
             array_1d<double, 3 > & displ          = i->FastGetSolutionStepValue(DISPLACEMENT);
             array_1d<double, 3 > & coor           = i->Coordinates();
             array_1d<double, 3 > & initial_coor   = i->GetInitialPosition();
             array_1d<double, 3 > & force          = i->FastGetSolutionStepValue(RHS);
             array_1d<double, 3 > & prev_force     = i->FastGetSolutionStepValue(RHS,1);
             const double mass                     = i->FastGetSolutionStepValue(NODAL_MASS);
```

```
            aux = delta_t / mass;

            new_accel = force / mass;
            prev_accel = prev_force / mass;

            if( ( i->pGetDof(DISPLACEMENT_X)->IsFixed() == false) && ( (i->IsFixed(VELOCITY_X))== false
            ) )
            {
                displ[0]  += delta_t * vel[0] + 0.5 * delta_t * delta_t * (prev_accel[0] +
                new_accel[0]);
                vel[0]    = vel[0] + 0.5 * delta_t * (prev_accel[0] + new_accel[0]);

                coor[0]   = initial_coor[0] + displ[0];
                prev_accel[0] = new_accel[0];

            }

            if( ( i->pGetDof(DISPLACEMENT_Y)->IsFixed() == false) && ( (i->IsFixed(VELOCITY_Y))== false
            ) )
            {
                displ[1]  += delta_t * vel[1] + 0.5 * delta_t * delta_t * (prev_accel[1] +
                new_accel[1]);
                vel[1]    = vel[1] + 0.5 * delta_t * (prev_accel[1] + new_accel[1]);

                coor[1]   = initial_coor[1] + displ[1];
                prev_accel[1] = new_accel[1];

            }

            if( (i->pGetDof(DISPLACEMENT_Z)->IsFixed() == false) && ( (i->IsFixed(VELOCITY_Z))== false
            ) )
            {
                displ[2]  += delta_t * vel[2] + 0.5 * delta_t * delta_t * (prev_accel[2] +
                new_accel[2]);
                vel[2]    = vel[2] + 0.5 * delta_t * (prev_accel[2] + new_accel[2]);

                coor[2]   = initial_coor[2] + displ[2];
                prev_accel[2] = new_accel[2];
            }
          }
        }
        KRATOS_CATCH(" ")
    }

     /// Turn back information as a string.
     virtual std::string Info() const
     {
       std::stringstream buffer;
       buffer << "ConstAverageAccelerationScheme" ;
       return buffer.str();
     }

     /// Print information about this object.
     virtual void PrintInfo(std::ostream& rOStream) const {rOStream <<
     "ConstAverageAccelerationScheme";}

     /// Print object's data.
     virtual void PrintData(std::ostream& rOStream) const {}

   protected:

   private:

     /// Assignment operator.
    ConstAverageAccelerationScheme& operator=(ConstAverageAccelerationScheme const& rOther)
    {
      return *this;
    }
```

```
     /// Copy constructor.
      ConstAverageAccelerationScheme(ConstAverageAccelerationScheme const& rOther)
      {
        *this = rOther;
      }

    }; // Class ConstAverageAccelerationScheme

  /// input stream function
  inline std::istream& operator >> (std::istream& rIStream,
                                    ConstAverageAccelerationScheme& rThis){return rIStream;}
  /// output stream function
  inline std::ostream& operator << (std::ostream& rOStream,
                                    const ConstAverageAccelerationScheme& rThis)
    {
      rThis.PrintInfo(rOStream);
      rOStream << std::endl;
      rThis.PrintData(rOStream);

      return rOStream;
    }
  ///@}

  ///@} addtogroup block

}  // namespace Kratos.

#endif // KRATOS_CONSTANT_ACERAGE_ACCELERATION_SCHEME_H_INCLUDED  defined
```

```
#if !defined(KRATOS_NEIGHBOURS_CALCULATOR )
#define  KRATOS_NEIGHBOURS_CALCULATOR

#include "includes/define.h"
#include "includes/model_part.h"
#include "spatial_containers/spatial_containers.h"
#include "containers/weak_pointer_vector.h"
#include "containers/pointer_vector.h"
#include "containers/pointer_vector_set.h"

