Czech Technical University in Prague, Faculty of Civil Engineering

&

Université Grenoble I – Joseph Fourier, École doctorale I-MEP2

PhD thesis

in computational mechanics

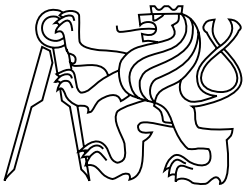
presented by

Václav Šmilauer

defended June 24th 2010

Cohesive Particle Model using the Discrete Element Method on the Yade Platform

defense committee		
Zdeněk Bittnar	professor, CTU Prague	president
Jan Vítek	professor, Metrostav	reviewer
Ali Limam	professor, INSA Lyon	reviewer
Bořek Patzák	assoc. professor, CTU Prague	examinator
Bruno Chareyre	assoc. professor, UJF Grenoble	examinator
Yijun Li	PhD, industry	examinator
Bernhard Winkler	PhD, industry	guest
Milan Jirásek	professor, CTU Prague	supervisor
Laurent Daudeville	professor, UJF Grenoble	supervisor



ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta stavební

ve spolupráci s Université Grenoble I – Joseph Fourier

Doktorský studijní program: Stavební inženýrství

Studijní obor: Fyzikální a materiálové inženýrství

Mgr. Ing. Václav Šmilauer

Cohesive Particle Model using the Discrete Element Method on the Yade Platform

Model kohezivních částic pomocí metody diskrétních prvků na platformě Yade

Disertační práce k získání akademického titulu Ph.D.

Školitelé: Prof. Ing. Milan Jirásek, DrSc.
Prof. Laurent Daudeville

Praha, Červen 2010

Acknowledgements

I am deeply indebted to numerous people at organizational, professional, inter-personal and intra-personal levels. Although they are too many to be mentioned here one by one, the concern of omitting someone does not present good reason to not mention anyone at all. Suffices to say that omissions are not disacknowledgements and that order of text should not suggest importance.

With the hope of giving back, directly or indirectly, all I received, I would like to express, therefore, my sincere thanks to (in no particular order)

- thesis supervisors Milan Jirásek at Czech Technical University in Prague, for his patience and personal and collegial attitude; Laurent Daudeville at Université Joseph Fourier in Grenoble, for his greatly appreciated support during difficult beginnings of the PhD,
- Ministère de l'enseignement supérieur et de la recherche, Grantová Agentura České republiky and the industrial partner for funding;
- my colleagues Anton Gladky, Sergei Dorofeenko, Bruno Chareyre, Jan Kozicki, Fréderic Victor Donzé, Chiara Modenese, David Mašín, Claudio Tamagnini, Michal Kotrč, Vít Šmilauer, Růžena Chamrová, Denis Davydov for questions, discussions and challenges;
- my parents;
- administrative support of Linda Fournier, Anne-Cécile Combe, Carole Didonato, Věra Pokorná, Alexandra Kurfürstová, Daniela Ambrožová, Veronika Kabilková, for giving human face to inhuman paperwork;
- my students, for their questions and criticism, and for the honor of becoming friend of several of them;
- authors of great open-source codes, especially: TEX and friends (XHLATEX), Python, GNU Compiler Collection, Boost libraries, Matplotlib, Linux kernel, Ubuntu, Debian, Vim, IPE, Bazaar, Launchpad.net;
- Lukáš Bernard and Rémi Cailletaud for professional and supportive IT backing;
- my friends Ludmila Divišová, Eva Romancovová, Helena Schalková, Berenika Kučerová, Mikuláš Kosák, Jan Kopecký, Jan Pospíšil, Marie Kindermannová, Klára Mesanyová, Jitka Špičková, Hana Šantrůčková, Meri Lampinen, Petr Máj, Stéphanie Vignand, Benedikt Vangeli, Jana Havlová, Jiří Holba, Daniel Suk, Justina Trlifajová, Michal Hrouzek, Michal Knotek, Martin Ježek, Marie Kriegelová, Kateřina Pekárková, Helena Svobodová for their support, tolerance, inspiration and everything else;
- my music soulmates Václav Hausenblas, David Čížek, Marek Novotný, Zbyněk Polívka, Jakub Klár, Denisa Martínková, Tereza Rejšková, Magdalena Klárová, Laurent Coq, Beata Hlavenková, Ondřej Pivec, Joel Frahm, Brad Mehldau, Michael Wollny, Kurt Rosenwinkel, Wayne Shorter, Joshua Redman, Sidonius Karez, Johann Sebastian Bach, John Eliot Gardiner for inspiration which would be infinite if my mind were infinitely open;
- Mariana Cecilie Svobodová for true bōdhisattva compassion;
- my teachers of openness Miloš Hrdý, Milada Hrdá, Tomáš Vystrčil, Ivan Špička, Mary Irene Bockover, Carl Gustav Jung, Marie-Louise von Franz, Rob Preece, Vladimír Holan, Robert Musil, Václav Černý, Irvin D. Yalom and others;
- Μουσική & शून्यता.

This thesis was elaborated jointly at:

Laboratoire 3S-R, Domaine Universitaire, BP53, F-38041 Grenoble Cedex 9, France.

Department of Structural Mechanics, Faculty of Civil Engineering, Czech Technical University in Prague, Thákurova 7, CZ-16629 Praha 6, Czech Republic.

Summary

This thesis describes implementation of particle-based model of concrete using the Discrete Element Method (DEM) using the Yade platform. DEM discretizes given domain using packing of (spherical) particles of which motion is governed via local contact laws and Newton's equations. Continuity is modeled via pre-established cohesive contacts between particles, while discontinuous behavior arises naturally from their failure. Concrete is modeled as a homogeneous material, where particles are purely discretization units (not representing granula, mortar or porosity); the local contact law features damage, plasticity and viscosity and calibration procedures are described in detail.

This model was implemented on the Yade platform, substantially enhanced in the course of our work and described for the first time in all its aspects here. As platform for explicit dynamic simulations, in particular the DEM, it is designed as highly modular toolkit of reusable algorithms. The computational part is written in c++ for efficiency and supports shared-memory parallel computation. Python, popular scripting language, is used for rapid and concise simulation setup, control and post-processing; Python also provides full access to most internal data. Good practices (documentation in particular) leading to sustainable development are encouraged by the framework.

Keywords: concrete, discrete element method, dynamic, numerical programming

Résumé

Cette thèse décrit un modèle de béton utilisant la méthode des éléments discrets (DEM) et le code de calcul Yade. La DEM discrétise un volume avec des particules (sphériques) dont les mouvements sont déterminés par des lois de comportement locales et les équations de Newton. La continuité du matériau est représentée par des contacts cohésifs entre les particules; les discontinuités apparaissent naturellement lors de l'endommagement de ces contacts. Le béton est considéré comme un matériau homogène; les particules ne sont qu'une méthode particulière de discrétisation et ne représentent pas la géométrie des granulats, du ciment ou des vides; la loi du comportement locale comprend l'endommagement, la plasticité et la viscosité; la calibration du modèle est décrite en détail.

Ce modèle a été implementé dans la plateforme Yade, profondément enrichie pendant ce travail; cette thèse décrit pour la première fois de manière complète le code de calcul Yade. Si Yade est prévu principalement pour la DEM, la modularité et la possibilité d'utiliser grandes parties du code dans le développement de nouvelles approches (re-utilisabilité) y sont tout de même des éléments importants. La partie calcul est programmée en c++ pour la performance et le calcul parallèle (mémoire partagée). Des scripts en langage python, l'un des plus répandus des langage de script, sont utilisés pour décrire les simulations de manière rapide et concise, contrôler l'exécution et post-traiter les résultats; Python permet l'accès aux données internes en cours de simulation. La pérennité des développements est encouragée par la plateforme, en particulier par l'exigence de documentation.

Mots clés: béton, méthode des éléments discrets, dynamique, programmation numérique

Contents

Notation

x	scalar
$\boldsymbol{\nu}$	3d vector
$\lvert\boldsymbol{\nu}\rvert$	(euclidean) norm of vector ν
$\widehat{\boldsymbol{\nu}}$	normalized vector $\boldsymbol{\nu}$, i.e. $\boldsymbol{\nu}/\lvert\boldsymbol{\nu}\rvert$
$\widetilde{\boldsymbol{\nu}}$	vector ν expressed in local coordinates
$\mathbf{u}\cdot\boldsymbol{\nu}$	scalar product of vectors $\mathbf{u}$ and $\boldsymbol{\nu}$
$\mathbf{u}\times\boldsymbol{\nu}$	vector product of $\mathbf{u}$, $\boldsymbol{\nu}$
$\mathbf{a}\otimes\mathbf{b}$	outer product of tensors $\mathbf{a}$, $\mathbf{b}$
δ_{ij}	the Kronecker delta, $\delta_{ij}=1\Leftrightarrow i=j$, $\delta_{ij}=0\Leftrightarrow i\neq j$
Δt	current timestep
x^{-}	$x(t-\Delta t)$
$x^{\ominus}$	$x\left(t-\frac{\Delta t}{2}\right)$
x°	$x(t)$, current value
$x^{\oplus}$	$x\left(t+\frac{\Delta t}{2}\right)$
x^{+}	$x(t+\Delta t)$
$q_{\mathbf{u}}$, q_{ϑ}	axis and angle decomposition of quaternion q
$\lVert q\rVert$	norm of quaternion q
q^{*}	conjugate of quaternion q ($\equiv$ inverse quaternion, if $\lvert q\rvert=1$)
$\emptyset$	empty set
$\lvert P\rvert$	cardinality of set P
$\mathcal{O}(n^2)$	algorithm with n^2 complexity
$\dot{x}$	$\partial x/\partial t$
$\ddot{x}$	$\partial^2 x/\partial t^2$
$\langle x\rangle$	$\min(0,x)$, positive part of x
$f\simeq g$	f is 1^{st} order approximate of g
$f\cong g$	f is 2^{nd} order approximate of g
$\hat{x}$, $\hat{y}$, $\hat{z}$	canonical base vectors of $\mathbb{R}^3$
$\hat{i}$, $\hat{j}$, $\hat{k}$	base imaginary numbers
$H(x)$	the Heaviside function, $H(x)=1\Leftrightarrow x\geq 0$, $H(x)=0\Leftrightarrow x<0$

Unless explicitily stated otherwise, it is assumed that quaternions have unit length.

E, G	macroscopic Young's and shear moduli [Pa]
K_N, K_T	spring normal and tangent stiffness [Nm^{-1}]
k_N, k_T	interaction normal and tangent moduli [Pa]
φ	friction angle

Introduction

This thesis is situated in the field of computational mechanics, which comprises mechanics, informatics and programming.

The goal of the research project was modeling of massive fracturing of concrete at small scale during high-rate processes. Traditional continuum-based modeling techniques, in particular the Finite Element Method (FEM), are designed (and efficient) for modeling continuum using discretization techniques, while discontinuities are introduced using relatively complicated extensions of the method (such as X-FEM). On the other hand, particle-based methods start from discrete entities and might obtain continuum-like behavior as an addition to the method. A discrete model with added continuous material features was chosen for our modeling task (rather than continuum-based model with added discontinuities), for discontinuous processes were predominant; because of high-rate effects, usage of a dynamic model was desirable. All those criteria led naturally to the Discrete Element Method (DEM) framework, in which the new concrete model (CPM, Concrete Particle Model) was formulated. This model was derived by applying concepts from continuum mechanics (plasticity, damage, viscosity) onto discrete particles, while trying to assure appropriate continuous behavior of particle arrangements which are sufficiently large to smear away individual particles.

As I spent the first year of PhD studies in Grenoble getting acquainted with Yade, a then-emerging open-source software platform targeted mainly at DEM, it was naturally the platform chosen for the implementation of the concrete model. Since my work on Yade during the first year concerned software engineering rather than mechanics, it was only later that I had to find out that Yade was not ready to be used as-is for serious modeling, by only plugging a new model into it. Substantial changes had to be made, which progressively covered all aspects of the program; that made me the lead developer of the Yade project in the 2007–2010 period, focusing on usability, documentation and performance.

The thesis is divided in two parts.

The first part pertains to mechanics. The DEM itself is situated amongst other modeling techniques in chapter 1. Then, the DEM is formulated mathematically in chapter 2. Chapter 3 presents the concrete model formulated in the DEM framework and implemented in Yade.

The second part is dedicated to Yade as software: it is presented from the point of view of a user and of programmer. Generated documentation for Yade classes and modules is in appendices (C and D), as it is unlikely to be read as continuous text. Besides that, some classes were documented by their respective authors and not me (see repository history for details); it is also for this reason that they are separated from the main text body.

In order to make this thesis useful beyond its defense, most parts of this thesis are conceived as part of online Yade documentation at `https://www.yade-dem.org/sphinx/`. Automatically generated documentation in appendices C and D is already part of it, while chapter 2 should become reference for the algorithms in the future.

Part I.

Concrete particle model

1. Discrete Element Method

1.1. Characterisation

Usage of particle models for mechanical problems originated in geomechanics in 1979, in a famous paper by Cundall & Strack named *A discrete numerical model for granular assemblies* [8]. Granular medium is modeled in a discrete fashion: circular non-deformable particles representing granula can collide, exerting forces on one another, while being governed by Newton's laws of dynamic equilibrium; these laws are integrated using an explicit scheme, proceeding by a given Δt at each step. Particles have both translational and rotational degrees of freedom. Forces coming from collision of particles are computed using penalty functions, which express simple spring-like contact, elastic in the normal sense (connecting both spheres' centers) and elasto-plastic with Mohr-Coulomb criterion in the perpendicular plane.

Since then, the initial formulation has been substantially enhanced in many ways, such as introduction of non-spherical particles, particle deformability, cohesion and fracturing. These features will be discussed later, after we establish the distinction between the Discrete Element Method (DEM) and other particle-based methods; naturally, such classification is only operational and does not imply inexistence or even impossibility of various intermediate positions.

Mass-spring models, where nodes have only 3 degrees of freedom and their contacts only transmit normal force. Mass is lumped into nodes without any associated volume, without collision detection and creation of new contacts; initial contacts are pre-determined. Such models were used to model solid fracture (where dynamic effects were predominant [70]) or elastic cloth behavior [50].

Rigid body-spring model (RBSM), where polygonal/polyhedral particles are connected with multiple spring elements across the neighbor's contact sides/areas; particles have no deformability on their own, their elasticity is represented by said spring elements [27]; an implicit integration scheme is employed. This method is similar to FEM with zero-thickness interface elements [4], but leads to a smaller stiffness matrix, since displacements of any point belonging to a particle are uniquely determined from displacements/rotations of the particle itself. Nagai et al. [41] uses elaborate plasticity functions for shear loading.

Lattice models family, where nodes are connected with truss or beam elements. Typically, nodes carry no mass and static equilibrium is sought; they do not occupy volume either, hence no new contacts between nodes will be created. Both regular and irregular lattices were studied. Properties of connecting elements are determined from local configuration, such as geometry of the Voronoï cell of each node and local material heterogeneity (e.g. mortar vs. aggregates in concrete).

Originally, lattice was representing elastic continuum; the equivalence was established for both truss [21] and beam [55] elements. Later, obvious enhancements such as brittle beam failure were introduced. Lattice models nicely show the *emergence* of relatively complex structural behavior, although fairly simple formulas govern local processes.

Some models find themselves on the border between DEM and lattice models, e.g. by considering sphere packing for finding initial contacts, but only finding a static solution later [17].

1.2. Feature variations

This initial formulation has seen numerous improvements and enhancements since; we roughly follow the feature classification in a nice overview by Bićanić [4].

1.2.1. Space dimension

Originally, 2d simulation space was used, as it reduces significantly computational costs. With the increase of available computing power, the focus has shifted to 3d space. The number of dimensions also qualitatively influences some phenomena, such as dilation and percolation.

1.2.2. Particle geometry

Discs (2d) and spheres (3d) were first choices for the ease of contact detection, as sphere overlap is determined from spatial distance of their centroids, without the need to consider their orientation. Approximating more complex shapes by spheres can be done by building up rigid aggregates ("clumps"), which might try to approximate real surfaces [49].

At further development stages, elliptical shapes, general quadrics and implicit superquadrics all have been used. The advantage is that, unlike for other complex shapes, the solid predicate (whether a given point is inside or outside) is computed very fast; however, to detect collision of 2 such shapes, one particle is usually discretized into a set of surface points, and the predicate of the other particle is applied on those points.

Polygons/polyhedra with explicit vertices are frequently used. Exact detection of contact might be tricky and has to distinguish combinations of features that enter the interaction: edge-edge, vertex-edge, vertex-facet, etc.

Surface singularities at vertices can be problematic, since direction of the repulsive force (penalty function) is not clearly defined. Several solutions are employed: rounding edges and vertices, replacing them with aligned spheres and cylinders; formulating the penalty function volumetrically (e.g. in the direction of the least principal moment of inertia of the overlapping volume, which is the direction the volume will decrease the fastest); using some more detailed knowledge about the particles in question, such as tracking the *common plane* defined by arrangement of vertices and faces during movement of particles [43].

Arbitrary discrete functions have been employed for particle shapes as well.

1.2.3. Contact detection algorithm

Naïve checking of all possible couples soon leads to performance issues with increasing number of particles, having $\mathcal{O}(n^2)$ complexity. Moreover, for complex shapes exact contact evaluation can be complicated. Therefore, the detection is generally done in 2 passes, the first one eliminating as many contacts as possible:

1. Possible contacts based on approximate volume representation are sought; different particle geometries within the simulation do not have to be distinguished at this stage explicitly. Most non-trivial algorithms are $\mathcal{O}(n \log n)$, which makes them unusable for very large number of particles although $\mathcal{O}(n)$ algorithms were already published [38, 40].

2. Possible contacts are evaluated, considering the exact geometry of particles. In this pass, all possible combinations of shapes must be handled.

1.2.4. Boundary conditions

Boundaries can be imposed at space level or at particle level.

Space boundaries are, in particular, periodic boundaries, where particles leaving the periodic cell on one side enter on the other side; for the periodicity condition to hold, the cell must be parallelepiped-shaped. The periodic boundary eliminates boundary-related distortions of simulations; it also prevents localization unless orientation of the cell matches that of the localization plane.

Particle-level boundaries may be as simple as fixing some particles in space; other boundaries, which aim at a more faithful representation of experimental setups, might be [4]

flexible where a chain of particles is tied together by links (keeping the circumference constant) or

hydrostatic where forces corresponding to constant hydrostatic stress are exerted on particles on the boundary.

In both cases, a definition of a particle "on the boundary" is needed; for spheres, Voronoï (Dirichlet) tessellation [3] might be used with weighting employed to account for different sphere radii.

1.2.5. Particle deformability

The initial Cundall's formulation supposed that particles themselves are rigid (their geometry undergoes no changes) and it is only at the interaction level that elastic behavior occurs.

Early attempts at deformability considered discrete elements as deformable quadrilaterals (Bićanić [4] calls this method "discrete finite elements"). Several further development branches were followed later:

Combined finite/discrete element method (FDEM) [39, 37] discretizes each discrete element internally into multiple finite elements. Special care must be taken to account for the interplay between external (discrete element boundary), internal (finite element stresses) and inertial (mass) forces. By allowing fracturing inside the FEM domain, the discrete element can be effectively crushed and then fall apart into multiple discrete elements. This method uses explicit integration and usual inter-particle contact handling via penalty functions, distributing external forces onto surface FE nodes.

Discontinuous deformation analysis (DDA) [58] superimposes polynomial approximation of the strain field on the movement of the rigid body centroid. Evolutions in this direction included increasing the polynomial order as well as dividing the discrete element in a number of sub-blocks with a lower-degree polynomial. Implicit integration is used, while taking contact constraints into account.

Non-rigid aggregates do not constitute a method on its own but account for deformable particles by clustering primitive, rigid particles using a special type of cohesive bonds, creating a lattice-like deformable solid representation.

1.2.6. Cohesion and fracturing

Cohesive interactions ("bonds") between particles have been used to represent non-granular media. If fracturing is to take place the formulation is usually derived from continuum elastic-plastic-damage models [4], though not necessarily [19]; such an approach only allows inter-particle fracture. Intra-particle fracture can be emulated with non-rigid aggregates or properly simulated in FDEM where localization and remeshing of the originally continuous FEM domain allows progressive fracturing [37].

1.2.7. Time integration scheme

Integrating motion equations in discrete elements system needs special consideration. Penalty functions expressing repulsive forces (for cohesion-less setups) have some order of discontinuity when contact occurs. This favors explicit integration methods, which are indeed used in the most discrete element codes. The numerical stability criterion reads $\Delta t < \sqrt{m/k}$, where m is mass and k is contact stiffness; this equation has physical meaning for corresponding continuum material, limiting distance of elastic wave propagation within one step, $\Delta x = \sqrt{E/\rho}\Delta t$, to the radius of spherical particle ($\Delta x \leq r$).

In implicit integration schemes, global stiffness matrix is assembled and dynamic equilibrium is sought; this allows for larger Δt values, but the computation is more complex. In DDA, to assure non-singularity of the matrix in the absence of contact between blocks, artificial low spring stiffness terms might have to be added [4].

1.3. Micro-macro behavior relations

Although for reasons of an easier fracture description continuum may be represented by particles with special bonds in such way that desired macroscopic behavior is obtained, the correspondence of bond-level and domain-level properties is far from clear and has been the subject of considerable research. The two levels are colloquially called *micro* and *macro*, although this bears no reference to "microscopic" scale as opposed to meso/macroscopic scale as otherwise used in material modeling.

Elastic properties (Young's modulus E and Poisson's ratio ν) already pose some problems as to what values should be expected based on given micro-parameters: stiffnesses in the normal (K_N) and shear (K_T) sense, in case of spherical DEM with 3-DoF contacts. It follows from dimensional analysis that $\nu = \nu(K_T/K_N)$ and $E = K_N f(K_T/K_N)$. Analytical solutions of this problem start from either of the following suppositions:

Regular lattice can be used for hand-derivation of macroscopic E and ν from bond-level strain-stress formulas. For 2D, the Potapov et al. [46] article derives macroscopic properties on 2 differently oriented hexagonal lattices and then shows they will converge when refined, making the limit value valid for any orientation; Potapov et al. [47] gives numerical evidence for the result. For 3D, Wang and Mora [69] derives equations on regular dense packing of spheres (hexagonal close-packed and face-centered cubic) using energy considerations.

General lattice. For the 3D case, the principle of energy conservation is used. External work (imposed via homogeneous strain field) and potential energy (expressed as stored elastic energy of bonds) are equaled, resulting in closed-form solution for E and ν. This can be done in a discrete fashion (by summing bonds between particles) or using integral form of the imaginary continuous "lattice". Such an approach is used by Liao et al. [34] and leads to integrals analogous to the microplane theory; the analogy is explicitly used by Kuhl et al. [31] in the context of DEM.

Liao et al. [34] shows, however, that the general homogeneous strain field biases the result and proposes a "best-fit" strategy (in both a discrete and integral form), showing that the actual numerical results are between the two; this is used, e.g., by Hentz et al. [20] for DEM-based concrete model calibration.

2. Problem formulation

In this chapter, we mathematically describe general features of explicit DEM simulations, with some reference to Yade implementation of these algorithms.

They are given roughly in the order as they appear in simulation; first, two particles might establish a new interaction, which consists in

1. detecting collision between particles;
2. creating new interaction and determining its properties (such as stiffness); they are either precomputed or derived from properties of both particles;

Then, for already existing interactions, the following is performed:

1. strain evaluation;
2. stress computation based on strains;
3. force application to particles in interaction.

This simplified description serves only to give meaning to the ordering of sections within this chapter. A more detailed description of this *simulation loop* is given later.

2.1. Collision detection

2.1.1. Generalities

Exact computation of collision configuration between two particles can be relatively expensive (for instance between **Sphere** and **Facet**). Taking a general pair of bodies i and j and their "exact"[1] spatial predicates (called **Shape** in Yade) represented by point sets P_i, P_j the detection generally proceeds in 2 passes:

1. fast collision detection using approximate predicate $\tilde{P}_i$ and $\tilde{P}_j$; they are pre-constructed in such a way as to abstract away individual features of P_i and P_j and satisfy the condition

$$\forall x \in \mathbb{R}^3 : x \in P_i \Rightarrow x \in \tilde{P}_i \tag{2.1}$$

   (likewise for P_j). The approximate predicate is called "bounding volume" (**Bound** in Yade) since it bounds any particle's volume from outside (by virtue of the implication). It follows that $(P_i \cap P_j) \neq \emptyset \Rightarrow (\tilde{P}_i \cap \tilde{P}_j) \neq \emptyset$ and, by applying *modus tollens*,

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) = \emptyset \Rightarrow \left(P_i \cap P_j\right) = \emptyset \tag{2.2}$$

   which is a candidate exclusion rule in the proper sense.

2. By filtering away impossible collisions in (2.2), more expensive, exact collision detection algorithms can be run on possible interactions, filtering out remaining spurious couples $(\tilde{P}_i \cap \tilde{P}_j) \neq \emptyset \wedge \left(P_i \cap P_j\right) = \emptyset$. These algorithms operate on P_i and P_j and have to be able to handle all possible combinations of shape types.

It is only the first step we are concerned with here.

[1] In the sense of precision admissible by numerical implementation.

2.1.2. Algorithms

Collision evaluation algorithms have been the subject of extensive research in fields such as robotics, computer graphics and simulations. They can be roughly divided in two groups:

Hierarchical algorithms which recursively subdivide space and restrict the number of approximate checks in the first pass, knowing that lower-level bounding volumes can intersect only if they are part of the same higher-level bounding volume. Hierarchy elements are bounding volumes of different kinds: octrees [26], bounding spheres [22], k-DOP's [28].

Flat algorithms work directly with bounding volumes without grouping them in hierarchies first; let us only mention two kinds commonly used in particle simulations:

Sweep and prune algorithm operates on axis-aligned bounding boxes, which overlap if and only if they overlap along all axes. These algorithms have roughly $\mathcal{O}(n \log n)$ complexity, where n is number of particles as long as they exploit *temporal coherence* of simulation (2.1.2).

Grid algorithms represent continuous R^3 space by a finite set of regularly spaced points, leading to very fast neighbor search; they can reach the $\mathcal{O}(n)$ complexity [38] and recent research suggests ways to overcome one of the major drawbacks of this method, which is the necessity to adjust grid cell size to the largest particle in the simulation (Munjiza et al. [40], the "multistep" extension).

Temporal coherence expresses the fact that motion of particles in simulation is not arbitrary but governed by physical laws. This knowledge can be exploited to optimize performance.

Numerical stability of integrating motion equations dictates an upper limit on Δt (sect. 2.5.6) and, by consequence, on displacement of particles during one step. This consideration is taken into account in Munjiza et al. [40], implying that any particle may not move further than to a neighboring grid cell during one step allowing the $\mathcal{O}(n)$ complexity; it is also explored in the periodic variant of the sweep and prune algorithm described below.

On a finer level, it is common to enlarge $\tilde{P}_i$ predicates in such a way that they satisfy the (2.1) condition during *several* timesteps; the first collision detection pass might then be run with stride, speeding up the simulation considerably. The original publication of this optimization by Verlet [65] used enlarged list of neighbors, giving this technique the name *Verlet list*. In general cases, however, where neighbor lists are not necessarily used, the term *Verlet distance* is employed.

2.1.3. Sweep and prune

Let us describe in detail the sweep and prune algorithm used for collision detection in Yade (class **InsertionSortCollider**). Axis-aligned bounding boxes (**Aabb**) are used as $\tilde{P}_i$; each **Aabb** is given by lower and upper corner $\in R^3$ (in the following, $\tilde{P}_i^{x0}$, $\tilde{P}_i^{x1}$ are minimum/maximum coordinates of $\tilde{P}_i$ along the x-axis and so on). Construction of **Aabb** from various particle **Shape**'s (such as **Sphere**, **Facet**, **Wall**) is straightforward, handled by appropriate classes deriving form **BoundFunctor** (**Bo1_Sphere_Aabb**, **Bo1_Facet_Aabb**, ...).

Presence of overlap of two **Aabb**'s can be determined from conjunction of separate overlaps of intervals along each axis (fig. 2.1):

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) \neq \emptyset \Leftrightarrow \bigwedge_{w \in \{x,y,z\}} \left[\left(\left(\tilde{P}_i^{w0}, \tilde{P}_i^{w1}\right) \cap \left(\tilde{P}_j^{w0}, \tilde{P}_j^{w1}\right)\right) \neq \emptyset\right] \tag{2.3}$$

where (a, b) denotes interval in R.

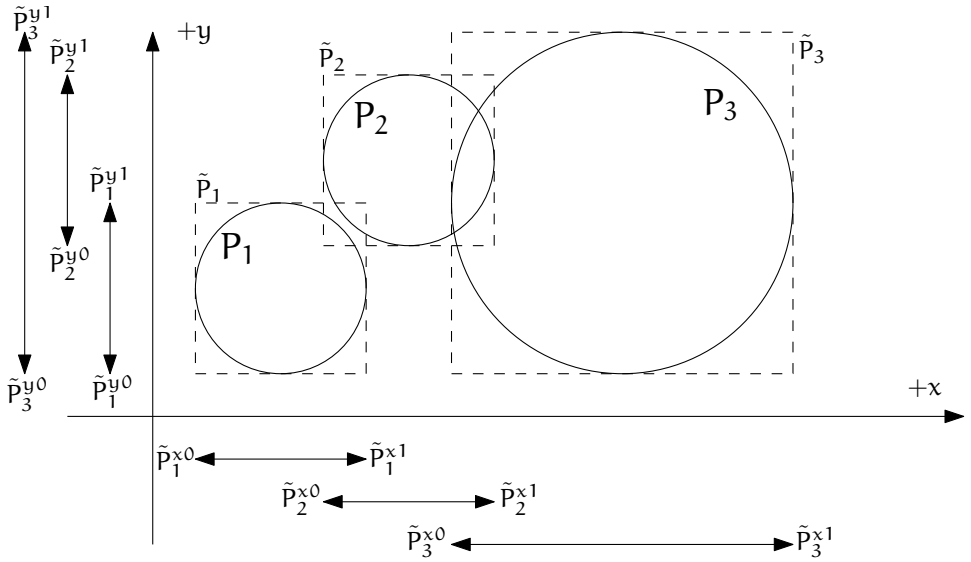


Figure 2.1.: Sweep and prune algorithm (shown in 2D), where **Aabb** of each sphere is represented by minimum and maximum value along each axis. Spatial overlap of **Aabb**'s is present if they overlap along all axes. In this case, $\tilde{P}_1 \cap \tilde{P}_2 \neq \emptyset$ (but note that $P_1 \cap P_2 = \emptyset$) and $\tilde{P}_2 \cap \tilde{P}_3 \neq \emptyset$.

The collider keeps 3 separate lists (arrays) L_w for each axis $w \in \{x, y, z\}$

$$L_w = \bigcup_i \left\{ \tilde{P}_i^{w0}, \tilde{P}_i^{w1} \right\} \tag{2.4}$$

where i traverses all particles. L_w arrays (sorted sets) contain respective coordinates of minimum and maximum corners for each **Aabb** (we call these coordinates *bound* in the following); besides bound, each of list elements further carries **id** referring to particle it belongs to, and a flag whether it is lower or upper bound.

In the initial step, all lists are sorted (using quicksort, average $\mathcal{O}(n \log n)$) and one axis is used to create initial interactions: the range between lower and upper bound for each body is traversed, while bounds in-between indicate potential **Aabb** overlaps which must be checked on the remaining axes as well.

At each successive step, lists are already pre-sorted. Inversions occur where a particle's coordinate has just crossed another particle's coordinate; this number is limited by numerical stability of simulation and its physical meaning (giving spatio-temporal coherence to the algorithm). The insertion sort algorithm swaps neighboring elements if they are inverted, and has complexity between $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$, for pre-sorted and unsorted lists respectively. For our purposes, we need only to handle inversions, which by nature of the sort algorithm are detected inside the sort loop. An inversion might signify:

- New overlap along the current axis, if an upper bound inverts (swaps) with a lower bound (i.e. that the upper bound with a higher coordinate was out of order in coming before the lower bound with a lower coordinate). Overlap along the other 2 axes is checked and if there is overlap along all axes, a new potential interaction is created.
- End of overlap along the current axis, if lower bound inverts (swaps) with an upper bound. If there is only potential interaction between the two particles in question, it is deleted.
- Nothing if both bounds are upper or both lower.

2.1.3.1. Aperiodic insertion sort

Let us show the sort algorithm on a sample sequence of numbers:

$$\| \quad 3 \qquad 7 \qquad 2 \qquad 4 \quad \| \tag{2.5}$$

Elements are traversed from left to right; each of them keeps inverting (swapping) with neighbors to the left, moving left itself, until any of the following conditions is satisfied:

($\leq$) the sorting order with the left neighbor is correct, or

($\|$) the element is at the beginning of the sequence.

We start at the leftmost element (the current element is marked $\boxed{i}$)

$$\| \quad \boxed{3} \quad 7 \quad 2 \quad 4 \quad \|. \tag{2.6}$$

It obviously immediately satisfies ($\|$), and we move to the next element:

$$\| \quad 3 \overset{\leq}{\leftarrow} \boxed{7} \quad 2 \quad 4 \quad \|. \tag{2.7}$$

Condition ($\leq$) holds, therefore we move to the right. The $\boxed{2}$ is not in order (violating ($\leq$)) and two inversions take place; after that, ($\|$) holds:

$$\begin{aligned} &\| \quad 3 \quad 7 \overset{\not\leq}{\leftarrow} \boxed{2} \quad 4 \quad \|, \\ &\| \quad 3 \overset{\not\leq}{\leftarrow} \boxed{2} \quad 7 \quad 4 \quad \|, \\ &\| \quad \boxed{2} \quad 3 \quad 7 \quad 4 \quad \|. \end{aligned} \tag{2.8}$$

The last element $\boxed{4}$ first violates ($\leq$), but satisfies it after one inversion

$$\begin{aligned} &\| \quad 2 \quad 3 \quad 7 \overset{\not\leq}{\leftarrow} \boxed{4} \quad \|, \\ &\| \quad 2 \quad 3 \overset{\leq}{\leftarrow} \boxed{4} \quad 7 \quad \|. \end{aligned} \tag{2.9}$$

All elements having been traversed, the sequence is now sorted.

It is obvious that if the initial sequence were sorted, elements only would have to be traversed without any inversion to handle (that happens in $\mathcal{O}(n)$ time).

For each inversion during the sort in simulation, the function that investigates change in **Aabb** overlap is invoked, creating or deleting interactions.

The periodic variant of the sort algorithm is described in sect. 2.6.1.3, along with other periodic-boundary related topics.

2.1.3.2. Optimization with Verlet distances

As noted above, Verlet [65] explored the possibility of running the collision detection only sparsely by enlarging predicates $\tilde{P}_i$.

In Yade, this is achieved by enlarging **Aabb** of particles by fixed relative length in all dimensions ΔL (**InsertionSortCollider.sweepLength**). Suppose the collider run last time at step m and the current step is n. **NewtonIntegrator** tracks maximum distance traversed by particles (via maximum velocity magnitudes $v^{\circ}_{max} = \max |\dot{u}^{\circ}_i|$ in each step, with the initial cummulative distance $L_{max} = 0$,

$$L^{\circ}_{max} = L^{-}_{max} + v^{\circ}_{max} \Delta t^{\circ} \tag{2.10}$$

triggering the collider re-run as soon as

$$L^{\circ}_{max} > \Delta L. \tag{2.11}$$

The disadvantage of this approach is that even one fast particle determines v°_{max}.

A solution is to track maxima per particle groups. The possibility of tracking each particle separately (that is what ESyS-Particle [14] does) seemed to us too fine-grained. Instead, we assign particles to b_n (**InsertionSortCollider.nBins**) *velocity bins* based on their current velocity magnitude. The bins' limit values are geometrical with the coefficient $b_c > 1$ (**InsertionSortCollider.binCoeff**), the maximum velocity being the current global velocity maximum v°_{max} (with some constraints on its change rate, to avoid large oscillations); for bin $i \in \{0, \dots, b_n\}$ and particle j:

$$v^\circ_{max} b_c^{-(i+1)} \leq |\dot{u}^\circ_j| < v_{max} b_c^{-i}. \tag{2.12}$$

(note that in this case, superscripts of b_c mean exponentiation). Equations (2.10)–(2.11) are used for each bin separately; however, when (2.11) is satisfied, full collider re-run is necessary and all bins' distances are reset.

Particles in high-speed oscillatory motion could be put into a slow bin if they happen to be at the point where their instantaneous speed is low, causing the necessity of early collider re-run. This is avoided by allowing particles to only go slower by one bin rather than several at once.

Results of using Verlet distance depend highly on the nature of simulation and choice of parameters **InsertionSortCollider.nBins** and **InsertionSortColldier.binCoeff**. The binning algorithm was specifically designed for simulating local fracture of larger concrete specimen; in that way, only particles in the fracturing zone, with greater velocities, had the **Aabb**'s enlarged, without affecting quasi-still particles outside of this zone. In such cases, up to 50% overall computation time savings were observed, collider being run every ≈100 steps in average.

2.2. Creating interaction between particles

Collision detection described above is only approximate. Exact collision detection depends on the geometry of individual particles and is handled separately. In Yade terminology, the **Collider** creates only *potential* interactions; potential interactions are evaluated exactly using specialized algorithms for collision of two spheres or other combinations. Exact collision detection must be run at every timestep since it is at every step that particles can change their mutual position (the collider is only run sometimes if the Verlet distance optimization is in use). Some exact collision detection algorithms are described in sect. 2.3; in Yade, they are implemented in classes deriving from **InteractionGeometryFunctor** (prefixed with **Ig2**).

Besides detection of geometrical overlap (which corresponds to **InteractionGeometry** in Yade), there are also non-geometrical properties of the interaction to be determined (**InteractionPhysics**). In Yade, they are computed for every new interaction by calling a functor deriving from **InteractionPhysicsFunctor** (prefixed with **Ip2**) which accepts the given combination of **Material** types of both particles.

2.2.1. Stiffnesses

Basic DEM interaction defines two stiffnesses: normal stiffness K_N and shear (tangent) stiffness K_T. It is desirable that K_N be related to fictitious Young's modulus of the particles' material, while K_T is typically determined as a given fraction of computed K_N. The K_T/K_N ratio determines macroscopic Poisson's ratio of the arrangement, which can be shown by dimensional analysis: elastic continuum has two parameters (E and ν) and basic DEM model also has 2 parameters with the same dimensions K_N and K_T/K_N; macroscopic Poisson's ratio is therefore determined solely by K_T/K_N and macroscopic Young's modulus is then proportional to K_N and affected by K_T/K_N.

Naturally, such analysis is highly simplifying and does not account for particle radius distribution, packing configuration and other possible parameters such as the interaction radius introduced later.

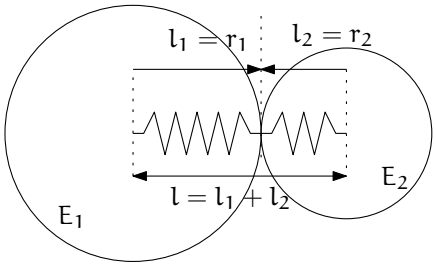


Figure 2.2.: Series of 2 springs representing normal stiffness of contact between 2 spheres.

2.2.1.1. Normal stiffness

The algorithm commonly used in Yade computes normal interaction stiffness as stiffness of two springs in serial configuration with lengths equal to the sphere radii (fig. 2.2.1.1).

Let us define distance $l = l_1 + l_2$, where l_i are distances between contact point and sphere centers, which are initially (roughly speaking) equal to sphere radii. Change of distance between the spehre centers Δl is distributed onto deformations of both spheres $\Delta l = \Delta l_1 + \Delta l_2$ proportionally to their compliances. Displacement change Δl_i generates force $F_i = K_i \Delta l_i$, where K_i assures proportionality and has physical meaning and dimension of stiffness; K_i is related to the sphere material modulus E_i and some length $\tilde{l}_i$ proportional to r_i.

$$\Delta l = \Delta l_1 + \Delta l_2 \tag{2.13}$$

$$K_i = E_i \tilde{l}_i \tag{2.14}$$

$$K_N \Delta l = F = F_1 = F_2 \tag{2.15}$$

$$K_N \left(\Delta l_1 + \Delta l_2\right) = F \tag{2.16}$$

$$K_N \left(\frac{F}{K_1} + \frac{F}{K_2}\right) = F \tag{2.17}$$

$$K_1^{-1} + K_2^{-1} = K_N^{-1} \tag{2.18}$$

$$K_N = \frac{K_1 K_2}{K_1 + K_2} \tag{2.19}$$

$$K_N = \frac{E_1 \tilde{l}_1 E_2 \tilde{l}_2}{E_1 \tilde{l}_1 + E_2 \tilde{l}_2} \tag{2.20}$$

The most used class computing interaction properties **Ip2_FrictMat_FrictMat_FrictPhys** uses $\tilde{l}_i = 2r_i$.

Some formulations define an equivalent cross-section A_{eq}, which in that case appears in the $\tilde{l}_i$ term as $K_i = E_i \tilde{l}_i = E_i \frac{A_{eq}}{l_i}$. Such is the case for the concrete model (**Ip2_CpmMat_CpmMat_CpmPhys**) described later, where $A_{eq} = \min(r_1, r_2)$.

For reasons given above, no pretense about equality of particle-level E_i and macroscopic modulus E should be made. Some formulations, such as [19], introduce parameters to match them numerically. This is not appropriate, in our opinion, since it binds those values to particular features of the sphere arrangement that was used for calibration.

2.2.2. Other parameters

Non-elastic parameters differ for various material models. Usually, though, they are averaged from the particles' material properties, if it makes sense. For instance, **Ip2_CpmMat_CpmMat_CpmPhys** averages most quantities, while **Ip2_FrictMat_FrictMat_FrictPhys** computes internal friction angle as $\varphi = \min(\varphi_1, \varphi_2)$ to avoid friction with bodies that are frictionless.