#include "custom_utilities/discrete_particle_configure.h"

namespace Kratos {

    template<
    std::size_t TDim,
    class TParticle
    >

    class Neighbours_Calculator {
    public:
        typedef DiscreteParticleConfigure < 3 > ConfigureType;
        typedef TParticle Particle;
        typedef typename Particle::Pointer ParticlePointer;
        typedef ModelPart::ElementsContainerType::ContainerType ParticleVector;
        typedef ParticleVector::iterator ParticleIterator;
        typedef ModelPart::ElementsContainerType ParticlePointerVector;
        typedef ParticlePointerVector::iterator ParticlePointerIterator;
        typedef ConfigureType::PointType PointType;
        typedef ConfigureType::DistanceIteratorType DistanceIteratorType;
        typedef ConfigureType::ContainerType ContainerType;
        typedef ConfigureType::PointerType PointerType;
        typedef ConfigureType::IteratorType IteratorType;
        typedef ConfigureType::ResultContainerType ResultContainerType;
        typedef ConfigureType::ResultPointerType ResultPointerType;
        typedef ConfigureType::ResultIteratorType ResultIteratorType;
        typedef ConfigureType::ContactPairType ContactPairType;
        typedef ConfigureType::ContainerContactType ContainerContactType;
        typedef ConfigureType::IteratorContactType IteratorContactType;
        typedef ConfigureType::PointerContactType PointerContactType;
        typedef ConfigureType::PointerTypeIterator PointerTypeIterator;
        typedef WeakPointerVector<Element> ParticleWeakVector;
        typedef typename ParticleWeakVector::iterator ParticleWeakIterator;
        typedef ParticleWeakVector::ptr_iterator ParticleWeakIteratorType_ptr;
        typedef std::vector<double> DistanceVector;
        typename DistanceVector::iterator DistanceIterator;
        typedef std::vector<array_1d<double, 3 > > TangDisplacementsVectorType;
        typedef TangDisplacementsVectorType::iterator TangDisplacementsIteratorType;

        //*******************************************************************************

        typedef Bucket < TDim, Particle, ParticlePointerVector> BucketType;

        typedef BinsObjectDynamic <ConfigureType> bins;

        /// Pointer definition of Neighbour_calculator

        virtual ~Neighbours_Calculator() {
        };

        static void Search_Neighbours(ContainerType& pElements, ProcessInfo& rCurrentProcessInfo, bool
    extension_option) {

            KRATOS_TRY

            double radius_extend = 0.0;
            if (extension_option) radius_extend = rCurrentProcessInfo[SEARCH_RADIUS_EXTENSION];
            const int case_OPTION = rCurrentProcessInfo[CASE_OPTION];
            bool delta_OPTION = false;
```

```
        bool continuum_simulation_OPTION = false;

        switch (case_OPTION) {
            case 0:
                delta_OPTION = false;
                continuum_simulation_OPTION = false;
                break;
            case 1:
                delta_OPTION = true;
                continuum_simulation_OPTION = false;
                break;
            case 2:
                delta_OPTION = true;
                continuum_simulation_OPTION = true;
                break;
            case 3:
                delta_OPTION = false;
                continuum_simulation_OPTION = true;
                break;
            default:
                delta_OPTION = false;
                continuum_simulation_OPTION = false;
        }

        boost::timer kdtree_construction;
        unsigned int MaximumNumberOfResults = 100;
        ResultContainerType Results(MaximumNumberOfResults);
        DistanceVector ResultsDistances(MaximumNumberOfResults);
        bins particle_bin(pElements.begin(), pElements.end());
        boost::timer search_time;
        //************************************************************************************
        ********************************************************************

        ResultIteratorType results_begin;
        DistanceIteratorType result_distances_begin;
        //loop over all of the particles in the list to perform search
        for (IteratorType particle_pointer_it = pElements.begin();
                particle_pointer_it != pElements.end(); ++particle_pointer_it)
{
            Element::GeometryType& geom = (*particle_pointer_it)->GetGeometry();
            double search_radius = (1.0 + radius_extend) * geom(0)->GetSolutionStepValue(RADIUS);

            //find all of the new particles within the radius
            //looks which of the new particles is inside the radius around the working particle

            results_begin = Results.begin();
            result_distances_begin = ResultsDistances.begin();

            (*particle_pointer_it)->GetValue(NUMBER_OF_NEIGHBOURS) =
            particle_bin.SearchObjectsInRadius(*(particle_pointer_it),
            search_radius, results_begin, result_distances_begin, MaximumNumberOfResults) - 1;

            // SAVING THE OLD NEIGHBOURS, FORCES, FAILURE TYPES AND NUMBER OF NEIGHBOURS.

            ParticleWeakVector TempNeighbours;
            TempNeighbours.swap((*particle_pointer_it)->GetValue(NEIGHBOUR_ELEMENTS));

            vector< array_1d<double, 3 > > TempContactForce;
            TempContactForce.swap((*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES));

            vector< double > TempContactFailureId;
            TempContactFailureId.swap((*particle_pointer_it)-
            >GetValue(PARTICLE_CONTACT_FAILURE_ID));

            vector< double > TempContactDelta;
            TempContactDelta.swap((*particle_pointer_it)->GetValue(PARTICLE_CONTACT_DELTA));

            vector< double > TempRotateSpringFailType;
            TempRotateSpringFailType.swap((*particle_pointer_it)-
            >GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE));
```

```
            vector< array_1d<double, 3 > > TempRotateSpringMoment;
            TempRotateSpringMoment.swap((*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT));

            int n_neighbours = (*particle_pointer_it)->GetValue(NUMBER_OF_NEIGHBOURS);

            // CLEARING AND INITIALITZING.

            (*particle_pointer_it)->GetValue(NEIGHBOUR_ELEMENTS).clear();
            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES).clear();
            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FAILURE_ID).clear();
            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_DELTA).clear();

            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE).clear();
            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT).clear();

            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES).resize(n_neighbours);
            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FAILURE_ID).resize(n_neighbours);
            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_DELTA).resize(n_neighbours);

            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE).resize(n_neighbours);
            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT).resize(n_neighbours);

            // GETTING NEW NEIGHBOURS

            int neighbour_counter = 0;

            for (ResultIteratorType neighbour_it = Results.begin(); neighbour_counter != n_neighbours + 1; ++neighbour_it)
{
                if ((*particle_pointer_it)->Id() != (*neighbour_it)->Id()) { //the bins search finds the particle itself

                    (*particle_pointer_it)->GetValue(NEIGHBOUR_ELEMENTS).push_back(*neighbour_it);

                    // LOOP TO EXTEND THE VECTORS AND SET A 0.0 VALUE EACH TIME

                    size_t Notemp = ((*particle_pointer_it)->GetValue(NEIGHBOUR_ELEMENTS)).size();

                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp - 1] = ZeroVector(3);
                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FAILURE_ID)[Notemp - 1] = 1;
                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_DELTA)[Notemp - 1] = 0.0;
                    (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE)[Notemp - 1] = 0.0;
                    (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)[Notemp - 1] = ZeroVector(3);

                    // LOOP OVER THE OLD NEIGHBOURS FOR EVERY NEIGHBOUR TO CHECK IF IT'S AN EXISTING ONE AND COPYING THE OLD DATA

                    int OldNeighbourCounter = 0;
                    for (ParticleWeakIterator old_neighbour = TempNeighbours.begin(); old_neighbour != TempNeighbours.end(); old_neighbour++)
                    {
                        {

                            if ((old_neighbour.base())->expired() == false) {
                                if ((*neighbour_it)->Id() == old_neighbour->Id())
                                {
                                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp-1][0] = TempContactForce[OldNeighbourCounter][0];
                                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp-1][1] = TempContactForce[OldNeighbourCounter][1];
                                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp-1][2] = TempContactForce[OldNeighbourCounter][2];

                                    (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FAILURE_ID)
```

```
                                    [Notemp-1] = TempContactFailureId[OldNeighbourCounter];

                                    (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)
                                    [Notemp-1][0] = TempRotateSpringMoment[OldNeighbourCounter][0];
                                    (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)
                                    [Notemp-1][1] = TempRotateSpringMoment[OldNeighbourCounter][1];
                                    (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)
                                    [Notemp-1][2] = TempRotateSpringMoment[OldNeighbourCounter][2];

                                    (*particle_pointer_it)-
                                    >GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE)[Notemp-1] =
                                    TempRotateSpringFailType[OldNeighbourCounter];

                                    break;
                                } //end of its an old one??
                            } //end of expired?
                        } // end of its myself
                        OldNeighbourCounter++;
                    } //loop old neighbours

                    if (delta_OPTION) {

                        // LOOP OVER THE INITIAL NEIGHBOURS FOR EVERY NEIGHBOUR TO CHECK IF IT'S AN
                        INITIAL ONE AND THEN COPYING THE DELTA DATA
                        int InitialNeighboursCounter = 0;

                        if (((*particle_pointer_it)->GetValue(INITIAL_NEIGHBOUR_ELEMENTS)).size() !=
                        0) {
                            for (ParticleWeakIterator ini_neighbour = ((*particle_pointer_it)-
                            >GetValue(INITIAL_NEIGHBOUR_ELEMENTS)).begin(); ini_neighbour !=
                            ((*particle_pointer_it)->GetValue(INITIAL_NEIGHBOUR_ELEMENTS)).end();
                            ini_neighbour++)
                            {
                                   if ((ini_neighbour.base())->expired() == false) {

                                        if ((*neighbour_it)->Id() == ini_neighbour->Id())
                                        {
                                             (*particle_pointer_it)-
                                             >GetValue(PARTICLE_CONTACT_DELTA)[Notemp-1] =
                                             (*particle_pointer_it)-
                                             >GetValue(PARTICLE_INITIAL_DELTA)
                                             [InitialNeighboursCounter];
                                             break;
                                        }
                                   }

                                InitialNeighboursCounter++;
                            } // for initial neighbours
                        } //if u have some intial neigh
                    }//deltaOPTION
                }//end of the: if((*particle_pointer_it)->Id() != (*neighbour_it)->Id()

                ++neighbour_counter;

            } // for each neighbour, neighbour_it.