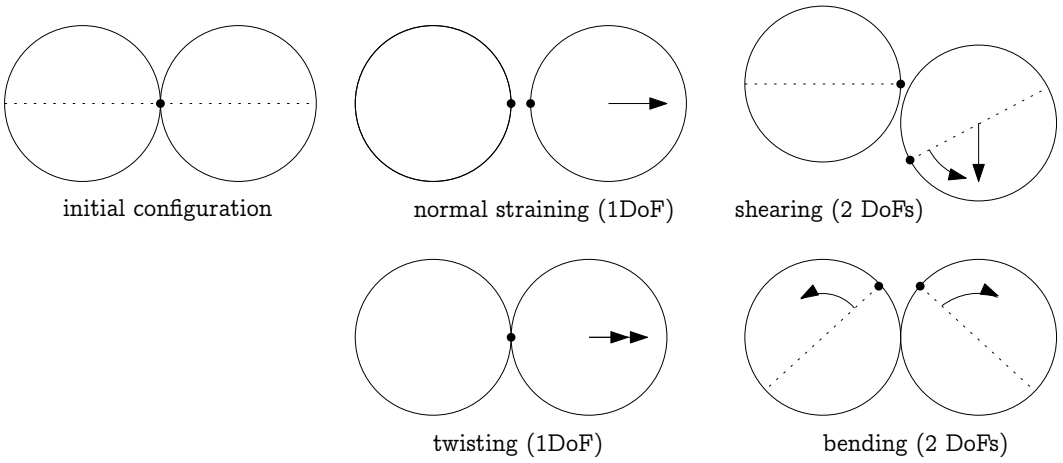


Figure 2.3.: Degrees of freedom of configuration of two spheres. Normal strain appears if there is a difference of linear velocity along the interaction axis (n); shearing originates from the difference of linear velocities perpendicular to n *and* from the part of $\omega_1 + \omega_2$ perpendicular to n; twisting is caused by the part of $\omega_1 - \omega_2$ parallel with n; bending comes from the part of $\omega_1 - \omega_2$ perpendicular to n.

2.3. Strain evaluation

In the general case, mutual configuration of two particles has 6 degrees of freedom (DoFs) just like a beam in 3D space: both particles have 6 DoFs each, but the interaction itself is free to move and rotate in space (with both spheres) having 6 DoFs itself; then $12 - 6 = 6$. They are shown at 2.3.

We will only describe normal and shear components of strain in the following, leaving torsion and bending aside. The reason is that most constitutive laws for contacts do not use the latter two.

2.3.1. Normal strain

2.3.1.1. Constants

Let us consider two spheres with *initial* centers $\bar{\mathbf{C}}_1$, $\bar{\mathbf{C}}_2$ and radii r_1, r_2 that enter into contact. The order of spheres within the contact is arbitrary and has no influence on the behavior. Then we define lengths

$$d_0 = |\bar{\mathbf{C}}_2 - \bar{\mathbf{C}}_1| \tag{2.21}$$

$$d_1 = r_1 + \frac{d_0 - r_1 - r_2}{2}, \qquad d_2 = d_0 - d_1. \tag{2.22}$$

These quantities are *constant* throughout the life of the interaction and are computed only once when the interaction is established. The distance d_0 is the *reference distance* and is used for the conversion of absolute displacements to dimensionless strain, for instance. It is also the distance where (for usual contact laws) there is neither repulsive nor attractive force between the spheres, whence the name *equilibrium distance*.

Distances d_1 and d_2 define reduced (or expanded) radii of spheres; geometrical radii r_1 and r_2 are used only for collision detection and may not be the same as d_1 and d_2, as shown in fig. 2.4. This difference is exploited in cases where the average number of contacts between spheres should be increased, e.g. to influence the response in compression or to stabilize the packing. In such case, interactions will be created also for spheres that do not geometrically overlap based on the *interaction radius* R_I, a dimensionless parameter determining „non-locality“ of contact detection. For $R_I = 1$, only spheres that touch are considered in contact; the general condition reads

$$d_0 \leq R_I(r_1 + r_2). \tag{2.23}$$

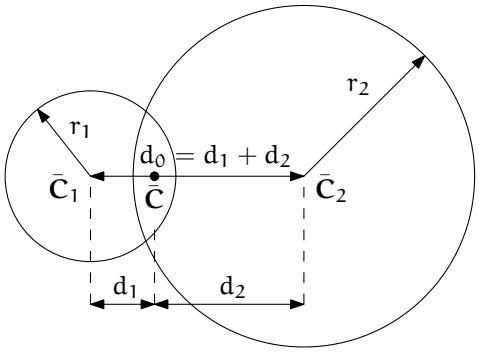


Figure 2.4.: Geometry of the initial contact of 2 spheres; this case pictures spheres which already overlap when the contact is created (which can be the case at the beginning of a simulation) for the sake of generality. The initial contact point $\bar{\mathbf{C}}$ is in the middle of the overlap zone.

The value of R_I directly influences the average number of interactions per sphere (percolation), which for some models is necessary in order to achieve realistic results. In such cases, **Aabb** (or $\tilde{P}_i$ predicates in general) must be enlarged accordingly (**Bo1_Sphere_Aabb.aabbEnlargeFactor**).

Contact cross-section. Some constitutive laws are formulated with strains and stresses (**Law2_Dem3DofGeom_CpmPhys_Cpm**, the concrete model described later, for instance); in that case, equivalent cross-section of the contact must be introduced for the sake of dimensionality. The exact definition is rather arbitrary; the CPM model (**Ip2_CpmMat_CpmMat_CpmPhys**) uses the relation

$$A_{eq} = \pi \min(r_1, r_2)^2 \tag{2.24}$$

which will be used to convert stresses to forces, if the constitutive law used is formulated in terms of stresses and strains. Note that other values than π can be used; it will merely scale macroscopic packing stiffness; it is only for the intuitive notion of a truss-like element between the particle centers that we choose A_{eq} representing the circle area. Besides that, another function than $\min(r_1, r_2)$ can be used, although the result should depend linearly on r_1 and r_2 so that the equation gives consistent results if the particle dimensions are scaled.

2.3.1.2. Variables

The following state variables are updated as spheres undergo motion during the simulation (as $\mathbf{C}_1^\circ$ and $\mathbf{C}_2^\circ$ change):

$$\mathbf{n}^\circ = \frac{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ}{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|} \equiv \widehat{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ} \tag{2.25}$$

$$\mathbf{C}^\circ = \mathbf{C}_1^\circ + \left(d_1 - \frac{d_0 - |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{2} \right) \mathbf{n}. \tag{2.26}$$

The contact point $\mathbf{C}^\circ$ is always in the middle of the spheres' overlap zone (even if the overlap is negative, when it is in the middle of the empty space between the spheres). The *contact plane* is always perpendicular to the contact plane normal $\mathbf{n}^\circ$ and passes through $\mathbf{C}^\circ$.

Normal displacement and strain can be defined as

$$u_N = |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ| - d_0, \tag{2.27}$$

$$\varepsilon_N = \frac{u_N}{d_0} = \frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0} - 1. \tag{2.28}$$

Since u_N is always aligned with $\mathbf{n}$, it can be stored as a scalar value multiplied by $\mathbf{n}$ if necessary.

For massively compressive simulations, it might be beneficial to use the logarithmic strain, such that the strain tends to $-\infty$ (rather than -1) as centers of both spheres approach. Otherwise, repulsive force

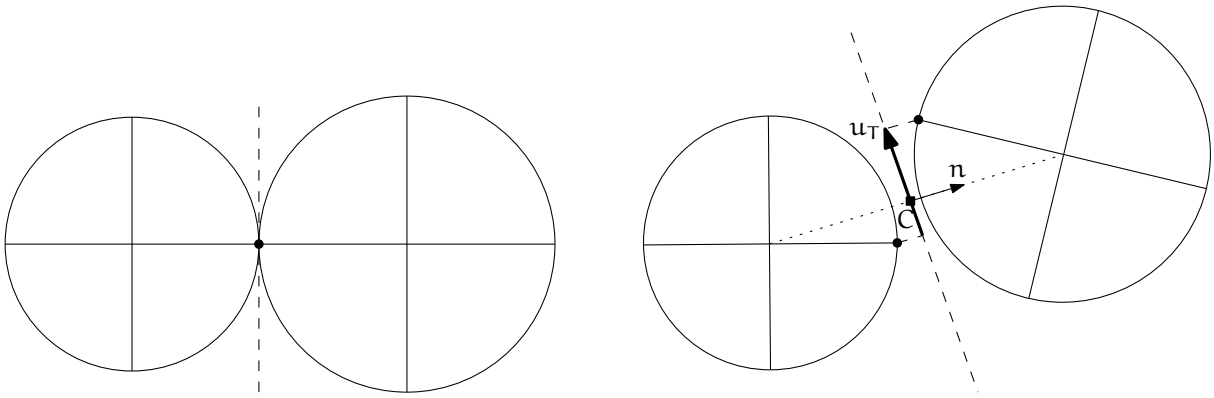


Figure 2.5.: Evolution of shear displacement $\mathbf{u}_T$ due to mutual motion of spheres, both linear and rotational. Left configuration is the initial contact, right configuration is after displacement and rotation of one particle.

would remain finite and the spheres could penetrate through each other. Therefore, we can adjust the definition of normal strain as follows:

$$\varepsilon_N = \begin{cases} \log\left(\frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0}\right) & \text{if } |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ| < d_0 \\ \frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0} - 1 & \text{otherwise.} \end{cases} \tag{2.29}$$

Such definition, however, has the disadvantage of effectively increasing rigidity (up to infinity) of contacts, requiring Δt to be adjusted, lest the simulation becomes unstable. Such dynamic adjustment is possible using a stiffness-based time-stepper (**GlobalStiffnessTimeStepper** in Yade).

2.3.2. Shear strain

A special (mis)feature of DEM is that each contact is oriented only by two points, current positions of both particles ($\mathbf{C}_1^\circ$ and $\mathbf{C}_2^\circ$). These only define contact normal $\mathbf{n}^\circ$ and the plane perpendicular to it, but no contact-local coordinate system. As a consequence, shear deformation $\mathbf{u}_T^\circ$ must always be expressed in global coordinates while satisfying the condition $\mathbf{u}_T^\circ \perp \mathbf{n}^\circ$.[2] In order to keep $\mathbf{u}_T$ consistent (e.g. that $\mathbf{u}_T$ must be constant if two spheres retain mutually constant configuration but move arbitrarily in space), then either $\mathbf{u}_T$ must track spheres' spatial motion or must (somehow) rely on sphere-local data exclusively.

These two possibilities lead to two algorithms of computing shear strains. They should give the same results (disregarding numerical imprecision), but there is a trade-off between computational cost of the incremental method and robustness of the total one.

Geometrical meaning of shear strain is shown in fig 2.5.

2.3.2.1. Incremental algorithm

The incremental algorithm is widely used in DEM codes and is described frequently.[3] Yade implements this algorithm in the **ScGeom** class. At each step, shear displacement $\mathbf{u}_T$ is updated; the update increment can be decomposed in 2 parts: motion of the interaction (i.e. $\mathbf{C}$ and $\mathbf{n}$) in global space and mutual motion of spheres.

1. Contact moves dues to changes of the spheres' positions $\mathbf{C}_1$ and $\mathbf{C}_2$, which updates current $\mathbf{C}^\circ$ and $\mathbf{n}^\circ$ as per (2.26) and (2.25). $\mathbf{u}_T^-$ is perpendicular to the contact plane at the previous step $\mathbf{n}^-$ and must be updated so that $\mathbf{u}_T^- + (\Delta\mathbf{u}_T) = \mathbf{u}_T^\circ \perp \mathbf{n}^\circ$; this is done by perpendicular projection to

[2] Let us note at this point that due to the absence of contact-local coordinates, plasticity conditions can only be formulated using σ_N and $|\boldsymbol{\sigma}_T|$, but not $\boldsymbol{\sigma}_T$.

[3] [36, 2].

the plane first (which might decrease $|\mathbf{u}_T|$) and adding what corresponds to spatial rotation of the interaction instead:

$$(\Delta\mathbf{u}_T)_1 = -\mathbf{u}_T^- \times (\mathbf{n}^- \times \mathbf{n}^\circ) \tag{2.30}$$

$$(\Delta\mathbf{u}_T)_2 = -\mathbf{u}_T^- \times \left(\frac{\Delta t}{2}\mathbf{n}^\circ \cdot (\boldsymbol{\omega}_1^{\ominus} + \boldsymbol{\omega}_2^{\ominus})\right)\mathbf{n}^\circ \tag{2.31}$$

2. Mutual movement of spheres, using only its part perpendicular to $\mathbf{n}^\circ$; $\boldsymbol{v}_{12}$ denotes mutual velocity of spheres at the contact point:

$$\boldsymbol{v}_{12} = \left(\boldsymbol{v}_2^{\ominus} + \boldsymbol{\omega}_2^- \times (-d_2\mathbf{n}^\circ)\right) - \left(\boldsymbol{v}_1^{\ominus} + \boldsymbol{\omega}_1^{\ominus} \times (d_1\mathbf{n}^\circ)\right) \tag{2.32}$$

$$\boldsymbol{v}_{12}^\perp = \boldsymbol{v}_{12} - (\mathbf{n}^\circ \cdot \boldsymbol{v}_{12})\mathbf{n}^\circ \tag{2.33}$$

$$(\Delta\mathbf{u}_T)_3 = -\Delta t\boldsymbol{v}_{12}^\perp \tag{2.34}$$

Finally, we compute

$$\mathbf{u}_T^\circ = \mathbf{u}_T^- + (\Delta\mathbf{u}_T)_1 + (\Delta\mathbf{u}_T)_2 + (\Delta\mathbf{u}_T)_3. \tag{2.35}$$

2.3.2.2. Total alogithm

The following algorithm, aiming at stabilization of response even with large rotation speeds or Δt approaching stability limit, was designed by the author of this thesis.[4] It is based on tracking original contact points (with zero shear) in the particle-local frame.

In this section, variable symbols implicitly denote their current values unless explicitly stated otherwise.

Shear strain may have two sources: mutual rotation of spheres or transversal displacement of one sphere with respect to the other. Shear strain does not change if both spheres move or rotate but are not in linear or angular motion mutually. To accurately and reliably model this situation, for every new contact the initial contact point $\bar{\mathbf{C}}$ is mapped into local sphere coordinates $(\mathbf{p}_{01}, \mathbf{p}_{02})$. As we want to determine the distance between both points (i.e. how long the trajectory in on both spheres' surfaces together), the shortest path from current $\mathbf{C}$ to the initial locally mapped point on the sphere's surface is „unrolled" to the contact plane $(\mathbf{p}'_{01}, \mathbf{p}'_{02})$; then we can measure their linear distance $\mathbf{u}_T$ and define shear strain $\boldsymbol{\varepsilon}_T = \mathbf{u}_T/d_0$ (fig. 2.6).

More formally, taking $\bar{\mathbf{C}}_i$, $\bar{q}_i$ for the sphere initial positions and orientations (as quaterions, see Appendix B) in global coordinates, the initial sphere-local contact point *orientation* (relative to sphere-local axis $\hat{\mathbf{x}}$) is remembered:

$$\bar{\mathbf{n}} = \widehat{\bar{\mathbf{C}}_1 - \bar{\mathbf{C}}_2}, \tag{2.36}$$

$$\bar{q}_{01} = \mathrm{Align}(\hat{\mathbf{x}}, \bar{q}_1^* \bar{\mathbf{n}} \bar{q}_1^{**}), \tag{2.37}$$

$$\bar{q}_{02} = \mathrm{Align}(\hat{\mathbf{x}}, \bar{q}_2^* (-\bar{\mathbf{n}}) \bar{q}_2^{**}). \tag{2.38}$$

(See Appendix B for definition of Align.)

After some spheres motion, the original point can be "unrolled" to the current contact plane:

$$q = \mathrm{Align}(\mathbf{n}, q_1 \bar{q}_{01} \hat{\mathbf{x}} (q_1 \bar{q}_{01})^*) \quad \text{(auxiliary)} \tag{2.39}$$

$$\mathbf{p}'_{01} = q_\vartheta d_1 (q_\mathbf{u} \times \mathbf{n}) \tag{2.40}$$

where $q_\mathbf{u}$, q_ϑ are axis and angle components of q and p'_{01} is the unrolled point. Similarly,

$$q = \mathrm{Align}(\mathbf{n}, q_2 \bar{q}_{02} \hat{\mathbf{x}} (q_2 \bar{q}_{02})^*) \tag{2.41}$$

$$\mathbf{p}'_{02} = q_\vartheta d_1 (q_\mathbf{u} \times (-\mathbf{n})). \tag{2.42}$$

[4] A similar algorithm based on total formulation, which covers additionally bending and torsion, was proposed in Wang [68].

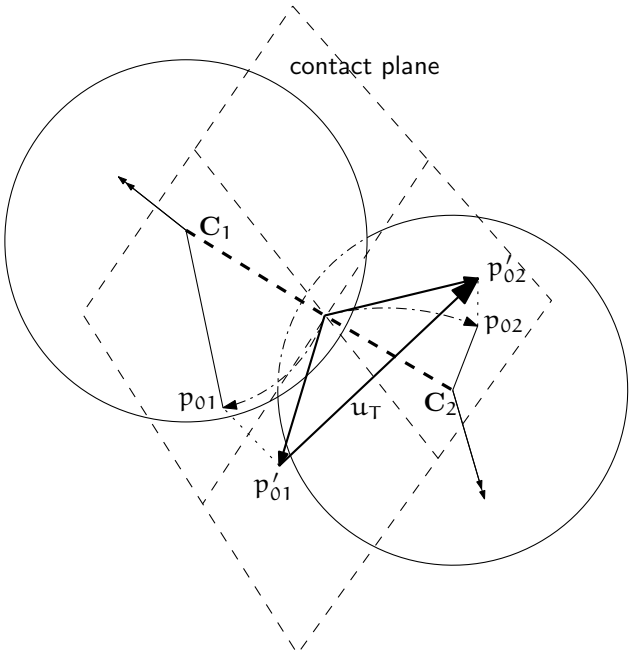


Figure 2.6.: Shear displacement computation for two spheres in relative motion.

Shear displacement and strain are then computed easily:

$$\mathbf{u}_T = \mathbf{p}'_{02} - \mathbf{p}'_{01} \tag{2.43}$$

$$\varepsilon_T = \frac{\mathbf{u}_T}{d_0} \tag{2.44}$$

When using material law with plasticity in shear, it may be necessary to limit maximum shear strain, in which case the mapped points are moved closer together to the requested distance (without changing $\hat{\mathbf{u}}_T$). This allows us to remember the previous strain direction and also avoids summation of increments of plastic strain at every step (fig. 2.7).

This algorithm is straightforwardly modified to facet-sphere interactions. In Yade, it is implemented by **Dem3DofGeom** and related classes.

2.4. Stress evaluation (example)

Once strain on a contact is computed, it can be used to compute stresses/forces acting on both spheres.

The constitutive law presented here is the most usual DEM formulation, originally proposed by Cundall. While the strain evaluation will be similar to algorithms described in the previous section regardless of stress evaluation, stress evaluation itself depends on the nature of the material being modeled. The constitutive law presented here is the most simple non-cohesive elastic case with dry friction, which Yade implements in **Law2_Dem3DofGeom_FrictPhys_Basic** (all constitutive laws derive from base class **LawFunctor**).

In DEM generally, some constitutive laws are expressed using strains and stresses while others prefer displacement/force formulation. The law described here falls in the latter category.

When new contact is established (discussed in 5.2.1.2) it has its properties (**InteractionPhysics**) computed from **Material**s associated with both particles. In the simple case of frictional material **FrictMat**, **Ip2_FrictMat_FrictMat_FrictPhys** creates a new **FrictPhys** instance, which defines normal stiffness K_N, shear stiffness K_T and friction angle φ.

At each step, given normal and shear displacements u_N, $\mathbf{u}_T$, normal and shear forces are computed (if

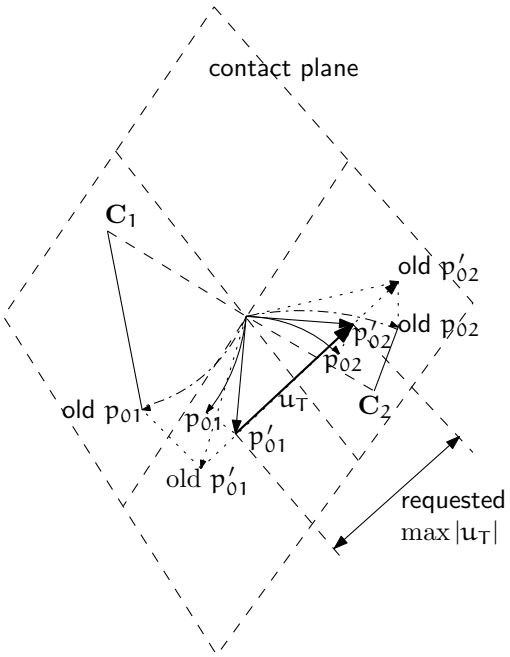


Figure 2.7.: Shear plastic slip for two spheres.

$u_N > 0$, the contact is deleted without generating any forces):

$$\mathbf{F}_N = K_N u_N \mathbf{n}, \tag{2.45}$$

$$\mathbf{F}_T^t = K_T \mathbf{u}_T \tag{2.46}$$

where $\mathbf{F}_N$ is normal force and $\mathbf{F}_T$ is trial shear force. A simple non-associated stress return algorithm is applied to compute final shear force

$$\mathbf{F}_T = \begin{cases} \mathbf{F}_T^t \frac{|\mathbf{F}_N| \tan\varphi}{\mathbf{F}_T^t} & \text{if } |\mathbf{F}_T| > |\mathbf{F}_N| \tan\varphi, \\ \mathbf{F}_T^t & \text{otherwise.} \end{cases} \tag{2.47}$$

Summary force $\mathbf{F} = \mathbf{F}_N + \mathbf{F}_T$ is then applied to both particles – each particle accumulates forces and torques acting on it in the course of each step. Because the force computed acts at contact point $\mathbf{C}$, which is difference from sphes' centers, torque generated by $\mathbf{F}$ must also be considered.

$$\mathbf{F}_1 += \mathbf{F} \qquad \mathbf{F}_2 += -\mathbf{F} \tag{2.48}$$

$$\mathbf{T}_1 += d_1(-\mathbf{n}) \times \mathbf{F} \qquad \mathbf{T}_2 += d_2 \mathbf{n} \times \mathbf{F}. \tag{2.49}$$

2.5. Motion integration

Each particle accumulates generalized forces (forces and torques) from the contacts in which it participates. These generalized forces are then used to integrate motion equations for each particle separately; therefore, we omit i indices denoting the i-th particle in this section.

The customary leapfrog scheme (also known as the Verlet scheme) is used, with some adjustments for rotation of non-spherical particles, as explained below. The "leapfrog" name comes from the fact that even derivatives of position/orientation are known at on-step points, whereas odd derivatives are known at mid-step points. Let us recall that we use a^-, a°, a^+ for on-step values of a at $t - \Delta t$, t and $t + \Delta t$ respectively; and $a^\ominus$, $a^\oplus$ for mid-step values of a at $t - \Delta t/2$, $t + \Delta t/2$.

Described integration algorithms are implemented in the **NewtonIntegrator** class in Yade.

2.5.1. Position

Integrating motion consists in using current acceleration $\ddot{\mathbf{u}}^\circ$ on a particle to update its position from the current value $\mathbf{u}^\circ$ to its value at the next timestep $\mathbf{u}^+$. Computation of acceleration, knowing current forces $\mathbf{F}$ acting on the particle in question and its mass m, is simply

$$\ddot{\mathbf{u}}^\circ = \mathbf{F}/m. \tag{2.50}$$

Using the 2^{nd} order finite difference with step Δt, we obtain

$$\ddot{\mathbf{u}}^\circ \cong \frac{\mathbf{u}^- - 2\mathbf{u}^\circ + \mathbf{u}^+}{\Delta t^2} \tag{2.51}$$

from which we express

$$\begin{aligned}\mathbf{u}^+ &= 2\mathbf{u}^\circ - \mathbf{u}^- + \ddot{\mathbf{u}}^\circ \Delta t^2 = \\ &= \mathbf{u}^\circ + \Delta t \underbrace{\left(\frac{\mathbf{u}^\circ - \mathbf{u}^-}{\Delta t} + \ddot{\mathbf{u}}^\circ \Delta t \right)}_{(\dagger)}.\end{aligned} \tag{2.52}$$

Typically, $\mathbf{u}^-$ is already not known (only $\mathbf{u}^\circ$ is); we notice, however, that

$$\dot{\mathbf{u}}^\ominus \simeq \frac{\mathbf{u}^\circ - \mathbf{u}^-}{\Delta t}, \tag{2.53}$$

i.e. the mean velocity during the previous step, which is known. Plugging this approximate into the $(\dagger)$ term, we also notice that mean velocity during the current step can be approximated as

$$\dot{\mathbf{u}}^\oplus \simeq \dot{\mathbf{u}}^\ominus + \ddot{\mathbf{u}}^\circ \Delta t, \tag{2.54}$$

which is $(\dagger)$; we arrive finally at

$$\mathbf{u}^+ = \mathbf{u}^\circ + \Delta t \left(\dot{\mathbf{u}}^\ominus + \ddot{\mathbf{u}}^\circ \Delta t \right). \tag{2.55}$$

The algorithm can then be written down by first computing current mean velocity $\dot{\mathbf{u}}^\oplus$ which we need to store for the next step (just as we use its old value $\dot{\mathbf{u}}^\ominus$ now), then computing the position for the next time step $\mathbf{u}^+$:

$$\dot{\mathbf{u}}^\oplus = \dot{\mathbf{u}}^\ominus + \ddot{\mathbf{u}}^\circ \Delta t, \tag{2.56}$$

$$\mathbf{u}^+ = \mathbf{u}^\circ + \dot{\mathbf{u}}^\oplus \Delta t. \tag{2.57}$$

Positions are known at times $i\Delta t$ (if Δt is constant) while velocities are known at $i\Delta t + \frac{\Delta t}{2}$. The facet that they interleave (jump over each other) in such way gave rise to the colloquial name "leapfrog" scheme.

2.5.2. Orientation (spherical)

Updating particle orientation q° proceeds in an analogous way to position update. First, we compute current angular acceleration $\dot{\boldsymbol{\omega}}^\circ$ from known current torque $\mathbf{T}$. For spherical particles where the inertia tensor is diagonal in any orientation (therefore also in current global orientation), satisfying $\mathbf{I}_{11} = \mathbf{I}_{22} = \mathbf{I}_{33}$, we can write

$$\dot{\boldsymbol{\omega}}_i^\circ = \mathbf{T}_i / \mathbf{I}_{11}, \tag{2.58}$$

We use the same approximation scheme, obtaining an equation analogous to (2.56)

$$\boldsymbol{\omega}^\oplus = \boldsymbol{\omega}^\ominus + \Delta t \dot{\boldsymbol{\omega}}^\circ. \tag{2.59}$$

The quaternion (see Appendix B) Δq representing rotation vector $\boldsymbol{\omega}^{\oplus}\Delta t$ is constructed, i.e. such that

$$(\Delta q)_{\vartheta} = |\boldsymbol{\omega}^{\oplus}|, \tag{2.60}$$

$$(\Delta q)_{\mathbf{u}} = \widehat{\boldsymbol{\omega}^{\oplus}} \tag{2.61}$$

Finally, we compute the next orientation q^{+} by rotation composition

$$q^{+} = \Delta q q^{\circ}. \tag{2.62}$$

2.5.3. Orientation (aspherical)

Integrating rotation of aspherical particles is considerably more complicated than their position, as their local reference frame is not inertial. Rotation of rigid body in the local frame, where inertia matrix I is diagonal, is described in the continuous form by Euler's equations ($i \in \{1, 2, 3\}$ and i, j, k are subsequent indices):

$$\mathbf{T}_i = I_{ii}\dot{\boldsymbol{\omega}}_i + (I_{kk} - I_{jj})\boldsymbol{\omega}_j\boldsymbol{\omega}_k. \tag{2.63}$$

Due to the presence of the current values of both $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$, they cannot be solved using the standard leapfrog algorithm (that was the case for translational motion and also for the spherical bodies' rotation where this equation reduced to $\mathbf{T} = I\dot{\boldsymbol{\omega}}$).

The algorithm presented here is described by Allen and Tildesley [1, pg. 84–89] and was designed by Fincham for molecular dynamics problems; it is based on extending the leapfrog algorithm by mid-step/on-step estimators of quantities known at on-step/mid-step points in the basic formulation. Although it has received criticism and more precise algorithms are known (Omelyan [44], Neto and Bellucci [42], Johnson et al. [25]), this one is currently implemented in Yade for its relative simplicity.

Each body has its local coordinate system based on the principal axes of inertia for that body. We use $\widetilde{\bullet}$ to denote vectors in local coordinates. The orientation of the local system is given by the current particle's orientation q° as a quaternion; this quaternion can be expressed as the (current) rotation matrix A. Therefore, every vector $\mathbf{a}$ is transformed as $\widetilde{\mathbf{a}} = q\mathbf{a}q^{*} = A\mathbf{a}$. Since A is a rotation (orthogonal) matrix, the inverse rotation $A^{-1} = A^{T}$.

For given particle in question, we know

$\widetilde{I}^{\circ}$ (constant) inertia matrix; diagonal, since in local, principal coordinates,

$\mathbf{T}^{\circ}$ external torque,

q° current orientation (and its equivalent rotation matrix A),

$\boldsymbol{\omega}^{\ominus}$ mid-step angular velocity,

$\mathbf{L}^{\ominus}$ mid-step angular momentum; this is an auxiliary variable that must be tracked in addition for use in this algorithm. It will be zero in the initial step.

Our goal is to compute new values of the latter three, that is $\mathbf{L}^{\oplus}$, q^{+}, $\boldsymbol{\omega}^{\oplus}$. We first estimate current angular momentum and compute current local angular velocity:

$$\mathbf{L}^{\circ} = \mathbf{L}^{\ominus} + \mathbf{T}^{\circ}\frac{\Delta t}{2}, \qquad \widetilde{\mathbf{L}}^{\circ} = A\mathbf{L}^{\circ}, \tag{2.64}$$

$$\mathbf{L}^{\oplus} = \mathbf{L}^{\ominus} + \mathbf{T}^{\circ}\Delta t, \qquad \widetilde{\mathbf{L}}^{\oplus} = A\mathbf{L}^{\oplus}, \tag{2.65}$$

$$\widetilde{\boldsymbol{\omega}}^{\circ} = \widetilde{I}^{\circ -1}\widetilde{\mathbf{L}}^{\circ}, \tag{2.66}$$

$$\widetilde{\boldsymbol{\omega}}^{\oplus} = \widetilde{I}^{\circ -1}\widetilde{\mathbf{L}}^{\oplus}. \tag{2.67}$$

$$\tag{2.68}$$

Then we compute $\dot{q}^\circ$, using q° and $\widetilde{\omega}^\circ$:

$$\begin{pmatrix} \dot{q}^\circ_w \\ \dot{q}^\circ_x \\ \dot{q}^\circ_y \\ \dot{q}^\circ_z \end{pmatrix} = \frac{1}{2} \begin{pmatrix} q^\circ_w & -q^\circ_x & -q^\circ_y & -q^\circ_z \\ q^\circ_x & q^\circ_w & -q^\circ_z & q^\circ_y \\ q^\circ_y & q^\circ_z & q^\circ_w & -q^\circ_x \\ q^\circ_z & -q^\circ_y & q^\circ_x & q^\circ_w \end{pmatrix} \begin{pmatrix} 0 \\ \widetilde{\omega}^\circ_x \\ \widetilde{\omega}^\circ_y \\ \widetilde{\omega}^\circ_z \end{pmatrix}, \tag{2.69}$$

$$q^\oplus = q^\circ + \dot{q}^\circ \frac{\Delta t}{2}. \tag{2.70}$$

We evaluate $\dot{q}^\oplus$ from $q^\oplus$ and $\widetilde{\omega}^\oplus$ in the same way as in (2.69) but shifted by $\Delta t/2$ ahead. Then we can finally compute the desired values

$$q^+ = q^\circ + \dot{q}^\oplus \Delta t, \tag{2.71}$$

$$\omega^\oplus = A^{-1} \widetilde{\omega}^\oplus \tag{2.72}$$

2.5.4. Clumps (rigid aggregates)

DEM simulations frequently make use of rigid aggregates of particles to model complex shapes [49] called *clumps*, typically composed of many spheres. Dynamic properties of clumps are computed from the properties of its members: the clump's mass m_c is summed over members, the inertia tensor I_c with respect to the clump's centroid is computed using the parallel axes theorem; local axes are oriented such that they are principal and inertia tensor is diagonal and clump's orientation is changed to compensate rotation of the local system, as to not change the clump members' positions in global space. Initial positions and orientations of all clump members in local coordinate system are stored.

In Yade (class **Clump**), clump members behave as stand-alone particles during simulation for purposes of collision detection and contact resolution, except that they have no contacts created among themselves within one clump. It is at the stage of motion integration that they are treated specially. Instead of integrating each of them separately, forces/torques on those particles F_i, T_i are converted to forces/torques on the clump itself. Let us denote r_i relative position of each particle with regards to clump's centroid, in global orientation. Then summary force and torque on the clump are

$$F_c = \sum F_i / m_c, \tag{2.73}$$

$$T_c = \sum r_i \times F_i + T_i. \tag{2.74}$$

Motion of the clump is then integrated, using aspherical rotation integration. Afterwards, clump members are displaced in global space, to keep their initial positions and orientations in the clump's local coordinate system. In such a way, relative positions of clump members are always the same, resulting in the behavior of a rigid aggregate.

2.5.5. Numerical damping

In simulations of quasi-static phenomena, it it desirable to dissipate kinetic energy of particles. Since most constitutive laws (including **Law_ScGeom_FrictPhys_Basic** shown above, sect. 2.4) do not include velocity-based damping (such as one in D'Addetta et al. [10]), it is possible to use artificial numerical damping. The formulation is described in ICG [23], although our version is slightly adapted. The basic idea is to decrease forces which increase the particle velocities and vice versa by $(\Delta F)_d$, comparing the current acceleration sense and particle velocity sense. This is done by component, which makes the damping scheme clearly non-physical, as it is not invariant with respect to coordinate system rotation; on the other hand, it is very easy to compute. Cundall proposed the form (we omit particle indices i since it applies to all of them separately):

$$\frac{(\Delta F)_{dw}}{F_w} = -\lambda_d \operatorname{sgn}(F_w \dot{u}^\ominus_w), \quad w \in \{x, y, z\} \tag{2.75}$$

where λ_d is the damping coefficient. This formulation has several advantages [19]:

- it acts on forces (accelerations), not constraining uniform motion;
- it is independent of eigenfrequencies of particles, they will be all damped equally;
- it needs only the dimensionless parameter λ_d which does not have to be scaled.

In Yade, we use the adapted form

$$\frac{(\Delta \mathbf{F})_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn} \mathbf{F}_w \underbrace{\left(\dot{u}_w^{\ominus} + \frac{\ddot{u}_w^{\circ} \Delta t}{2} \right)}_{\simeq \dot{u}_w^{\circ}}, \tag{2.76}$$

where we replaced the previous mid-step velocity $\dot{u}^{\ominus}$ by its on-step estimate in parentheses. This is to avoid locked-in forces that appear if the velocity changes its sign due to force application at each step, i.e. when the particle in question oscillates around the position of equilibrium with $2\Delta t$ period.

In Yade, damping (2.76) is implemented in the **NewtonIntegrator** engine; the damping coefficient λ_d is **NewtonIntegrator.damping**.

2.5.6. Stability considerations

The leapfrog integration scheme is conditionally stable, i.e. not magnifying errors, provided $\Delta t < \Delta t_{cr}$ where Δt_{cr} is the *critical timestep*, above which the integration is unstable. Usually, Δt is taken as a fraction of Δt_{cr}; this fraction is called the *timestep safety factor*, with meaningful values $\in \langle 0, 1)$.

2.5.6.1. Critical timestep (translational)

In order to ensure stability for the explicit integration sceheme, an upper limit is imposed on Δt:

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{2.77}$$

where ω_{max} is the highest angular eigenfrequency within the system.

Single mass-spring system with mass m and stiffness K is governed by the equation

$$m\ddot{x} = -Kx, \tag{2.78}$$

where x is displacement from the mean (equilibrium) position. The solution of harmonic oscillation is $x(t) = A\cos(\omega t + \varphi)$ where phase φ and amplitude A are determined by initial conditions. The angular frequency

$$\omega^{(1)} = \sqrt{\frac{m}{K}} \tag{2.79}$$

does not depend on initial conditions. Since there is one single mass, $\omega_{max}^{(1)} = \omega^{(1)}$. Plugging (2.79) into (2.77), we obtain

$$\Delta t_{cr}^{(1)} = 2/\omega_{max}^{(1)} = 2\sqrt{K/m} \tag{2.80}$$

for a single oscillator.

In general mass-spring system, the highest frequency occurs if two connected masses m_i, m_j are in opposite motion; let us suppose they have equal velocities (which is conservative) and they are connected by a spring with stiffness K_i: displacement Δx_i of m_i will be accompained by $\Delta x_j = -\Delta x_i$ of m_j, giving $\Delta F_i = -K_i(\Delta x_i - (-\Delta x_i)) = -2K_i\Delta x_i$. That results in apparent stiffness $K_i^{(2)} = 2K_i$, giving maximum angular frequency of the whole system

$$\omega_{max} = \max_i \sqrt{K_i^{(2)}/m_i}. \tag{2.81}$$

The overall critical timestep is then

$$\Delta t_{cr} = \frac{2}{\omega_{max}} = \min_i 2\sqrt{\frac{m_i}{K_i^{(2)}}} = \min_i 2\sqrt{\frac{m_i}{2K_i}} = \min_i \sqrt{2}\sqrt{\frac{m_i}{K_i}}. \tag{2.82}$$

This equation can be used in R^3 simulations by evaluating the critical timestep per-axis and requiring

$$\Delta t_{cr} = \min \Delta t_{cr\,w}, \quad w \in \{x, y, z\}. \tag{2.83}$$

In DEM simulations, per-particle stiffness $\mathbf{K}_i$ is determined from the stiffnesses of contacts in which it participates [7]. Suppose each contact has normal stiffness K_{Nj}, shear stiffness $K_{Tj} = \xi K_{Nj}$ and is oriented by normal $\mathbf{n}_j$. $\mathbf{K}_i$ is then sum of contributions of all contacts in which it participates (indices j), with terms along axes equal to

$$\mathbf{K}_{iw} = \sum_j (K_{Nj} - K_{Tj})\mathbf{n}_{jw}^2 + K_{Tj} = \sum_j K_{Nj}\left((1-\xi)\mathbf{n}_{jw}^2 + \xi\right) \tag{2.84}$$

with $w \in \{x, y, z\}$. Equations (2.82), (2.83) and (2.84) determine Δt_{cr} in a simulation; it is implemented by the **GlobalStiffnessTimeStepper** engine in Yade.

This approach is simplifying, since only translational terms of the stiffness matrix are considered (which is the limiting factor typically) and eigenvalues are estimated from diagonal terms only. Rigorous derivation of critical timestep is possible from overall stiffness and mass matrices; we have to leave it for posterity at this place, for time constraints.

There is one important condition that $\omega_{max} > 0$: if there are no contacts between particles and $\omega_{max} = 0$, we would obtain value $\Delta t_{cr} = \infty$. While formally correct, this value is numerically erroneous: we were silently supposing that stiffness remains constant during each timestep, which is not true if contacts are created as particles collide. In case of no contact, therefore, stiffness must be pre-estimated based on future interactions, as shown in the next section.

2.5.6.2. Estimation of Δt_{cr} by wave propagation speed

Estimating timestep in absence of interactions is based on the connection between interaction stiffnesses and the particle's properties. Note that in this section, symbols E and ρ refer exceptionally to Young's modulus and density of *particles*, not of macroscopic arrangement.

In Yade, particles have associated **Material** which defines density ρ (**Material.density**), and also may define (in **ElastMat** and derived classes) particle's "Young's modulus" E (**ElastMat.young**). ρ is used when particle's mass m is initially computed from its ρ, while E is taken in account when creating new interaction between particles, affecting stiffness K_N. Knowing m and K_N, we can estimate (2.84) for each particle; we obviously neglect

- number of interactions per particle N_i; for "reasonable" radius distribution, however, there is a geometrically imposed upper limit (6 for a packing of spheres with equal radii, for instance);
- the exact relationship the between particles' rigidities E_i, E_j, supposing only that K_N is somehow proportional to them.

By defining E and ρ, particles have continuum-like quantities. Explicit integration schemes for continuum equations impose a critical timestep based on sonic speed $\sqrt{E/\rho}$; the elastic wave must not propagate farther than the minimum distance of integration points l_{min} during one step. Since E, ρ are parameters of the elastic continuum and l_{min} is fixed beforehand, we obtain

$$\Delta t_{cr}^{(c)} = l_{min}\sqrt{\frac{\rho}{E}}. \tag{2.85}$$

For our purposes, we define E and ρ for each particle separately; l_{min} can be replaced by the sphere's radius R_i; technically, $l_{min} = 2R_i$ could be used, but because of possible interactions of spheres and facets

(which have zero thickness), we consider $l_{min} = R_i$ instead. Then

$$\Delta t_{cr}^{(p)} = \min_i R_i \sqrt{\frac{\rho_i}{E_i}}. \tag{2.86}$$

This algorithm is implemented in the **utils.PWaveTimeStep()** function.

Let us compare this result to (2.82); this necessitates making several simplifying hypotheses:

- all particles are spherical and have the same radius R;
- the sphere's material has the same E and ρ
- the average number of contacts per sphere is N;
- the contacts have sufficiently uniform spatial distribution around each particle;
- the $\xi = K_N/K_T$ ratio is constant for all interactions;
- contact stiffness K_N is computed from E using a formula of the form

$$K_N = E\pi' R', \tag{2.87}$$

  where π' is some constant depending on the algorithm in use[5] and R' is half-distance between spheres in contact, equal to R for the case of interaction radius $R_I = 1$. If $R_I = 1$ (and $R' \equiv R$ by consequence), all interactions will have the same stiffness K_N. In other cases, we will consider K_N as the average stiffness computed from average R' (see below).

As all particles have the same parameters, we drop the i index in the following formulas.

We try to express the average per-particle stiffness from (2.84). It is a sum over all interactions where K_N and ξ are scalars that will not rotate with interaction, while $\mathbf{n}_w$ is w-th component of unit interaction normal $\mathbf{n}$. Since we supposed uniform spatial distribution, we can replace $\mathbf{n}_w^2$ by its average value $\overline{\mathbf{n}}_w^2$. Recognizing components of $\mathbf{n}$ as direction cosines, the average values of $\mathbf{n}_w^2$ is $1/3$.

Moreover, since all directions are equal, we can write the per-body stiffness as $K = \mathbf{K}_w$ for all $w \in \{x, y, z\}$. We obtain

$$K = \sum K_N \left((1-\xi)\frac{1}{3} + \xi \right) = \sum K_N \frac{1-2\xi}{3} \tag{2.88}$$

and can put constant terms (everything) in front of the summation. $\sum 1$ equals the number of contacts per sphere, i.e. N. Arriving at

$$K = N K_N \frac{1-2\xi}{3}, \tag{2.89}$$

we substitute K into (2.82) using (2.87):

$$\Delta t_{cr} = \sqrt{2}\sqrt{\frac{m}{K}} = \sqrt{2}\sqrt{\frac{\frac{4}{3}\pi R^3 \rho}{NE\pi' R \frac{1-2\xi}{3}}} = \underbrace{R\sqrt{\frac{\rho}{E}}}_{\Delta t_{cr}^{(p)}} 2\sqrt{\frac{\pi/\pi'}{N(1-2\xi)}}. \tag{2.90}$$

The ratio of timestep $\Delta t_{cr}^{(p)}$ predicted by the p-wave velocity and numerically stable timestep Δt_{cr} is the inverse value of the last (dimensionless) term:

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{N(1+\xi)}{\pi/\pi'}}. \tag{2.91}$$

Actual values of this ratio depend on characteristics of packing N, $K_N/K_T = \xi$ ratio and the way of computing contact stiffness from particle rigidity. Let us show it for two models in Yade:

[5] For example, $\pi' = \pi/2$ in the concrete particle model (**Ip2_CpmMat_CpmMat_CpmPhys**), while $\pi' = 2$ in the classical DEM model (**Ip2_FrictMat_FrictMat_FrictPhys**) as implemented in Yade.

Concrete particle model computes contact stiffness from the equivalent area A_{eq} first (2.24),

$$A_{eq} = \pi R^2 \qquad K_N = \frac{A_{eq} E}{d_0}. \tag{2.92}$$

d_0 is the initial contact length, which will be, for interaction radius (2.23) $R_I > 1$, in average larger than $2R$. For $R_I = 1.5$ (sect. 3.3.2), we can roughly estimate $\overline{d}_0 = 1.25 \cdot 2R = \frac{5}{2}R$, getting

$$K_N = E\left(\frac{2}{5}\pi\right)R \tag{2.93}$$

where $\frac{2}{5}\pi = \pi'$ by comparison with (2.87).

Interaction radius $R_I = 1.5$ leads to average $N \approx 12$ interactions per sphere for dense packing of spheres with the same radius R. $\xi = 0.2$ is calibrated (sect. 3.3.2) to match the desired macroscopic Poisson's ratio $\nu = 0.2$.

Finally, we obtain the ratio

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{12(1-2\cdot 0.2)}{\frac{\pi}{(2/5)\pi}}} = 3.39, \tag{2.94}$$

showing significant overestimation by the p-wave algorithm.

Non-cohesive dry friction model is the basic model proposed by Cundall explained in 2.4. Supposing almost-constant sphere radius R and rather dense packing, each sphere will have $N = 6$ interactions on average (that corresponds to maximally dense packing of spheres with a constant radius). If we use the **Ip2_FrictMat_FrictMat_FrictPhys** class, we have $\pi' = 2$, as $K_N = E2R$; we again use $\xi = 0.2$ (for lack of a more significant value). In this case, we obtain the result

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{6(1-2\cdot 0.2)}{\pi/2}} = 3.02 \tag{2.95}$$

which again overestimates the numerical critical timestep.

To conclude, p-wave timestep gives estimate proportional to the real Δt_{cr}, but in the cases shown, the value of about $\Delta t = 0.3\Delta t_{cr}^{(p)}$ should be used to guarantee stable simulation.

2.5.6.3. Non-elastic Δt constraints

Let us note at this place that not only Δt_{cr} assuring numerical stability of motion integration is a constraint. In systems where particles move at relatively high velocities, position change during one timestep can lead to non-elastic irreversible effects such as damage. The Δt needed for reasonable result can be lower Δt_{cr}. We have no rigorously derived rules for such cases.

2.6. Periodic boundary conditions

While most DEM simulations happen in R^3 space, it is frequently useful to avoid boundary effects by using periodic space instead. In order to satisfy periodicity conditions, periodic space is created by repetition of parallelepiped-shaped cell. In Yade, periodic space is implemented in the **Cell** class. The cell is determined by

- the reference size $\mathbf{s}$ (**Cell.refSize**), giving reference cell configuration (which is always perpendicular): axis-aligned cuboid with corners $(0,0,0)$ and $\mathbf{s}$;
- the transformation matrix T (**Cell.trsf**).

The transformation matrix T can hold arbitrary linear transformation composed of scaling, rotation and shear. Volume change of the cell is given by $\det T$. The cell can be manipulated by directly changing its transformation matrix T and its reference size $\mathbf{s}$.

Additionally, we define teransformation gradient $\nabla\boldsymbol{v}$ (**Cell.velGrad**) which can be automatically integrated at every step using the Euler scheme

$$\mathrm{T}^{+} = \mathrm{T}^{\circ} + \nabla\boldsymbol{v}\Delta t. \tag{2.96}$$

Along with the automatic integration of cell transformation, there is an option to homothetically displace all particles so that $\nabla\boldsymbol{v}$ is swept linearly over the whole simulation (enabled via **NewtonIntegrator.homotheticCellResize**). This avoids all boundary effects coming from change of the transformation.

2.6.1. Collision detection in periodic cell

In usual implementations, particle positions are forced to be inside the cell by wrapping their positions if they get over the boundary (so that they appear on the other side). As we wanted to avoid abrupt changes of position (it would make particle's velocity inconsistent with step displacement change), a different method was chosen.

2.6.1.1. Approximate collision detection

Pass 1 collision detection (based on sweep and prune algorithm, sect. 2.1.3) operates on axis-aligned bounding boxes (**Aabb**) of particles. During the collision detection phase, bounds of all **Aabb**s are wrapped inside the cell in the first step. At subsequent runs, every bound remembers by how many cells it was initially shifted from coordinate given by the **Aabb** and uses this offset repeatedly as it is being updated from **Aabb** during particle's motion. Bounds are sorted using the periodic insertion sort algorithm (sect. 2.6.1.3), which tracks periodic cell boundary $\|$.

Upon inversion of two **Aabb**'s, their collision along all three axes is checked, wrapping real coordinates inside the cell for that purpose.

This algorithm detects collisions as if all particles were inside the cell but without the need of constructing "ghost particles" (to represent periodic image of a particle which enters the cell from the other side) or changing the particle's positions.

It is required by the implementation (and partly by the algorithm itself) that particles do not span more than half of the current cell size along any axis; the reason is that otherwise two (or more) contacts between both particles could appear, on each side. Since Yade identifies contacts by **Body.id** of both bodies, they would not be distinguishable.

In presence of shear, the sweep-and-prune collider could not sort bounds independently along three axes: collision along x axis depends on the mutual position of particles on the y axis. Therefore, bounding boxes *are expressed in transformed coordinates* which are perpendicular in the sense of collision detection. This requires some extra computation: **Aabb** of sphere in transformed coordinates will no longer be cube, but cuboid, as the sphere itself will appear as ellipsoid after transformation. Inversely, the sphere in simulation space will have a parallelepiped bounding "box", which is cuboid around the ellipsoid in transformed axes (the **Aabb** has axes aligned with transformed cell basis). This is shown at fig. 2.9.

The restriction of a single particle not spanning more than half of the transformed axis becomes stringent as **Aabb** is enlarged due to shear. Considering **Aabb** of a sphere with radius r in the cell where $x' \equiv x$, $z' \equiv z$, but $\angle(y, y') = \varphi$, the x-span of the **Aabb** will be multiplied by $1/\cos\varphi$. For the infinite shear $\varphi \to \pi/2$, which can be desirable to simulate, we have $1/\cos\varphi \to \infty$. Fortunately, this limitation can be easily circumvented by realizing the quasi-identity of all periodic cells which, if repeated in space, create the same grid with their corners: the periodic cell can be flipped, keeping all particle interactions intact, as shown in fig 2.8. It only necessitates adjusting the **Interaction.cellDist** of interactions and re-initialization of the collider (**Collider::invalidatePersistentData**). Cell flipping is implemented in the **utils.flipCell()** function.

This algorithm is implemented in **InsertionSortCollider** and is used whenever simulation is periodic (**Omega.isPeriodic**); individual **BoundFunctor**s are responsible for computing sheared **Aabb**s; currently it is implemented for spheres and facets (in **Bo1_Sphere_Aabb** and **Bo1_Facet_Aabb** respectively).

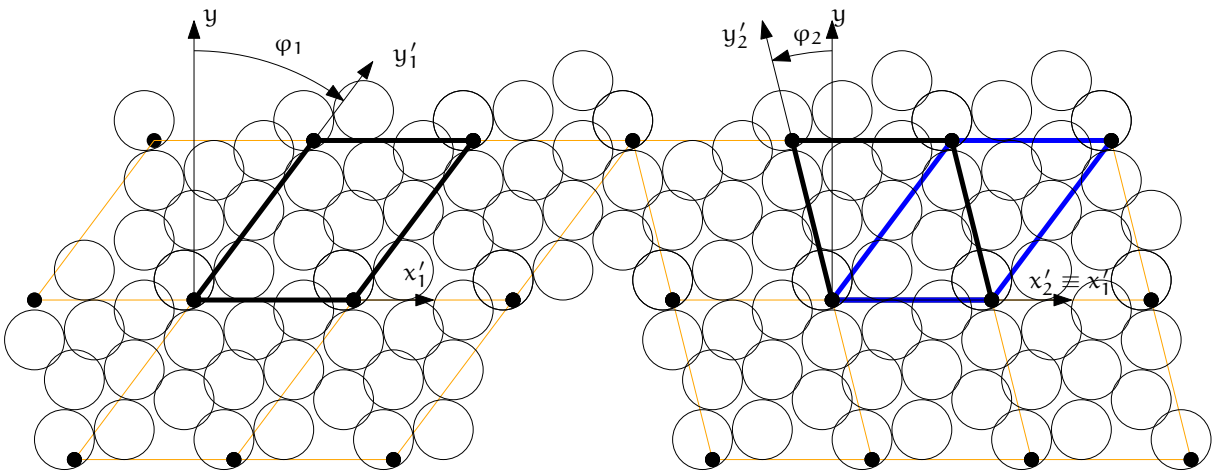


Figure 2.8.: Flipping cell (**utils.flipCell()**) to avoid infinite stretch of the bounding boxes' spans with growing φ. Cell flip does not affect interactions from the point of view of the simulation. The periodic arrangement on the left is the same as the one on the right, only the cell is situated differently between identical grid points of repetition; at the same time $|\varphi_2| < |\varphi_1|$ and sphere bounding box's x-span stretched by $1/\cos\varphi$ becomes smaller. Flipping can be repeated, making effective infinite shear possible.

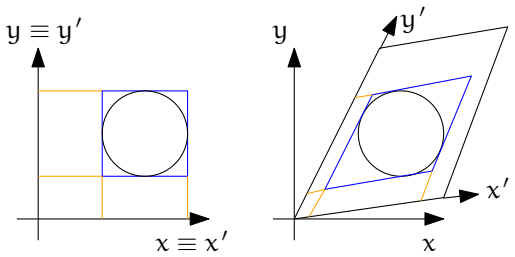


Figure 2.9.: Constructing axis-aligned bounding box (**Aabb**) of a sphere in simulation space coordinates (without periodic cell – left) and transformed cell coordinates (right), where collision detection axes x', y' are not identical with simulation space axes x, y. Bounds' projection to axes is shown by orange lines.

2.6.1.2. Exact collision detection

When the collider detects approximate contact (on the **Aabb** level) and the contact does not yet exist, it creates *potential* contact, which is subsequently checked by exact collision algorithms (depending on combination of **Shape**s). Since particles can interact over many periodic cells (recall we never change their positions in simulation space), the collider embeds the relative cell coordinate of particles in the interaction itself (**Interaction.cellDist**) as *integer* vector $\mathbf{c}$. Multiplying current cell size $\mathbf{Ts}$ by $\mathbf{c}$ component-wise, we obtain particle offset $\Delta\boldsymbol{x}$ in aperiodic $\mathbb{R}^3$; this value is passed (from **InteractionDispatchers**) to the functor computing exact collision (**InteractionGeometryFunctor**), which adds it to the position of the particle **Interaction.id2**.

By storing the integral offset $\mathbf{c}$, $\Delta\boldsymbol{x}$ automatically updates as cell parameters change.

2.6.1.3. Periodic insertion sort algorithm

The extension of sweep and prune algorithm (described in sect. 2.1.3) to periodic boundary conditions is non-trivial. Its cornerstone is a periodic variant of the insertion sort algorithm, which involves keeping track of the "period" of each boundary; e.g. taking period $\langle 0, 10)$, then $8_1 \equiv -2_2 < 2_2$ (subscript indicating period). Doing so efficiently (without shuffling data in memory around as bound wraps from one period to another) requires moving period boundary rather than bounds themselves and making the comparison work transparently at the edge of the container.

This algorithm was also extended to handle non-orthogonal periodic **Cell** boundaries by working in transformed rather than Cartesian coordinates; this modifies computation of **Aabb** from Cartesian coordinates in which bodies are positioned (treated in detail in sect. 2.6.1.1).

The sort algorithm is tracking **Aabb** extrema along all axes. At the collider's initialization, each value is assigned an integral period, i.e. its distance from the cell's interior expressed in the cell's dimension along its respective axis, and is wrapped to a value inside the cell. We put the period number in subscript.

Let us give an example of coordinate sequence along x axis:[6]

$$4_1 \qquad 12_2 \quad \| \quad -1_2 \qquad -2_4 \qquad 5_0 \tag{2.97}$$

with cell x-size $s_x = 10$. The 4_1 value then means that the real coordinate x_i of this extremum is $x_i + 1 \cdot 10 = 4$, i.e. $x_i = -4$. The $\|$ symbol denotes the periodic cell boundary.

Sorting starts from the first element in the cell, i.e. right of $\|$, and inverts elements as in the aperiodic variant. The rules are, however, more complicated due to the presence of the boundary $\|$:

- ($\leq$) stop inverting if neighbors are ordered;
- ($\|\bullet$) current element left of $\|$ is below 0 (lower period boundary); in this case, decrement element's period, decrease its coordinate by s_x and move $\|$ right;
- ($\bullet\|$) current element right of $\|$ is above s_x (upper period boundary); increment element's period, increase its coordinate by s_x and move $\|$ left;
- ($\nleftrightarrow$) inversion across $\|$ must subtract s_x from the left coordinate during comparison. If the elements are not in order, they are swapped, but they must have their periods changed as they traverse $\|$. Apply ($\|\circ$) if necessary;
- ($\|\circ$) if after ($\nleftrightarrow$) the element that is now right of $\|$ has $x_i < s_x$, decrease its coordinate by s_x and decrement its period. Do not move $\|$.

[6] In real case, the number of elements would be even, as there is maximum and minimum value couple for each particle; this demonstration only shows the sorting algorithm, however.

In the first step, ($\|\bullet$) is applied, and inversion with 12_2 happens; then we stop because of ($\leq$):

$$\begin{array}{llllll} 4_1 & 12_2 & \| & \boxed{-1_2} & -2_4 & 5_0, \\ 4_1 & 12_2 & \overset{\not\leq}{\leftarrow} & \boxed{9_1} & \| \; -2_4 & 5_0, \\ 4_1 & \overset{\leq}{\leftarrow} \boxed{9_1} & & 12_2 & \| \; -2_4 & 5_0. \end{array} \tag{2.98}$$

We move to next element $\boxed{-2_4}$; first, we apply ($\|\bullet$), then invert until ($\leq$):

$$\begin{array}{llllll} 4_1 & 9_1 & 12_2 & \| & \boxed{-2_4} & 5_0, \\ 4_1 & 9_1 & 12_2 & \overset{\not\leq}{\leftarrow} & \boxed{8_3} \; \| & 5_0, \\ 4_1 & 9_1 & \overset{\not\leq}{\leftarrow} \boxed{8_3} & & 12_2 \; \| & 5_0, \\ 4_1 & \overset{\leq}{\leftarrow} \boxed{8_3} & 9_1 & & 12_2 \; \| & 5_0. \end{array} \tag{2.99}$$

The next element is $\boxed{5_0}$; we satisfy ($\nparallel$), therefore instead of comparing $12_2 > 5_0$, we must do $(12_2 - s_x) = 2_3 \leq 5$; we adjust periods when swapping over $\|$ and apply ($\|\circ$), turning 12_2 into 2_3; then we keep inverting, until ($\leq$):

$$\begin{array}{llllll} 4_1 & 8_3 & 9_1 & 12_2 \overset{\not\leq}{\leftarrow} \| & \boxed{5_0}, \\ 4_1 & 8_3 & 9_1 \overset{\not\leq}{\leftarrow} & \boxed{5_{-1}} \; \| & 2_3, \\ 4_1 & 8_3 \overset{\not\leq}{\leftarrow} & \boxed{5_{-1}} & 9_1 \; \| & 2_3, \\ 4_1 \overset{\leq}{\leftarrow} & \boxed{5_{-1}} & 8_3 & 9_1 \; \| & 2_3. \end{array} \tag{2.100}$$

We move (wrapping around) to $\boxed{4_1}$, which is ordered:

$$\boxed{4_1} \quad 5_{-1} \quad 8_3 \quad 9_1 \; \| \; 2_3 \qquad (\geq) \tag{2.101}$$

and so is the last element

$$4_1 \overset{\leq}{\leftarrow} \boxed{5_{-1}} \quad 8_3 \quad 9_1 \; \| \; 2_3. \tag{2.102}$$

2.7. Computational aspects

2.7.1. Cost

The DEM computation using an explicit integration scheme demands a relatively high number of steps during simulation, compared to implicit scehemes. The total computation time Z of simulation spanning T seconds (of simulated time), containing N particles in volume V depends on:

- linearly, the number of steps $i = T/(s_t \Delta t_{cr})$, where s_t is timestep safety factor; Δt_{cr} can be estimated by p-wave velocity using E and ρ (sect. 2.5.6.2) as $\Delta t_{cr}^{(p)} = r\sqrt{\frac{\rho}{E}}$. Therefore

$$i = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}. \tag{2.103}$$

- the number of particles N; for fixed value of simulated domain volume V and particle radius r

$$N = p\frac{V}{\frac{4}{3}\pi r^3}, \tag{2.104}$$

  where p is packing porosity, roughly $\frac{1}{2}$ for dense irregular packings of spheres of similar radius.

  The dependency is not strictly linear (which would be the best case), as some algorithms do not scale linearly; a case in point is the sweep and prune collision detection algorithm introduced in sect. 2.1.3, with scaling roughly $\mathcal{O}(N \log N)$.

  The number of interactions scales with N, as long as packing characteristics are the same.

- the number of computational cores n_{cpu}; in the ideal case, the dependency would be inverse-linear were all algorithms parallelized (in Yade, collision detection is not).

Let us suppose linear scaling. Additionally, let us suppose that the material to be simulated (E, ρ) and the simulation setup (V, T) are given in advance. Finally, dimensionless constants s_t, p and n_{cpu} will have a fixed value. This leaves us with one last degree of freedom, r. We may write

$$Z \propto iN\frac{1}{n_{cpu}} = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}p\frac{V}{\frac{4}{3}\pi r^3}\frac{1}{n_{cpu}} \propto \frac{1}{r}\frac{1}{r^3} = \frac{1}{r^4}. \tag{2.105}$$

This (rather trivial) result is essential to realize DEM scaling; if we want to have finer results, refining the "mesh" by halving r, the computation time will grow $2^4 = 16$ times.

For very crude estimates, one can use a known simulation to obtain a machine "constant"

$$\mu = \frac{Z}{Ni} \tag{2.106}$$

with the meaning of time per particle and per timestep (in the order of 10^{-6} s for current machines). μ will be only useful if simulation characteristics are similar and non-linearities in scaling do not have major influence, i.e. N should be in the same order of magnitude as in the reference case.

2.7.2. Result indeterminism

It is naturally expected that running the same simulation several times will give exactly the same results: although the computation is done with finite precision, round-off errors would be deterministically the same at every run. While this is true for *single-threaded* computation where exact order of all operations is given by the simulation itself, it is not true anymore in *multi-threaded* computation which is described in detail in later sections.

The straight-forward manner of parallel processing in explicit DEM is given by the possibility of treating interactions in arbitrary order. Strain and stress is evaluated for each interaction independently, but forces from interactions have to be summed up. If summation order is also arbitrary (in Yade, forces are accumulated for each thread in the order interactions are processed, then summed together), then the results can be slightly different. For instance

```
(1/10.)+(1/13.)+(1/17.)=0.23574660633484162
(1/17.)+(1/13.)+(1/10.)=0.23574660633484165
```

As forces generated by interactions are assigned to bodies in quasi-random order, summary force F_i on the body can be different between single-threaded and multi-threaded computations, but also between different runs of multi-threaded computation with exactly the same parameters. Exact thread scheduling by the kernel is not predictable since it depends on asynchronous events (hardware interrupts) and other unrelated tasks running on the system; and it is thread scheduling that ultimately determines summation order of force contributions from interactions.

2.7.2.1. Numerical damping influence

The effect of summation order can be significantly amplified by the usage of a *discontinuous* damping function in **NewtonIntegrator** given in (2.76) as

$$\frac{(\Delta \mathbf{F})_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn} \mathbf{F}_w \left(\dot{u}_w^{\ominus} + \frac{\ddot{u}_w^{\circ} \Delta t}{2} \right). \tag{2.107}$$

If the sgn argument is close to zero then the least significant finite precision artifact can determine whether the equation (relative increment of $\mathbf{F}_w$) is $+\lambda_d$ or $-\lambda_d$. Given commonly used values of $\lambda_d = 0.4$, it means that such artifact propagates from least significant place to the most significant one at once.

3. Concrete particle model

This chapter presents a new model of concrete formulated within the DEM framework presented in chapter 2.

Contact law is formulated on the level of individual contacts between 2 particles. However, since desired results are macroscopic and not observed until larger simulations are performed, motivations for introducing various contact-level features of the model might not be obvious until after reading sect. 3.3 dedicated to calibration.

3.1. Discrete concrete models overview

Before describing the CPM model, we would like to give a brief overview of some of the main particle-based, DEM and non-DEM, concrete models.

Lattice models. Vervuurt [66, pg. 14–16] gives an overview of older lattice models used for concrete. Later lattice models of concrete aim usually at meso-scale modeling, where aggregates and mortar are distinguished by differing beam parameters. Beam configuration can be regular (leading to failure anisotropy), random [35], or generated to match observed statistical distribution [33].

Lilliu and van Mier [35] uses brittle beam behavior, simply removing broken beams from simulation; Leite et al. [33] uses tensile softening to avoid dependence of global softening (which is observed even with brittle behavior of beams) on beam size. Neither of these models focuses on capturing compression behavior.

Cusatis et al. [9] presents a rather sophisticated model, in which properties of lattice beams are computed from the geometry of Voronoï cells of each node. Granulometry is supposed to have substantial influence on confinement behavior; as it is not fully considered by the model, the beam-transversal stress history influences shear rigidity instead. Compression behavior is captured fairly well.

Rigid body – spring models are, to our knowledge, used infrequently for concrete. Nagai et al. [41] presents a mesoscopic (distinguishing aggregates and mortar) model, though only formulated in 2D.

Discrete element models of concrete are rather rare. Most activity (judging by publications) comes from teams formed around Frédéric V. Donzé. His work (using SDEC software mentioned in sect. 4.1) first targeted at 2D DEM[5]. Later, exploiting the dynamic nature of DEM led to fast concrete dynamics in 3D [19, 20] and impact simulation [59]. In order to reduce computational costs, elastic FEM for the non-damaged subdomain is used, with some tricks to avoid spurious dynamic effects on the DEM/FEM boundary [54].

With both lattice and DEM models, arriving at reasonable compression behavior is non-trivial; it seems to stem from the fact that 1D elements (beams in lattice, bonds in DEM) have no notion of an overall stress state at their vicinity, but only insofar as it is manifested in the direction of the respective element. Cusatis et al. [9] works around this by explicitly introducing the influence of transversal stress. Hentz [19, sect. 5.3], on the other hand, blocks rotations of spherical elements to achieve a higher and more realistic f_c/f_t ratio, but it is questionable whether there is physically sound reason for such an adjustment.

3.2. Model description

3.2.1. Cohesive and non-cohesive contacts

We use the word interaction to denote both cohesive (material bond) and non-cohesive (contact two spheres meeting during their motion) interactions, as they are governed by the same equations. The non-cohesive contact only differs by that it is considered as fully damaged from the very start, by setting the damage variable $\omega = 1$; since damage effectively prevent tensile forces in the interaction, the result is that the two spheres interact only while in tension (geometrically overlapping) and disappears when they geometrically separate. On the other hand, cohesive contact, which is always created at the beginning of the simulation, is created in virgin, undamaged state.

3.2.2. Contact parameters

As already explained above, most parameters of an interaction are computed as averages of particle material properties. That is also true for the CPM model, with the respective **material** and **interaction physics** classes.

Those computed as averages (from values that are typically identical, since particles share the same material) are c_{T0}, ε_0, ε_s, k_N (from particles' moduli E), τ_d, M_d, τ_{pl}, M_{pl}, σ_0, φ.[1]

On the other hand, shear modulus k_T is computed from k_N using the ratio in **G_over_E**. ε_f is computed from ε_0 by multiplying it by dimensionless **relDuctility**, which is $\varepsilon_f/\varepsilon_0$. Several parameters are

Finally, some parameters are set in **law functor**, hence the same for all interactions. Those are Y_0 and $\tilde{K}_s$.

Density of the particle material is set to $\rho = 4800\,\mathrm{kgm}^{-3}$, as we have to compensate for porosity of the packing, which a little above 50% for spheres of the same radius and we take continuum concrete density roughly $2400\,\mathrm{kgm}^{-3}$.

3.2.3. Normal stresses

The normal stress-strain law is formulated within the framework of damage mechanics:

$$\sigma_N = [1 - \omega H(\varepsilon_N)] k_N \varepsilon_N. \tag{3.1}$$

Here, k_N is the normal modulus (model parameter, [Pa]), and $\omega \in \langle 0, 1\rangle$ is the damage variable. The Heaviside function $H(\varepsilon_N)$ deactivates damage influence in compression, which physically corresponds to crack closure.

The damage variable ω is evaluated using the *damage evolution function* g (fig. 3.1):

$$\omega = g(\kappa) = 1 - \frac{\varepsilon_f}{\kappa} \exp\left(-\frac{\kappa - \varepsilon_0}{\varepsilon_f}\right) \tag{3.2}$$

$$\kappa = \max \tilde{\varepsilon} \tag{3.3}$$

$$\tilde{\varepsilon} = \sqrt{\langle \varepsilon_N \rangle^2 + \xi_1^2 |\varepsilon_T|^2}, \tag{3.4}$$

where $\tilde{\varepsilon}$ is the equivalent strain responsible for damage ($\langle \varepsilon_N \rangle$ signifies the positive part of ε_N) and ξ_1 is a dimensionless coefficient weighting the contribution of the shear strain ε_T to damage. However, comparative studies indicate that the optimal value is $\xi_1 \equiv 0$, hence equation (3.4) simplifies to $\tilde{\varepsilon} = \langle \varepsilon_N \rangle$. κ is the maximum equivalent strain over the whole history of the contact.

Parameter ε_0 is the limit elastic strain, and the product $K_T \varepsilon_0$ corresponds to the local tensile strength at the level of one contact (in general different from the macroscopic tensile strength). Parameter ε_f is

[1] The meaning of symbols used here is explained in the following text.

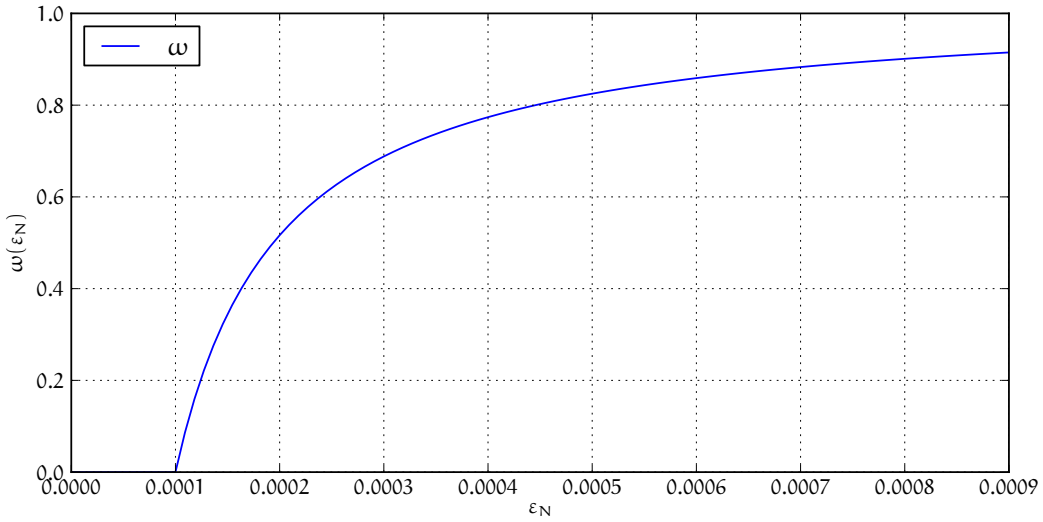


Figure 3.1.: Damage ω evolution function $\omega = g(\kappa_D)$, where $\kappa_D = \max \varepsilon_N$ (using $\varepsilon_0 = 0.0001$, $\varepsilon_f = 30\varepsilon_0$).

related to the slope of the softening part of the normal strain-stress diagram (fig. 3.2) and must be larger than ε_0.

3.2.3.1. Compressive plasticity

To better capture confinement effect, we introduce, as an extension to the model, hardening in plasticity in compression. Using material parameter $\varepsilon_s < 0$, we define $\sigma_s = k_N \varepsilon_s$. If $\sigma < \sigma_s$, then $\tilde{K}_s k_N$ is taken for tangent stiffness, $\tilde{K}_s \in \langle 0, 1 \rangle$ and plastic strain ε_N^{pl} is incremented accordingly. This introduces 2 new parameters ε_s (stress at which the hardening branch begins) and $\tilde{K}_s$ (relative modulus of the hardening branch) that must be calibrated (see 3.3) and one internal variable ε_N^{pl}.

Extended strain-stress diagram for multiple loading cycles is shown at fig. 3.3. Note that this feature is activated only in compression, whereas damage is activated only in tension. Therefore, there is no need to specially consider interaction between both.

3.2.3.2. Visco-damage

In order to model time-dependent phenomena, viscosity is introduced in tension by adding viscous overstress σ_{Nv} to (3.1). As we suppose it to be related to a limited rate of crack propagation, it cannot depend on total strain rate; rather, we split total strain into the elastic strain ε_e and the damage part ε_d. Since $\varepsilon_e = \sigma_N / k_N$, we have

$$\varepsilon_d = \varepsilon_N - \frac{\sigma_N}{k_N} \tag{3.5}$$

We then postulate the overstress in the form

$$\sigma_{Nv}(\dot{\varepsilon}_d) = k_N \varepsilon_0 \langle \tau_d \dot{\varepsilon}_{Nd} \rangle^{M_d}, \tag{3.6}$$

where $k_N \varepsilon_0$ is rate-independent tensile strength (introduced for the sake of dimensionality), τ_d is characteristic time for visco-damage and M_d is a dimensionless exponent. The $\langle \dots \rangle$ operator denotes positive part; therefore, for $\dot{\varepsilon}_{Nd} \leq 0$, viscous overstress vanishes. 2 new parameters τ_d and M_d were introduced.

The normal stress equation then reads

$$\sigma_N = [1 - \omega H(\varepsilon_N)]\, k_N \varepsilon_N + \sigma_{Nv}(\dot{\varepsilon}_{Nd}). \tag{3.7}$$

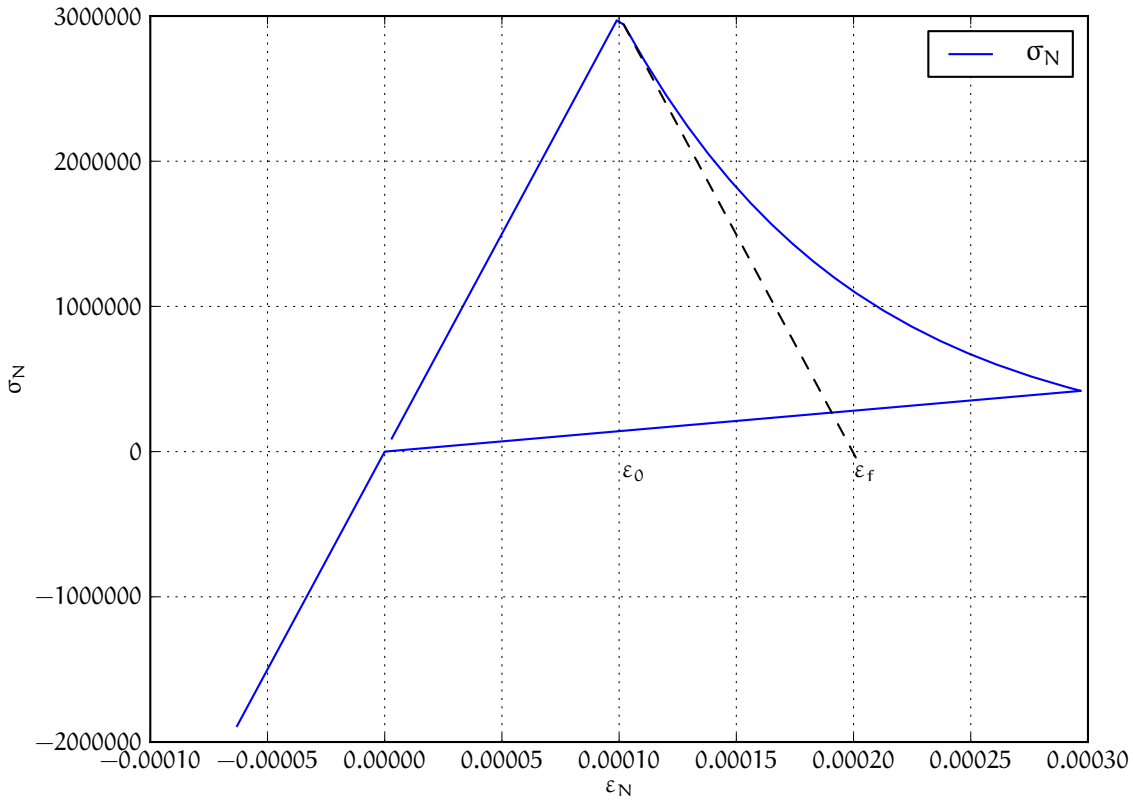


Figure 3.2.: Strain-stress diagram in the normal direction.

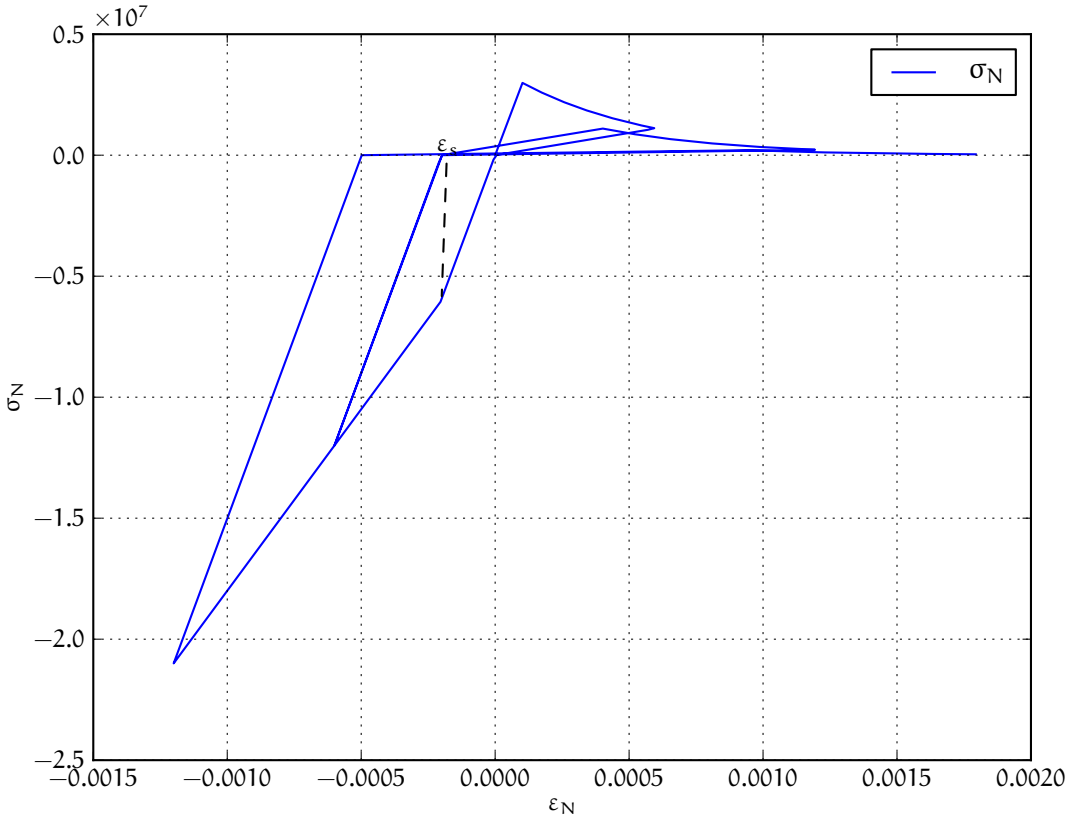


Figure 3.3.: Strain-stress diagram in normal direction, loaded cyclically in tension and compression; it shows several features of the model: (1) damage in tension, but not damage in compression (governed by the ω internal variable) (2) plasticity in compression, starting at strain ε_s; reduced (hardening) modulus is $\tilde{K}_s k_N$.

In a finite step computation, we can evaluate the damage variable ω in a rate-independent manner first; then, we compute the increment of ε_d satisfying equations (3.5–3.7). As usual, we use $\bullet^\circ$, $\bullet^+$ to denote value of $\bullet$ at t, $t+\Delta t$ (at the beginning and at the end of the current timestep) respectively. We approximate the damage strain rate by the difference formula

$$\dot{\varepsilon}_d = \frac{\varepsilon_d^+ - \varepsilon_d^\circ}{\Delta t} = \frac{\Delta\varepsilon_d}{\Delta t} \tag{3.8}$$

and substitute (3.6) and (3.7) into (3.5) obtaining

$$\varepsilon_d^\circ + \Delta\varepsilon_d = \varepsilon_N^+ - (1-\omega)\varepsilon_N^+ - \frac{k_N \varepsilon 0}{k_N}\left\langle \frac{\tau_d \Delta\varepsilon_d}{\Delta t}\right\rangle^{M_d}, \tag{3.9}$$

which can be written as

$$\Delta\varepsilon_d + \varepsilon_0 \left\langle \frac{\tau_d \Delta\varepsilon_d}{\Delta t}\right\rangle^{M_d} = \omega\varepsilon_N^+ - \varepsilon_d^\circ. \tag{3.10}$$

During unloading, i.e. $\Delta\varepsilon_d \le 0$, the power term vanishes, leading to

$$\Delta\varepsilon_d = \omega\varepsilon_N^+ - \varepsilon_d^\circ, \tag{3.11}$$

applicable if $\omega\varepsilon_N^+ \le \varepsilon_d^\circ$.

In the contrary case, (3.10) must be solved, but $\langle\cdots\rangle$ can be replaced by $(\cdots)$ since the term is now known to be positive. We divide (3.10) by its right-hand side and apply logarithm. We transform the unknown $\Delta\varepsilon_d$ into a new unknown

$$\beta = \ln \frac{\Delta\varepsilon_d}{\omega\varepsilon_N^+ - \varepsilon_d^\circ}, \tag{3.12}$$

obtaining equation

$$\ln\left(e^\beta + c e^{M_d \beta}\right) = 0, \tag{3.13}$$

with

$$c = (1-\omega)\varepsilon_0 \left(\omega\varepsilon_N^+ - \varepsilon_d^\circ\right)^{M_d - 1} \left(\frac{\tau_d}{\Delta t}\right)^{M_d}. \tag{3.14}$$

For positive c and M_d, the term $\ln(e^\beta + c e^{M_d\beta})$ from (3.13) is convex, increasing and positive at $\beta = 0$. As a consequence, the Newton method with starting point at $\beta = 0$[2] always converges monotonically to a unique negative solution of β; from it we compute

$$\Delta\varepsilon_d = \left(\omega\varepsilon_N^+ - \varepsilon_d^\circ\right) e^\beta. \tag{3.15}$$

Finally, the term $\sigma_{Nd}(\dot{\varepsilon}_d) = \sigma_{Nd}\left(\frac{\Delta\varepsilon_d}{\Delta t}\right)$ in 3.6 can be evaluated and plugged into 3.7.

The effect of viscosity on damage for one contact is shown on fig. 3.4; calibration of the new parameters τ_d and M_d is described in sect. 3.3.

3.2.3.3. Isotropic confinement

During calibration, we faced the necessity to simulate confined compression setups. Applying boundary confinement on a specimen composed of many particles is not straightforward, since thera are necessary strong local effects. We then found a way to introduce isotropic confinement at contact level, by pre-adjusting ε_N and post-adjusting σ_N; the supposition is that the confinement value σ_0 is negative and does not lead to immediate damage to contacts.