            //ADDING NOT FOUND NEIGHBOURS (the ones with negative identation still in tensile
            contact are not detected, but they are on the old neighbours list).

            int TempNeighbourCounter = 0;

            for (ParticleWeakIterator temp_neighbour = TempNeighbours.begin(); temp_neighbour !=
            TempNeighbours.end(); temp_neighbour++)
{
                if (TempContactFailureId[TempNeighbourCounter] == 0) // if they are not detached.
                {
                    if ((temp_neighbour.base())->expired() == false)
                    {
                        if ((*particle_pointer_it)->Id() != temp_neighbour->Id()) {
```

```
                        bool AlreadyAdded = false; //identifying if they are already found ot
                        not.

                        for (ParticleWeakIterator new_neighbour = (*particle_pointer_it)-
                        >GetValue(NEIGHBOUR_ELEMENTS).begin();
                                new_neighbour != (*particle_pointer_it)-
                                >GetValue(NEIGHBOUR_ELEMENTS).end(); new_neighbour++) {

                            if (new_neighbour->Id() == (temp_neighbour)->Id()) {

                                AlreadyAdded = true; //for the ones already found in the new
                                search.
                                break;
                            }
                        }

                        if (AlreadyAdded == false) //for the ones not included!
                        {

                            (*particle_pointer_it)-
                            >GetValue(NEIGHBOUR_ELEMENTS).push_back(TempNeighbours(TempNeighbour
                            Counter)); //adding the not found neighbours.

                            size_t Notemp = (*particle_pointer_it)-
                            >GetValue(PARTICLE_CONTACT_FORCES).size();
                            (*particle_pointer_it)-
                            >GetValue(PARTICLE_CONTACT_FORCES).resize(Notemp + 1); // adding one
                            more space for every missing neighbour.
                            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp][0]
                            = TempContactForce[TempNeighbourCounter][0]; //copying properties.
                            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp][1]
                            = TempContactForce[TempNeighbourCounter][1];
                            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FORCES)[Notemp][2]
                            = TempContactForce[TempNeighbourCounter][2];

                            (*particle_pointer_it)-
                            >GetValue(PARTICLE_CONTACT_FAILURE_ID).resize(Notemp + 1);
                            (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_FAILURE_ID)
                            [Notemp] = TempContactFailureId[TempNeighbourCounter];

                            (*particle_pointer_it)-
                            >GetValue(PARTICLE_ROTATE_SPRING_MOMENT).resize(Notemp + 1); //
                            adding one more space for every missing neighbour.
                            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)
                            [Notemp][0] = TempRotateSpringMoment[TempNeighbourCounter][0];
                            //copying properties.
                            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)
                            [Notemp][1] = TempRotateSpringMoment[TempNeighbourCounter][1];
                            (*particle_pointer_it)->GetValue(PARTICLE_ROTATE_SPRING_MOMENT)
                            [Notemp][2] = TempRotateSpringMoment[TempNeighbourCounter][2];

                            (*particle_pointer_it)-
                            >GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE).resize(Notemp + 1);
                            (*particle_pointer_it)-
                            >GetValue(PARTICLE_ROTATE_SPRING_FAILURE_TYPE)[Notemp] =
                            TempRotateSpringFailType[TempNeighbourCounter];

                            if (delta_OPTION) {
                                (*particle_pointer_it)-
                                >GetValue(PARTICLE_CONTACT_DELTA).resize(Notemp + 1);
                                (*particle_pointer_it)->GetValue(PARTICLE_CONTACT_DELTA)[Notemp]
                                = TempContactDelta[TempNeighbourCounter];
                            }

                            (*particle_pointer_it)->GetValue(NUMBER_OF_NEIGHBOURS)++;
                        }
                    }// end its myself???
                }//if not expired
            } //if not detached
```

```
                    TempNeighbourCounter++;
                }//loop over tempneigh
            }//Loop for evey particle as a base.

            KRATOS_CATCH("")
        }// Search_Neighbours

        virtual std::string Info() const {
            return "neighbour_calculator";
        }

        virtual void PrintInfo(std::ostream& rOStream) const {
        }

        virtual void PrintData(std::ostream& rOStream) const {
        }

    protected:

    private:

        inline void Clear(ModelPart::NodesContainerType::iterator node_it, int step_data_size) {
            unsigned int buffer_size = node_it->GetBufferSize();
            for (unsigned int step = 0; step < buffer_size; step++) {
                //getting the data of the solution step
                double* step_data = (node_it)->SolutionStepData().Data(step);
                //copying this data in the position of the vector we are interested in
                for (int j = 0; j < step_data_size; j++) {
                    step_data[j] = 0.0;
                }
            }
        }

        Neighbours_Calculator & operator=(Neighbours_Calculator const& rOther);

    }; // Class Neighbours_calculator

} // namespace Kratos.

#endif // KRATOS_NEIGHBOURS_CALCULATOR  defined
```

```cpp
#ifndef PARTICLE_CONFIGURE_H
#define PARTICLE_CONFIGURE_H

// System includes

// System includes
#include <string>
#include <iostream>
#include <cmath>
#include "utilities/spatial_containers_configure.h"

namespace Kratos
{

template <class TParticle>
class ParticleConfigure
{

public:

    enum {Dimension = 3};
    typedef TParticle                                                   ParticleType;
    typedef Point< 3, double>                                           PointType;
    typedef typename ParticleType::DistanceIteratorType                 DistanceIteratorType;
    typedef typename ParticleType::Pointer                              PointerType;
    typedef typename std::vector<typename ParticleType::Pointer>        ContainerType;
    typedef typename std::vector<PointerType>::iterator                 IteratorType;
    typedef ContainerType                                               ResultContainerType;
    typedef IteratorType                                                ResultIteratorType;

    /// Contact Pairs
    typedef ContactPair<PointerType>                                    ContactPairType;
    typedef  std::vector<ContactPairType>                               ContainerContactType;
    typedef  typename ContainerContactType::iterator                    IteratorContactType;
    typedef  typename ContainerContactType::value_type                  PointerContactType;

    /// Pointer definition of SpatialContainersConfigure
    KRATOS_CLASS_POINTER_DEFINITION(ParticleConfigure);

    ParticleConfigure() {};
    virtual ~ParticleConfigure() {}

    //**************************************************************************************
    static inline void CalculateBoundingBox(const PointerType& rObject, PointType& rLowPoint, PointType&
rHighPoint)
    {
        rLowPoint = *(rObject->GetPointerToCenterNode());
        rHighPoint = *(rObject->GetPointerToCenterNode());
        double radius = rObject->GetRadius();
        for(std::size_t i = 0; i < 3; i++)
        {
            rLowPoint[i]  += -radius;
            rHighPoint[i] += radius;
        }
    }

    static inline void CalculateBoundingBox(const PointerType& rObject, PointType& rLowPoint, PointType&
rHighPoint, const double& Radius)
    {
        rLowPoint = *(rObject->GetPointerToCenterNode());
        rHighPoint = *(rObject->GetPointerToCenterNode());
        for(std::size_t i = 0; i < 3; i++)
        {
            rLowPoint[i]  += -Radius;
            rHighPoint[i] += Radius;
        }
    }
    //**************************************************************************************
    static inline bool Intersection(const PointerType& rObj_1, const PointerType& rObj_2)
```

```
    {
        array_1d<double, 3> rObj_2_to_rObj_1 = rObj_1->GetPosition() - rObj_2->GetPosition();
        double distance_2 = rObj_2_to_rObj_1[0] * rObj_2_to_rObj_1[0] + rObj_2_to_rObj_1[1] *
        rObj_2_to_rObj_1[1] + rObj_2_to_rObj_1[2] * rObj_2_to_rObj_1[2];
        //distance_2 is the inter-center distance squared (from the definition of distance in search-
        structure.h, with operator (,))
        double radius_1 = rObj_1->GetRadius();
        double radius_2 = rObj_2->GetRadius();
        double radius_sum = radius_1 + radius_2;
        bool intersect = (distance_2 - radius_sum * radius_sum) <= 0;
        return intersect;
    }

    static inline bool Intersection(const PointerType& rObj_1, const PointerType& rObj_2, double Radius)
    {
        array_1d<double, 3> rObj_2_to_rObj_1 = rObj_1->GetPosition() - rObj_2->GetPosition();
        double distance_2 = rObj_2_to_rObj_1[0] * rObj_2_to_rObj_1[0] + rObj_2_to_rObj_1[1] *
        rObj_2_to_rObj_1[1] + rObj_2_to_rObj_1[2] * rObj_2_to_rObj_1[2];
        //distance_2 is the inter-center distance squared (from the definition of distance in search-
        structure.h, with operator (,))
        double radius_1 = Radius;es
        double radius_2 = rObj_2->GetRadius();
        double radius_sum = radius_1 + radius_2;
        bool intersect = (distance_2 - radius_sum * radius_sum) <= 0;
        return intersect;
    }
    //*******************************************************************************************