Given a prescribed confinement value σ_0, we adjust ε_N value *before* computing normal and shear stresses:

$$\varepsilon_N' = \varepsilon_N + \begin{cases} \sigma_0/k_N & \text{if } \sigma_0 > k_N\varepsilon_s, \\ \varepsilon_s + \frac{\sigma_0 - k_N\varepsilon_s}{k_N \tilde{K}_s} & \text{otherwise,} \end{cases} \tag{3.16}$$

[2] Using pre-determined maximum number of steps and given absolute tolerance δ, we start with $\beta = 0$. Then at each step, we compute $a = c\exp(N\beta) + \exp(N)$ and $f = \log(a)$. If $|f| < \delta$, then β is the desired solution. Otherwise, we update $\beta \leftarrow \beta - f/((cN\exp N\beta + \exp\beta)/a)$ and continue with the next step.

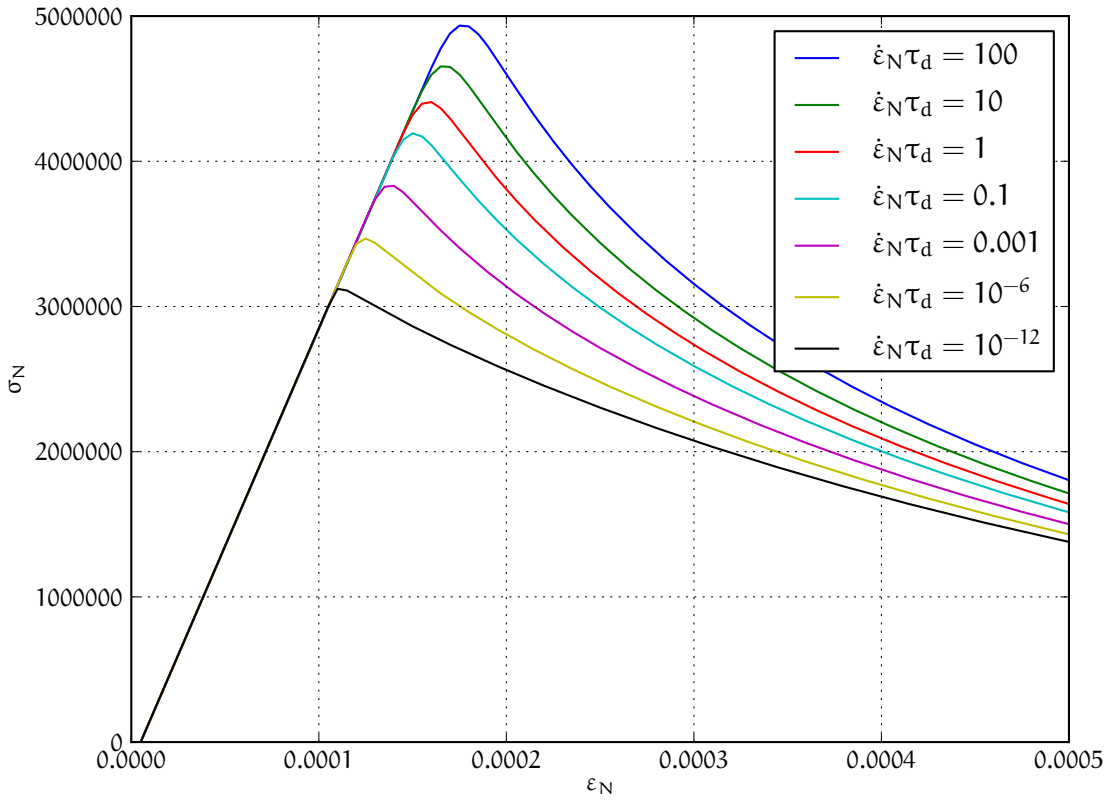


Figure 3.4.: Strain-stress curve in tension with different rates of loading; the parameters used here are $\tau_d = 1\,\mathrm{s}$ and $M_d = 0.1$.

where the second case takes in account compressive plasticity. The constitutive law then uses the adjusted value ε'_N. At the end, computed normal stress is adjusted back to

$$\sigma'_N = \sigma_N - \sigma_0. \tag{3.17}$$

before being applied on particle in contact.

3.2.4. Shear stresses

For the shear stress we use plastic constitutive law

$$\boldsymbol{\sigma}_T = k_T(\boldsymbol{\varepsilon}_T - \boldsymbol{\varepsilon}_{Tp}) \tag{3.18}$$

where ε_{Tp} is the plastic strain on the contact and k_T is shear contact modulus computed from k_N as the ratio k_T/k_N is fixed (usually 0.2, see the sect. 3.3).

In the DEM formulation (large strains), however, ε_{Tp} is not stored and mapped contact points on element surfaces are moved instead as explained above, sect. 2.3.2.2.

The shear stress is limited by the yield function (fig. 3.5)

$$f(\sigma_N, \boldsymbol{\sigma}_T) = |\boldsymbol{\sigma}_T| - r_{pl} = |\boldsymbol{\sigma}_T - (c_T - \sigma_N \tan\varphi), \qquad c_T = c_{T0}(1-\omega) \tag{3.19}$$

where material parameters c_{T0} and φ are initial cohesion and internal friction angle, respectively. The initial cohesion c_{T0} is reduced to the current cohesion c_T using damage state variable ω. Note that we split the plasticity function in a part that depends on $\boldsymbol{\sigma}_T$ and another part which depends on already known values of ω and σ_N; the latter is denoted r_{pl}, radius of the plasticity surface in given σ_N plane.

The plastic flow rule

$$\dot{\boldsymbol{\varepsilon}}_{Tp} = \dot{\lambda}\frac{\boldsymbol{\sigma}_T}{|\boldsymbol{\sigma}_T|}, \tag{3.20}$$

λ being plastic multiplier, is associated in the plane of shear stresses but not with respect to the normal stress (fig. 3.5). The advantage of using a non-associated flow rule is computational. At every step, σ_N can be evaluated directly, followed by a direct evaluation of $\boldsymbol{\sigma}_T$; stress return in shear stress plane reduces to simple radial return and does not require any iterations as $f(\sigma_N, \boldsymbol{\sigma}_T) = 0$ is satisfied immediately.

In the implementation, numerical evaluation starts from current value of $\boldsymbol{\varepsilon}_T$. Trial stress $\boldsymbol{\sigma}_T^t = \boldsymbol{\varepsilon}_T k_T$ is computed and compared with current plasticity surface radius r_{pl} from (3.19). If $|\boldsymbol{\sigma}_T^t| > r_{pl}$, the radial stress return is performed; since we do not store $\boldsymbol{\varepsilon}_{Tp}$, $\boldsymbol{\varepsilon}_T$ is updated as well in such case:

$$\boldsymbol{\sigma}_T = r_{pl}\widehat{\boldsymbol{\sigma}_T} \tag{3.21}$$

$$\boldsymbol{\varepsilon}'_T = \widehat{\boldsymbol{\varepsilon}_T}\frac{|\boldsymbol{\sigma}_T|}{|\boldsymbol{\sigma}_T^t|} \tag{3.22}$$

If $|\boldsymbol{\sigma}_T^t| \le r_{pl}$, there is no plastic slip and we simply assign $\boldsymbol{\sigma}_T = \boldsymbol{\sigma}_T^t$ without $\boldsymbol{\varepsilon}_T$ update.

3.2.4.1. Confinement extension

As in the case of normal stress, we introduce an extension to better capture confinement effect and to prevent shear locking under extreme compression: instead of using linear plastic surface we replace it by logarithmic surface in the compression part, which has C_1 continuity with the linear surface in tension; the continuity condition avoids pathologic behavior around the switch point and also reduces number of new parameters. Instead of (3.19), we use

$$f(\sigma_N, \sigma_T) = \begin{cases} |\sigma_T| - (c_{T0}(1-\omega) - \sigma_N \tan\varphi) & \text{if } \sigma_N \ge 0 \\ |\sigma_T| - c_{T0}\left[(1-\omega) + Y_0 \tan\varphi \log\left(\frac{-\sigma_N}{c_{T0}Y_0} + 1\right)\right] & \text{if } \sigma_N < 0, \end{cases} \tag{3.23}$$

which introduces a new dimensionless parameter Y_0 determining how fast the logarithm deviates from the original, linear form. The function is shown at fig. 3.6.

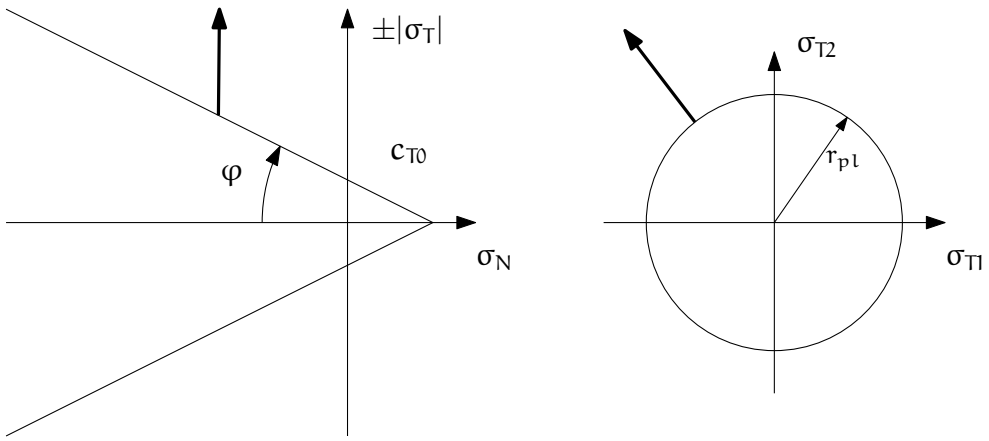


Figure 3.5.: Linear yield surface and plastic flow rule.

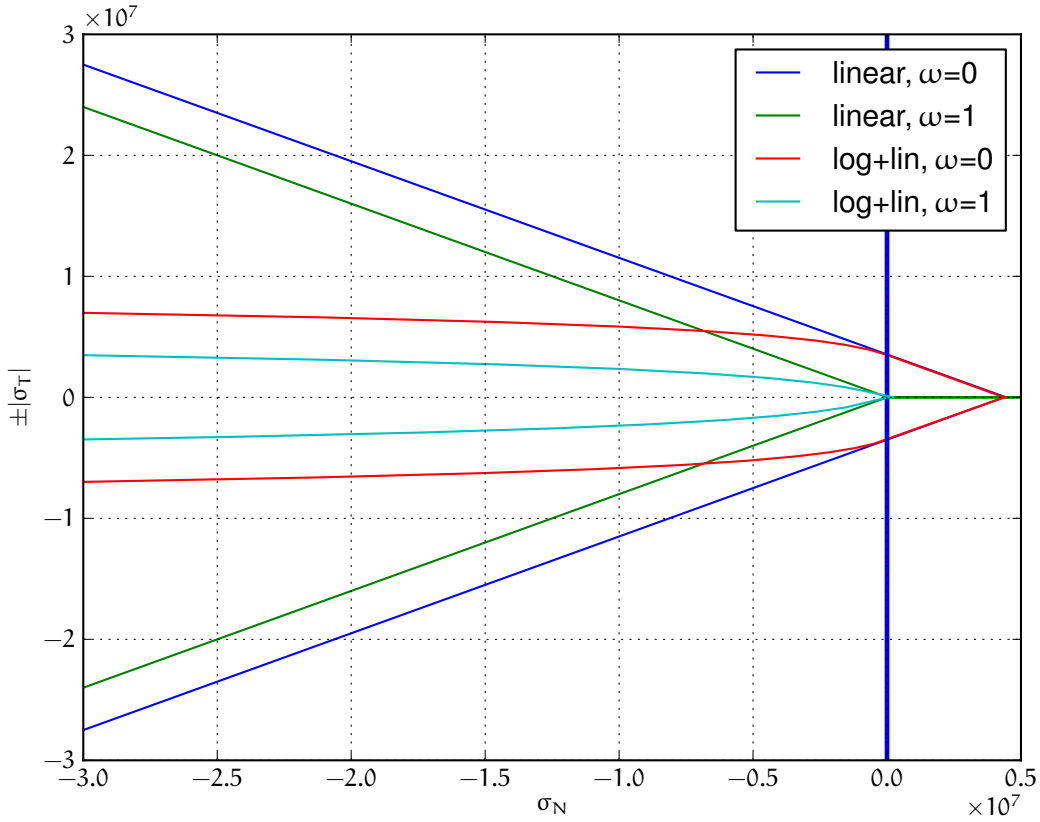


Figure 3.6.: Comparison of linear and logarithmic (in compression) yield surfaces, for both virgin and damaged material.

3.2.4.2. Visco-plasticity

Viscosity in shear uses similar ideas as visco-damage from sect. 3.2.3.2; the value of r_t in (3.19) is augmented depending on rate of plastic flow following the equation

$$r'_{pl} = r_{pl} + c_{T0}\left(\tau_{pl}\dot{\varepsilon}_{Tp}\right)^{M_{pl}} = r_{pl} + c_{T0}\left(\tau_{pl}\frac{\Delta\varepsilon_{Tp}}{\Delta t}\right)^{M_{pl}} \tag{3.24}$$

where τ_{pl} is characteristic time for visco-plasticity, M_{pl} is a dimensionless exponent (both to be calibrated. c_{T0} is undamaged cohesion introduced for the sake of dimensionality.

Similar solution strategy as for visco-damage is used, with different substitutions

$$\beta = \ln\left(\frac{|\boldsymbol{\sigma}_T^t| - r'_{pl}}{|\boldsymbol{\sigma}_T^t| - r_{pl}}\right), \tag{3.25}$$

$$c = c_{T0}\left(|\boldsymbol{\sigma}_T^t| - r_{pl}\right)^{M_{pl}-1}\left(\frac{\tau_{pl}}{k_T\Delta t}\right)^{M_{pl}}. \tag{3.26}$$

The equation to solve is then formally the same as (3.13)

$$\ln\left(e^{\beta} + ce^{M_{pl}\beta}\right) = 0. \tag{3.27}$$

After finding β using the Newton-Raphson method, trial stress $\boldsymbol{\sigma}_T^t$ is scaled by the factor

$$\frac{r'_{pl}}{|\boldsymbol{\sigma}_T^t|} = 1 - e^{\beta}\left(1 - \frac{r_{pl}}{|\boldsymbol{\sigma}_T^t|}\right) \tag{3.28}$$

to obtain the final shear stress value

$$\boldsymbol{\sigma}_T = \frac{r'_{pl}}{|\boldsymbol{\sigma}_T^t|}\boldsymbol{\sigma}_T^t. \tag{3.29}$$

3.2.5. Applying stresses on particles

Resulting stresses σ_N and and $\boldsymbol{\sigma}_T$ are computed at the current contact point $\mathbf{C}^\circ$. Summary force $\mathbf{F}_\Sigma = A_{eq}(\sigma_N\mathbf{n} + \boldsymbol{\sigma}_T)$ has to be applied to particle's positions $\mathbf{C}_1^\circ$, $\mathbf{C}_2^\circ$, exerting force and torque:

$$\mathbf{F}_1 = \mathbf{F}_\Sigma, \qquad \mathbf{T}_1 = (\mathbf{C}^\circ - \mathbf{C}_1^\circ)\times\mathbf{F}_\Sigma, \tag{3.30}$$

$$\mathbf{F}_2 = -\mathbf{F}_\Sigma, \qquad \mathbf{T}_2 = -(\mathbf{C}^\circ - \mathbf{C}_1^\circ)\times\mathbf{F}_\Sigma. \tag{3.31}$$

Forces and torques on particles from multiple interactions accumulate during one computation step.

3.2.6. Contact model summary

The computation described above proceeds in the following order:

Isotropic confinement σ_0 is applied if active. This consists in adjusting normal strain to either $\varepsilon_N \leftarrow \varepsilon_N + \sigma_0$ (if $\sigma_0 \geq k_N\varepsilon_s$) or $\varepsilon_N \leftarrow \varepsilon_s + (\sigma_0 - k_N\varepsilon_s)/(k_N\tilde{K}_s)$.

Normal stress σ_N. $\tilde{\varepsilon} = \langle\varepsilon_N\rangle$ is computed, then the history variable is updated $\kappa \leftarrow \max(\kappa, \tilde{\varepsilon})$; κ is the state variable from which the current damage $\omega = g(k) = 1 - (\varepsilon_f/\kappa)\exp(-(\kappa - \varepsilon_0)/\varepsilon_f)$ is evaluated; for non-cohesive contacts, however, we set $\omega = 1$. For cohesive contacts with damage disabled (**CpmPhys.neverDamage**), we set $\omega = 0$.

The state variable ε_N^{pl} (initially zero) holds the normal plastic strain; we use it to compute the elastic part of the current strain $\varepsilon_N^{el} = \varepsilon_N - \varepsilon_N^{pl}$.

Normal stress is computed using the equation of damage mechanics $\sigma_N = (1 - H(\varepsilon_N^{el})\omega)k_N\varepsilon_N^{el}$.

Whether compressive hardening is present is determined; $\sigma_{Ns} = k_N(\varepsilon_s + \tilde{K}_s(\varepsilon_N - \varepsilon_s))$ is pre-evaluated; then if $\varepsilon_N^{el} < \varepsilon_s$ and $\sigma_{Ns} > \sigma_N$, normal stress and normal plastic strains are updated $\varepsilon_N^{pl} \leftarrow \varepsilon_N^{pl} + (\sigma_N - \sigma_{Ns})/k_N$, $\sigma_N \leftarrow \sigma_{Ns}$. If the condition is not satisfied, compressive plasticity has no effect and does not have to be specifically considered.

Shear stress σ_T. First, trial value is computed simply from $\boldsymbol{\sigma}_T \leftarrow k_T\varepsilon_T$. This value will be compared with the radius of plastic surface r_{pl} for given, already known σ_N. As there are different plasticity functions for tension and compression, we obtain $r_{pl} = c_{T0}(1-\omega) - \sigma_N \tan\varphi$ for $\sigma_N \geq 0$; in compression, the logarithmic surface makes the formula more complicated, giving $r_{pl} = c_{T0}\,[(1-\omega) + Y_0 \tan\varphi(-\sigma_N/(c_{T0}Y_0) + 1)]$.

If the trial stress is beyond the admissible range, i.e. $|\boldsymbol{\sigma}_T| > r_pl$, plastic flow will take place. Since the total formulation for strain is used, we update the $\boldsymbol{\varepsilon}_T$ to have the same direction, but the magnitude of $|\boldsymbol{\sigma}_T|/k_T$.

Without visco-plasticity (the default), a simple update $\boldsymbol{\sigma}_T \leftarrow (\sigma_T/|\sigma_T|)r_{pl}$ is performed during the plastic flow. If visco-plasticity is used, we update $\boldsymbol{\sigma}_T \leftarrow s\boldsymbol{\sigma}_T$, s being a scaling scalar. It is computed as $s = 1 - e^{\beta}(1 - r_{pl}/|\boldsymbol{\sigma}_T|)$, where β is solved with Newton-Raphson iteration as described above, with the coefficient $c = c_{T0}(\tau_{pl}/(k_T\Delta t))^{M_{pl}}(|\boldsymbol{\sigma}_T| - r_{pl})^{M_{pl}-1}$ and the exponent M_{pl}.

Viscous normal overstress $\sigma_{N\nu}$ is applied for cohesive contacts only. As explained above, it is effective for non-zero damage rate. When damage is not growing (i.e. the state variable $\varepsilon_d \geq \varepsilon_N\omega$, where ε_d is initially zero), we simply update $\varepsilon_d \leftarrow \varepsilon_N\omega$, and the overstress is zero

Otherwise, the viscosity equation has to be solved using the coefficient $c = \varepsilon_0(1-\omega)(\tau_d/\Delta t)^{M_d}(\varepsilon_N\omega - \varepsilon_d)^{M_d-1}$ and the exponent $N = M_d$; once β is solved with the Newton-Raphson method as shown above, we update $\varepsilon_d \leftarrow \varepsilon_d + (\varepsilon_N\omega - \varepsilon_d)e^{\beta}$ and finally obtain $\sigma_{N\nu} = (\varepsilon_N\omega - \varepsilon_d)k_N$. Then the overstress is applied via $\sigma_N \leftarrow \sigma_N + \sigma_{N\nu}$.

3.3. Parameter calibration

The model comprises two sets of parameters that determine elastic and inelastic behavior. In order to match simulations to experiments, a procedure to calibrate these parameters must be given. Since elastic properties are independent of inelastic ones, they are calibrated first.

Model parameters can be summarized as follows:

1. geometry

r	sphere radius
R_I	interaction radius

2. elasticity

k_N	normal contact stiffness
k_T/k_N	relative shear contact stiffness

3. damage and plasticity

ε_0	limit elastic strain
ε_f	parameter of damage evolution function
C_{T0}	shear cohesion of undamaged material
φ	internal friction angle

4. confinement

Y_0	parameter for plastic surface evolution in compression
ε_s	hardening strain in compression
$\tilde{K}_s$	relative hardening modulus in compression

5. rate-dependence

   τ_d characteristic time for visco-damage
   M_d dimensionless visco-damage exponent
   τ_{pl} characteristic time for visco-plasticity
   M_{pl} dimensionless visco-plasticity exponent

Macroscopic properties should be matched to prescribed values by running simulation on sufficiently large specimen. Let us give overview of them, in the order of calibration:

1. elastic properties, which depend on only geometry and elastic parameters (using grouping from the list above)

   E Young's modulus,
   ν Poisson's ratio

2. inelastic properties, depending (in addition) on damage and plasticity parameters:

   f_t tensile strength
   f_c compressive strength
   G_f fracture energy (conventional definition shown in fig. 3.7)

3. confinement properties; they appear only in high confinement situations and can be calibrated without having substantial impact on already calibrated inelastic properties. We do not describe them quantitatively; fitting simulation and experimental curves is used instead.

4. rate-dependence properties; they appear only in high-rate situations, therefore are again calibrated after inelastic properties independently. As in the previous case, a simple fitting approach is used here.

3.3.1. Simulation setup

In order to calibrate macroscopic properties, simulations with multiple particles have to be run. This allows to smooth away different orientation of individual contacts and gives apparent continuum-like behavior.

We were running simple strain-controlled tension/compression (**UniaxialStrainer**) test on a 1:1:2 cuboid-shaped specimen of 2000 spheres.[3] Straining is applied in the direction of the longest dimension, on boundary particles; they are identified, on the "positive" and "negative" end of the specimen, by distance from bounding box of the specimen; as result, roughly one layer of spheres is considered as support on each side. Distance between (some) two spheres on each end along the strained axis determines the reference length l_0; specimen elongation is computed from their current distance divided by l_0 during subsequent simulation. Straining imposes displacement on support particles along strained axis, symmetrically on either end of the specimen (half on the "positive" and half on the "negative" boundary particles), while all their other degrees of freedom are kept free, including perpendicular translations, leading to simulation of frictionless supports.

Axial force F is computed by averaging sums of forces on support particles from both supports F^+ and $-F^-$. Divided by specimen cross-section A, average stress is obtained. The cross-section area is estimated as either cross-section of the specimen's bounding box (for cuboid specimen) or as minimum of several areas A_i of convex hull around particles intersecting perpendicular plane at different coordinates along the axis (for non-prismatic specimen) – see fig. 3.8.

Such tension/compression test can be found in the **examples/concrete/uniax.py** script.

Periodic boundary conditions were not implemented in Yade until later stages of the thesis (the **Cell** class). In such case, determining deformation and cross-section area is much simpler, as it exists ob-

[3] Later, the test was being done on hyperboloid-shaped specimen, to pre-determine fracturing area, while avoiding boundary effects.

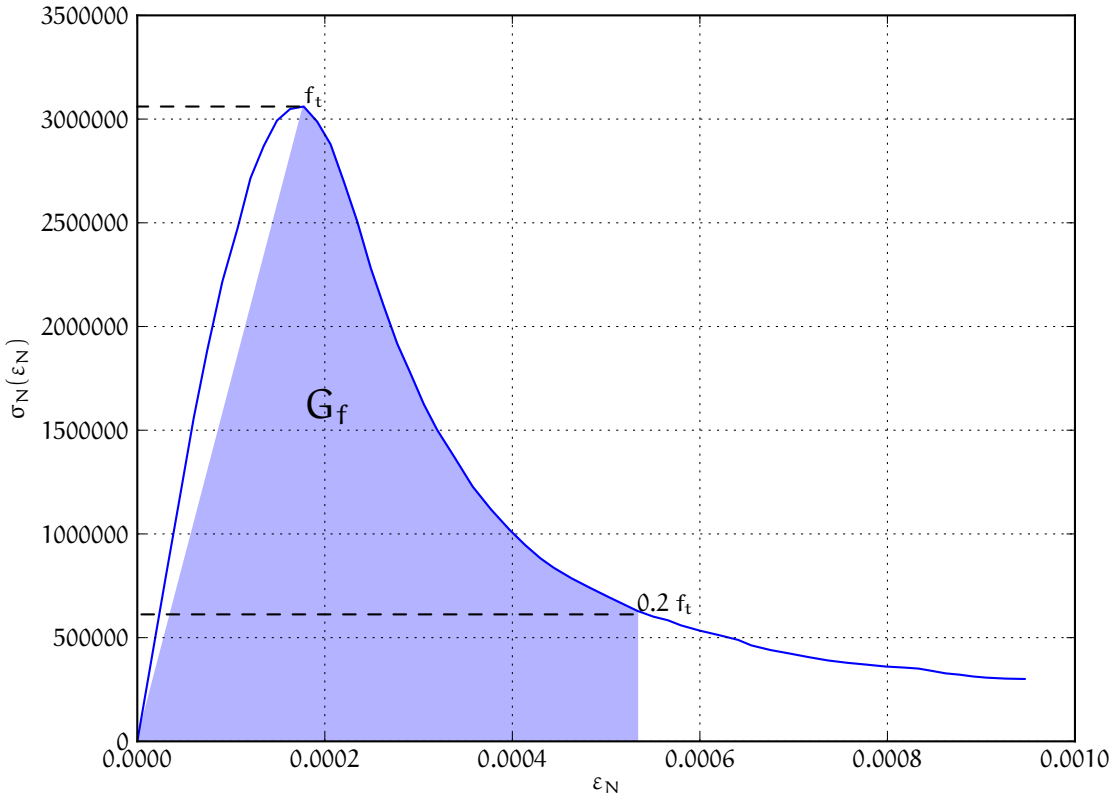


Figure 3.7.: Conventional definition of fracture energy of our own, which goes only to $0.2 f_t$ on the strain-stress curve.

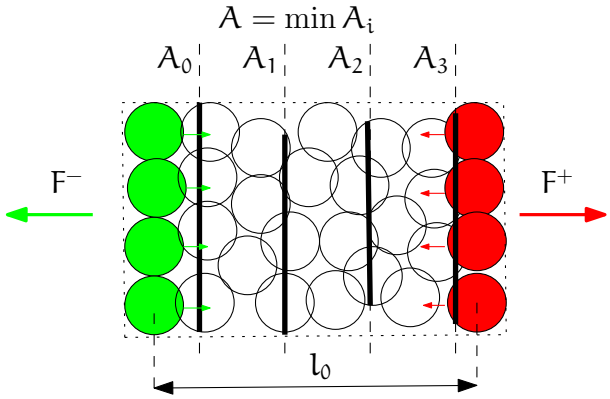


Figure 3.8.: Simplified scheme of the uniaxial tension/compression setup. Strained spheres, negative and positive support, are shown in green and red respectively. Cross-section area A is minimum of convex hull's areas A_i.

jectively, embodies in the periodic cell size. Computing stress is equally trivial: first, vector of sum of all forces on interactions in the cell (taking tensile forces as positive and compressive as negative) is computed, then divided by-component by perpendicular areas of the cell. This is handled by **PeriTriaxCompressor** and **PeriTriaxController** classes.

3.3.1.1. Stress tensor evaluation in arbitrary volume

Computation of stress from reaction forces is not suitable for cases where the loading scenario is not as straight-forwardly defined as in the case of uniaxial tension/compression. For general case, an equation for stress tensor can be derived. Using the work of Kuhl et al. [31], eqs. (33) and (35), we have

$$\boldsymbol{\sigma} = \frac{1}{V}\sum_{c\in V}[\mathbf{F}^c_\Sigma \otimes (\mathbf{C}_2 - \mathbf{C}_1)]^{\mathrm{sym}} = \tag{3.32}$$

$$= \frac{1}{V}\sum_{c\in V} l^c \left[\mathbf{N}^c F^c_N + \mathbf{T}^{c\,\mathrm{T}} F^c_T\right] \tag{3.33}$$

where V is the considered volume containing interactions with the c index. For each interaction, there is $l = |\mathbf{C}_2 - \mathbf{C}_1|$, $F_\Sigma = F_N\mathbf{n} + \mathbf{F}_T$, with all variables assuming their current value. We use 2^{nd}-order normal projection tensor $\mathbf{N} = \mathbf{n}\otimes\mathbf{n}$ which, evaluated component-wise, gives

$$\mathbf{N}_{ij} = \mathbf{n}_i\mathbf{n}_j. \tag{3.34}$$

The 3^{rd}-order tangential projection tensor $\mathbf{T}^{\mathrm{T}} = \mathbf{I}_{\mathrm{sym}}\cdot\mathbf{n} - \mathbf{n}\otimes\mathbf{n}\otimes\mathbf{n}$ is written by components

$$\mathbf{T}^{\mathrm{T}}_{ijk} = \frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})\,\mathbf{n}_l - \mathbf{n}_i\mathbf{n}_j\mathbf{n}_k = \tag{3.35}$$

$$= \frac{\delta_{ik}\mathbf{n}_j}{2} + \frac{\delta_{jk}\mathbf{n}_i}{2} - \mathbf{n}_i\mathbf{n}_j\mathbf{n}_k. \tag{3.36}$$

Plugging these expressions into (3.33) gives

$$\boldsymbol{\sigma}_{ij} = \sum_{c\in V}\mathbf{n}^c_i\mathbf{n}^c_j F^c_N + \frac{\mathbf{n}^c_j\mathbf{F}^c_{Ti}}{2} + \frac{\mathbf{n}^c_i\mathbf{F}^c_{Tj}}{2} - \mathbf{n}^c_i\mathbf{n}^c_j\underbrace{\mathbf{n}^c_k\mathbf{F}^c_{Tk}}_{=0,\ \text{since}\ \mathbf{n}^c\perp\mathbf{F}^c_T} \tag{3.37}$$

Results from this formula were slightly lower than stress obtained from support reaction forces. It is likely due to small number of interaction in V; we were considering an interaction inside if the contact point was inside spherical V, which can also happen for an interaction between two spheres outside V; some weighting function could be used to avoid V boundary problems.

Boundary effect is avoided for periodic cell (**Cell**), where the volume V is defined by its size and all interaction would are summed together.

This algorithm is implemented in the **eudoxos.InteractionLocator.macroAroundPt()** method.

3.3.2. Geometry and elastic parameters

Let us recall the parameters that influence the elastic response of the model:

Radius r. The radius is considered to be the same for all the spheres, for the following two reasons:

1. The time step of the computation (which is one of the main factors determining computational costs) depends on the smallest critical time step for all bodies. Small elements have a smaller critical time step, therefore they would negatively impact the performance.
2. A direct correlation of macroscopic and contact-level properties is based on the assumption that the sphere radii are the same.

Interaction radius R_I is the relative distance determining the „non-locality“ of contact detection. For $R_I = 1$, only spheres that touch are considered as being in contact. In general, the condition reads

$$d_0 \leq R_I(r_1 + r_2). \tag{3.38}$$

The value of R_I directly influences the average number of interactions per sphere (percolation). For our purposes, we recommend to use $R_I = 1.5$, which gives the average of ≈12 interactions per sphere for packing with density > 0.5.

This value was determined experimentally based on the average number of interactions; it stabilizes the packing with regards to contact-level phenomena (damage) and makes the model, in a way, “non-local”.

R_I also importantly influences the f_t/f_c ratio, which was the original motivation for increasing its value from 1.

R_I only serves to create initial (cohesive) interactions in the packing; after the initial step, interactions having been established, it is reset to 1. A disadvantage is that fractured material which becomes compact again (such as dust compaction) will have a smaller elastic stiffness, since it will have a smaller number of contacts per sphere.

k_N **and** k_T are contact moduli in the normal and shear directions introduced above.

These 4 parameters should be calibrated in such way that the given macroscopic properties E and ν are matched. It can be shown by dimensional analysis that ν depends on the dimensionless ratio k_N/k_T and, if R_I is fixed, Young's modulus is proportional to k_N (at fixed k_N/k_T).

By analogy with the microplane theory, the dependence can be derived analytically (see [31]) as

$$\nu = \frac{k_N - k_T}{4k_N + k_T} = \frac{1 - k_T/k_N}{4 + k_T/k_N}, \tag{3.39}$$

which matches quite well the results our simulations (fig. 3.9). Stránský et al. [63] reports similar numerical results, which get closer to theoretical values as R_I grows.

For E, similar equations can be derived, leading to

$$\frac{E}{k_N} = \frac{\sum A_i L_i}{3V} \frac{2 + 3\frac{k_T}{k_N}}{4 + \frac{k_T}{k_N}}, \tag{3.40}$$

where A_i is cross-sectional area of contact number i, L_i is its length and V is the volume of space in which the spheres are placed (total volume of the given sample). The first fraction, volume ratio, is determined solely by the interaction radius R_I; therefore, E depends linearly on K_N.

In our case, however, we simply run elastic simulation to determine the actual E/k_N ratio (3.40). To obtain desired macroscopic modulus of E^*, the value of k_N is scaled by E^*/E.

3.3.2.1. Measuring macroscopic elastic properties

Measuring linear properties in dynamic simulations faces 2 sources of non-linearity:

1. Dynamic oscillations may influence results if strain rate is too high. This can be avoided by stopping loading at some point and letting kinetic energy dissipate by using numerical damping (**NewtonIntegrator.damping**).
2. Early non-linear behavior might disturb the results. For avoiding damage, special flag **CpmMat.neverDamage** was introduced to the material, causing it to never damage. To prevent plasticity, loading to low strains is necessary. However, due to $R_I = 1.5$, there will be no plastic behavior (rearranging particles under initial load, which would make the response artificially more compliant) until later loading stages.

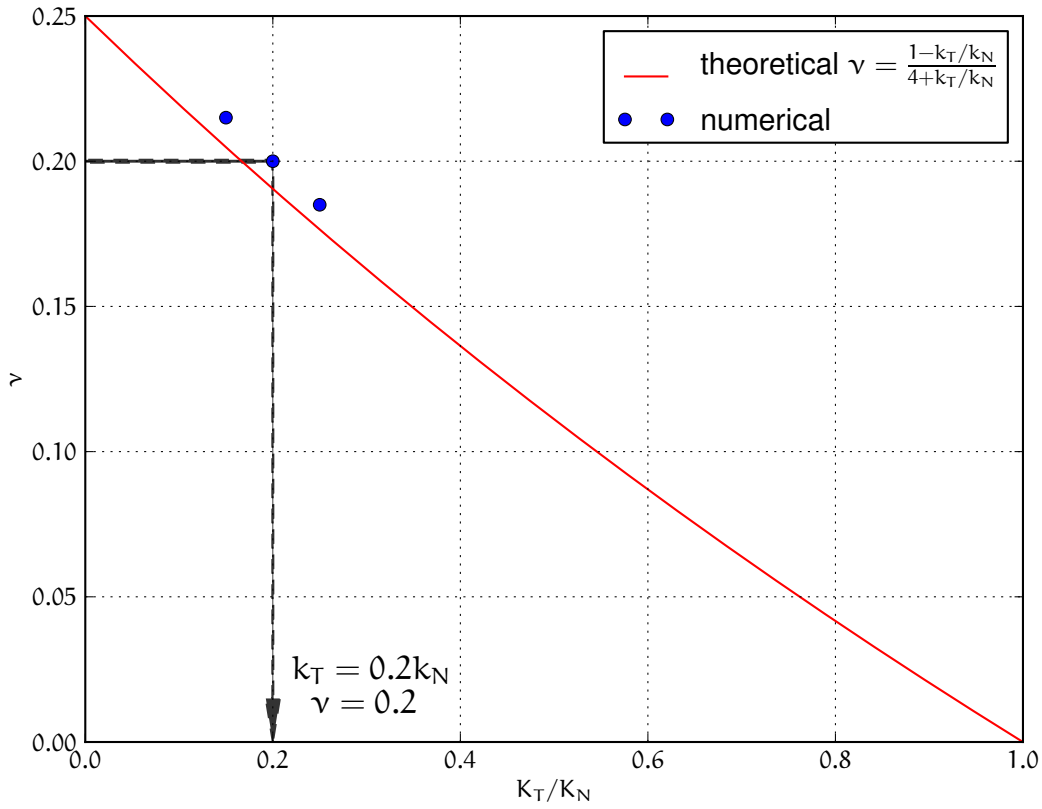


Figure 3.9.: Relationship between k_T/k_N and ν.

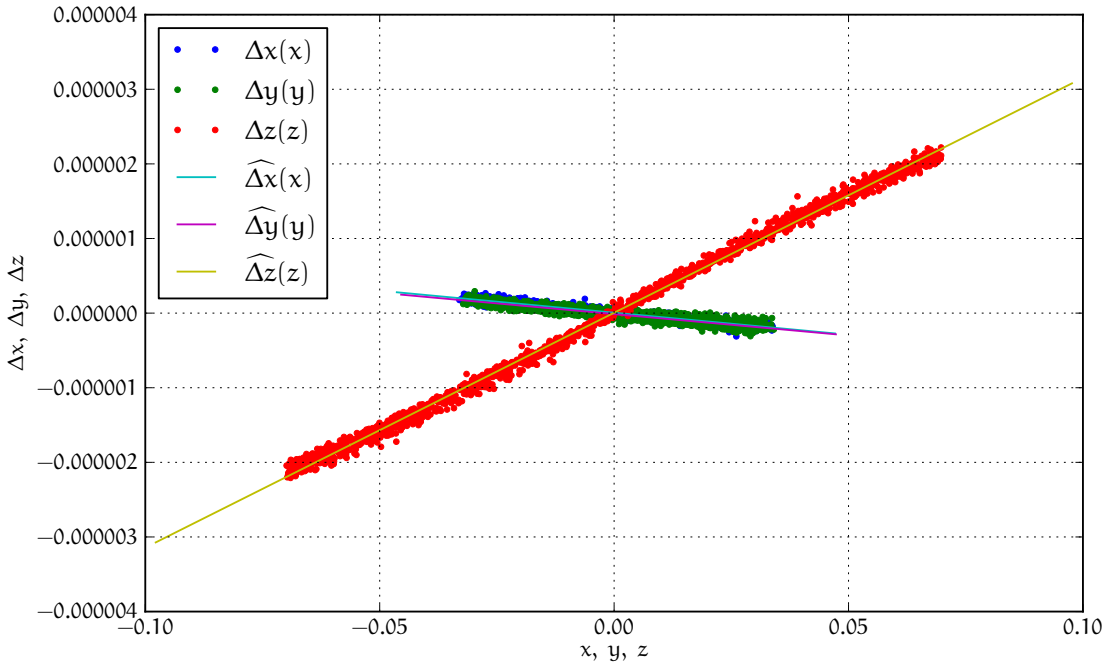


Figure 3.10.: Displacements during uniaxial tension test, plotted against position on respective axis. The slope of the regression $\widehat{\Delta x}(x)$ is the average ε_x in the specimen. Straining was applied in the direction of the z axis (as $\varepsilon_z > 0$) in the case pictured.

Young's modulus can be evaluated in a straight-forward way from its definition σ_i/ε_i, if $i \in \{x, y, z\}$ is the strained axis.

Poisson's ratio. The original idea of measuring specimen dilation by tracking displacement of some boundary spheres was quickly abandoned, as it was giving highly unstable response due to local irregularities and boundary effects. Later, a simple and reliable way was found, consisting in correlation between average axial and transversal displacements.

Taking $w \in \{x, y, z\}$, we evaluate displacement from the initial position $\Delta w(w)$ for all particles. To avoid boundary effect, only sufficient number of particles inside the specimen can be considered. The slope of linear regression $\widehat{\Delta w}(w)$ has the meaning of average ε_w, shown in fig. 3.10. If z is the strained axis, Poisson's ratio is then computed as

$$\nu = \frac{-\frac{1}{2}(\varepsilon_x + \varepsilon_y)}{\varepsilon_z}. \tag{3.41}$$

The algorithms described are implemented in the **eudoxos.estimatePoissonYoung()** function.

3.3.3. Damage and plasticity parameters

Once the elastic parameters are calibrated, inelastic parameters ε_0, ε_f, c_{T0} and φ should be adjusted such that we obtain the desired macroscopic properties f_t, f_c, G_f. The calibration procedure is as follows:

1. We transform model parameters to be dimensionless and material properties to be normalized:

   parameters $\frac{\varepsilon_f}{\varepsilon_0}$ (relative ductility), $\frac{c_{T0}}{k_T \varepsilon_0}$, φ; (ε_0 is left as-is)

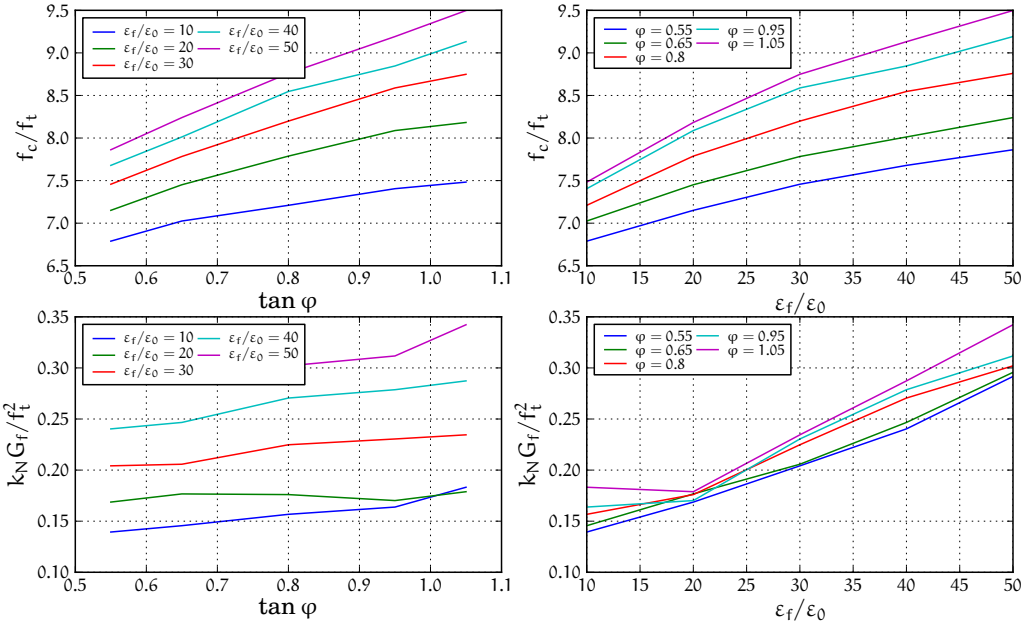


Figure 3.11.: Cross-dependencies of $\varepsilon_0/\varepsilon_f$, EG_f/f_t^2 and $\tan\varphi$. Since $\tan\varphi$ has little influence on $k_N G_f/f_t^2$ (lower left), first $\varepsilon_f/\varepsilon_0$ can be set based on desired $k_N G_f/f_t^2$ (lower right), then $\tan\varphi$ is determined so that wanted f_c/f_t ratio is obtained (upper left).

properties $\frac{f_c}{f_t}$ (strength ratio), $\frac{k_N G_f}{f_t^2}$ (characteristic length); (f_t is lest as-is)

There is one additional degree of freedom on both sides (ε_0 and f_t), which we will use later.