    static inline bool  IntersectionBox(const PointerType& rObject,  const PointType& rLowPoint, const
    PointType& rHighPoint)
    {
//        double separation_from_particle_radius_ratio = 0.1;
        array_1d<double, 3> center_of_particle = rObject->GetPosition();
        double radius = rObject->GetRadius();
        bool intersect = (rLowPoint[0] - radius <= center_of_particle[0] && rLowPoint[1] - radius <=
        center_of_particle[1] && rLowPoint[2] - radius <= center_of_particle[2] &&
                          rHighPoint[0] + radius >= center_of_particle[0] && rHighPoint[1] + radius >=
                          center_of_particle[1] && rHighPoint[2] + radius >= center_of_particle[2]);
        return  intersect;
    }

    static inline bool  IntersectionBox(const PointerType& rObject,  const PointType& rLowPoint, const
    PointType& rHighPoint, const double& Radius)
    {
//        double separation_from_particle_radius_ratio = 0.1;
        array_1d<double, 3> center_of_particle = rObject->GetPosition();
        double radius = Radius;
        bool intersect = (rLowPoint[0] - radius <= center_of_particle[0] && rLowPoint[1] - radius <=
        center_of_particle[1] && rLowPoint[2] - radius <= center_of_particle[2] &&
                          rHighPoint[0] + radius >= center_of_particle[0] && rHighPoint[1] + radius >=
                          center_of_particle[1] && rHighPoint[2] + radius >= center_of_particle[2]);
        return  intersect;
    }

    //*******************************************************************************************

    static inline void Distance(const PointerType& rObj_1, const PointerType& rObj_2, double& distance)
    {
        array_1d<double, 3> center_of_particle1 = rObj_1->GetPosition();
        array_1d<double, 3> center_of_particle2 = rObj_2->GetPosition();

        distance = sqrt((center_of_particle1[0] - center_of_particle2[0]) * (center_of_particle1[0] -
        center_of_particle2[0]) +
                        (center_of_particle1[1] - center_of_particle2[1]) * (center_of_particle1[1] -
                        center_of_particle2[1]) +
                        (center_of_particle1[2] - center_of_particle2[2]) * (center_of_particle1[2] -
                        center_of_particle2[2]) );
    }
    //*******************************************************************************************
```

```
    /// Turn back information as a string.
    virtual std::string Info() const
    {
        return " Spatial Containers Configure for Particles";
    }

    /// Print information about this object.
    virtual void PrintInfo(std::ostream& rOStream) const {}

    /// Print object's data.
    virtual void PrintData(std::ostream& rOStream) const {}

protected:

private:

    /// Assignment operator.
    ParticleConfigure& operator=(ParticleConfigure const& rOther);

    /// Copy constructor.
    ParticleConfigure(ParticleConfigure const& rOther);

}; // Class ParticleConfigure

/// input stream function
template <class TParticle>
inline std::istream& operator >> (std::istream& rIStream, ParticleConfigure<TParticle> & rThis)
{
    return rIStream;
}

/// output stream function
template <class TParticle>
inline std::ostream& operator << (std::ostream& rOStream, const ParticleConfigure<TParticle>& rThis)
{
    rThis.PrintInfo(rOStream);
    rOStream << std::endl;
    rThis.PrintData(rOStream);

    return rOStream;
}
///@}

}   // namespace Kratos.
#endif  /* PARTICLE_CONFIGURE_H */
```

```
from KratosMultiphysics import *
from KratosMultiphysics.DEMApplication import *
# Check that KratosMultiphysics was imported in the main script
#CheckForPreviousImport(

def AddVariables(model_part):