2. Since there is one additional parameter on the material model side, we fix c_{T0} to a known good value. It was shown that it has the least influence on macroscopic properties, hence the choice.

3. From graphs showing the parameter/property dependence, we set $\varepsilon_f/\varepsilon_0$ to get the desired $k_N G_f/f_t^2$ (fig. 3.11 lower right), since the only remaining parameter φ has (almost) no influence on $k_N G_f/f_t^2$ (fig. 3.11 lower left).

4. We set $\tan\varphi$ such that we obtain the desired f_c/f_t (fig. 3.11 upper left).

5. We use the remaining degree of freedom to scale the stress-strain diagram to get the absolute values using *radial scaling* (fig. 3.12). By dimensional analysis it can be shown that

$$f_t = k_N \varepsilon_0 \Psi\left(\frac{\varepsilon_f}{\varepsilon_0}, \frac{c_{T0}}{k_T \varepsilon_0}, \varphi\right). \tag{3.42}$$

Since k_N is already determined, it is only ε_0 that will directly determine f_t.

3.3.4. Confinement parameters

Calibrating three confinement-related parameters ε_s, $\tilde{K}_s$ and Y_0 is not algorithmic, but rather a trial-and-error process. On the other hand, typically it will be enough to calibrate the parameters for some generic confinement data, both for the lack of availability of exact measurements and for at best fuzzy matching that can be achieved. The chief reason is that the bilinear relationship for plasticity in compression is far from perfect and could be refined by using a smooth function; in our case, however, the confinement extension was only meant to mitigate high strength overestimation under confinement, not to accurately predict behavior under such conditions. Introducing more complicated functions would further increase the number of parameters, which was not desirable.

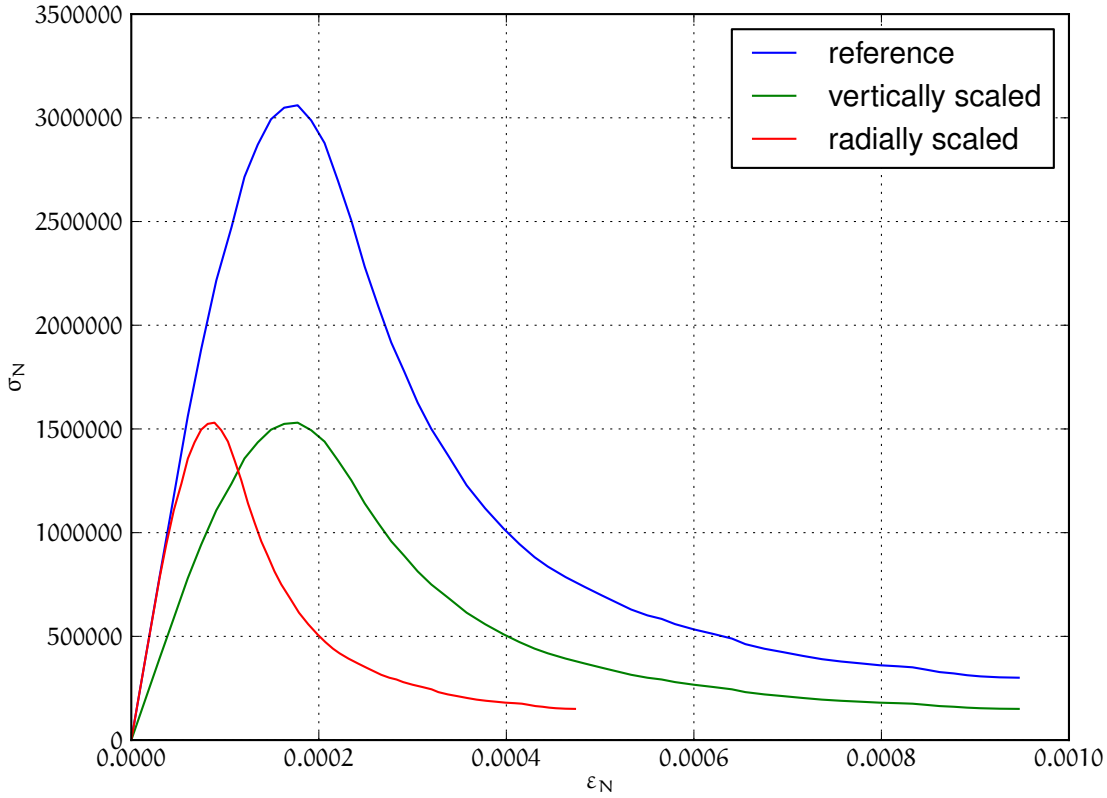


Figure 3.12.: Radial and vertical scaling of the stress-strain diagram; vertical scaling is used during calibration and is achieved by changing the value of ε_0.

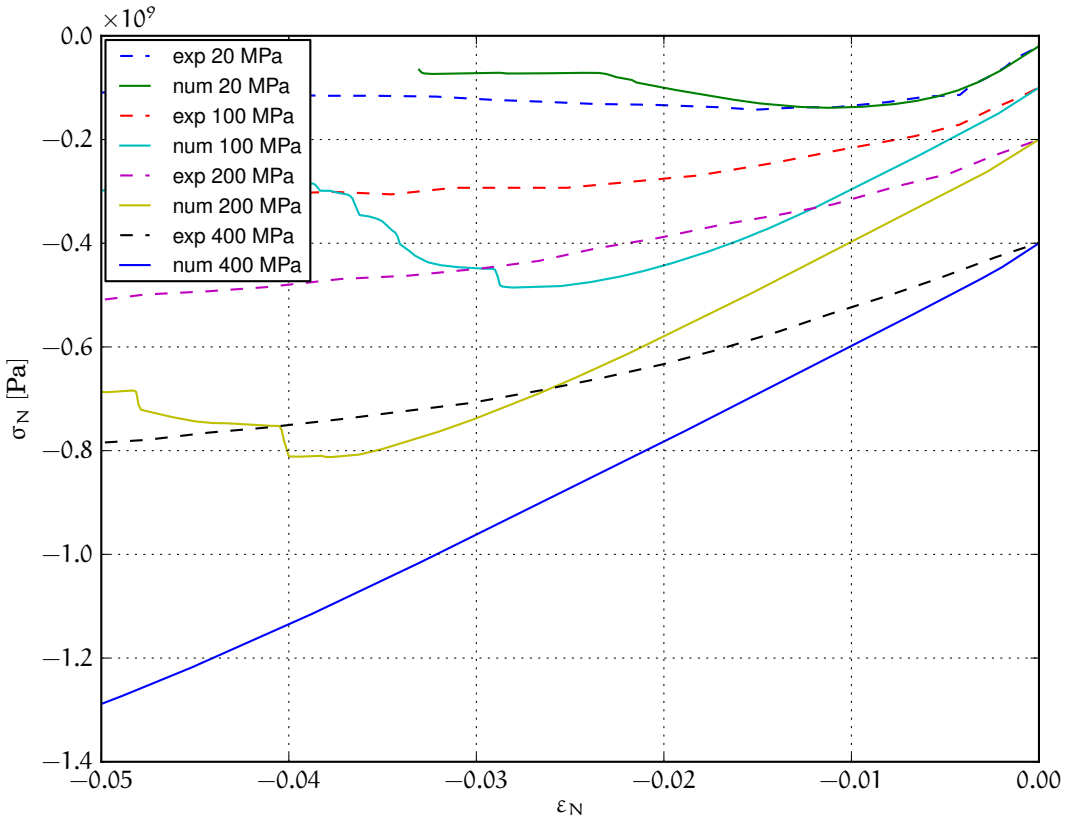


Figure 3.13.: Confined compression, comparing experimental data and simulation without the confinement extensions of the model. Experimental results (dashed) from Caner and Bažant [6].

The experimental data we use come from [6] and [16].

Consider confined strain-stress diagrams at fig. 3.13 exhibiting unrealistic behavior under high confinement (-400 MPa). Parameters ε_s and $\tilde{K}_s$ will influence at which point the curve will get to the hardening branch and what will be its tangent modulus (fig. 3.3). The Y_0 parameter determines evolution of plasticity surface in compression (fig 3.6). We recommend the following values of the parameters:

$$\varepsilon_s = -3 \cdot 10^{-3}, \qquad \tilde{K}_s = 0.3, \qquad Y_0 = 0.1, \tag{3.43}$$

which give curves in fig. 3.14. It was observed when running multiple simulations that results under high confinement depend greatly on the exact packing configuration, specimen shape and specimen size; therefore, the values given above should be taken with grain of salt.

During simulation, the confinement effect was introduced on the contact level, in the constitutive law itself, as described in sect. 3.2.3.3; the confinement is therefore isotropic and without boundary influence.

Cross-dependencies. Confinement properties may, to certain extent, have influence on inelastic properties. If that happens, reiterating the calibration with new confinement properties should give wanted results quickly.

3.3.5. Rate-dependence parameters

The visco-damage behavior in tension introduced two parameters, characteristic time τ_d and exponent M_d. There is no calibration procedure developed for them, as measuring the response is experimentally

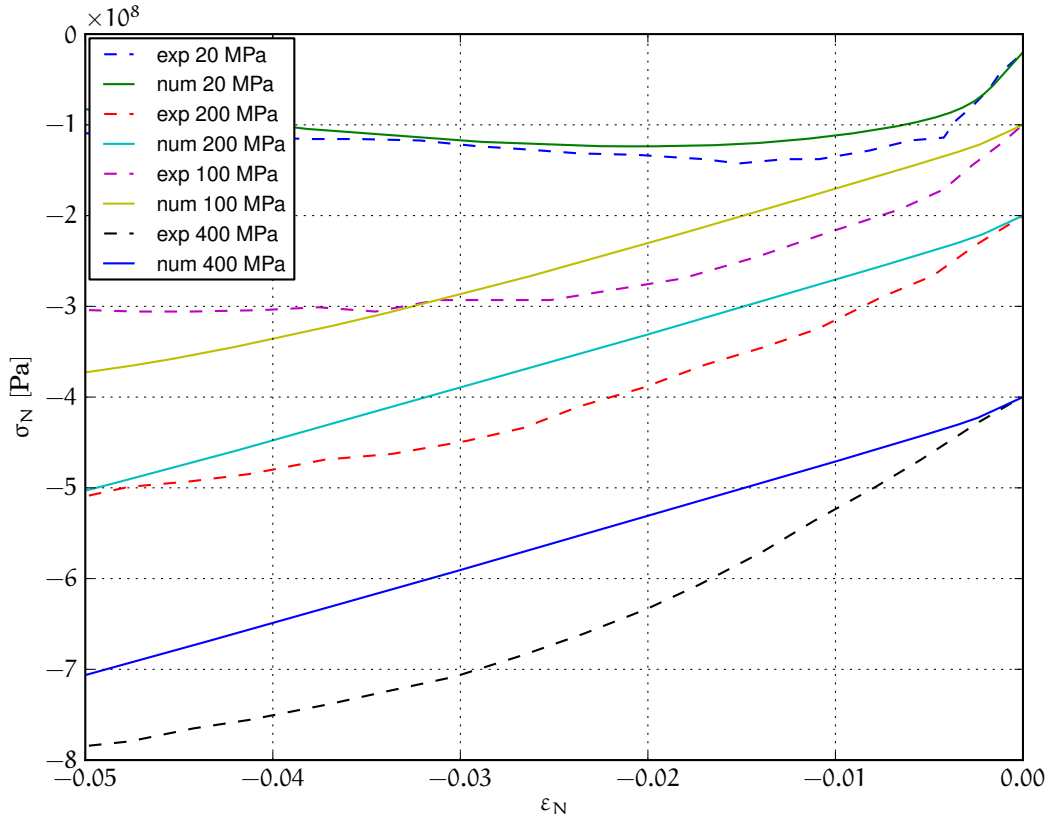


Figure 3.14.: Experimental data and simulation in confined compression, using confinement extensions of the model. Cf. fig 3.13 for influence of those extensions.

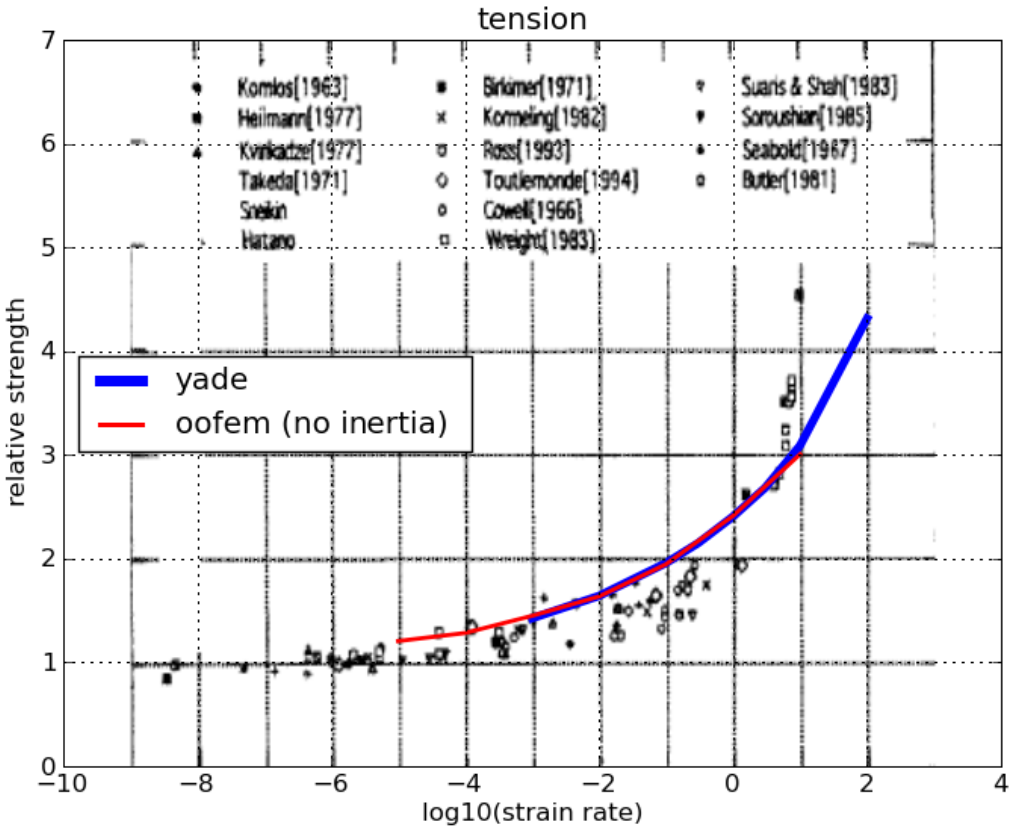


Figure 3.15.: Experimental data and simulation results for tension.

challenging and the scatter of results is rather high. Instead, we determined those two parameters by a trial-and-error procedure so that the resulting curve approximately fits the experimental data cloud — we use figures from [53], which are in turn based on published experiments.

The resulting curves are show in figs. 3.15 and 3.16. Because DEM computation would be very slow (large number of steps, determined by critical timestep) for slow rates, those results were computed with the same model implemented in the OOFEM framework (using a static implicit FEM model); this also served to verify that both implementations give identical results. For high loading rates, Yade's results deviate, since there is inertial mass that begins to play an important role.

The values that we recommend to use are

$$\tau_d = 1000\,\mathrm{s} \qquad\qquad M_d = 0.3.$$

Calibration of visco-plastic parameters was rather simple: we found out that it has no beneficial effect on results; therefore, visco-plasticity should be deactivated.[4]

[4] In the implementation, this is done by setting τ_{pl} to an arbitrary non-positive value.

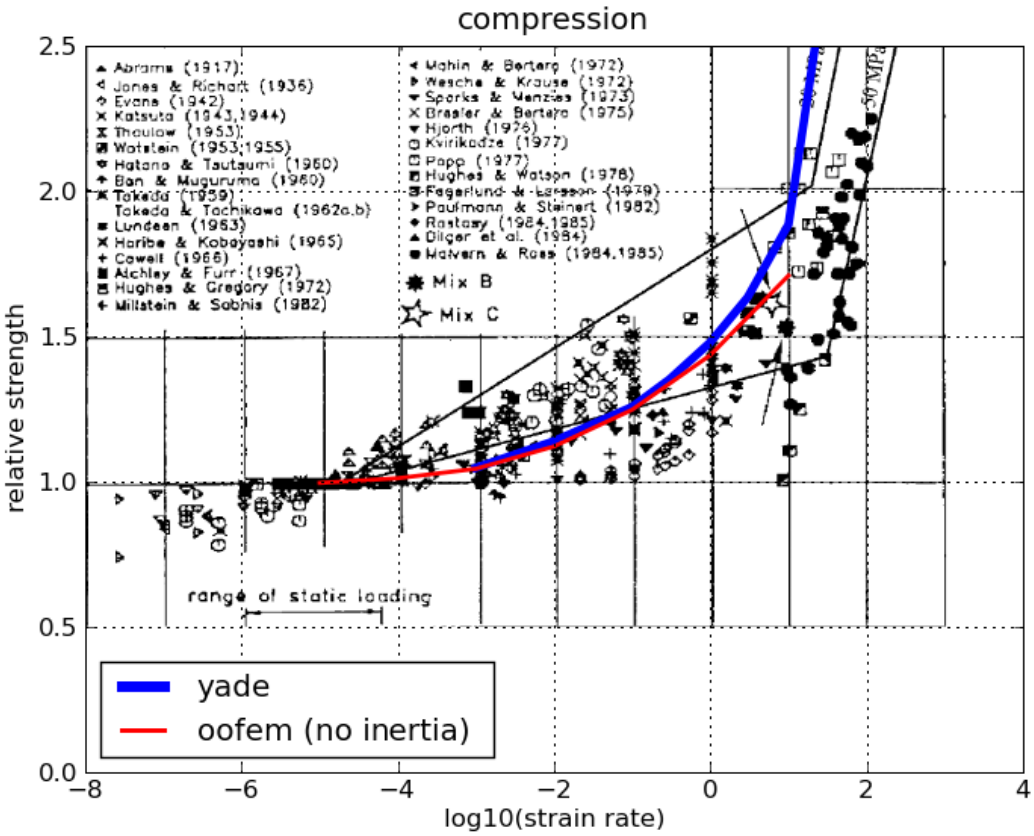


Figure 3.16.: Experimental data and simulation results for compression.

Part II.

The Yade platform

4. Overview

4.1. History

In 1990, Frédéric Donzé started "Spherical Discrete Element Code" (SDEC, Donzé [11]), using rigid spherical particles in 3d, with explicit leap-frog integration scheme, using elastic-brittle behavior in the normal sense and Mohr-Coulomb criterion for shear.

Already in mid-1990s, SDEC inspired first version of ESyS-Particle [14], which was focusing on high-performance computing and was developed in-house at that time.

Having perceived inflexibility of SDEC in face of new emerging methods and method couplings, Donzé initiated [71] conception of Yade, flexible platform for dynamic computing in 2004. The initial implementation was done in c++ by Jan Kozicki and Olivier Galizzi, back-then both working at the University of Grenoble. The platform was supposed to unite multiple different methods (FEM, DEM, lattice models, mass-spring models, realtime rigidbody dynamics), but with the exception of DEM and lattice models, they all stagnated at proof-of-concept stage.

Many advanced programming concepts were used (polymorphism, multiple dispatch, plugins); the original framework was described in (belated) papers Kozicki and Donzé [29, 30].

Arguably, the framework was over-designed with much more framework than functionality, poorly documented; that was criticised by the author of this paper [67]. In years 2007–2010, I became the the *de facto* lead developer and Yade underwent several significant changes:

Python scripting interface using boost::python was the most important and involved change and warrants its dedicated section 4.2.4.

Code cleanup, removing unused, incomplete or poorly functioning classes. Most classes were renamed for consistency, and many abstraction layers were removed. This made Yade perhaps less flexible, but more functional.

Documentation within the source code, setting up wiki.

Community moving development resources to launchpad.net, animating mailing lists, initiating dedicated website `https://www.yade-dem.org`. The community currently counts (based on list subscription counts) 60 users and 27 developers.

Parallel computation on shared-memory multiprocessors, using relatively non-intrusive OpenMP framework. This makes the computation (roughly) 5.5 × faster on 8 cores. Lots of other performance-related improvements were done as well.

4.2. Software architecture

The original Yade design was described in Kozicki and Donzé [29]; although some parts are already refactored, it still contains useful overview of patters that gave rise to Yade. The current architecture is described, from programmer's point of view, in the *Programmer's manual*, chapter 7.

The framework has layered structure; any part can only depend on a same-level or a lower-level part:

Libraries is the lowest level defining functionality which is not related specifically to simulations; this entails mathematics (vector, quaternion, simple matrix algebra), serialization (storing arbitrary class instances to file and restoring them later), multimethods (described below), OpenGL-related functions for 3d display, import functions for foreign file formats (STereo Litography), etc.

External libraries also count to this layer. Yade tries to delegate as much as possible to other (high-quality) open-source codes; in particular, various boost.org libraries are used extensively.

Core layer defines the most abstract simulation-related classes, such as **Scene**, **Body** (particle), **Interaction**, **Engine**. It contains the executable as well (which is itself in python) and could be run, though uselessly, without higher levels.

Common layer defines data structures useful for different simulation methods, not only DEM, such as spherical particle, elastic material, approximate collision detection, generic OpenGL renderer.

Specialized layer contains functionality for particular simulation methods. This layer used to contain DEM, FEM, lattice, basic realtime-rigidbody, ... implementations; at this moment, only DEM is used, making the distinction between "common" and "specialized" layers less obvious than it was originally planned.

The framework is implemented in c++ and is trying to use many features c++ offers, where it makes sense; readers not familiar with them can find introductory paragraphs in Appendix A.

4.2.1. Documentation

As is the case with many research programs, documentation was severely lacking [67]. Although Doxygen comments were in place, they were unpractical due to

1. Heavy use of templates, creating many unreadable symbols,
2. Length of generation and size of documentation files (of which chief part was occupied by PNG images of call graphs for each class)
3. Lack of integration with python, of which need didn't emerge until python started to be used seriously.

Sphinx was chosen as the tool for documenting python code (since Python 2.5, it is the documentation tool for Python itself as well, see `http://docs.python.org/`). Documentation for c++ classes and their attributes is put in special preprocessor macros; as most c++ classes are mirrored in Python automatically, this documentation becomes docstring of that object in python, allowing effectively to create documentation for the c++ class by documenting the wrapper class in python. Sphinx allows fairly complicated markup (math, images, lists, footnotes, citations) and, most importantly, easily integrates automatically-generated documentation with hand-written, continuous text. Finally, Sphinx handles multiple output formats; its LaTeX output made it possible to include large parts of Yade documentation (available online at `https://www.yade-dem.org/sphinx/`) directly in this thesis.

Although the documentation is still far from being complete, it is already useful for newcomers as well as moderately experienced users.

4.2.2. Modularity

The original modularity idea was that users will locally compile their own classes, independent of those already distributed. In practice, this has never been the case, as the code was not stable and bug-free enough to make the separation of core and external plugins meaningful. Despite that, plugins are conveniently used to conditionally disable parts of functionality, quickly change only part of code without recompiling the rest and so on. Plugin functionality is composed of several otherwise-unrelated features:

1. Plugin can be queried what classes they provide, when being loaded.

2. Classes can be instantiated based on their name. Classes are not binary objects as e.g. functions are (of which pointer can be retrieved via a **dlsym(...)** call); therefore, *factory function*, which returns pointer to new instance every time called, must be defined for each class.

A special object, usually called *class factory*, keeps track of all factory functions; given an identifier of class (which is usually its name as string), it finds associated factory function, calls it and returns pointer to the new instance. User is then responsible to cast this pointer from generic type[1] to appropriate special type.

4.2.3. Serialization

Serialization refers to converting arbitrary in-memory object to is serial representation, from which it can be reconstructed again (deserialization). All objects that are able to serialize themselves derive from **Serializable** class and can enumerate attributes describing their state which must be serialized; those attributes must be again serializable: either a **Serializable**, or primitive serializable type (number, vector, string, etc).

Since the whole simulation (**Scene** class) is a **Serializable**, it can be saved to file (using fairly readable XML representation) and reloaded later.

Currently, Yade uses its own serialization framework. Using (much faster) **boost::serialization** instead is already possible with special compile-time switches and will be preferred in the future.

4.2.4. Python interface

Scripting languages in general provide fast, easy and flexible way to control numerical programs [32]. Python was the language of choice for Yade, used for simulation setup (which proved to be the most future-compatible format), control, debug-inspection and post-processing.

Integration of Python into Yade evolved in several (unplanned) steps over years:

1. Primitive and explicit wrapping of a few basic functions controlling the simulation.
2. Explicit wrapping of base classes which was abusing serialization interface for attribute access (hence all attributes had to be converted to string first, then back). Derived classes were only instantiated using special syntax.
3. Explicit wrapping of base classes, but providing native implementations of attribute access via macros; instantiation of derived classes was handled via proxy metaclasses constructed from Yade's internal RTTI information.
4. Implicit wrapping of all classes, using macros in class declarations, including native attribute access, documentation, and proper type hierarchy reflected in python; class objects are injected into dedicated namespace by special virtual functions called when plugin class is being registered at startup.

   Shared pointers,[2] which are used for memory management throughout Yade, integrate seamlessly with **boost::python**, as they are used automatically to manage lifetime of wrapped objects, avoiding big potential source of invalid memory accesses.

Plans for the future include seamless deriving of python classes from c++ classes including virtual function calls by using the special **boost::python::wrapper** class.

[1] Generic pointer type can be **void***, but also parent class of all factorable types; in Yade, such class is named **Factorable** and returned pointers are of type **shared_ptr**⟨**Factorable**⟩.

[2] Shared pointers are provided by **boost::shared_ptr** and are already standardized in the updated C++09 standard, currently accessible under namespace of Technical Report 1 (**tr1::shared_ptr**) on compilers that support it.

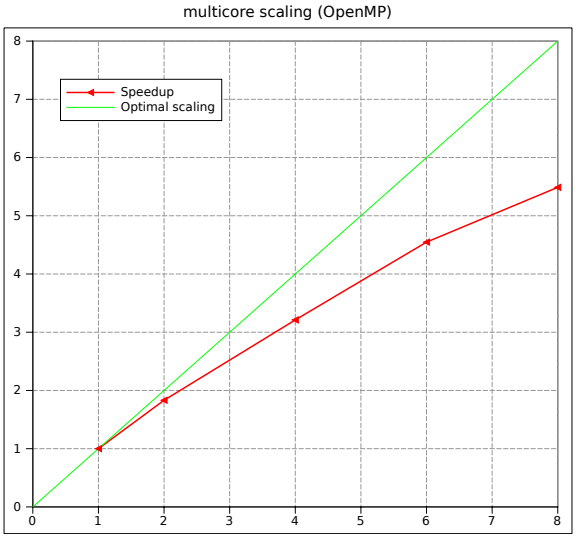


Figure 4.1.: Scalability of OpenMP parallelization in a 10k-particle simulation. Some parts of the simulation loop (**InsertionSortCollider**) are not parallelized at all, which is, along with overhead of OpenMP itself, reason for sub-optimal performance.

4.2.5. Parallel computation

DEM simulations with explicit integration scheme are said to be easily parallelized [4]. This is theoretically true, although there are numerous problems encountered in practice:

- unlike in FEM codes, substantial amount of time is spent in collision detection (both passes); this algorithm must be parallelized separately (if necessary);
- since global matrix is not assembled, existing parallel matrix solvers cannot be used;
- large number of short timesteps is used in explicit code; for small number of particles ($<10\,\mathrm{k}$), time per step is relatively small; for distributed-memory systems, this increases relative time synchronizing subdomains and accents the role of network latency as well.

As parallel computation in Yade was developed in an incremental fashion from originally serial code, non-intrusive parallel techniques were evaluated. The then-emerging OpenMP standard was chosen and several parts of the interaction loop were parallelized. This implies that Yade provides only shared-memory parallelism, but the availability of commodity multi-core processors makes this still non-negligible gain; scalability is shown at fig. 4.1.

In our case, 2 parts of the simulation loop are parallelized by default; in typical simulations, parallelizing other engines internally will have no effect or might even hurt performance due to constant overhead of entering the parallel section.

1. The **InteractionDispatchers** part of the loop; this is the one that takes the most time and it makes sense to process each interaction completely, improving memory locality [12] and decreasing parallelization overhead. There are no cross-dependencies allowed between interactions and special **ForceContainer** was designed to assure lock-free write-access of forces acting on bodies by concurrent threads. Write-access to other areas are relatively rare (e.g. **State**) and are protected via mutexes.
2. **NewtonIntegrator** processes all bodies, again without further dependencies between them.

In addition, **ParallelEngine** is provided, which runs arbitrary subordinate engines in parallel; considering possible concurrency issues is up to the user in such case, though.

4.2.6. Dispatchers and functors

Frequently, dispatching to functors depending on exact argument types (generalization of virtual functions) is necessary; a typical example is handling collisions between different combinations of shapes:

```
InteractionGeometryDispatcher([
  Ig2_Sphere_Sphere_Dem3DofGeom(),
  Ig2_Facet_Sphere_Dem3DofGeom()
]),
```

The dispatcher will ask its functors for types they receive (**Sphere**+**Sphere** in the first case, **Facet**+**Sphere** in the second). Number of types the dispatcher decides upon is *arity* of the functor; in our case, $n = 2$. Calling the dispatcher performs the following behind scenes:[3]

1. Determine types of all relevant (n) arguments; this is implemented using regular virtual functions, which returns *class index* statically associated with each class.

2. Lookup in n-dimensional *dispatch matrix* containing (shared) functor pointers. If successful, the functor is called, being passed appropriate arguments.

   Direct lookup failure results in attempting dispatch for base classes of the argument's types recursively; if successful, functor minimizing type distance[4] is called; matrix entries are filled to return that functor for direct argument types next time immediately (caching).

   If recursive lookup fails, exception is thrown.

Although this algorithm is relatively "fast" (2 virtual function calls, functor pointer dereference, functor's virtual function call), the fact of it being performed several thousand times in each step (of which number is typically in the order of 10^5) makes it sensible to improve it further.

Our implementation finally caches functor pointers inside simulation objects, making dispatch matrix lookup necessary only at the first call for each interaction. All further calls reduce to pointer dereference and functor (virtual) call.[5]

[3] As noted in Appendix A, multivirtual functions are not part of the c++ standard. Even if it were the case, however, we want to be able to provide different functors for resolving the call, which wouldn't be (most likely) possible with regular multi-methods.

[4] Sum of distances of argument type and type the functor accepts for all dimensions of the dispatch matrix.

[5] Because this is still not optimal, there are plans to move algorithms of functors into static functions; their (non-virtual) address could be cached and called without any overhead then.

5. Introduction

5.1. Getting started

Before you start moving around in Yade, you should have some prior knowledge.

- Basics of command line in your Linux system are necessary for running yade. Look on the web for tutorials.
- Python language; we recommend the official Python tutorial. Reading further documents on the topis, such as Dive into Python will certainly not hurt either.

You are advised to try all commands described yourself. Don't be afraid to experiment.

5.1.1. Starting yade

Yade is being run primarily from terminal; the name of command is **yade**. [1] (In case you did not install from package, you might need to give specific path to the command [2]):

```
$ yade
Welcome to Yade bzr1984
TCP python prompt on localhost:9001, auth cookie `sdksuy'
TCP info provider on localhost:21000
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:
```

These initial lines give you some information about

- version (**bzr1984**); always state this version you use if you seek help in the community or report bug;
- some information for *Remote control*, which you are unlikely to need now;
- basic help for the command-line that just appeared (**Yade [1]:**).

Type **quit()**, **exit()** or simply press **^D** to quit Yade.

The command-line is ipython, python shell with enhanced interactive capabilities; it features persistent

[1] The executable name can carry a suffix, such as version number (**yade-0.20**), depending on compilation options. Packaged versions on Debian systems always provide the plain **yade** alias, by default pointing to latest stable version (or latest snapshot, if no stable version is installed). You can use **update-alternatives** to change this.

[2] In general, Unix *shell* (command line) has environment variable **PATH** defined, which determines directories searched for executable files if you give name of the file without path. Typically, $PATH contains **/usr/bin/**, **/usr/local/bin**, **/bin** and others; you can inspect your **PATH** by typing **echo $PATH** in the shell (directories are separated by **:**).

If Yade executable is not in directory contained in **PATH**, you have to specify it by hand, i.e. by typing the path in front of the filename, such as in **/home/user/bin/yade** and similar. You can also navigate to the directory itself (**cd ~/bin/yade**, where ~ is replaced by your home directory automatically) and type **./yade** then (the **.** is the current directory, so **./** specifies that the file is to be found in the current directory).

To save typing, you can add the directory where Yade is installed to your **PATH**, typically by editing **~/.profile** (in normal cases automatically executed when shell starts up) file adding line like **export PATH=/home/user/bin:$PATH**. You can also define an *alias* by saying **alias yade="/home/users/bin/yade"** in that file.

Details depend on what shell you use (bash, zsh, tcsh, ...) and you will find more information in introductory material on Linux/Unix.

history (remembers commands from your last sessions), searching and so on. See ipython's documentation for more details.

Typically, you will not type Yade commands by hand, but use *scripts*, python programs describing and running your simulations. Let us take the most simple script that will just print "Hello world!":

```
print "Hello world!"
```

Saving such script as **hello.py**, it can be given as argument to yade:

```
$ yade script.py
Welcome to Yade bzr1986
TCP python prompt on localhost:9001, auth cookie `askcsu'
TCP info provider on localhost:21000
Running script hello.py                                          ## the script is being run
Hello world!                                                     ## output from the script
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:
```

Yade will run the script and then drop to the command-line again. [3] If you want Yade to quit immediately after running the script, use the **-x** switch:

```
$ yade -x script.py
```

There is more command-line options than just **-x**, run **yade -h** to see all of them.

5.1.2. Creating simulation

To create simulation, one can either use a specialized class of type FileGenerator to create full scene, possibly receiving some parameters. Generators are written in c++ and their role is limited to well-defined scenarios. For instance, to create triaxial test scene:

```
Yade [4]: TriaxialTest(numberOfGrains=200).load()

Yade [5]: len(O.bodies)
 ->  [5]: 184
```

Generators are regular yade objects that support attribute access.

It is also possible to construct the scene by a python script; this gives much more flexibility and speed of development and is the recommended way to create simulation. Yade provides modules for streamlined body construction, import of geometries from files and reuse of common code. Since this topic is more involved, it is explained in the *User's manual*.

5.1.3. Running simulation

As explained above, the loop consists in running defined sequence of engines. Step number can be queried by **O.iter** and advancing by one step is done by **O.step()**. Every step advances *virtual time* by current timestep, **O.dt**:

```
Yade [7]: O.iter
 ->  [7]: 0

Yade [8]: O.time
 ->  [8]: 0.0
```

[3] Plain Python interpreter exits once it finishes running the script. The reason why Yade does the contrary is that most of the time script only sets up simulation and lets it run; since computation typically runs in background thread, the script is technically finished, but the computation is running.

```
Yade [9]: O.dt=1e-4

Yade [10]: O.step()

Yade [11]: O.iter
 ->  [11]: 1

Yade [12]: O.time
 ->  [12]: 0.0001
```

Normal simulations, however, are run continuously. Starting/stopping the loop is done by **O.run()** and **O.pause()**; note that **O.run()** returns control to Python and the simulation runs in background; if you want to wait for it finish, use **O.wait()**. Fixed number of steps can be run with **O.run(1000)**, **O.run(1000,True)** will run and wait. To stop at absolute step number, **O.stopAtIter** can be set and **O.run()** called normally.

```
Yade [13]: O.run()

Yade [14]: O.pause()

Yade [15]: O.iter
 ->  [15]: 1920

Yade [16]: O.run(100000,True)

Yade [17]: O.iter
 ->  [17]: 101920

Yade [18]: O.stopAtIter=500000

Yade [19]: O.wait()

Yade [20]: O.iter
 ->  [20]: 101920
```

5.1.4. Saving and loading

Simulation can be saved at any point to (optionally compressed) XML file. With some limitations, it is generally possible to load the XML later and resume the simulation as if it were not interrupted. Note that since XML is merely readable dump of Yade's internal objects, it might not (probably will not) open with different Yade version.

```
Yade [21]: O.save('/tmp/a.xml.bz2')

Yade [22]: O.reload()

Yade [24]: O.load('/tmp/another.xml.bz2')
```

The principal use of saving the simulation to XML is to use it as temporary in-memory storage for checkpoints in simulation, e.g. for reloading the initial state and running again with different parameters (think tension/compression test, where each begins from the same virgin state). The functions **O.saveTmp()** and **O.loadTmp()** can be optionally given a slot name, under which they will be found in memory:

```
Yade [25]: O.saveTmp()

Yade [26]: O.loadTmp()

Yade [27]: O.saveTmp('init') ## named memory slot
```

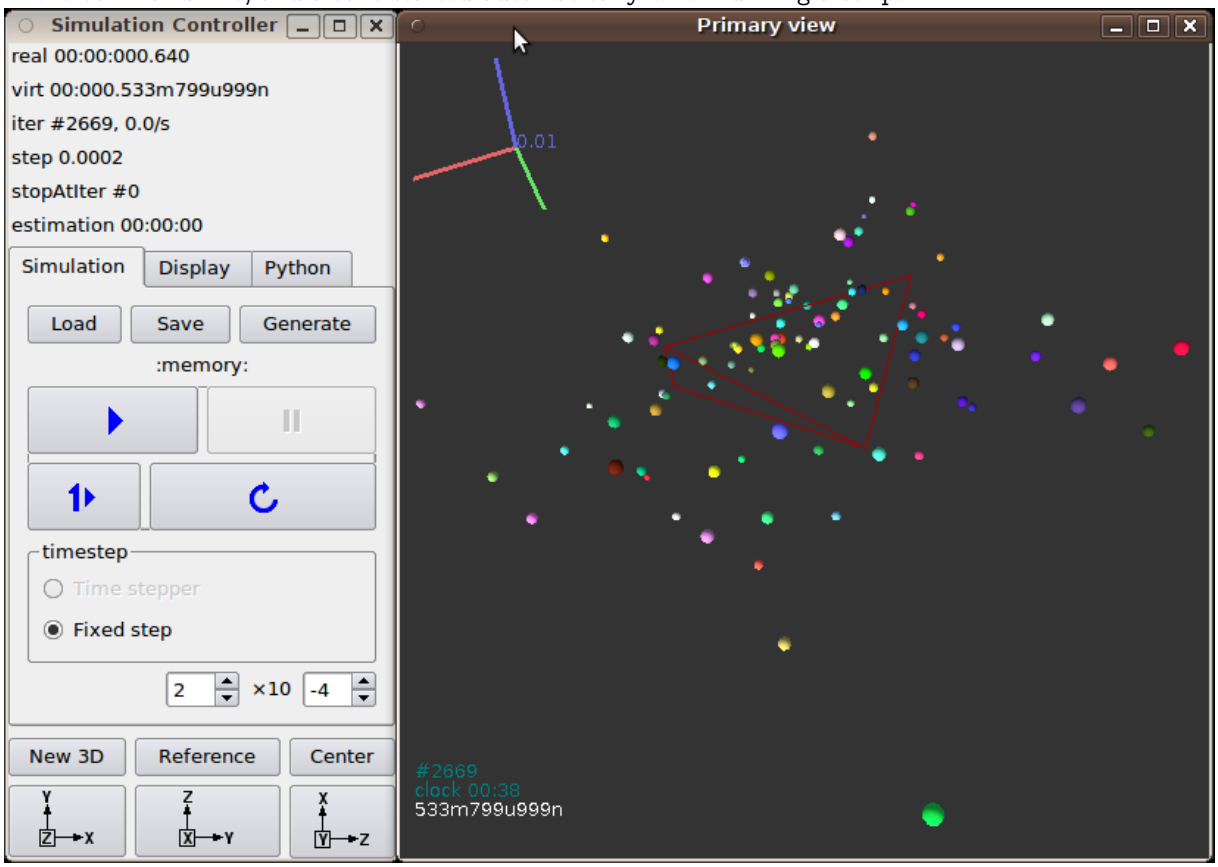
```
Yade [28]: O.loadTmp('init')
```

Simulation can be reset to empty state by **O.reset()**.

It can be sometimes useful to run different simulation, while the original one is temporarily suspended, e.g. when dynamically creating packing. **O.switchWorld()** toggles between the primary and secondary simulation.

5.1.5. Graphical interface

Yade can be optionally compiled with qt3-based graphical interface. It can be started by pressing **F12** in the command-line, and also is started automatically when running a script.



The windows with buttons is called **Controller** (can be invoked by **yade.qt.Controller()** from python):

1. The *Simulation* tab is mostly self-explanatory, and permits basic simulation control.
2. The *Display* tab has various rendering-related options, which apply to all opened views (they can be zero or more, new one is opened by the *New 3D* button).
3. The *Python* tab has only a simple text entry area; it can be useful to enter python commands while the command-line is blocked by running script, for instance.