    model_part.AddNodalSolutionStepVariable(DISPLACEMENT)
    model_part.AddNodalSolutionStepVariable(VELOCITY)
    model_part.AddNodalSolutionStepVariable(RHS)
    model_part.AddNodalSolutionStepVariable(APPLIED_FORCE)
    model_part.AddNodalSolutionStepVariable(RADIUS)
    model_part.AddNodalSolutionStepVariable(PARTICLE_DENSITY)
    model_part.AddNodalSolutionStepVariable(PARTICLE_STIFFNESS)
    model_part.AddNodalSolutionStepVariable(YOUNG_MODULUS)
    model_part.AddNodalSolutionStepVariable(POISSON_RATIO)
    model_part.AddNodalSolutionStepVariable(NODAL_MASS)
    model_part.AddNodalSolutionStepVariable(PARTICLE_COEF_RESTITUTION)
    model_part.AddNodalSolutionStepVariable(PARTICLE_ZETA)
    model_part.AddNodalSolutionStepVariable(IS_STRUCTURE)
    model_part.AddNodalSolutionStepVariable(PARTICLE_MATERIAL)
    model_part.AddNodalSolutionStepVariable(PARTICLE_CONTINUUM)
    model_part.AddNodalSolutionStepVariable(PARTICLE_COHESION)
    model_part.AddNodalSolutionStepVariable(PARTICLE_FRICTION)
    model_part.AddNodalSolutionStepVariable(PARTICLE_TENSION)
    model_part.AddNodalSolutionStepVariable(PARTICLE_LOCAL_DAMP_RATIO)
    model_part.AddNodalSolutionStepVariable(PARTICLE_FAILURE_ID)
    model_part.AddNodalSolutionStepVariable(PARTICLE_INERTIA)
    model_part.AddNodalSolutionStepVariable(ANGULAR_VELOCITY)
    model_part.AddNodalSolutionStepVariable(PARTICLE_MOMENT)
    model_part.AddNodalSolutionStepVariable(PARTICLE_MOMENT_OF_INERTIA)
    model_part.AddNodalSolutionStepVariable(PARTICLE_ROTATION_ANGLE)

    print "variables for the explicit solver added correctly"

def AddDofs(model_part):

    for node in model_part.Nodes:
    #adding dofs
        node.AddDof(DISPLACEMENT_X,REACTION_X);
        node.AddDof(DISPLACEMENT_Y,REACTION_Y);
        node.AddDof(DISPLACEMENT_Z,REACTION_Z);
        node.AddDof(VELOCITY_X,REACTION_X);
        node.AddDof(VELOCITY_Y,REACTION_Y);
        node.AddDof(VELOCITY_Z,REACTION_Z);

    print "dofs for the DEM solution added correctly"

class ExplicitStrategy:

    def __init__(self,model_part,domain_size):

        self.model_part                     = model_part
        self.domain_size                    = domain_size
        self.damping_ratio                  = 0.00;
        self.penalty_factor                 = 10.00
        self.max_delta_time                 = 0.05;
        self.fraction_delta_time            = 0.90;
        self.MoveMeshFlag                   = True;
        self.time_scheme                    = FowardEulerScheme();
        self.gravity                        = Vector(3)
        self.gravity[0] = 0.0
        self.gravity[1] = -9.81
        self.gravity[2] = 0.0
        self.delta_time                     = 0.00001;
```

```
        #type of problem:

        self.delta_OPTION                   = False
        self.continuum_simulating_OPTION    = False
        self.case_OPTION                    = 0
        self.rotation_OPTION                = 0
        self.rotation_spring_OPTION         = 0

        #problem specific parameters

        self.force_calculation_type_id      =1
        self.damp_id                        =1
        self.search_radius_extension        = 0.0
        self.dummy_switch                   =0

        #problem utilities
        self.enlargement_factor             = 1;
        self.n_step_search                  = 1;
        self.safety_factor                  = 1; #for critical time step

    ########################################################################

    def Initialize(self):

        self.model_part.ProcessInfo.SetValue(GRAVITY, self.gravity)
        self.model_part.ProcessInfo.SetValue(DELTA_TIME, self.delta_time)

        if(self.delta_OPTION==True):
            if(self.continuum_simulating_OPTION==True): self.case_OPTION = 2
            else: self.case_OPTION = 1
        elif(self.delta_OPTION==False):
            if(self.continuum_simulating_OPTION==False): self.case_OPTION = 0
            else: self.case_OPTION = 3

        self.model_part.ProcessInfo.SetValue(CASE_OPTION, self.case_OPTION)
        self.model_part.ProcessInfo.SetValue(ROTATION_OPTION, self.rotation_OPTION)
        self.model_part.ProcessInfo.SetValue(ROTATION_SPRING_OPTION, self.rotation_spring_OPTION)
        self.model_part.ProcessInfo.SetValue(FORCE_CALCULATION_TYPE, self.force_calculation_type_id)
        self.model_part.ProcessInfo.SetValue(DAMP_TYPE, self.damp_id)
        self.model_part.ProcessInfo.SetValue(SEARCH_RADIUS_EXTENSION, self.search_radius_extension)
        self.model_part.ProcessInfo.SetValue(DUMMY_SWITCH, self.dummy_switch)