3d views can be controlled using mouse and keyboard shortcuts; help is displayed if you press the **h** key while in the 3d view. Note that having the 3d view open can slow down running simulation significantly, it is meant only for quickly checking whether the simulation runs smoothly. Advanced post-processing is described in dedicated section.

5.2. Architecture overview

In the following, a high-level overview of Yade architecture will be given. As many of the features are directly represented in simulation scripts, which are written in Python, being familiar with this language will help you follow the examples. For the rest, this knowledge is not strictly necessary and you can ignore code examples.

5.2.1. Data and functions

To assure flexibility of software design, yade makes clear distinction of 2 families of classes: *data* components and *functional* components. The former only store data without providing functionality, while the latter define functions operating on the data. In programming, this is known as *visitor* pattern (as functional components "visit" the data, without being bound to them explicitly).

Entire simulation, i.e. both data and functions, are stored in a single **Scene** object. It is accessible through the Omega class in python (a singleton), which is by default stored in the **O** global variable:

```
Yade [32]: O.bodies       # some data components
 ->  [32]: <yade.wrapper.BodyContainer object at 0x29ed320>

Yade [33]: len(O.bodies)  # there are no bodies as of yet
 ->  [33]: 0

Yade [34]: O.engines      # functional components, empty at the moment
 ->  [34]: []
```

5.2.1.1. Data components

Bodies Yade simulation (class Scene) is represented by Bodies, their Interactions and resultant generalized forces (all stored internally in special containers).

Each Body comprises the following:

Shape represents particle's geometry (neutral with regards to its spatial orientation), such as Sphere, Facet or inifinite Wall; it usually does not change during simulation.

Material stores characteristics pertaining to mechanical behavior, such as Young's modulus or density, which are independent on particle's shape and dimensions; usually constant, might be shared amongst multiple bodies.

State contains state variable variables, in particular spatial position and orientation, linear and angular velocity, linear and angular accelerator; it is updated by the integrator at every step.

Derived classes can hold additional data, e.g. averaged damage.

Bound is used for approximate ("pass 1") contact detection; updated as necessary following body's motion. Currently, Aabb is used most often as Bound. Some bodies may have no Bound, in which case they are exempt from contact detection.

(In addition to these 4 components, bodies have several more minor data associated, such as Body::id or Body::mask.)

All these four properties can be of different types, derived from their respective base types. Yade frequently makes decisions about computation based on those types: Sphere + Sphere collision has to be treated differently than Facet + Sphere collision. Objects making those decisions are called Dispatcher's and are essential to understand Yade's functioning; they are discussed below.

Explicitly assigning all 4 properties to each particle by hand would be not practical; there are utility functions defined to create them with all necessary ingredients. For example, we can create sphere particle

using utils.sphere:

```
Yade [35]: s=utils.sphere(center=[0,0,0],radius=1)

Yade [36]: s.shape, s.state, s.mat, s.bound
 ->  [36]:
(<Sphere instance at 0x2a8f030>,
 <State instance at 0x2eb4b90>,
 <FrictMat instance at 0x24500e0>,
 <Aabb instance at 0x2ce9590>)

Yade [37]: s.state.pos
 ->  [37]: Vector3(0,0,0)

Yade [38]: s.shape.radius
 ->  [38]: 1.0
```

We see that a sphere with material of type FrictMat (default, unless you provide another Material) and bounding volume of type Aabb (axis-aligned bounding box) was created. Its position is at origin and its radius is 1.0. Finally, this object can be inserted into the simulation; and we can insert yet one sphere as well.

```
Yade [39]: O.bodies.append(s)
 ->  [39]: 0

Yade [40]: O.bodies.append(utils.sphere([0,0,2],.5))
 ->  [40]: 1
```

In each case, return value is Body.id of the body inserted.

Since till now the simulation was empty, its id is 0 for the first sphere and 1 for the second one. Saving the id value is not necessary, unless you want access this particular body later; it is remembered internally in Body itself. You can address bodies by their id:

```
Yade [41]: O.bodies[1].state.pos
 ->  [41]: Vector3(0,0,2)

Yade [42]: O.bodies[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/home/vaclav/ydoc/<ipython console> in <module>()

IndexError: Body id out of range.
```

Adding the same body twice is, for reasons of the id uniqueness, not allowed:

```
Yade [43]: O.bodies.append(s)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/home/vaclav/ydoc/<ipython console> in <module>()

IndexError: Body already has id 0 set; appending such body (for the second time) is not allowed.
```

Bodies can be iterated over using standard python iteration syntax:

```
Yade [44]: for b in O.bodies:
     ....:    print b.id,b.shape.radius
     ....:
0 1.0
1 0.5
```

Interactions Interactions are always between pair of bodies; usually, they are created by the collider based on spatial proximity; they can, however, be created explicitly and exist independently of distance. Each interaction has 2 components:

InteractionGeometry holding geometrical configuration of the two particles in collision; it is updated automatically as the particles in question move and can be queried for various geometrical characteristics, such as penetration distance or shear strain.

Based on combination of types of Shapes of the particles, there might be different storage requirements; for that reason, a number of derived classes exists, e.g. for representing geometry of contact between Sphere+Sphere, Facet+Sphere etc.

InteractionPhysics representing non-geometrical features of the interaction; some are computed from Materials of the particles in contact using some averaging algorithm (such as contact stiffness from Young's moduli of particles), others might be internal variables like damage.

Suppose now interactions have been already created. We can access them by the id pair:

```
Yade [48]: O.interactions[0,1]
 ->  [48]: <Interaction instance at 0x267f4f0>

Yade [49]: O.interactions[1,0]     # order of ids is not important
 ->  [49]: <Interaction instance at 0x267f4f0>

Yade [50]: i=O.interactions[0,1]

Yade [51]: i.id1,i.id2
 ->  [51]: (0, 1)

Yade [52]: i.geom
 ->  [52]: <Dem3DofGeom_SphereSphere instance at 0x2f786b0>

Yade [53]: i.phys
 ->  [53]: <FrictPhys instance at 0x2666f10>

Yade [54]: O.interactions[100,10111]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/home/vaclav/ydoc/<ipython console> in <module>()

IndexError: No such interaction
```

Generalized forces Generalized forces include force, torque and forced displacement and rotation; they are stored only temporarliy, during one computation step, and reset to zero afterwards. For reasons of parallel computation, they work as accumulators, i.e. only can be added to, read and reset.

```
Yade [55]: O.forces.f(0)
 ->  [55]: Vector3(0,0,0)

Yade [56]: O.forces.addF(0,Vector3(1,2,3))

Yade [57]: O.forces.f(0)
 ->  [57]: Vector3(1,2,3)
```

You will only rarely modify forces from Python; it is usually done in c++ code and relevant documentation can be found in the Programmer's manual.

5.2.1.2. Function components

In a typical DEM simulation, the following sequence is run repeatedly:

- reset forces on bodies from previous step
- approximate collision detection (pass 1)
- detect exact collisions of bodies, update interactions as necessary
- solve interactions, applying forces on bodies
- apply other external conditions (gravity, for instance).
- change position of bodies based on forces, by integrating motion equations.

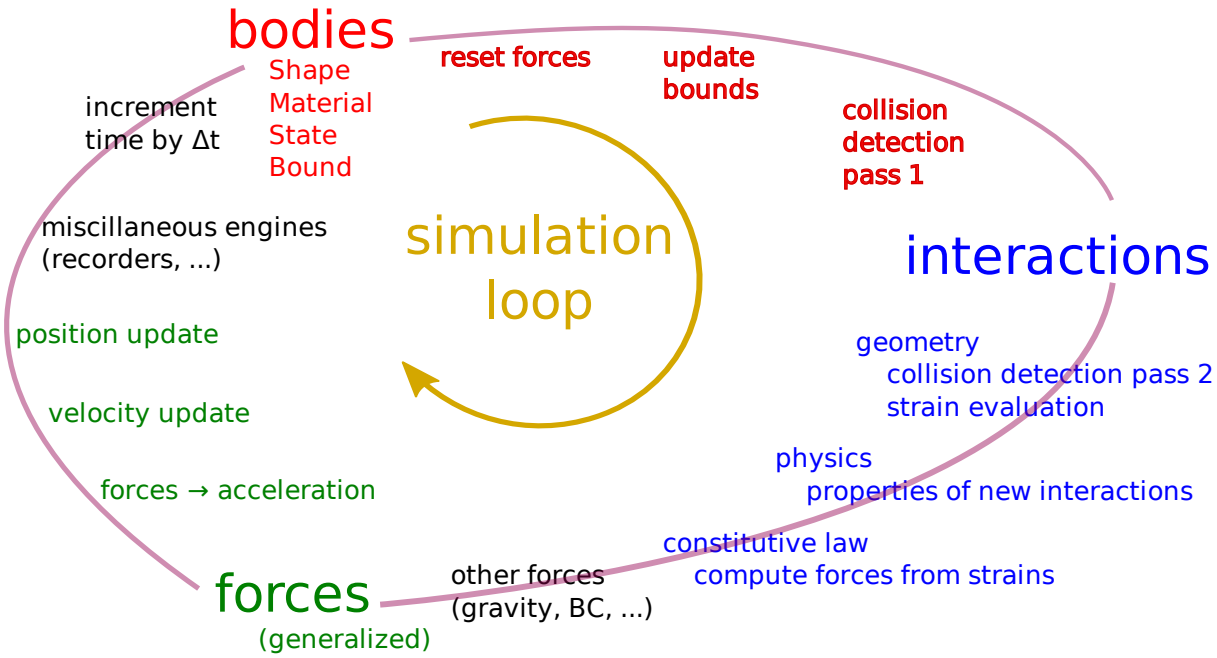


Figure 5.1.: Typical simulation loop; each step begins at body-cented bit at 11 o'clock, continues with interaction bit, force application bit, miscillanea and ends with time update.

Each of these actions is represented by an Engine, functional element of simulation. The sequence of engines is called *simulation loop*.

Engines Simulation loop, shown at img. img-yade-iter-loop, can be described as follows in Python (details will be explained later); each of the **O.engine** items is instance of a type deriving from Engine:

```
O.engines=[
        # reset forces
        ForceResetter(),
        # approximate collision detection, create interactions
        BoundDispatcher([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
        InsertionSortCollider(),
        # handle interactions
        InteractionDispatchers(
                [Ig2_Sphere_Sphere_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_Dem3Dof_Elastic_Elastic()],
        ),
        # apply other conditions
        GravityEngine(gravity=(0,0,-9.81)),
        # update positions using Newton's equations
        NewtonIntegrator()
]
```

There are 3 fundamental types of Engines:

GlobalEngines operating on the whole simulation (e.g. GravityEngine looping over all bodies and applying force based on their mass)

PartialEngine operating only on some pre-selected bodies (e.g. ForceEngine applying constant force to some bodies)

Dispatchers do not perform any computation themselves; they merely call other functions, represented by function objects, Functors. Each functor is specialized, able to handle certain object types, and will be dispatched if such obejct is treated by the dispatcher.

Dispatchers and functors For approximate collision detection (pass 1), we want to compute bounds for all bodies in the simulation; suppose we want bound of type axis-aligned bounding box. Since the exact algorithm is different depending on particular shape, we need to provide functors for handling all specific cases. The line:

```
BoundDispatcher([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()])
```

creates a BoundDispatcher. It traverses all bodies and will, based on shape type of each body, dispatch one of the functors to create/update bound for that particular body. In the case shown, it has 2 functors, one handling spheres, another facets.

The name is composed from several parts: **Bo** (functor creating Bound), which accepts **1** type Sphere and creates an Aabb (axis-aligned bounding box; it is derived from Bound). The Aabb objects are used by InsertionSortCollider, which does the actual approximate collision detection. All **Bo1** functors derive from BoundFunctor.

The next part, reading

```
InteractionDispatchers(
        [Ig2_Sphere_Sphere_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_Dem3Dof_Elastic_Elastic()],
),
```

hides 3 internal dispatchers within the InteractionDispatchers engine; they all operate on interactions and are, for performance reasons, put together:

InteractionGeometryDispatcher uses the first set of functors (**Ig2**), which are dispatched based on combination of **2** Shapes objects. Dispatched functor resolves exact collision configuration and creates InteractionGeometry (whence **Ig** in the name) associated with the interaction, if there is collision. The functor might as well fail on approximate interactions, indicating there is no real contact between the bodies, even if they did overlap in the approximate collision detection.

1. The first functor, Ig2_Sphere_Sphere_Dem3DofGeom, is called on interaction of 2 Spheres and creates Dem3DofGeom instance, if appropriate.

2. The second functor, Ig2_Facet_Sphere_Dem3DofGeom, is called for interaction of Facet with Sphere and might create (again) a Dem3DofGeom instance.

All **Ig2** functors derive from InteractionGeometryFunctor (they are documented at the same place).

InteractionPhysicsDispatcher dispatches to the second set of functors based on combination of **2** Materials; these functors return return InteractionPhysics instance (the **Ip** prefix). In our case, there is only 1 functor used, Ip2_FrictMat_FrictMat_FrictPhys, which create FrictPhys from 2 FrictMat's.

Ip2 functors are derived from InteractionPhysicsFunctor.

LawDispatcher dispatches to the third set of functors, based on combinations of InteractionGeometry and InteractionPhysics (wherefore **2** in their name again) of each particular interaction, created by preceding functors. The **Law2** functors represent "constitutive law"; they resolve the interaction by computing forces on the interacting bodies (repulsion, attraction, shear forces, ...) or otherwise update interaction state variables.

Law2 functors all inherit from LawFunctor.

There is chain of types produced by earlier functors and accepted by later ones; the user is responsible to satisfy type requirement (see img. img-dispatch-loop). An exception (with explanation) is raised in the contrary case.

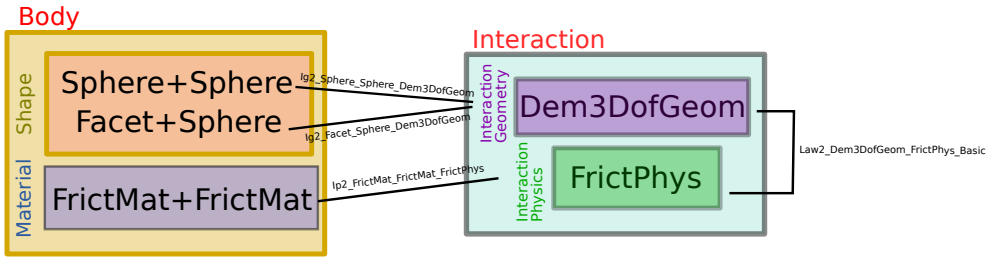


Figure 5.2.: Chain of functors producing and accepting certain types. In the case shown, the **Ig2** functors produce Dem3DofGeom instances from all handled Shape combinations; the **Ig2** functor produces FrictMat. The constitutive law functor **Law2** accepts the combination of types produced. Note that the types are stated in the functor's class names.

6. User's manual

6.1. Scene construction

6.1.1. Triangulated surfaces

Yade integrates with the the GNU Triangulated Surface library, exposed in python via th 3rd party gts module. GTS provides variety of functions for surface manipuation (coarsening, tesselation, simlification, import), to be found in its documentation.

GTS surfaces are geometrical objects, which can be inserted into simulation as set of particles whose Body.shape is of type Facet – single triangulation elements. pack.gtsSurface2Facets can be used to convert GTS surface triangulation into list of bodies ready to be inserted into simulation via **O.bodies.append**.

Facet particles are created by default as non-Body.dynamic (they have zero inertial mass). That means that they are fixed in space and will not move if subject to forces. You can however

- prescribe arbitrary movement to facets using a PartialEngine (such as TranslationEngine or RotatioNEngine);
- assign explicitly mass and inertia to that particle;
- make that particle part of a clump and assign mass and inertia of the clump itself (described below).

Note: Facets can only (currently) interact with spheres, not with other facets, even if they are *dynamic*. Collision of 2 facets will not create interaction, therefore no forces on facets.

6.1.1.1. Import

Yade currently offers 3 formats for importing triangulated surfaces from external files, in the ymport module:

ymport.gts text file in native GTS format.

ymport.stl STereoLitography format, in either text or binary form; exported from Blender, but from many CAD systems as well.

ymport.gmsh. text file in native format for GMSH, popular open-source meshing program.

If you need to manipulate surfaces before creating list of facets, you can study the py/ymport.py file where the import functions are defined. They are rather simple in most cases.

6.1.1.2. Parametric construction

The gts module provides convenient way of creating surface by vertices, edges and triangles.

Frequently, though, the surface can be conveniently described as surface between polylines in space. For instance, cylinder is surface between two polygons (closed polylines). The pack.sweptPolylines2gtsSurface offers the functionality of connecting several polylines with triangulation.

Note: The implementation of pack.sweptPolylines2gtsSurface is rather simplistic: all polylines must be of the same length, and they are connected with triangles between points following their indices within each polyline (not by distance). On the other hand, points can be co-incident, if the **threshold** parameter is positive: degenerate triangles with vertices closer that **threshold** are automatically elimintated.

Manipulating lists efficiently (in terms of code length) requires being familiar with list comprehensions in python.

Another examples can be found in examples/mill.py (fully parametrized) or examples/funnel.py (with hardcoded numbers).

6.1.2. Sphere packings

Representing a solid of an arbitrary shape by arragment of spheres presents the problem of sphere packing, i.e. spatial arrangment of sphere such that given solid is approximately filled with them. For the purposes of DEM simulation, there can be several requirements.

1. Distribution of spheres' radii. Arbitrary volume can be filled completely with spheres provided there are no restrictions on their radius; in such case, number of spheres can be infinite and their radii approach zero. Since both number of particles and minimum sphere radius (via critical timestep) determine computation cost, radius distribution has to be given mandatorily. The most typical distribution is uniform: mean±dispersion; if dispersion is zero, all spheres will have the same radius.
2. Smooth boundary. Some algorithms treat boundaries in such way that spheres are aligned on them, making them smoother as surface.
3. Packing density, or the ratio of spheres volume and solid size. It is closely related to radius distribution.
4. Coordination number, (average) number of contacts per sphere.
5. Isotropy (related to regularity/irregularity); packings with preferred directions are usually not desirable, unless the modeled solid also has such preference.
6. Permissible Spheres' overlap; some algorithms might create packing where spheres slightly overlap; since overlap usually causes forces in DEM, overlap-free packings are sometimes called "stress-free".

6.1.2.1. Volume representation

There are 2 methods for representing exact volume of the solid in question in Yade: boundary representation and constructive solid geometry. Despite their fundamental differences, they are abstracted in Yade in the Predicate class. Predicate provides the following functionality:

1. defines axis-aligned bounding box for the associated solid (optionally defines oriented bounding box);
2. can decide whether given point is inside or outside the solid; most predicates can also (exactly or approximately) tell whether the point is inside *and* satisfies some given padding distance from the represented solid boundary (so that sphere of that volume doesn't stick out of the solid).

Constructive Solid Geometry (CSG) CSG approach describes volume by geometric *primitives* or primitive solids (sphere, cylinder, box, cone, ...) and boolean operations on them. Primitives defined in Yade include inCylinder, inSphere, inEllipsoid, inHyperboloid, notInNotch.

For instance, hyperboloid (dogbone) specimen for tension-compression test can be constructed in this way (shown at img. img-hyperboloid):

```
from yade import pack

## construct the predicate first
pred=pack.inHyperboloid(centerBottom=(0,0,-.1),centerTop=(0,0,.1),radius=.05,skirt=.03)
## alternatively: pack.inHyperboloid((0,0,-.1),(0,0,.1),.05,.03)

## pack the predicate with spheres (will be explained later)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=3.5e-3)

## add spheres to simulation
O.bodies.append(spheres)
```

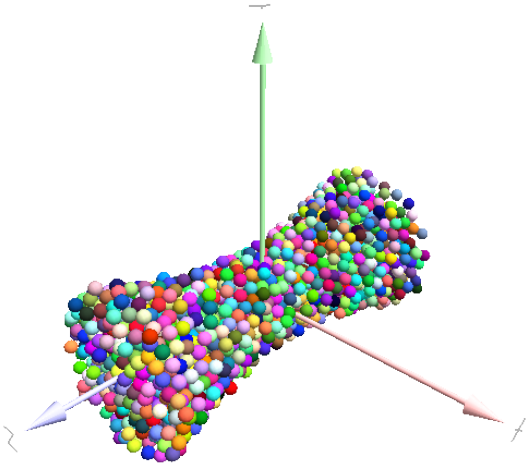


Figure 6.1.: Specimen constructed with the pack.inHyperboloid predicate, packed with pack.randomDensePack.

Boundary representation (BREP) Representing a solied by its boundary is much more flexible than CSG volumes, but is mostly only approximate. Yade interfaces to GNU Triangulated Surface Library (GTS) to import surfaces readable by GTS, but also to construct them explicitly from within simulation scripts. This makes possible parametric construction of rather complicated shapes; there are functions to create set of 3d polylines from 2d polyline (pack.revolutionSurfaceMeridians), to triangulate surface between such set of 3d polylines (pack.sweptPolylines2gtsSurface).

For example, we can construct a simple funnel (examples/funnel.py, shown at img-funnel):

```
from numpy import linspace
from yade import pack

# angles for points on circles
thetas=linspace(0,2*pi,num=16,endpoint=True)

# creates list of polylines in 3d from list of 2d projections
# turned from 0 to π
meridians=pack.revolutionSurfaceMeridians(
        [[(3+rad*sin(th),10*rad+rad*cos(th)) for th in thetas] for rad in linspace(1,2,num=10)],
        linspace(0,pi,num=10)
)

# create surface
surf=pack.sweptPolylines2gtsSurface(
        meridians+
        +[[Vector3(5*sin(-th),-10+5*cos(-th),30) for th in thetas]]  # add funnel top
)
```

```
# add to simulation
O.bodies.append(pack.gtsSurface2Facets(surf))
```

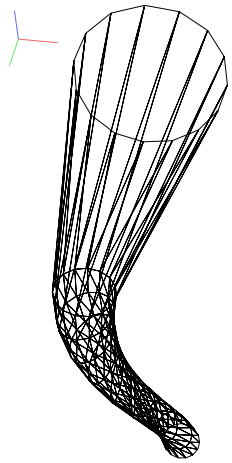


Figure 6.2.: Triangulated funnel, constructed with the examples/funnel.py script.

GTS surface objects can be used for 2 things:

1. pack.gtsSurface2Facets function can create the triangulated surface (from Facet particles) in the simulation itself, as shown in the funnel example. (Triangulated surface can also be imported directly from a STL file using ymport.stl.)
2. pack.inGtsSurface predicate can be created, using the surface as boundary representation of the enclosed volume.

The scripts/test/gts-horse.py (img. img-horse) shows both possibilities; first, a GTS surface is imported:

```
import gts
surf=gts.read(open('horse.coarse.gts'))
```

That surface object is used as predicate for packing:

```
pred=pack.inGtsSurface(surf)
O.bodies.append(pack.regularHexa(pred,radius=radius,gap=radius/4.))
```

and then, after being translated, as base for triangulated surface in the simulation itself:

```
surf.translate(0,0,-(aabb[1][2]-aabb[0][2]))
O.bodies.append(pack.gtsSurface2Facets(surf,wire=True))
```

Boolean operations on predicates Boolean operations on pair of predicates (noted **A** and **B**) are defined:

- intersection **A & B** (conjunction): point must be in both predicates involved.
- union **A | B** (disjunction): point must be in both predicates involved.
- difference **A - B** (conjunction with second predicate negated): the point must be in the first predicate and not in the second one.
- symmetric difference **A ^ B** (exclusive disjunction): point must be in exactly one of the two predicates.

Composed predicates also properly define their bounding box. For example, we can take box and remove cylinder from inside, using the **A - B** operation (img. img-predicate-difference):

Figure 6.3.: Imported GTS surface (horse) used as packing predicate (top) and surface constructed from facets (bottom). See http://www.youtube.com/watch?v=PZVruIlUX1A for movie of this simulation.

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4)
```

6.1.2.2. Packing algorithms

Algorithms presented below operate on geometric spheres, defined by their center and radius. With a few exception documented below, the procedure is as follows:

1. Sphere positions and radii are computed (some functions use volume predicate for this, some do not)
2. utils.sphere is called for each position and radius computed; it receives extra keyword arguments of the packing function (i.e. arguments that the packing function doesn't specify in its definition; they are noted ****kw**). Each utils.sphere call 0creates actual Body objects with Sphere shape. List of Body objects is returned.
3. List returned from the packing function can be added to simulation using **O.bodies.append**.

Taking the example of pierced box:

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4,wire=True,color=(0,0,1),material=1)
```

Keyword arguments **wire**, **color** and **material** are not declared in pack.randomDensePack, therefore will be passed to utils.sphere, where they are also documented. **spheres** is now list of Body objects, which we add to the simulation:

```
O.bodies.append(spheres)
```

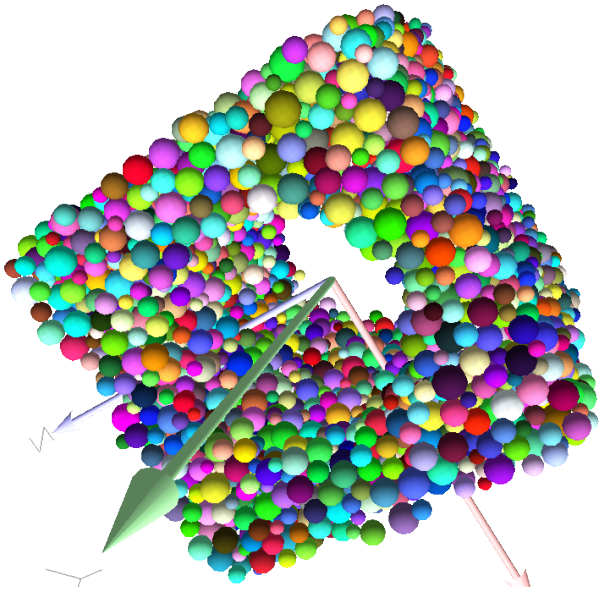


Figure 6.4.: Box with cylinder removed from inside, using difference of these two predicates.

Packing algorithms described below produce dense packings. If one needs loose packing, pack.SpherePack class provides functions for generating loose packing, via its pack.SpherePack.makeCloud method. It is used internally for generating initial configuration in dynamic algorithms. For instance:

```
from yade import pack
sp=pack.SpherePack()
sp.makeCloud(minCorner=(0,0,0),maxCorner=(3,3,3),rMean=.2,rRelFuzz=.5)
```

will fill given box with spheres, until no more spheres can be placed. The object can be used to add spheres to simulation:

```
for c,r in sp: O.bodies.append(utils.sphere(c,r))
```

or, in a more pythonic way, with one single **O.bodies.append** call:

```
O.bodies.append([utils.sphere(c,r) for c,r in sp])
```

Geometric Geometric algorithms compute packing without performing dynamic simulation; amogh their advantages are

- speed;
- spheres touch exactly, there are no overlaps (what some people call "stress-free" packing);

their chief disadvantage is that radius distribution cannot be prescribed exactly, save in specific cases (regular packings); sphere radii are given by the algorithm, which already makes the system determined. If exact radius distribuition is important for your problem, consider dynamic algorithms instead.

Regular Yade defines packing generators for spheres with constant radii, which can be used with volume predicates as described above. They are dense orthogonal packing (pack.regularOrtho) and dense hexagonal packing (pack.regularHexa). The latter creates so-called "hexagonal close packing", which achieves maximum density (http://en.wikipedia.org/wiki/Close-packing_of_spheres).

Clear disadvantage of regular packings is that they have very strong directional preferences, which might not be an issue in some cases.

Irregular Random geometric algorithms do not integrate at all with volume predicates described above; rather, they take their own boundary/volume definition, which is used during sphere positioning. On the other hand, this makes it possible for them to respect boundary in the sense of making spheres touch it at appropriate places, rather than leaving empty space in-between.

SpherePadder constructs dense sphere packing based on pre-computed tetrahedron mesh; it is documented in SpherePadder documentation; sample script is in scripts/test/SpherePadder.py. SpherePadder does not return Body list as other algorithms, but a pack.SpherePack object; it can be iterated over, adding spheres to the simulation, as shown in its documentation.

GenGeo is library (python module) for packing generation developed with ESyS-Particle. It creates packing by random insertion of spheres with given radius range. Inserted spheres touch each other exactly and, more importantly, they also touch the boundary, if in its neighbourhood. Boundary is represented as special object of the GenGeo library (Sphere, cylinder, box, convex polyhedron, ...). Therefore, GenGeo cannot be used with volume represented by yade predicates as explained above.

Packings generated by this module can be imported directly via ymport.gengeo, or from saved file via ymport.gengeoFile. There is an example script scripts/test/genCylLSM.py. Full documentation for GenGeo can be found at ESyS documentation website.

To our knowledge, the GenGeo library is not currently packaged. It can be downloaded from current subversion repository

```
svn checkout https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo
```

then following instruction in the **INSTALL** file.

Dynamic The most versatile algorithm for random dense packing is provided by pack.randomDensePack. Initial loose packing of non-overlapping spheres is generated by randomly placing placing them in cuboid volume, with radii given by requested (currently only uniform) radius distribution. When no more spheres can be inserted, the packing is compressed, then uncompressed (see py/pack.py for exact values of these "stresses"), byt running a DEM simulation; Omega.switchWorld is used to not affect existing simulation). Finally, resulting packing is clipped using provided predicate, as explained above.

By its nature, this method might take relatively long; and there are 2 provisions to make the computation time shorter:

- If number of spheres using the **spheresInCell** parameter is specified, only smaller specimen with *periodic* boundary is created and then repeated as to fill the predicate. This can provide high-quality packing with low regularity, depending on the **spheresInCell** parameter (value of several thousands is recommended).
- Providing **memoizeDb** parameter will make pack.randomDensePack first look into provided file (SQLite database) for packings with similar parameters. On success, the packing is simply read from database and returned. If there is no similar pre-existent packing, normal procedure is run, and the result is saved in the database before being returned, so that subsequent calls with same parameters will return quickly.

If you need to obtain full periodic packing (rather than packing clipped by predicate), you can use pack.randomPeriPack.

In case of specific needs, you can create packing yourself, "by hand". For instance, packing boundary can be constructed from facets, letting randomly positioned spheres in space fall down under gravity.

6.1.3. Adding particles

The BodyContainer holds Body objects in the simulation; it is accessible as **O.bodies**.

6.1.3.1. Creating Body objects

Body objects are only rarely constructed by hand by their components (Shape, Bound, State, Material); instead, convenience functions utils.sphere, utils.facet and utils.wall are used to create them. Using these functions also ensures better future compatibility, if internals of Body change in some way. These functions receive geometry of the particle and several other characteristics. See their documentation for details. If the same Material is used for several (or many) bodies, it can be shared by adding it in **O.materials**, as explained below.

6.1.3.2. Defining materials

The **O.materials** object (instance of Omega.materials) holds defined shared materials for bodies. It only supports addition, and will typically hold only a few instance (though there is no limit).

label given to each material is optional, but can be passed to utils.sphere and other functions forconstructing body. The value returned by **O.materials.append** is and **id** of the material, which can be also passed to utils.sphere – it is a little bit faster than using label, though not noticeable for small number of particles and perhaps less convenient.

If no Material is specified when calling utils.sphere, the *last* defined material is used; that is a convenient default. If no material is defined yet (hence there is no last material), a default material will be created using utils.defaultMaterial; this should not happen for serious simulations, but is handy in simple scripts, where exact material properties are more or less irrelevant.

```
Yade [124]: len(O.materials)
 ->  [124]: 0

Yade [125]: idConcrete=O.materials.append(FrictMat(young=30e9,poisson=.2,frictionAngle=.6,label="concrete"))

Yade [126]: O.materials[idConcrete]
 ->  [126]: <FrictMat instance at 0x2d09140>

# uses the last defined material
Yade [128]: O.bodies.append(utils.sphere(center=(0,0,0),radius=1))
 ->  [128]: 0

# material given by id
Yade [130]: O.bodies.append(utils.sphere((0,0,2),1,material=idConcrete))
 ->  [130]: 1

# material given by label
Yade [132]: O.bodies.append(utils.sphere((0,2,0),1,material="concrete"))
 ->  [132]: 2

Yade [133]: idSteel=O.materials.append(FrictMat(young=210e9,poisson=.25,frictionAngle=.8,label="steel"))

Yade [134]: len(O.materials)
 ->  [134]: 2

# implicitly uses "steel" material, as it is the last one now
Yade [136]: O.bodies.append(utils.facet([(1,0,0),(0,1,0),(-1,-1,0)]))
 ->  [136]: 3
```

6.1.3.3. Adding multiple particles

As shown above, bodie are added one by one or several at the same time using the **append** method:

```
Yade [138]: O.bodies.append(utils.sphere((0,0,0),1))
 ->  [138]: 0
```

```
Yade [139]: O.bodies.append(utils.sphere((0,0,2),1))
 ->  [139]: 1

# this is the same, but in one function call
Yade [141]: O.bodies.append([
     .....:    utils.sphere((0,0,0),1),
     .....:    utils.sphere((0,0,2),1)
     .....: ])
 ->  [144]: [2, 3]
```

Many functions introduced in preceding sections return list of bodies which can be readily added to the simulation, including

- packing generators, such as pack.randomDensePack, pack.regularHexa
- surface function pack.gtsSurface2Facets
- import functions ymport.gmsh, ymport.stl, ...

As those functions use utils.sphere and utils.facet internally, they accept additional argument passed to those function. In particular, material for each body is selected following the rules above (last one if not specified, by label, by index, etc.).

6.1.3.4. Clumping particles together

In some cases, you might want to create rigid aggregate of individual particles (i.e. particles will retain their mutual position during simulation); a special function BodyContainer.appendClumped is designed for this task; for instance, we might add 2 spheres tied together:

```
Yade [146]: O.bodies.appendClumped([
     .....:    utils.sphere([0,0,0],1),
     .....:    utils.sphere([0,0,2],1)
     .....: ])
 ->  [149]: (2, [0, 1])

Yade [150]: len(O.bodies)
 ->  [150]: 3

Yade [151]: O.bodies[1].isClumpMember, O.bodies[2].clumpId
 ->  [151]: (True, 2)

Yade [152]: O.bodies[2].isClump, O.bodies[2].clumpId
 ->  [152]: (True, 2)
```

appendClumped returns a tuple of **(clumpId,[memberId1,memberId2])**: clump is internally represented by a special Body, referenced by clumpId of its members (see also isClump, isClumpMember and isStandalone).

6.1.4. Creating interactions

In typical cases, interactions are created during simulations as particles collide. This is done by a Collider detecting approximate contact between particles and then an InteractionGeometryFunctor detecting exact collision.

Some material models (such as the concrete model) rely on initial interaction network which is denser than geometrical contact of spheres: sphere's radii as "enlarged" by a dimensionless factor called *interaction radius* to create this initial network. This is done typically in this way (see examples/concrete/uniax.py for an example):

1. Approximate collision detection is adjusted so that approximate contacts are detected also between particles within the interaction radius are detected. This consists in setting value of Bo1_Sphere_Aabb.aabbEnlargeFactor to the interaction radius value.

2. The geometry functor (**Ig2**) would normally say that "there is no contact" if given 2 spheres that are not in contact. Therefore, the same value as for Bo1_Sphere_Aabb.aabbEnlargeFactor must be given to it. (Either Ig2_Sphere_Sphere_Dem3DofGeom.distFactor or Ig2_Sphere_Sphere_ScGeom.interactionDetectionFactor, depending on the functor that is in use.

   Note that only Sphere + Sphere interactions are supported; there is no parameter analogous to distFactor in Ig2_Facet_Sphere_Dem3DofGeom. This is on purpose, since the interaction radius is meaningful in bulk material represented by sphere packing, whereas facets usually represent boundary conditions which should be exempt from this dense interaction network.

3. Run one single step of the simulation so that the initial network is created.

4. Reset interaction radius in both **Bo1** and **Ig2** functors to their default value again.

5. Continue the simulation; interactions that are already established will not be deleted (the **Law2** functor in usepermitting).

In code, such scenario might look similar to this one (labeling is explained in *Labeling things*):

```
intRadius=1.5

O.engines=[
   ForceResetter(),
   BoundDispatcher([
      # enlarge here
      Bo1_Sphere_Aabb(aabbEnlargeFactor=intRadius,label='bo1s'),
      Bo1_Facet_Aabb(),
   ]),
   InsertionSortCollider(),
   InteractionDispatchers(
      [
         # enlarge here
         Ig2_Sphere_Sphere_Dem3DofGeom(distFactor=intRadius,label='ig2ss'),
         Ig2_Facet_Sphere_Dem3DofGeom(),
      ],
      [Ip2_CpmMat_CpmMat_CpmPhys()],
      [Law2_Dem3DofGeom_CpmPhys_Cpm(epsSoft=0)], # deactivated
   ),
   NewtonIntegrator(damping=damping,label='damper'),
]

# run one single step
O.step()

# reset interaction radius to the default value
# see documentation of those attributes for the meaning of negative values
bo1s.aabbEnlargeFactor=-1
ig2ss.distFactor=-1

# now continue simulation
O.run()
```

6.1.4.1. Individual interactions on demand

It is possible to create an interaction between a pair of particles independently of collision detection using utils.createInteraction. This function looks for and uses matching **Ig2** and **Ip2** functors. Interaction will be created regardless of distance between given particles (by passing a special parameter to the **Ig2** functor to force creation of the interaction even without any geometrical contact). Appropriate constitutive law

should be used to avoid deletion of the interaction at the next simulation step.

```
Yade [154]: O.materials.append(FrictMat(young=3e10,poisson=.2,density=1000))
 ->  [154]: 0

Yade [155]: O.bodies.append([
     .....:    utils.sphere([0,0,0],1),
     .....:    utils.sphere([0,0,1000],1)
     .....: ])
 ->  [158]: [0, 1]

# only add InteractionDispatchers, no other engines are needed now
Yade [159]: O.engines=[
     .....:    InteractionDispatchers(
     .....:        [Ig2_Sphere_Sphere_Dem3DofGeom(),],
     .....:        [Ip2_FrictMat_FrictMat_FrictPhys()],
     .....:        [] # not needed now
     .....:    )
     .....: ]

Yade [166]: i=utils.createInteraction(0,1)

# created by functors in InteractionDispatchers
Yade [167]: i.geom, i.phys
 ->  [167]:
(<Dem3DofGeom_SphereSphere instance at 0x2699190>,
 <FrictPhys instance at 0x3a69160>)
```

This method will be rather slow if many interaction are to be created (the functor lookup will be repeated for each of them). In such case, ask on yade-dev@lists.launchpad.net to have the utils.createInteraction function accept list of pairs id's as well.

6.1.5. Base engines

A typical DEM simulation in Yade does at least the following at each step (see *Function components* for details):

1. Reset forces from previous step
2. Detect new collisions
3. Handle interactions
4. Apply forces and update positions of particles

Each of these points corresponds to one or several engines:

```
O.engines=[
   ForceResetter(),          # reset forces
   BoundDispatcher(),        # update bounding boxes, for use by the next engine
   InsertionSortCollider(),  # detect new collisions
   InteractionDispatchers([...],[...],[...]) # handle interactions
   NewtonIntegrator()        # apply forces and update positions
]
```

The order of engines is important. In majority of cases, you will put any additional engine after InteractionDispatchers:

- if it apply force, it should come before NewtonIntegrator, otherwise the for will never be effective.
- if it makes use of bodies' positions, it should also come before NewtonIntegrator, otherwise, positions at the next step will be used (this might not be critical in many cases, such as output for visualization

with VTKRecorder).

The O.engines sequence must be always assigned at once (the reason is in the fact that although engines themselves are passed by reference, the sequence is *copied* from c++ to Python or from Python to c++). This includes modifying an existing **O.engines**; therefore

```
O.engines.append(SomeEngine()) # wrong
```

will not work;

```
O.engines=O.engines+[SomeEngine()] # ok
```

must be used instead. For inserting an engine after position #2 (for example), use python slice notation:

```
O.engines=O.engines[:2]+[SomeEngine()]+O.engines[2:]
```

6.1.5.1. Functors choice

In the above example, we omited functors, only writing ellipses **...** instead. As explained in *Dispatchers and functors*, there are 4 kinds of functors and associated dispatchers. User can choose which ones to use, though the choice must be consistent.

Bo1 functors **Bo1** functors must be chosen depending on the collider in use.

At this moment (May 2010), the most common choice is InsertionSortCollider, which uses Aabb; functors creating Aabb must be used in that case. Depending on particle shapes in your simulation, choose appropriate functors:

```
O.engines=[...,
   BoundDispatcher([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
   InsertionSortCollider(),
   ...
]
```

Using more functors than necessary (such as Bo1_Facet_Aabb if there are no facets in the simulation) has no performance penalty. On the other hand, missing functors for existing shapes will cause those bodies to not collider with other bodies (they will freely interpenetrate).

There are other colliders as well, though their usage is only experimental:

- SpatialQuickSortCollider is correctness-reference collider operating on Aabb; it is significantly slower than InsertionSortCollider.
- PersistentTriangulationCollider only works on spheres; it does not use a BoundDispatcher, as it operates on spheres directly.
- FlatGridCollider is proof-of-concept grid-based collider, which computes grid positions internally (no BoundDispatcher either)

Ig2 functors **Ig2** functor choice (all of the derive from InteractionGeometryFunctor) depends on

1. shape combinations that should collide; for instance:

   ```
   InteractionDispatchers([Ig2_Sphere_Sphere_Dem3DofGeom()],[],[])
   ```

   will handle collisions for Sphere + Sphere, but not for Facet + Sphere – if that is desired, an additional functor must be used:

```
InteractionDispatchers([
   Ig2_Sphere_Sphere_Dem3DofGeom(),
   Ig2_Facet_Sphere_Dem3DofGeom()
],[],[])
```

Again, missing combination will cause given shape combinations to freely interpenetrate one another.

2. InteractionGeometry type accepted by the **Law2** functor (below); it is the first part of functor's name after **Law2** (for instance, Law2_Dem3DofGeom_CpmPhys_Cpm accepts Dem3DofGeom). This is (for most cases) either Dem3DofGeom (total shear formulation) or ScGeom (incremental shear formulation). For ScGeom, the above example would simply change to:

```
InteractionDispatchers([
   Ig2_Sphere_Sphere_ScGeom(),
   Ig2_Facet_Sphere_ScGeom()
],[],[])
```

Ip2 functors **Ip2** functors (deriving from InteractionPhysicsFunctor) must be chosen depending on

1. Material combinations within the simulation. In most cases, **Ip2** functors handle 2 instances of the same Material class (such as Ip2_FrictMat_FrictMat_FrictPhys for 2 bodies with FrictMat)
2. InteractionPhysics accepted by the constitutive law (**Law2** functor), which is the second part of the **Law2** functor's name (e.g. Law2_ScGeom_FrictPhys_Basic accepts FrictPhys)

Note: Unlike with **Bo1** and **Ig2** functors, unhandled combination of Materials is an error condition signaled by an exception.

Law2 functor(s) **Law2** functor was the ultimate criterion for the choice of **Ig2** and **Ig2** functors; there are no restrictions on its choice in itself, as it only applies forces without creating new objects.

In most simulations, only one **Law2** functor will be in use; it is possible, though, to have several of them, dispatched based on combination of InteractionGeometry and InteractionPhysics produced previously by **Ig2** and **Ip2** functors respectively (in turn based on combination of Shapes and Materials).

Note: As in the case of **Ip2** functors, receiving a combination of InteractionGeometry and InteractionPhysics which is not handled by any **Law2** functor is an error.

Examples Let us give several example of the chain of created and accepted types.

Basic DEM model Suppose we want to use the Law2_ScGeom_FrictPhys_Basic constitutive law. We see that

1. the **Ig2** functors most create ScGeom. Since we have spheres and walls in the simulation, we will need functors accepting Sphere + Sphere and Wall + Sphere combinations. We don't want interactions between walls themselves (as a matter of fact, there is no such functor anyway). That gives us Ig2_Sphere_Sphere_ScGeom and Ig2_Wall_Sphere_ScGeom (as a matter of facet, there is no such functor now, although it is planned)
2. the **Ip2** functors should create FrictPhys. Looking at InteractionPhysicsFunctors, there is only Ip2_FrictMat_FrictMat_FrictPhys. That obliges us to use FrictMat for particles.

The result will be therefore:

```
InteractionDispatchers(
   [Ig2_Sphere_Sphere_ScGeom(),Ig2_Wall_Sphere_ScGeom()],
   [Ip2_FrictMat_FrictMat_FrictPhys()],
```

```
   [Law2_ScGeom_FrictPhys_Basic()]
)
```

Concrete model In this case, our goal is to use the Law2_Dem3DofGeom_CpmPhys_Cpm constitutive law.

- We use spheres and facets in the simulation, which selects **Ig2** functors accepting those types and producing Dem3DofGeom: Ig2_Sphere_Sphere_Dem3DofGeom and Ig2_Facet_Sphere_Dem3DofGeom.
- We have to use Material which can be used for creating CpmPhys. We find that CpmPhys is only created by Ip2_CpmMat_CpmMat_CpmPhys, which determines the choice of CpmMat for all particles.

Therefore, we will use:

```
InteractionDispatchers(
   [Ig2_Sphere_Sphere_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
   [Ip2_CpmMat_CpmMat_CpmPhys()],
   [Law2_Dem3DofGeom_CpmPhys_Cpm()]
)
```

6.1.6. Imposing conditions

In most simulations, it is not desired that all particles float freely in space. There are several ways of imposing boundary conditions that block movement of all or some particles with regard to global space.

6.1.6.1. Motion constraints

- Body.dynamic determines whether a body will be moved by NewtonIntegrator; it is mandatory for bodies with zero mass, where applying non-zero force would result in infinite displacement.

  Facets are case in the point: utils.facet makes them non-dynamic by default, as they have zero volume and zero mass (this can be changed, by passing **dynamic=True** to utils.facet or setting Body.dynamic; setting State.mass to a non-zero value must be done as well). The same is true for utils.wall.

  Making sphere non-dynamic is achieved simply by:

  ```
  utils.sphere([x,y,z],radius,dynamic=False)
  ```

  Note: There is an open bug #398089 to define exactly what the **dynamic** flag does. Please read it before writing a new engine relying on this flag.

- State.blockedDOFs permits selective blocking of any of 6 degrees of freedom in global space. For instance, a sphere can be made to move only in the xy plane by saying:

  ```
  Yade [169]: O.bodies.append(utils.sphere((0,0,0),1))
   ->  [169]: 0

  Yade [170]: O.bodies[0].state.blockedDOFs=['z','rx','ry']
  ```

  In contrast to Body.dynamic, blockedDOFs will only block forces (and acceleration) in that direction being effective; if you prescribed linear or angular velocity, they will be applied regardless of blockedDOFs. (This is also related to bug #398089 mentioned above)

It might be desirable to constrain motion of some particles constructed from a generated sphere packing, following some condition, such as being at the bottom of a specimen; this can be done by looping over

all bodies with a conditional:

```
for b in O.bodies:
   # block all particles with z coord below .5:
   if b.state.pos[2]<.5: b.dynamic=False
```

Arbitrary spatial predicates introduced above can be expoited here as well:

```
from yade import pack
pred=pack.inAlignedBox(lowerCorner,upperCorner)
for b in O.bodies:
   if b.shape.name!=Sphere: continue # skip non-spheres
   # ask the predicate if we are inside
   if pred(b.state.pos,b.shape.radius): b.dynamic=False
```

6.1.6.2. Boundary controllers

Engines deriving from BoundaryController impose boundary conditions during simulation, either directly, or by influencing several bodies. You are referred to their individual documentation for details, though you might find interesting in particular

- UniaxialStrainer for applying strain along one axis at constant rate; useful for plotting strain-stress diagrams for uniaxial loading case. See examples/concrete/uniax.py for an example.
- TriaxialStressController which applies prescribed stress/strain along 3 perpendicular axes on cuboid-shaped packing using 6 walls (Box objects) (ThreeDTriaxialEngine is generalized such that it allows independent value of stress along each axis)
- PeriTriaxController for applying stress/strain along 3 axes independently, for simulations using periodic boundary conditions (Cell)

6.1.6.3. Field appliers

Engines deriving from FieldApplier acting on all particles. The one most used is GravityEngine applying uniform acceleration field.

6.1.6.4. Partial engines

Engines deriving from PartialEngine define the subscribedBodies attribute determining bodies which will be affected. Several of them warrant explicit mention here:

- TranslationEngine and RotationEngine for applying constant speed linear and rotational motion on subscribers.
- ForceEngine and TorqueEngine applying given values of force/torque on subscribed bodies at every step.
- StepDisplacer for applying generalized displacement delta at every timestep; designed for precise control of motion when testing constitutive laws on 2 particles.

If you need an engine applying non-constant value instead, there are several interpolating engines (InterpolatingDirectedForceEngine for applying force with varying magnitude, InterpolatingSpiralEngine for applying spiral displacement with varying angular velocity and possibly others); writing a new interpolating engine is rather simple using examples of those that already exist.

6.1.7. Convenience features

6.1.7.1. Labeling things

Engines and functors can define that **label** attribute. Whenever the **O.engines** sequence is modified, python variables of those names are created/update; since it happens in the **___builtins___** namespaces, these names are immediately accessible from anywhere. This was used in *Creating interactions* to change interaction radius in multiple functors at once.

> **Warning:** Make sure you do not use label that will overwrite (or shadow) an object that you already use under that variable name. Take care not to use syntactically wrong names, such as "er*452" or "my engine"; only variable names permissible in Python can be used.

6.1.7.2. Simulation tags

Omega.tags is a dictionary (it behaves like a dictionary, although the implementation in c++ is different) mapping keys to labels. Contrary to regular python dictionaries that you could create,

- **O.tags** is *saved and loaded with simulation*;
- **O.tags** has some values pre-initialized.

After Yade startup, **O.tags** contains the following:

```
Yade [172]: dict(O.tags) # convert to real dictionary
 ->  [172]:
{'author': 'V??clav~??milauer~(vaclav@flux)',
 'description': '',
 'id': '20100531T100723p5665',
 'isoTime': '20100531T100723'}
```

author Real name, username and machine as obtained from your system at simulation creation

id Unique identifier of this Yade instance (or of the instance which created a loded simulation). It is composed of date, time and process number. Useful if you run simulations in parallel and want to avoid overwriting each other's outputs; embed **O.tags['id']** in output filenames (either as directory name, or as part of the file's name itself) to avoid it. This is explained in *batch-output-separate* in detail.

isoTime Time when simulation was created (with second resolution).

You can add your own tags by simply assigning value, with the restriction that the left-handside object must be a string:

```
Yade [173]: O.tags['anythingThat I lik3']='whatever'

Yade [174]: O.tags['anythingThat I lik3']
 ->  [174]: 'whatever'
```

6.1.7.3. Saving python variables

Python variable lifetime is limited; in particular, if you save simulation, variables will be lost after reloading. Yade provides limited support for data persistence for this reason (internally, it uses special values of **O.tags**). The functions in question are utils.saveVars and utils.loadVars.

utils.saveVars takes dictionary (variable names and their values) and a *mark* (identification string for the variable set); it saves the dictionary inside the simulation. These variables can be re-created (after the simulation was loaded from a XML file, for instance) in the **___builtin___** namespace by calling utils.loadVars with the same identification *mark*:

```
Yade [175]: a=45; b=pi/3

Yade [176]: utils.saveVars('ab',a=a,b=b)

# save simulation (we could save to disk just as well)
Yade [176]: O.saveTmp()

# change variable's values
Yade [176]: a=21; b='foo'

Yade [179]: O.loadTmp()

Yade [180]: utils.loadVars('ab')

# GOTCHA! Local variables shadow builtin variables
Yade [181]: print a,b
21 foo

# these are our builtins now
Yade [181]: print __builtins__.a, __builtins__.b
45 1.0471975512

# deletes local variables
Yade [183]: del a,b

Yade [184]: print a,b
45 1.0471975512
```

Enumeration of variables can be tedious if they are many; creating local scope (which is a function definition in Python, for instance) can help:

```
def setGeomVars():
        radius=a*4
        thickness=22
        p_t=4/3*pi
        dim=Vector3(1.23,2.2,3)
        #
        # define as much as you want here
        # it all appears in locals() (and nothing else does)
        #
        utils.saveVars('geom',loadNow=True,**locals())

setGeomVars()
# since we used loadNow=True, the **locals() were re-created in the __builtin__ scope
# therefore we can use them now
```

Note: Only types that can be pickled can be passed to utils.saveVars.

Warning: The utils.saveVars mechanism inherits all problems of global variables (such as shadowing, as shown above). Take care when using it. In particular, conflicting names can he far-reaching consequences.

6.2. Controlling simulation

6.2.1. Tracking variables

6.2.1.1. Running python code

A special engine PeriodicPythonRunner can be used to periodically call python code, specified via the **command** parameter. Periodicity can be controlled by specifying computation time (**realPeriod**), virutal time (**virtPeriod**) or iteration number (**iterPeriod**).

For instance, to print kinetic energy at every step (using utils.kineticEnergy) every 5 seconds, this engine will be p

PeriodicPythonRunner(command="print 'kinetic energy',utils.kineticEnergy()",realPeriod=5)

For running more complex commands, it is convenient to define an external function and only call it from within the engine. Since the **command** is run in the script's namespace, functions defined within scripts can be called. Let us print information on interaction between bodies 0 and 1 periodically:

```
def intrInfo(id1,id2):
        try:
                i=O.interactions[id1,id2]
                # assuming it is a CpmPhys instance
                print id1,id2,i.phys.sigmaN
        except:
                # in case the interaction doesn't exist (yet?)
                print "No interaction between",id1,id2
O.engines=[...,
        PeriodicPythonRunner(command="intrInfo(0,1)",realPeriod=5)
]
```

More useful examples will be given below.

The plot module provides simple interface and storage for tracking various data. Although originally conceived for plotting only, it is widely used for tracking variables in general.

The data are in plot.data dictionary, which maps variable names to list of their values; the plot.addData function is used to add them.

```
Yade [186]: from yade import plot

Yade [187]: plot.data
 ->  [187]: {}

Yade [188]: plot.addData(sigma=12,eps=1e-4)

# not adding sigma will add a NaN automatically
# this assures all variables have the same number of records
Yade [189]: plot.addData(eps=1e-3)

# adds NaNs to already existing sigma and eps columns
Yade [190]: plot.addData(force=1e3)

Yade [191]: plot.data
 ->  [191]:
{'eps': [0.0001, 0.001, nan],
 'force': [nan, nan, 1000.0],
 'sigma': [12, nan, nan]}

# retrieve only one column
Yade [192]: plot.data['eps']
 ->  [192]: [0.0001, 0.001, nan]

# get maximum eps
Yade [193]: max(plot.data['eps'])
 ->  [193]: 0.001
```

New record is added to all columns at every time plot.addData is called; this assures that lines in different columns always match. The special value **nan** or **NaN** (Not a Number) is inserted to mark the record

invalid.

Note: It is not possible to have two columns with the same name, since data are stored as a dictionary.

To record data periodically, use PeriodicPythonRunner. This will record the z coordinate and velocity of body #1, iteration number and simulation time (every 20 iterations):

```
O.engines=O.engines+[PeriodicPythonRunner(command='myAddData()', iterPeriod=20)]

from yade import plot
def myAddData():
        b=O.bodies[1]
        plot.addData(z1=b.state.pos[2], v1=b.state.vel.norm(), i=O.iter, t=O.time)
```

Note: Arbitrary string can be used as column label for data. If it cannot be used as keyword name for plot.addData (since it is a python keyword (**for**), or has spaces inside (**my funny column**), you can pass dictionary to plod.addData instead:

```
plot.addData(z=b.state.pos[2],**{'my funny column':b.state.vel.norm()})
```

An exception are columns having leading of trailing whitespaces. They are handled specially in plot.plots and should not be used (see below).

Labels can be conveniently used to access engines in the **myAddData** function:

```
O.engines=[...,
        UniaxialStrainer(...,label='strainer')
]
def myAddData():
        plot.addData(sigma=strainer.stress,eps=strainer.strain)
```

In that case, naturally, the labeled object must define attributes which are used (UniaxialStrainer.strain and UniaxialStrainer.stress in this case).

6.2.1.2. Plotting variables

Above, we explained how to track variables by storing then using plot.addData. These data can be readily used for plotting. Yade provides a simple, quick to use, plotting in the plot module. Naturally, since direct access to underlying data is possible via plot.data, these data can be processed in any way.

The plot.plots dictionary is a simple specification of plots. Keys are x-axis variable, and values are tuple of y-axis variables, given as strings that were used for plot.addData; each entry in the dictionary represents a separate figure:

```
plot.plots={
        'i':('t',),     # plot t(i)
        't':('z1','v1') # z1(t) and v1(t)
}
```

Actual plot using data in plot.data and plot specification of plot.plots can be triggered by invoking the plot.plot function.

Note: Yade does not feature live-updates of figures (as much as it would be nice). If you are strong in python, you are welcome to take up this challenege.

Multiple figures Since plot.plots is a dictionary, multiple entries with the same key (x-axis variable) would not be possible, since they overwrite each other:

```
Yade [194]: plot.plots={
     .....:     'i':('t',),
```

```
     .....:     'i':('z1','v1')
     .....: }

Yade [198]: plot.plots
 ->  [198]: {'i': ('z1', 'v1')}
```

You can, however, distinguish them by prepending/appending space to the x-axis variable, which will be removed automatically when looking for the variable in plot.data – both x-axes will use the **i** column:

```
Yade [199]: plot.plots={
     .....:     'i':('t',),
     .....:     'i ':('z1','v1') # note the space in 'i '
     .....: }

Yade [203]: plot.plots
 ->  [203]: {'i': ('t',), 'i ': ('z1', 'v1')}
```

Split y1 y2 axes To avoid big range differences on the y axis, it is possible to have left and right y axes separate (like **axes x1y2** in gnuplot). This is achieved by inserting **None** to the plot specifier; variables coming before will be plot normally (on the left y-axis), while those after will appear on the right:

```
plot.plots={'i':('z1',None,'v1')}
```

Exporting Plots can be exported to external files for later post-processing via that plot.saveGnuplot function.

- Data file is saved (compressed using bzip2) separately from the gnuplot file, so any other programs can be used to process them. In particular, the **numpy.genfromtxt** (documented here) can be useful to import those data back to python; the decompression happens automatically.
- The gnuplot file can be run through gnuplot to produce the figure; see plot.saveGnuplot documentation for details.

6.2.2. Stop conditions

For simulations with pre-determined number of steps, number of steps can be prescribed:

    # absolute iteration number O.stopAtIter=35466 O.run() O.wait()

or

```
# number of iterations to run from now
O.run(35466,True) # wait=True
```

causes the simulation to run 35466 iterations, then stopping.

Frequently, decisions have to be made based on evolution of the simulation itself, which is not yet known. In such case, a function checking some specific condition is called periodically; if the condition is satisfied, **O.pause** or other functions can be called to stop the stimulation. See documentation for Omega.run, Omega.pause, Omega.step, Omega.stopAtIter for details.

For simulations that seek static equilibrium, the _utils.unbalancedForce can provide a useful metrics (see its documentation for details); for a desired value of **1e-2** or less, for instance, we can use:

```
def checkUnbalanced():
        if utils.unbalancedForce<1e-2: O.pause()

O.engines=O.engines+[PeriodicPythonRunner(command="checkUnbalanced",iterPeriod=100)]
```

```
# this would work as well, without the function defined apart:
#   PeriodicPythonRunner(command="if utils.unablancedForce<1e-2: O.pause()",iterPeriod=100)

O.run(); O.wait()
# will continue after O.pause() will have been called
```

Arbitrary functions can be periodically checked, and they can also use history of variables tracked via plot.addData. For example, this is a simplified version of damage control in examples/concrete/uniax.py; it stops when current stress is lower than half of the peak stress:

```
O.engines=[...,
        UniaxialStrainer=(...,label='strainer'),
        PeriodicPythonRunner(command='myAddData()',iterPeriod=100),
        PeriodicPythonRunner(command='stopIfDamaged()',iterPeriod=100)
]

def myAddData():
        plot.addData(t=O.time,eps=strainer.strain,sigma=strainer.stress)

def stopIfDamaged():
        currSig=plot.data['sigma'][-1] # last sigma value
        maxSig=max(plot.data['sigma']) # maximum sigma value
        # print something in any case, so that we know what is happening
        print plot.data['eps'][-1],currSig
        if currSig<.5*maxSig:
                print "Damaged, stopping"
                print 'gnuplot',plot.saveGnuplot(O.tags['id'])
                import sys
                sys.exit(0)

O.run(); O.wait()
# this place is never reached, since we call sys.exit(0) directly
```

6.2.2.1. Checkpoints

Ocasionally, it is useful to revert to simulation at some past point and continue from it with different parameters. For instance, tension/compression test will use the same initial state but load it in 2 different directions. Two functions, Omega.saveTmp and Omega.loadTmp are provided for this purpose; *memory* is used as storage medium, which means that saving is faster, and also that the simulation will disappear when Yade finishes.

```
O.saveTmp()
# do something
O.saveTmp('foo')
O.loadTmp()      # loads the first state
O.loadTmp('foo') # loads the second state
```

Warning: **O.loadTmp** cannot be called from inside an engine, since *before* loading a simulation, the old one must finish the current iteration; it would lead to deadlock, since **O.loadTmp** would wait for the current iteration to finish, while the current iteration would be blocked on **O.loadTmp**.
A special trick must be used: a separate function to be run after the current iteration is defined and is invoked from an independent thread launched only for that purpose:

```
O.engines=[...,PeriodicPythonRunner('myFunc()',iterPeriod=345)]

def myFunc():
        if someCondition:
                import thread
                # the () are arguments passed to the function
                thread.start_new_thread(afterIterFunc,())
def afterIterFunc():
        O.pause(); O.wait() # wait till the iteration really finishes
        O.loadTmp()

O.saveTmp()
O.run()
```

6.2.3. Remote control

Yade can be controlled remotely over network. At yade startup, the following lines appear, among other messages:

```
TCP python prompt on localhost:9000, auth cookie `dcekyu'
TCP info provider on localhost:21000
```

They inform about 2 ports on which connection of 2 different kind is accepted.

6.2.3.1. Python prompt

TCP python prompt is telnet server with authenticated connection, providing full python command-line. It listens on port 9000, or higher if already occupied (by another yade instance, for example).

Using the authentication cookie, connection can be made:

```
$ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Enter auth cookie: dcekyu
__   __    ____                  __  _____ ____ ____
\ \ / /_ _|  _ \  ___    ___   / / |_   _/ ___|  _ \
 \ V / _` | | | |/ _ \  / _ \ / /    | || |   | |_) |
  | | (_| | |_| |  __/ | (_) / /     | || |___|  __/
  |_|\__,_|____/ \___|  \___/_/      |_| \____|_|

(connected from 127.0.0.1:40372)
>>>
```

The python pseudo-prompt >>> lets you write commands to manipulate simulation in variety of ways as usual. Two things to notice:

1. The new python interpreter (>>>) lives in a namespace separate from **Yade [1]:** command-line. For your convenience, **from yade import *** is run in the new python instance first, but local and global variables are not accessible (only builtins are).

2. The (fake) >>> interpreter does not have rich interactive feature of IPython, which handles the usual command-line **Yade [1]:**; therefore, you will have no command history, **?** help and so on.

Note: By giving access to python interpreter, full control of the system (including reading user's files) is possible. For this reason, **connection are only allowed from localhost**, not over network remotely.

> **Warning:** Authetication cookie is trivial to crack via bruteforce attack. Although the listener stalls for 5 seconds after every failed login attempt (and disconnects), the cookie could be guessed by trial-and-error during very long simulations on a shared computer.

6.2.3.2. Info provider

TCP Info provider listens at port 21000 (or higher) and returns some basic information about current simulation upon connection; the connection terminates immediately afterwards. The information is python dictionary represented as string (serialized) using standard pickle module.

This functionality is used by the batch system (described below) to be informed about individual simulation progress and estimated times. If you want to access this information yourself, you can study core/main/yade-multi.in for details.

6.2.4. Batch queuing and execution (yade-multi)

Yade features light-weight system for running one simulation with different parameters; it handles assignment of parameter values to python variables in simulation script, scheduling jobs based on number of available and required cores and more. The whole batch consists of 2 files:

simulation script regular Yade script, which calls utils.readParamsFromTable to obtain parameters from parameter table. In order to make the script runnable outside the batch, readParamsFromTable takes default values of parameters, which might be overridden from the parameter table.

utils.readParamsFromTable knows which parameter file and which line to read by inspecting the **PARAM_TABLE** environment variable, set by the batch system.

parameter table simple text file, each line representing one parameter set. This file is read by utils.readParamsFromTable (using utils.TableParamReader class), called from simulation script, as explained above.

The batch can be run as

```
yade-multi parameters.table simulation.py
```

and it will intelligently run one simulation for each parameter table line.

6.2.4.1. Example

This example is found in scripts/multi.table and scripts/multi.py.

Suppsoe we want to study influence of parameters *density* and *initialVelocity* on position of a sphere falling on fixed box. We create parameter table like this:

```
description density initialVelocity # first non-empty line are column headings
reference   2400    10
hi_v           =    20              # = to use value from previous line
lo_v           =     5
# comments are allowed
hi_rho      5000    10
# blank lines as well:
```

```
hi_rho_v       =    20
hi_rh0_lo_v    =     5
```

Each line give one combination of these 2 parameters and assigns (which is optional) a *description* of this simulation.

In the simulation file, we read parameters from table, at the beginning of the script; each parameter has default value, which is used if not specified in the parameters file:

```
from yade import utils
utils.readParamsFromTable(
        gravity=-9.81,
        density=2400,
        initialVelocity=20,
        noTableOk=True     # use default values if not run in batch
)
print gravity, density, initialVelocity
```

after the call to utils.readParamsFromTable, corresponding python variables are created and can be readily used in the script, e.g.

```
GravityEngine(gravity=(0,0,gravity))
```

Warning: New variables are created in the **builtin** namespace, in order to be visible from everywhere. Avoid name clash with existing variables, such as **Vector3** or **time**, your script might be rendered unfunctional.

Let us see what happens when running the batch:

```
$ yade-multi multi.table multi.py
Will run `/usr/local/bin/yade-trunk' on `multi.py' with nice value 10, output redirected to `multi.@.log', 4 jo
Will use table `multi.table', with available lines 2, 3, 4, 5, 6, 7.
Will use lines  2 (reference), 3 (hi_v), 4 (lo_v), 5 (hi_rho), 6 (hi_rho_v), 7 (hi_rh0_lo_v).
Master process pid 7030
```

These lines inform us about general batch information: nice level, log file names, how many cores will be used (4); table name, and line numbers that contain parameters; finally, which lines will be used; master PID is useful for killing (stopping) the whole batch with the **kill** command.

```
Job summary:
   #0 (reference/4): PARAM_TABLE=multi.table:2 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x mult
   #1 (hi_v/4): PARAM_TABLE=multi.table:3 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x multi.py
   #2 (lo_v/4): PARAM_TABLE=multi.table:4 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x multi.py
   #3 (hi_rho/4): PARAM_TABLE=multi.table:5 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x multi.p
   #4 (hi_rho_v/4): PARAM_TABLE=multi.table:6 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x multi
   #5 (hi_rh0_lo_v/4): PARAM_TABLE=multi.table:7 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x mu
```

displays all jobs with command-lines that will be run for each of them. At this moment, the batch starts to be run.

```
#0 (reference/4) started on Tue Apr 13 13:59:32 2010
#0 (reference/4) done    (exit status 0), duration 00:00:01, log multi.reference.log
#1 (hi_v/4) started on Tue Apr 13 13:59:34 2010
#1 (hi_v/4) done    (exit status 0), duration 00:00:01, log multi.hi_v.log
#2 (lo_v/4) started on Tue Apr 13 13:59:35 2010
#2 (lo_v/4) done    (exit status 0), duration 00:00:01, log multi.lo_v.log
#3 (hi_rho/4) started on Tue Apr 13 13:59:37 2010
#3 (hi_rho/4) done    (exit status 0), duration 00:00:01, log multi.hi_rho.log
#4 (hi_rho_v/4) started on Tue Apr 13 13:59:38 2010
#4 (hi_rho_v/4) done    (exit status 0), duration 00:00:01, log multi.hi_rho_v.log
#5 (hi_rh0_lo_v/4) started on Tue Apr 13 13:59:40 2010
```

```
#5 (hi_rh0_lo_v/4) done    (exit status 0), duration 00:00:01, log multi.hi_rh0_lo_v.log
```

information about job status changes is being printed, until:

```
All jobs finished, total time  00:00:08
Log files:
multi.reference.log multi.hi_v.log multi.lo_v.log multi.hi_rho.log multi.hi_rho_v.log multi.hi_rh0_lo_v.log
Bye.
```

6.2.4.2. Separating output files from jobs

As one might output data to external files during simulation (using classes such as VTKRecorder, it is important to name files in such way that they are not overwritten by next (or concurrent) job in the same batch. A special tag **O.tags['id']** is provided for such purposes: it is comprised of date, time and PID, which makes it always unique (e.g. **20100413T144723p7625**); additional advantage is that alphabetical order of the **id** tag is also chronological.

For smaller simulations, prepending all output file names with **O.tags['id']** can be sufficient:

```
utils.saveGnuplot(O.tags['id'])
```

For larger simulations, it is advisable to create separate directory of that name first, putting all files inside afterwards:

```
os.mkdir(O.tags['id'])
O.engines=[
        # …
        VTKRecorder(fileName=O.tags['id']+'/'+'vtk'),
        # …
]
# …
O.saveGnuplot(O.tags['id']+'/'+'graph1')
```

6.2.4.3. Controlling parallel compuation

Default total number of available cores is determined from **/proc/cpuinfo** (provided by Linux kernel); in addition, if **OMP_NUM_THREADS** environment variable is set, minimum of these two is taken. The **-j**/--**jobs** option can be used to override this number.

By default, each job uses all available cores for itself, which causes jobs to be effectively run in parallel. Number of cores per job can be globally changed via the --**job-threads** option.

Table column named **!OMP_NUM_THREADS** (**!** prepended to column generally means to assign *environment variable*, rather than python variable) controls number of threads for each job separately, if it exists.

If number of cores for a job exceeds total number of cores, warning is issued and only the total number of cores is used instead.

6.2.4.4. Merging gnuplot from individual jobs

Frequently, it is desirable to obtain single figure for all jobs in the batch, for comparison purposes. Somewhat heiristic way for this functionality is provided by the batch system. **yade-multi** must be run with the --**gnuplot** option, specifying some file name that will be used for the merged figure:

```
yade-trunk --gnuplot merged.gnuplot multi.table multi.py
```

Data are collected in usual way during the simulation (using plot.addData) and saved to gnuplot file via plot.saveGnuplot (it creates 2 files: gnuplot command file and compressed data file). The batch system *scans*, once the job is finished, log file for line of the form **gnuplot [something]**. Therefore, in onrder to print this *magic line* we put:

```
print 'gnuplot',plot.saveGnuplot(O.tags['id'])
```

and the end of the script, which prints:

```
gnuplot 20100413T144723p7625.gnuplot
```

to the output (redirected to log file).

This file itself contains single graph:

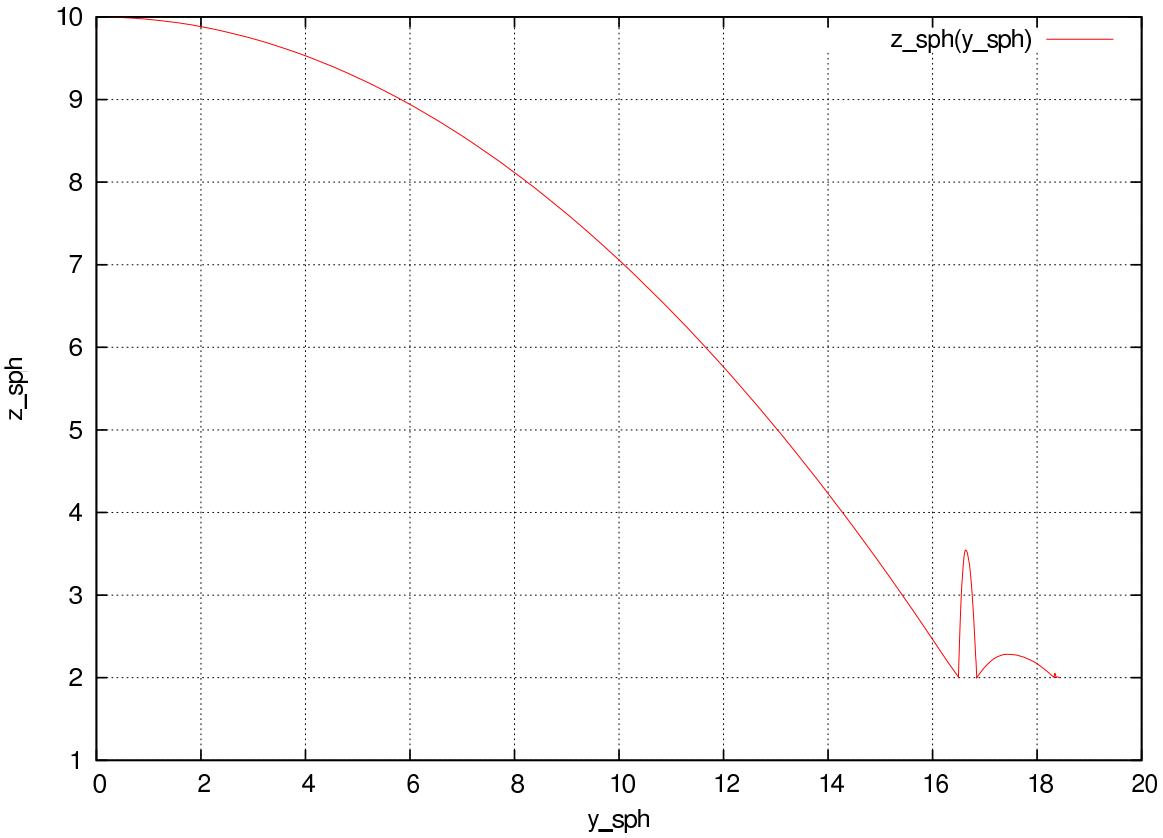


Figure 6.5.: Figure from single job in the batch.

At the end, the batch system knows about all gnuplot files and tries to merge them together, by assembling the **merged.gnuplot** file.

6.2.4.5. HTTP overview

While job is running, the batch system presents progress via simple HTTP server running at port 9080, which can be acessed from regular web browser by requesting the **http://localhost:9080** URL. This page can be accessed remotely over network as well.

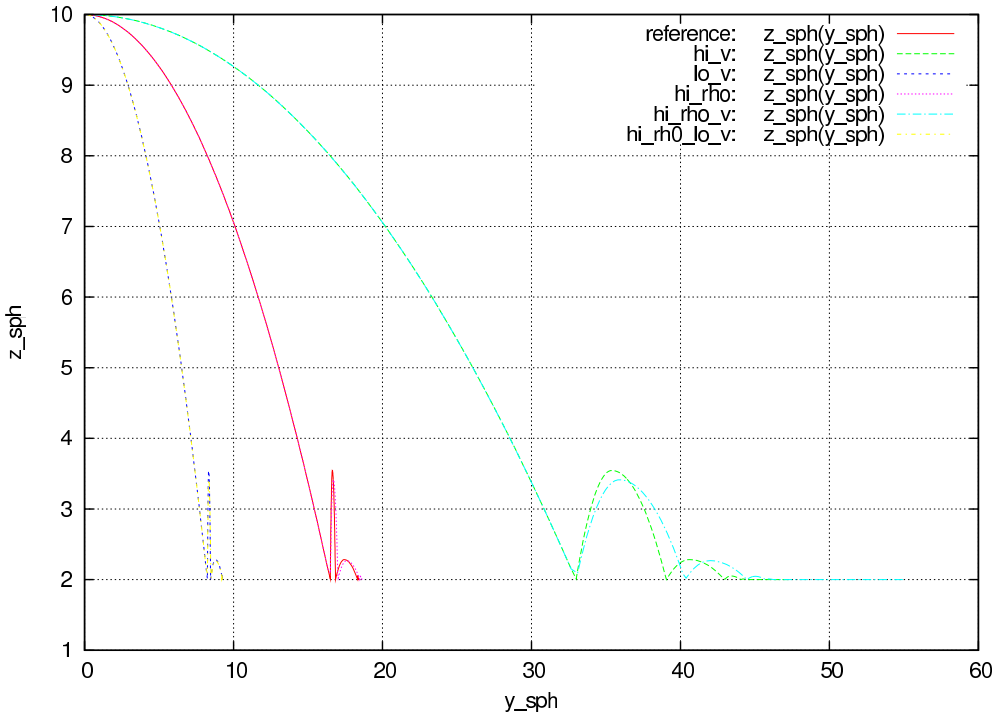


Figure 6.6.: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.

Running for 00:10:19, since Tue Apr 13 16:17:11 2010.

Pid 9873

4 slots available, 4 used, 0 free.

Jobs

4 total, **2** running, **1** done

id	status	info	slots	command
_geomType=B	00:10:19	96.33% done step 9180/9530 avg 14.9596/sec 10267 bodies 65506 intrs	2	PARAM_TABLE=iParams.table:2 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=B.log 2>&1
_geomType=smallA	00:09:53	(no info)	2	PARAM_TABLE=iParams.table:3 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallA.log 2>&1
_geomType=smallB	00:00:24	6.95% done step 694/9985 avg 35.8212/sec 9021 bodies 58352 intrs	2	PARAM_TABLE=iParams.table:4 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallB.log 2>&1
_geomType=smallC	(pending)	(no info)	2	PARAM_TABLE=iParams.table:5 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallC.log 2>&1

Figure 6.7.: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

6.3. Postprocessing

Not yet written. Describe how to use plot, post2d and VTKRecorder with Paraview.

6.4. Extending Yade

How to write new constitutive law: not yet written, see https://yade-dem.org/wiki/ConstitutiveLawHowto.

6.5. Troubleshooting

6.5.1. Crashes

It is possible that you encounter crash of Yade, i.e. Yade terminates with error message such as

```
Segmentation fault (core dumped)
```

without further explanation. Frequent causes of such conditions are

- program error in Yade itself;
- fatal condition in you particular simulation (such as impossible dispatch);
- problem with graphics card driver.

Try to reproduce the error (run the same script) with debug-enabled version of Yade. Debugger will be automatically launched at crash, showing backtrace of the code (in this case, we triggered crash by hand):

```
Yade [1]: import os,signal
Yade [2]: os.kill(os.getpid(),signal.SIGSEGV)
SIGSEGV/SIGABRT handler called; gdb batch file is `/tmp/yade-YwtfRY/tmp-0'
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
[Thread debugging using libthread_db enabled]
[New Thread 0x7f0fb1268710 (LWP 16471)]
[New Thread 0x7f0fb29f2710 (LWP 16470)]
[New Thread 0x7f0fb31f3710 (LWP 16469)]


…
```

What looks as cryptic message is valuable information for developers to locate source of the bug. In particular, there is (usually) line <**signal handler called**>; lines below it are source of the bug (at least very likely so):

```
Thread 1 (Thread 0x7f0fcee53700 (LWP 16465)):
#0  0x00007f0fcd8f4f7d in __libc_waitpid (pid=16497, stat_loc=<value optimized out>, options=0) at ../sysdeps/ur
#1  0x00007f0fcd88c7e9 in do_system (line=<value optimized out>) at ../sysdeps/posix/system.c:149
#2  0x00007f0fcd88cb20 in __libc_system (line=<value optimized out>) at ../sysdeps/posix/system.c:190
#3  0x00007f0fcd0b4b23 in crashHandler (sig=11) at core/main/pyboot.cpp:45
```

```
#4  <signal handler called>
#5  0x00007f0fcd87ed57 in kill () at ../sysdeps/unix/syscall-template.S:82
#6  0x000000000051336d in posix_kill (self=<value optimized out>, args=<value optimized out>) at ../Modules/posi
#7  0x00000000004a7c5e in call_function (f=Frame 0x1c54620, for file <ipython console>, line 1, in <module> (),
#8  PyEval_EvalFrameEx (f=Frame 0x1c54620, for file <ipython console>, line 1, in <module> (), throwflag=<value
```

If you think this might be error in Yade, file a bug report as explained below. Do not forget to attach *full* yade output from terminal, including startup messages and debugger output – select with right moust button, paste with middle button to the bugreport to to a file and attach it. Attach your simulation script as well.

6.5.2. Reporting bugs

Bugs are general name for defects (functionality shortcomings, misdocumentation, crashes) or feature requests. They are tracked at http://bugs.launchpad.net/yade.

When reporting a new bug, be as specific as possible; state version of yade you use, sustem version and so on, as explained in the above section on crashes.

6.5.3. Getting help

6.5.3.1. Mailing lists

Yade has two mailing-lists. Both are hosted at http://www.launchpad.net and before posting, you must register to Launchpad and subscribe to the list by adding yourself to "team" of the same name running the list.

yade-users@lists.launchpad.net is general help list for Yade users. Add yourself to yade-users team so that you can post messages. List archive is available.

yade-dev@lists.launchpad.net is for discussions about Yade development; you must be member of yade-dev team to post. This list is archived as well.

Read How To Ask Questions The Smart Way before posting. Do not forget to state what *version* of yade you use (shown when you start yade), what operating system (such as Ubuntu 10.04), and if you have done any local modifications to source code.

6.5.3.2. Questions and answers

Launchpad provides interface for giving questions at https://answers.launchpad.net/yade/ which you can use instead of mailing lists; at the moment, it functionally somewhat overlaps with yade-users, but has the advantage of tracking whether a particular question has already been answered.

6.5.3.3. Wiki

http://www.yade-dem.org/wiki/

6.5.3.4. Private and/or paid support

You might contact developers by their private mail (rather than by mailing list) if you do not want to disclose details on the mailing list. This is also a suitable method for proposing financial reward for implementation of a substantial feature that is not yet in Yade – typically, though, we will request this

feature to be part of the public codebase once completed, so that the rest of the community can benefit from it as well.

7. Programmer's manual

7.1. Build system

Yade uses [scons] build system for managing the build process. It takes care of configuration, compilation and installation. SCons is written in python and its build scripts are in python, too. SCons complete documentation can be found in its manual page.

7.1.1. Pre-build configuration

We use **$** to denote build variable in strings in this section; in SCons script, they can be used either by writing **$variable** in strings passed to SCons functions, or obtained as attribute of the **Environment** instance **env**, i.e. **env['variable']**; we use the formed in running text here.

In order to allow parallel installation of multiple yade versions, the installation location follows the pattern **$PREFIX/lib/yade$SUFFIX** for libraries and **$PREFIX/bin/yade$SUFFIX** for executables (in the following, we will refer only to the first one). **$SUFFIX** takes the form **-$version$variant**, which further allows multiple different builds of the same version (typically, optimized and debug builds). For instance, the default debug build of version 0.5 would be installed in **/usr/local/lib/yade-0.5-dbg/**, the executable being **/usr/local/bin/yade-0.5-dbg**.

The build process takes place outside the source tree, in directory reffered to as **$buildDir** within those scripts. By default, this directory is **../build-$SUFFIX**.

Each build depends on a number of configuration parameters, which are stored in mutually independent *profiles*. They are selected according to the **profile** argument to scons (by default, the last profile used, stored in **scons.current-profile**). Each profile remembers its non-default variables in **scons.profile-$profile**.

There is a number of configuration parameters; you can list all of them by **scons -h**. The following table summarizes only a few that are the most used.