        #creating the solution strategy
        self.solver = ExplicitSolverStrategy(self.model_part, self.domain_size,  self.damping_ratio,
        self.fraction_delta_time, self.delta_time, self.n_step_search, self.safety_factor,
                                             self.MoveMeshFlag, self.delta_OPTION,
                                             self.continuum_simulating_OPTION, self.time_scheme)

        self.solver.Initialize()
        self.model_part.ProcessInfo.SetValue(DUMMY_SWITCH, 1)

    ########################################################################
    def Solve(self):
        (self.solver).Solve()

     ########################################################################

    def Calculate_Model_Surrounding_Bounding_Box(self, enlargement_factor):
        self.solver.BoundingBoxUtility()
```

References

1. Cundall, P.A., *A computer model for simulating progressive, large-scale movements in blockly rock systems.*, in *Symposium Soc. Internat Mécanique des Roches* 1971: Nancy. p. 2-8.
2. Pande, G., Beer, G. and Williams, J.R., *Numerical Modeling in Rock Mechanics.* John Wiley and Sons, 1990.
3. Munjiza, A., *The combined Finite-Discrete Element Method*. 2004: John Wiley & Sons.
4. Cundall, P.A. and O.D. L–Strack, *A discrete numerical model for granular assemblies.* Geotechnique, 1979. 29(1): p. 47-65.
5. Walizer, L.E. and J.F. Peters, *A bounding box search algorithm for DEM simulation.* Computer Physics Communications, 2011. 182(2): p. 281-288.
6. Perkins, E. and J.R. Williams, *CGrid: neighbor searching for many body simulation*, in *4th Int. Conf. on Analysis of Discontinuous Deformation*. 2001: Glasgow, UK.
7. Munjiza, A. and K.R.F. Andrews, *NBS contact detection algorithm for bodies of similar size.* International Journal for Numerical Methods in Engineering, 1998. 43: p. 131-49.
8. Raschdorf, S. and M. Kolonko, *Loose octree: a datastructure for the simulation of polydisperse particle packings*. 2009, Clausthal University of Technology.
9. G. Nezami, E., et al., *Shortest link method for contact detection in discrete element method.* International Journal for Numerical and Analytical Methods in Geomechanics, 2006. 30(8): p. 783-801.
10. Nezami, E.G., et al., *A fast contact detection algorithm for 3-D discrete element method.* Computers and Geotechnics, 2004. 31(7): p. 575-587.
11. Feng, Y., *Discrete Element Methods – Theory & Practice.* International Symposium / UK-China Summer School on Discrete Element Methods and Numerical Modelling of Discontinuum Mechanics (Beijing DEM'08), 2008.
12. Han, K., Y.T. Feng, and D.R.J. Owen, *Polygon-based contact resolution for superquadrics.* International Journal for Numerical Methods in Engineering, 2006. 66: p. 485-501.
13. Yung-ming, C., C. Wensheng, and G. Xiurun. *Procedure to detect the contact of three-dimensional blocks using penetration edges method*. in *Discrete Element Methods. Numerical Modeling of Discontinua*. 2002. Santa Fe, New Mexico, USA: American Society of Civil Engineers.
14. Song, Y., R. Turton, and F. Kayihan, *Contact detection algorithms for DEM simulations of tablet-shaped particles.* Powder Technology, 2006. 161 p. 32 - 40.
15. H. Kruggel-Emden, M.S., S. Wirtza, and V. Scherera., *Selection of an appropriate time integration scheme for the discrete element method (dem).* Computers & Chemical Engineering, 2008: p. 32(10):2263{2279, .
16. Bray., C.O.S.a.J.D., *Selecting a suitable time step for discrete element simulations that use the central difference time integration scheme.* Engineering Computations, 2004: p. 21(2/3/4):278-303.
17. Hsieh, Y.L., H. Huang, T. and Jeng, F., *Interpretations on how the macroscopic mechanical behavior of sandstone affected by microscopic properties.* Engineering Geology, 2008. 110.
18. Tavarez, F.A. and M.E. Plesha, *Discrete element method for modelling solid and particulate materials.* International Journal for Numerical Methods in Engineering, 2006. 70 (4): p. 379 - 404.
19. Huang, H.a.E.D., *Intrinsic length scales in tool-rock interaction.* International Journal of Geomechanics, 2008. 8(1):3944.
20. Huang, H., *Discrete element modeling of tool-rock interaction. .* 1999.

21. Labra, C. and E. Oñate, *High-density sphere packing for discrete element method simulations.* COMMUNICATIONS IN NUMERICAL METHODS IN ENGINEERING, 2008.
22. Rojek, J., et al., *Discrete Element Modelling of Rock Cutting*
Particle-Based Methods, E. Oñate and R. Owen, Editors. 2011, Springer Netherlands. p. 247-267.