PREFIX [default: /usr/local] installation prefix (**PREFIX** preprocessor macro; **yade.config.prefix** in python

version [bzr revision (e.g. bzr1899)] first part of suffix (**SUFFIX** preprocessor macro; **yade.config.suffix** in python)]

variant [*(empty)*] second part of suffix

buildPrefix [..] where to create **build-$SUFFIX** directory

debug [*False* (0)] add debugging symbols to output, enable stack traces on crash

optimize [*True* (1)] optimize binaries (#defines NDEBUG; assertions eliminated; YADE_CAST and YADE_PTR_CAST are static casts rather than dynamic; LOG_TRACE and LOG_DEBUG are eliminated)

CPPPATH [/usr/include/vtk-5.2:/usr/include/vtk-5.4] additional colon-separated paths for preprocessor (for atypical header locations). Required by some libraries, such as VTK (reflected by the default)

LIBPATH [*(empty)*] additional colon-separated paths for linker

CXX [g++] compiler executable

CXXFLAGS [*(empty)*] additional compiler flags (may are added automatically)

jobs [4] number of concurrent compilations to run

brief [*True* (1)] only show brief notices about what is being done rather than full command-lines during compilation

linkStrategy [monolithic] whether to link all plugins in one shared library (**monolithic**) or in one file per plugin (**per-class**); the first option is faster for overall builds, while the latter one makes recompilation of only part of Yade faster; granularity of monolithic build can be changed with the **chunkSize** parameter, which determines how many files are compiled at once.

features [log4cxx,opengl,gts,openmp] optional comma-separated features to build with (details below; each defines macro **YADE_$FEATURE**; available as lowercased list **yade.config.features** at runtime

7.1.1.1. Library detection

When the **scons** command is run, it first checks for presence of all required libraries. Some of them are *essential*, other are *optional* and will be required only if features that need them are enabled.

Essentials

compiler Obviously c++ compiler is necessary. Yade relies on several extensions of **g++** from the [gcc] suite and cannot (probably) be built with other compilers.

boost [boost] is a large collection of peer-reviewed c++ libraries. Yade currently uses thread, date_time, filesystem, iostreams, regex, serialization, program_options, foreach, python; typically the whole boost bundle will be installed. If you need functionality from other modules, you can make presence of that module mandatory. Only be careful about relying on very new features; due to range of systems yade is or might be used on, it is better to be moderately conservative (read: roughly 3 years backwards compatibility).

python [python] is the scripting language used by yade. Besides [boost::python]_, yade further requires

- [ipython] (terminal interaction)
- [matplotlib] (plotting)
- [numpy] (matlab-like numerical functionality and accessing numpy arrays from **c/c++** efficiently)

Optional libraries (features) The *features* parameter controls optional functionality. Each enabled feature defines preprocessor macro *YADE_FEATURE* (name uppercased) to enable selective exclude/include of parts of code. In some cases, it would be meaningless to compile some file at all (e.g. *VTKRecorder* without the *vtk* feature). This can be controller using the *YADE_REQUIRE_FEATURE* places in the respective implementation file (see the *Linking* section for more details).

log4cxx (YADE_LOG4CXX) Enable flexible logging system ([log4cxx]), which permits to assign logging levels on per-class basis; doesn't change API, only redefines *LOG_INFO* and other macros accordingly; see *log4cxx* for details.

opengl (YADE_OPENGL) Enable 3d rendering as well as the Qt3-based graphical user interface (in addition to python console).

vtk (YADE_VTK) Enable functionality using Visualization Toolkit ([vtk]; e.g. VTKRecorder exporting to files readable with ParaView).

openmp (YADE_OPENMP) Enable parallelization using OpenMP, non-intrusive shared-memory parallelization framework; it is only supported for **g++** > 4.0. Parallel computation leads to significant performance increase and should be enabled unless you have a special reason for not doing so (e.g. single-core machine). See *upyade-parallel* for details.

gts (YADE_GTS) Enable functionality provided by GNU Triangulated Surface library ([gts]) and build PyGTS, its python interface; used for surface import and construction.

cgal (YADE_CGAL) Enable functionality provided by Computation Geometry Algorithms Library ([cgal]); triangulation code in MicroMacroAnalyser and PersistentTriagulationCollider ses its routines.

other There might be more features added in the future. Always refer to **scons -h** output for possible values.

> **Warning:** Due to a long-standing bug in log4cxx, using log4cxx will make yade crash at every exit. We work-around this partially by disabling the crash handler for regular exits, but process exit status will still be non-zero. The batch system (*yade-multi*) detects successful runs by looking at magic line "Yade: normal exit." in the process' standard output.

Before compilation, SCons will check for presence of libraries required by their respective features [1]. Failure will occur if a respective library isn't found. To find out what went wrong, you can inspect **../build-$SUFFIX/config.log** file; it contains exact commands and their output for all performed checks.

Note: Features are not auto-detected on purpose; otherwise problem with library detection might build Yade without expected features, causing specifically problems for automatized builds.

7.1.2. Building

Yade source tree has the following structure (omiting **debian**, **doc**, **examples** and **scripts** which don't participate in the build process); we shall call each top-level component *module*:

```
attic/        ## code that is not currently functional and might be removed unless resurrected
   lattice/      ## lattice and lattice-like models
   snow/         ## snow model (is really a DEM)
core/         ## core simulation building blocks
extra/        ## miscillanea
gui/          ## user interfaces
   qt3/          ## graphical user interface based on qt3 and OpenGL
   py/           ## python console interface (phased out)
lib/          ## support libraries, not specific to simulations
pkg/          ## simulation-specific files
   common/       ## generally useful classes
   dem/          ## classes for Discrete Element Method
py/           ## python modules
```

Each directory on the top of this hierarchy (except **pkg**, which is treated specially – see below) contains file **SConscript**, determining what files to compile, how to link them together and where should they be installed. Within these script, a scons variable **env** (build **Environment**) contains all the configuration parameters, which are used to influence the build process; they can be either obtained with the **[]** operator, but scons also replaces **$var** strings automatically in arguments to its functions:

```
if 'opengl' in env['features']:
        env.Install('$PREFIX/lib/yade$SUFFIX/',[
                # ...
        ])
```

[1] Library checks are defined inside the **SConstruct** file and you can add your own, should you need it.

7.1.2.1. Header installation

To allow flexibility in source layout, SCons will copy (symlink) all headers into flattened structure within the build directory. First 2 components of the original directory are joind by dash, deeper levels are discarded (in case of **core** and **extra**, only 1 level is used). The following table makes gives a few examples:

Original header location	Included as
core/Scene.hpp	**<yade/core/Scene.hpp>**
lib/base/Logging.hpp	**<yade/lib-base/Logging.hpp>**
lib/serialization/Serializable.hpp	**<yade/lib-serialization/Serializable.hpp>**
pkg/dem/DataClass/SpherePack.hpp	**<yade/pkg-dem/SpherePack.hpp>**
gui/qt3/QtGUI.hpp	**<yade/gui-qt3/QtGUI.hpp>**

It is advised to use **#include<yade/module/Class.hpp>** style of inclusion rather than **#include"Class.hpp** even if you are in the same directory.

7.1.2.2. What files to compile

SConscript files in **lib**, **core**, **gui**, **py** and **extra** explicitly determine what files will be built.

Automatic compilation In the **pkg/** directory, situation is different. In order to maximally ease addition of modules to yade, all ***.cpp** files are *automatically scanned* by SCons and considered for compilation. Each file may contain multiple lines that declare features that are necessary for this file to be compiled:

```
YADE_REQUIRE_FEATURE(vtk);
YADE_REQUIRE_FEATURE(gts);
```

This file will be compiled only if *both* **vtk** and **gts** features are enabled. Depending on current feature set, only selection of plugins will be compiled.

It is possible to disable compilation of a file by requiring any non-existent feature, such as:

```
YADE_REQUIRE_FEATURE(temporarily disabled 345uiysdijkn);
```

The **YADE_REQUIRE_FEATURE** macro expands to nothing during actual compilation.

Note: The source scanner was written by hand and is not official part of SCons. It is fairly primitive and in particular, it doesn't interpret c preprocessor macros, except for a simple non-nested feature-checks like **#ifdef YADE_*/#ifndef YADE_*** #endif.

7.1.2.3. Linking

The order in which modules might depend on each other is given as follows:

module	resulting shared library	dependencies
lib	**libyade-support.so**	can depend on external libraries, may **not** depend on any other part of Yade.
core	**libcore.so**	**yade-support**; *may* depend on external libraries.
pkg	**libplugins.so** for monolithic builds, **libClass.so** for per-class (per-plugin) builds.	**core**, **yade-support**; may **not** depend on external libraries explcitly (only implicitly, by adding the library to global linker flags in **SConstruct**)
extra	(undefined)	(arbitrary)
gui	**libQtGUI.so**, **libPythonUI.so**	**lib**, **core**, **pkg**
py	(many files)	**lib**, **core**, **pkg**, external

Because **pkg** plugins might be linked differently depending on the **linkStrategy** option, **SConscript** files that need to explicitly declare the dependency should use provided **linkPlugins** function which returns libraries in which given plugins will be defined:

```
env.SharedLibrary('_packSpheres',['_packSpheres.cpp'],
        SHLIBPREFIX='',
        LIBS=env['LIBS']+[linkPlugins(['Shop','SpherePack']),]
),
```

Note: **env['LIBS']** are libraries that all files are linked to and they should always be part of the **LIBS** parameter.

Since plugins in **pkg** are not declared in any **SConscript** file, other plugins they depend on are again found *automatically* by scannig their **#include** directives for the pattern **#include<yade/module/Plugin.hpp>**. Again, this works well in normal circumastances, but is not necessarily robust.

See scons manpage for meaning of parameters passed to build functions, such as **SHLIBPREFIX**.

7.2. Conventions

The following rules that should be respected; documentation is treated separately.

- general
  - C++ source files have **.hpp** and **.cpp** extensions (for headers and implementation, respectively).
  - All header files should have the **#pragma once** multiple-inclusion guard.
  - Try to avoid **using namespace ...** in header files.
  - Use tabs for indentation. While this is merely visual in **c++**, it has semantic meaning in python; inadverently mixing tabs and spaces can result in syntax errors.
- capitalization style
  - Types should be always capitalized. Use CamelCase for composed names (**GlobalEngine**). Underscores should be used only in special cases, such as functor names.
  - Class data members and methods must not be capitalized, composed names should use use lowercased camelCase (**glutSlices**). The same applies for functions in python modules.
  - Preprocessor macros are uppercase, separated by underscores; those that are used outside the core take (with exceptions) the form **YADE_***, such as *YADE_CLASS_BASE_DOC_* macro family*.
- programming style
  - Be defensive, if it has no significant performance impact. Use assertions abundantly: they don't affect performance (in the optimized build) and make spotting error conditions much easier.
  - Use logging abundantly. Again, LOG_TRACE and LOG_DEBUG are eliminated from optimized code; unless turned on explicitly, the ouput will be suppressed even in the debug build (see below).
  - Use YADE_CAST and YADE_PTR_CAST where you want type-check during debug builds, but fast casting in optimized build.
  - Initialize all class variables in the default constructor. This avoids bugs that may manifest randomly and are difficult to fix. Initializing with NaN's will help you find otherwise unitialized variable. (This is taken care of by *YADE_CLASS_BASE_DOC_* macro family* macros

for user classes)

7.2.1. Class naming

Although for historial reasons the naming scheme is not completely consistent, these rules should be obeyed especially when adding a new class.

GlobalEngines and PartialEngines GlobalEngines should be named in a way suggesting that it is a performer of certain action (like ForceResetter, InsertionSortCollider, Recorder); if this is not appropriate, append the **Engine** to the characteristics (GravityEngine). PartialEngines have no special naming convention different from GlobalEngines.

Dispatchers Names of all dispatchers end in **Dispatcher**. The name is composed of type it creates or, in case it doesn't create any objects, its main characteristics. Currently, the following dispatchers [2] are defined:

dispatcher	arity	dispatch types	created type	functor type	functor prefix
BoundDispatcher	1	Shape	Bound	BoundFunctor	**Bo1**
InteractionGeometryDispatcher	2 (symmetric)	2 × Shape	InteractionGeometry	InteractionGeometryFunctor	**Ig2**
InteractionPhysicsDispatcher	2 (symmetric)	2 × Material	InteractionPhysics	InteractionPhysicsFunctor	**Ip2**
LawDispatcher	2 (asymmetric)	InteractionGeometry, InteractionPhysics	*(none)*	LawFunctor	**Law2**

Respective abstract functors for each dispatchers are BoundFunctor, InteractionGeometryFunctor, InteractionPhysicsFunctor and LawFunctor.

Functors Functor name is composed of 3 parts, separated by underscore.

1. prefix, composed of abbreviated functor type and arity (see table above)
2. Types entering the dispatcher logic (1 for unary and 2 for binary functors)
3. Return type for functors that create instances, simple characteristics for functors that don't create instances.

To give a few examples:

- Bo1_Sphere_Aabb is a BoundFunctor which is called for Sphere, creating an instance of Aabb.
- Ig2_Facet_Sphere_Dem3DofGeom is binary functor called for Facet and Sphere, creating and instace of Dem3DofGeom.
- Law2_Dem3Dof_CpmPhys_Cpm is binary functor (LawFunctor) called for types Dem3Dof (Geom) and CpmPhys.

[2] Not considering OpenGL dispatchers, which might be replaced by regular virtual functions in the future.

7.2.2. Documentation

Documenting code properly is one of the most important aspects of sustained development.

Read it again.

Most code in research software like Yade is not only used, but also read, by developers or even by regular users. Therefore, when adding new class, always mention the following in the documentation:

- purpose
- details of the functionality, unless obvious (algorithms, internal logic)
- limitations (by design, by implementation), bugs
- bibliographical reference, if using non-trivial published algorithms (see below)
- references to other related classes
- hyperlinks to bugs, blueprints, wiki or mailing list about this particular feature.

As much as it is meaningful, you should also

- update any other documentation affected
- provide a simple python script demonstrating the new functionality in **scripts/test**.

Historically, Yade was using Doxygen for in-source documentation. This documentation is still available (by running **scons doc**), but was rarely written and used by programmers, and had all the disadvantages of auto-generated documentation. Then, as Python became ubiquitous in yade, python was documented using epydoc generator. Finally, hand-written documentation (this one) started to be written using Sphinx, which was developed originally for documenting Python itself. Disadvantages of the original scatter were different syntaxes, impossibility for cross-linking, non-interactivity and frequently not being up-to-date.

7.2.2.1. Sphinx documentation

Most c++ classes are wrapped in Python, which provides good introspection and interactive documentation (try writing **Material?** in the ipython prompt; or **help(CpmState)**).

Syntax of documentation is [rest] (reStructuredText, see reStructuredText Primer). It is the same for c++ and python code.

- Documentation of c++ classes exposed to python is given as 3rd argument to *YADE_CLASS_BASE_DOC_* * *macro family* introduced below.
- Python classes/functions are documented using regular python docstrings. Besides explaining functionality, meaning and types of all arguments should also be documented. Short pieces of code might be very helpful. See the utils module for an example.

In addition to standard ReST syntax, yade provides several shorthand macros:

:yref: creates hyperlink to referenced term, for instance:

```
:yref:`CpmMat`
```

becomes CpmMat; link name and target can be different:

```
:yref:`Material used in the CPM model<CpmMat>`
```

yielding Material used in the CPM model.

:ysrc: creates hyperlink to file within the source tree (to its latest version in the repository), for instance

core/Cell.hpp. Just like with *:yref:*, alternate text can be used with

```
:ysrc:`Link text<target/file>`
```

like this.

|ycomp| is used in attribute description for those that should not be provided by the used, but are auto-computed instead; **|ycomp|** expands to *(auto-computed)*.

|yupdate| marks attributes that are periodically update, being subset of the previous. **|yupdate|** expands to *(auto-updated)*.

$...$ delimits inline math expressions; they will be replaced by:

```
:math:`...`
```

and rendered via LaTeX. To write a single dollar sign, escape it with backslash **\$**.

Displayed mathematics (standalone equations) can be inserted as explained in Math support in Sphinx.

7.2.2.2. Bibliographical references

As in any scientific documentation, references to publications are very important. To cite an article, add it to BibTeX file in doc/references.bib, using the BibTeX format. Please adhere to the following conventions:

1. Keep entries in the form **Author2008** (**Author** is the first author), **Author2008b** etc if multiple articles from one author;
2. Try to fill mandatory fields for given type of citation;
3. Do not use **\'{i}** funny escapes for accents, since they will not work with the HTML output; put everything in straight utf-8.

In your docstring, the **Author2008** article can be cited by **[Author2008]_**; for example:

```
According to [Allen1989]_, the integration scheme …
```

will be rendered as

According to [1], the intergration scheme ...

7.2.2.3. Separate class/function documentation

Some c++ might have long or content-rich documentation, which is rather inconvenient to type in the c++ source itself as string literals. Yade provides a way to write documentation separately in py/_extraDocs.py file: it is executed after loading c++ plugins and can set **___doc___** attribute of any object directly, overwriting docstring from c++. In such (exceptional) cases:

1. Provide at least a brief description of the class in the c++ code nevertheless, for people only reading the code.
2. Add notice saying "This class is documented in detail in the py/_extraDocs.py file".
3. Add documentation to py/_etraDocs.py in this way:

```
module.YourClass.__doc__='''
        This is the docstring for YourClass.

        Class, methods and functions can be documented this way.
```

```
        .. note:: It can use any syntax features you like.

'''
```

Note: Boost::python embeds function signatures in the docstring (before the one provided by the user). Therefore, before creating separate documentation of your function, have a look at its **__doc__**-attribute and copy the first line (and the blank lie afterwards) in the separate docstring. The first line is then used to create the function signature (arguments and return value).

7.2.2.4. Local documentation

Note: At some future point, this documentation will be integrated into yade's sources. This section should be updated accordingly in that case.

To generate Yade's documentation locally, get a copy of the ydoc branch via bzr, then follow instructions in the README file.

7.2.2.5. Internal c++ documentation

[doxygen] was used for automatic generation of c++ code. Since user-visible classes are defined with sphinx now, it is not meaningful to use doxygen to generate overall documentation. However, take care to document well internal parts of code using regular comments, including public and private data members.

7.3. Support framework

Besides the framework provided by the c++ standard library (including STL), boost and other dependencies, yade provides its own specific services.

7.3.1. Pointers

7.3.1.1. Shared pointers

Yade makes extensive use of shared pointers **shared_ptr**. [3] Although it probably has some performance impacts, it greatly simplifies memory management, ownership management of c++ objects in python and so forth. To obtain raw pointer from a **shared_ptr**, use its **get()** method; raw pointers should be used in case the object will be used only for short time (during a function call, for instance) and not stored anywhere.

Python defines thin wrappers for most c++ Yade classes (for all those registered with *YADE_CLASS_BASE_DOC_* macro family* and several others), which can be constructed from **shared_ptr**; in this way, Python reference counting blends with the **shared_ptr** reference counting model, preventing crashes due to python objects pointing to c++ objects that were destructed in the meantime.

7.3.1.2. Typecasting

Frequently, pointers have to be typecast; there is choice between static and dynamic casting.

[3] Either **boost::shared_ptr** or **tr1::shared_ptr** is used, but it is always imported with the **using** statement so that unqualified **shared_ptr** can be used.

- **dynamic_cast** (**dynamic_pointer_cast** for a **shared_ptr**) assures cast admissibility by checking runtime type of its argument and returns NULL if the cast is invalid; such check obviously costs time. Invalid cast is easily caught by checking whether the pointer is NULL or not; even if such check (e.g. **assert**) is absent, dereferencing NULL pointer is easily spotted from the stacktrace (debugger output) after crash. Moreover, **shared_ptr** checks that the pointer is non-NULL before dereferencing in debug build and aborts with "Assertion 'px!=0' failed." if the check fails.

- **static_cast** is fast but potentially dangerous (**static_pointer_cast** for **shared_ptr**). Static cast will return non-NULL pointer even if types don't allow the cast (such as casting from **State*** to **Material***); the consequence of such cast is interpreting garbage data as instance of the class cast to, leading very likely to invalid memory access (segmentation fault, "crash" for short).

To have both speed and safety, Yade provides 2 macros:

YADE_CAST expands to **static_cast** in optimized builds and to **dynamic_cast** in debug builds.

YADE_PTR_CAST expands to **static_pointer_cast** in optimized builds and to **dynamic_pointer_cast** in debug builds.

7.3.2. Basic numerics

The floating point type to use in Yade Real, which is by default typedef for **double**. [4]

Yade uses the Eigen library for computations. It provides classes for 2d and 3d vectors, quaternions and 3x3 matrices templated by number type; their specialization for the **Real** type are typedef'ed with the "r" suffix, and occasionally useful integer types with the "i" suffix:

- **Vector2r**, **Vector2i**
- **Vector3r**, **Vector3i**
- **Quaternionr**
- **Matrix3r**

Yade additionally defines a class named Se3r, which contains spatial position (**Vector3r Se3r::position**) and orientation (**Quaternionr Se3r::orientation**), since they are frequently used one with another, and it is convenient to pass them as single parameter to functions.

Eigen provides full rich linear algebra functionality. Some code firther uses the [cgal] library for computational geometry.

In Python, basic numeric types are wrapped and imported from the **miniEigen** module; the types drop the **r** type qualifier at the end, the syntax is otherwise similar. **Se3r** is not wrapped at all, only converted automatically, rarely as it is needed, from/to a **(Vector3,Quaternion)** tuple/list.

```
# cross product
Yade [61]: Vector3(1,2,3).cross(Vector3(0,0,1))
 ->  [61]: Vector3(2,-1,0)

# construct quaternion from axis and angle
Yade [63]: Quaternion(Vector3(0,0,1),pi/2)
 ->  [63]: Quaternion((0,0,1),1.5707963267948966)
```

Note: Quaternions are internally stored as 4 numbers. Their usual human-readable representation is, however, (normalized) axis and angle of rotation around that axis, and it is also how they are input/output in Python. Raw internal values can be accessed using the **[0]** ... **[3]** element access (or **.W()**, **.X()**, **.Y()** and **.Z()** methods), in both c++ and Python.

[4] Historically, it was thought that Yade could be also run with single precision based on build-time parameter; it turned out however that the impact on numerical stability was such disastrous that this option is not available now. There is, however, **QUAD_PRECISION** parameter to scons, which will make **Real** a typedef for **long double** (extended precision; quad precision in the proper sense on IA64 processors); this option is experimental and is unlikely to be used in near future, though.

7.3.3. Run-time type identification (RTTI)

Since serialization and dispatchers need extended type and inheritance information, which is not sufficiently provided by standard RTTI. Each yade class is therefore derived from Factorable and it must use macro to override its virtual functions providing this extended RTTI:

YADE_CLASS_BASE_DOC(Foo,Bar Baz,"Docstring) creates the following virtual methods (mediated via the **REGISTER_CLASS_AND_BASE** macro, which is not user-visible and should not be used directly):

- **std::string getClassName()** returning class name (**Foo**) as string. (There is the **typeid(instanceOrType).name()** standard c++ construct, but the name returned is compiler-dependent.)
- **unsigned getBaseClassNumber()** returning number of base classes (in this case, 2).
- **std::string getBaseClassName(unsigned i=0)** returning name of i-th base class (here, **Bar** for i=0 and **Baz** for i=1).

> **Warning:** RTTI relies on virtual functions; in order for virtual functions to work, at least one virtual method must be present in the implementation (**.cpp**) file. Otherwise, virtual method table (vtable) will not be generated for this class by the compiler, preventing virtual methods from functioning properly.

Some RTTI information can be accessed from python:

```
Yade [65]: yade.system.childClasses('Shape')
 ->  [65]: set(['Box', 'Facet', 'Sphere', 'Tetra', 'Wall'])

Yade [66]: Sphere().name            ## getClassName()
 ->  [66]: 'Sphere'
```

7.3.4. Serialization

Serialization serves to save simulation to file and restore it later. This process has several necessary conditions:

- classes know which attributes (data members) they have and what are their names (as strings);
- creating class instances based solely on its name;
- knowing what classes are defined inside a particular shared library (plugin).

This functionality is provided by 3 macros and 2 virtual functions; details are provided below.

Serializable::preProcessAttributes *Optional* class virtual function. See *Attribute registration*.

Prepare attributes for being (de)serialized.

Serializable::postProcessAttributes *Optional* class virtual function.

Process attributes after being (de)serialized. See *Attribute registration*.

YADE_CLASS_BASE_DOC_* Inside the class declaration (i.e. in the **.hpp** file within the **class Foo { /* ... */};** block). See *Attribute registration*.

Enumerate class attributes that should be saved and loaded; associate each attribute with its literal name, which can be used to retrieve it. See *YADE_CLASS_BASE_DOC_* macro family*.

Additionally documents the class in python, adds methods for attribute access from python, and documents each attribute.

REGISTER_SERIALIZABLE In header file, but *after* the class declaration block. See *Class factory*.

Associate literal name of the class with functions that will create its new instance (ClassFactory).

YADE_PLUGIN In the implementation **.cpp** file. See *Plugin registration*.

Declare what classes are declared inside a particular plugin at time the plugin is being loaded (yade startup).

7.3.4.1. Attribute registration

All (serializable) types in Yade are one of the following:

- Type deriving from Serializable, which provide information on how to serialize themselves via overriding the Serializable::registerAttributes method; it declares data members that should be serialzed along with their literal names, by which they are identified. This method then invokes **registerAttributes** of its base class (until **Serializable** itself is reached); in this way, derived classes properly serialize data of their base classes.

  This funcionality is hidden behind the macro *YADE_CLASS_BASE_DOC_* * *macro family* used in class declaration body (header file), which takes base class and list of attributes:

  ```
  YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"class documentation",((type1,attribute1,initValue1,"Docum
  ```

  Note that attributes are encodes in double parentheses, not separated by commas. Empty attribute list can be given simply by **YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"documentation",)** (the last comma is mandatory), or by omiting **ATTRS** from macro name and last parameter altogether.

- Fundamental type: strings, various number types, booleans, Vector3r and others. Their "handlers" (serializers and deserializers) are defined in **lib/serialization**.

- Standard container of any serializable objects.

- Shared pointer to serializable object.

Currently, Yade relies on its own serialization system (in **lib/serialization** and **lib/serialization-xml**), but there are plans to use boost::serialization instead. This implementation detail is hidden behind the helper macros for regular use.

Warning: Yade's serialization system lacks some functionality, notably
- tracking shared pointers
- serialization of some containers (std::map or std::pair, for instance).

Such functionality must be explicitly emulated if desired, until Yade switches to boost::serialization.

Note: **YADE_CLASS_BASE_DOC_ATTRS** also generates code for attribute access from python; this will be discussed later. Since this macro serves both purposes, the consequence is that attributes that are serialized can always be accessed from python.

Yade also provides callback for before/after (de) serialization, virtual functions Serializable::preProcessAttributes and Serializable::postProcessAttributes, which receive one **bool deserializing** argument (**true** when deserializing, **false** when serializing). Their default implementation in Serializable doesn't do anything, but their typical use is:

- converting some non-serializable internal data structure of the class (such as multi-dimensional array, hash table, array of pointers) into a serializable one (pre-processing) and fill this non-serializable structure back after deserialization (post-processing); for instance, InteractionContainer uses these hooks to ask its concrete implementation to store its contents to a unified storage (**vector<shared_ptr<Interaction> >**) before serialization and to restore from it after deserialization.

- precomputing non-serialized attributes from the serialized values; e.g. Facet computes its (local) edge normals and edge lengths from vertices' coordinates.

7.3.4.2. Class factory

Each serializable class must use **REGISTER_SERIALIZABLE**, which defines function to create that class by **ClassFactory**. **ClassFactory** is able to instantiate a class given its name (as string), which is necessary for deserialization.

Although mostly used internally by the serialization framework, programmer can ask for a class instantiation using **shared_ptr<Factorable> f=ClassFactory::instance().createShared("ClassName");**, casting the returned **shared_ptr<Factorable>** to desired type afterwards. Serializable itself derives from Factorable, i.e. all serializable types are also factorable (It is possible that different mechanism will be in place if boost::serialization is used, though.)

7.3.4.3. Plugin registration

Yade loads dynamic libraries containing all its functionality at startup. ClassFactory must be taught about classes each particular file provides. **YADE_PLUGIN** serves this purpose and, contrary to *YADE_CLASS_BASE_DOC_* macro family*, must be place in the implementation (.cpp) file. It simple enumerates classes that are provided by this file:

```
YADE_PLUGIN((ClassFoo)(ClassBar));
```

Note: You must use parentheses around the class name even if there is only one (preprocessor limitation): **YADE_PLUGIN((classFoo));**. If there is no class in this file, do not use this macro at all.

Internally, this macro creates function **registerThisPluginClasses_** declared specially as **__attribute__((constructor))** (see GCC Function Attributes); this attributes makes the function being executed when the plugin is loaded via **dlopen** from **ClassFactory::load(...)**. It registers all factorable classes from that file in the *Class factory*.

Note: Classes that do not derive from **Factorable**, such as **Shop** or **SpherePack**, are not declared with **YADE_PLUGIN**.

This is an example of a serializable class header:

```
/*! Homogeneous gravity field; applies gravity×mass force on all bodies. */
class GravityEngine: public GlobalEngine{
        public:
                virtual void action();
        // registering class and its base for the RTTI system
        YADE_CLASS_BASE_DOC_ATTRS(GravityEngine,GlobalEngine,
                // documentation visible from python and generated reference documentation
                "Homogeneous gravity field; applies gravity×mass force on all bodies.",
                // enumerating attributes here, include documentation
                ((Vector3r,gravity,Vector3r::ZERO,"acceleration, zero by default [kgms ⁻²]"))
        );
};
// registration function for ClassFactory
REGISTER_SERIALIZABLE(GravityEngine);
```

and this is the implementation:

```
#include<yade/pkg-common/GravityEngine.hpp>
#include<yade/core/Scene.hpp>

// registering the plugin
```

```
YADE_PLUGIN((GravityEngine));

void GravityEngine::action(){
        /* do the work here */
}
```

We can create a mini-simulation (with only one GravityEngine):

```
Yade [67]: O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]

Yade [68]: O.save('abc.xml')
```

and the XML looks like this:

```
<Yade>
        <scene _className_="Scene"  tags="[author=V??clav~??milauer~(vaclav@flux) isoTime=20100531T100722 id=201
                <grpRelationData />
                <engines size="1">
                        <engines _className_="GravityEngine"  label="" gravity="{0 0 -9.8100000000000005}" />
                </engines>
                <initializers size="0" />
                <bodies _className_="BodyContainer" >
                        <body size="0" />
                </bodies>
                <interactions _className_="InteractionContainer"  serializeSorted="0">
                        <interaction size="0" />
                </interactions>
                <materials size="0" />
                <miscParams size="0" />
                <dispParams size="0" />
                <cell _className_="Cell"  refSize="{1 1 1}" trsf="{1 0 0 0 1 0 0 0 1}" velGrad="{0 0 0 0 0 0 0 0
        </scene>
</Yade>
```

Warning: Since XML files closely reflect implementation details of Yade, they will not be compatible between different versions. Use them only for short-term saving of scenes. Python is *the* high-level description Yade uses.

7.3.4.4. Python attribute access

The macro *YADE_CLASS_BASE_DOC_* * *macro family* introduced above is (behind the scenes) also used to create functions for accessing attributes from Python. As already noted, set of serialized attributes and set of attributes accessible from Python are identical. Besides attribute access and the **[]** operator access, these wrapper classes imitate also other functionality of regular python dictionaries:

```
Yade [69]: s=Sphere()

Yade [70]: s.radius              ## read-access
 ->  [70]: nan

Yade [71]: s.radius=4.           ## write access

Yade [72]: s.keys()              ## show all available keys
 ->  [72]: ['radius', 'color', 'wire', 'highlight']

Yade [73]: for k in s.keys(): print s[k]  ## iterate over keys, print their values
     ....:
4.0
Vector3(1,1,1)
False
```

```
False

Yade [74]: s.has_key('radius')            ## same as: 'radius' in s.keys()
 ->  [74]: True

Yade [75]: s.dict()                       ## show dictionary of both attributes and values
 ->  [75]: {'color': Vector3(1,1,1), 'highlight': False, 'radius': 4.0, 'wire': False}

## only very rarely needed; calls Serializable::postProcessAttributes(bool deserializing):
Yade [77]: s.postProcessAttributes(False)
```

7.3.5. YADE_CLASS_BASE_DOC_* macro family

There is several macros that hide behind them the functionality of *Sphinx documentation*, *Run-time type identification (RTTI)*, *Attribute registration*, *Python attribute access*, plus automatic attribute initialization and documentation. They are all defined as shorthands for base macro **YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY** with some arguments left out. They must be placed in class declaration's body (**.hpp** file):

```
#define YADE_CLASS_BASE_DOC(klass,base,doc) \
        YADE_CLASS_BASE_DOC_ATTRS(klass,base,doc,)
#define YADE_CLASS_BASE_DOC_ATTRS(klass,base,doc,attrs) \
        YADE_CLASS_BASE_DOC_ATTRS_CTOR(klass,base,doc,attrs,)
#define YADE_CLASS_BASE_DOC_ATTRS_CTOR(klass,base,doc,attrs,ctor) \
        YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(klass,base,doc,attrs,ctor,)
#define YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(klass,base,doc,attrs,ctor,py) \
        YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,,ctor,py)
#define YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,init,ctor,py) \
        YADE_CLASS_BASE_DOC_ATTRS_DEPREC_INIT_CTOR_PY(klass,base,doc,attrs,,init,ctor,py)
```

Expected parameters are indicated by macro name components separated with underscores. Their meaning is as follows:

klass (unquoted) name of this class (used for RTTI and python)

base (unquoted) name of the base class (used for RTTI and python)

doc docstring of this class, written in the ReST syntax. This docstring will appear in generated documentation (such as CpmMat). It can be as long as necessary, but sequences interpreted by c++ compiler must be properly escaped (therefore some backslashes must be doubled, like in $\sigma = \epsilon E$:

```
":math:`\\sigma=\\epsilon E"
```

Use **\n** and **\t** for indentation inside the docstring. Hyperlink the documentation abundantly with **yref** (all references to other classes should be hyperlinks).

See *Sphinx documentation* for syntax details.

attrs Attribute must be written in the form of parethesized list:

```
((type1,attr1,initValue1,"Attribute 1 documentation"))
((type2,attr2,,"Attribute 2 documentation"))  // initValue unspecified
```

This will expand to

1. data members declaration in c++ (note that all attributes are *public*):

```
public: type1 attr1;
        type2 attr2;
```

2. Initializers of the default (argument-less) constructor, for attributes that have non-empty **initValue**:

```
Klass(): attr1(initValue1), attr2() { /* constructor body */ }
```

   No initial value will be assigned for attribute of which initial value is left empty (as is for attr2 in the above example). Note that you still have to write the commas.

3. Registration of the attribute in the serialization system

4. **Registration of the attribute in python, so that it can be accessed as klass().name1.** The attribute will be read-write; to avoid this, override it by read-only attribute of the same name in the py section (see below).

   This attribute will carry the docstring provided, along with knowledge of the initial value. You can add text description to the default value using the comma operator of c++ and casting the char* to (void):

```
((Real,dmgTau,((void)"deactivated if negative",-1),"Characteristic time for normal viscosit
```

   leading to CpmMat::dmgTau.

   The attribute is registered via **boost::python::add_property** specifying **return_by_value** policy rather than **return_internal_reference**, which is the default when using **def_readwrite**. The reason is that we need to honor custom converters for those values; see note in *Custom converters* for details.

deprec List of deprecated attribute names. The syntax is

```
((oldName1,newName1,"Explanation why renamed etc."))
((oldName2,newName2,"! Explanation why removed and what to do instaed."))
```

This will make accessing **oldName1** attribute *from Python* return value of **newName**, but displaying warning message about the attribute name change, displaying provided explanation. This happens whether the access is read or write.

If the explanation's first character is **!** (*bang*), the message will be displayed upon attribute access, but exception will be thrown immediately. Use this in cases where attribute is no longer meaningful or was not straightforwardly replaced by another, but more complex adaptation of user's script is needed. You still have to give **newName2**, although its value will never be used – you can use any variable you like, but something must be given for syntax reasons).

> **Warning:** Due to compiler limitations, this feature only works if Yade is compiled with gcc >= 4.4. In the contrary case, deprecated attribute functionality is disabled, even if such attributes are declared.

init Parethesized list of the form:

```
((attr3,value3)) ((attr4,value4))
```

which will be expanded to initializers in the default ctor:

```
Klass(): /* attributes declared with the attrs argument */ attr4(value4), attr5(value5) { /* constructo
```

The purpose of this argument is to make it possible to initialize constants and references (which are not declared as attributes using this macro themselves, but separately), as that cannot be done in constructor body. This argument is rarely used, though.

ctor will be put directly into the generated constructor's body. Mostly used for calling createIndex(); in the constructor.

Note: The code must not contain commas ouside parentheses (since preprocessor uses commas to separate macro arguments). If you need complex things at construction time, create a separate init() function and call it from the constructor instead.

py will be appeneded directly after generated python code that registers the class and all its attributes. You can use it to make accessible data member which you do not want to be serialized, or to override an already-existing attribute of the same name in order to make it read-only from python (taken from CpmPhys):

```
.def_readonly("omega",&CpmPhys::omega,"Damage internal variable")
.def_readonly("Fn",&CpmPhys::Fn,"Magnitude of normal force.")
```

def_readonly will not work for custom types (such as std::vector), as it bypasses conversion registry; see *Custom converters* for details.

Changing some attributes might render other data within the object inconsistent (cache coherency); for instance, Cell must update Cell.size each time Cell.trsf is updated, since Cell.size is not computed every time, but cached value is used instead. In that case, getter and setter functions must be defined, which will, besides getting/setting value of the attribute itself also run appropriate cache update function; we override refSize that was already registered; **integrateAndUpdate(0)** triggers cache update in this case:

```
class Cell{
public:
        Vector3r getRefSize(){ return refSize; }
        void setRefSize(const Vector3r& s){ refSize=s; integrateAndUpdate(0); }
YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(Cell,Serializable,"doc",
        /* … */
        ((Vector3r,refSize,Vector3r(1,1,1),"[will be overridden below]")),
        /* ctor */,
        /* py */
                .add_property("refSize",&Cell::getRefSize,&Cell::setRefSize,"Reference size of the cell.
        );
};
```

7.3.5.1. Static attributes

Some classes (such as OpenGL functors) are instantiated automatically; since we want their attributes to be persistent throughout the session, they are static. To expose class with static attributes, use the **YADE_CLASS_BASE_DOC_STATICATTRS** macro. Attribute syntax is the same as for **YADE_CLASS_BASE_DOC_ATTRS**:

```
class SomeClass: public BaseClass{
        YADE_CLASS_BASE_DOC_STATICATTRS(SomeClass,BaseClass,"Documentation of SomeClass",
                ((Type1,attr1,default1,"doc for attr1"))
                ((Type2,attr2,default2,"doc for attr2"))
        );
};
```

additionally, you *have* to allocate memory for static data members in the **.cpp** file (otherwise, error about undefined symbol will appear when the plugin is loaded):

There is no way to expose class that has both static and non-static attributes using **YADE_CLASS_BASE_*** macros. You have to expose non-static attributes normally and wrap static attributes separately in the **py** parameter.

7.3.5.2. Returning attribute by value or by reference

When attribute is passed from c++ to python, it can be passed either as

- value: new python object representing the original c++ object is constructed, but not bound to it; changing the python object doesn't modify the c++ object, unless explicitly assigned back to it, where inverse conversion takes place and the c++ object is replaced.
- reference: only reference to the underlying c++ object is given back to python; modifying python object will make the c++ object modified automatically.

The way of passing attributes given to **YADE_CLASS_BASE_DOC_ATTRS** in the **attrs** parameter is determined automatically in the following manner:

- **Vector3, Vector3i, Vector2, Vector2i, Matrix3 and Quaternion objects are passed by *reference*. For instance::**
  O.bodies[0].state.pos[0]=1.33

  will assign correct value to **x** component of position, without changing the other ones.
- **Yade classes (all that use shared_ptr when declared in python: all classes deriving from Serializable declared**
  O.engines[4].damping=.3

  will change damping parameter on the original engine object, not on its copy.
- **All other types are passed by *value*. This includes, most importantly, sequence types declared in *Custom con***
  O.engines[4]=NewtonIntegrator()

  will *not* work as expected; it will replace 5th element of a *copy* of the sequence, and this change will not propagate back to c++.

7.3.6. Multiple dispatch

Multiple dispatch is generalization of virtual methods: a Dispatcher decides based on type(s) of its argument(s) which of its Functors to call. Numer of arguments (currently 1 or 2) determines *arity* of the dispatcher (and of the functor): unary or binary. For example:

```
BoundDispatcher([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()])
```

creates dispatcher BoundDispatcher (a Dispatcher1D), with 2 functors; they receive **Sphere** or **Facet** instances and create **Aabb**. This code would look like this in c++:

```
shared_ptr<BoundDispatcher> boundDispatcher=shared_ptr<BoundDispatcher>(new BoundDispatcher);
boundDispatcher->add(new Bo1_Sphere_Aabb());
boundDispatcher->add(new Bo1_Facet_Aabb());
```

There are currenly 4 predefined dispatchers (see dispatcher-names) and corresponding functor types. They are inherit from template instantiations of **Dispatcher1D** or **Dispatcher2D** (for functors, **Functor1D** or **Functor2D**). These templates themselves derive from **DynlibDispatcher** (for dispatchers) and **FunctorWrapper** (for functors).

7.3.6.1. Example: InteractionGeometryDispatcher

Let's take (the most complicated perhaps) InteractionGeometryDispatcher. InteractionGeometryFunctor, which is dispatched based on types of 2 Shape instances (a Functor2D), takes a number of arguments and returns bool. The functor "call" is always provided by its overridden **Functor::go** method; it always receives the dispatched instances as first argument(s) (2 × **const shared_ptr<Shape>&**) and a number of other arguments it needs:

```
class InteractionGeometryFunctor: public Functor2D<
   bool,                                  //return type
   TYPELIST_7(const shared_ptr<Shape>&,   // 1st class for dispatch
      const shared_ptr<Shape>&,           // 2nd class for dispatch
      const State&,                       // other arguments passed to ::go
      const State&,                       // …
```

```
      const Vector3r&,                    // …
      const bool&,                        // …
      const shared_ptr<Interaction>&      // …
   )
>
```

The dispatcher is declared as follows:

```
class InteractionGeometryDispatcher: public Dispatcher2D<
   Shape,                        // 1st class for dispatch
   Shape,                        // 2nd class for dispatch
   InteractionGeometryFunctor,   // functor type
   bool,                         // return type of the functor

   // follow argument types for functor call
   // they must be exactly the same as types
   // given to the InteractionGeometryFunctor above.
   TYPELIST_7(const shared_ptr<Shape>&,
      const shared_ptr<Shape>&,
      const State&,
      const State&,
      const Vector3r&,
      const bool &,
      const shared_ptr<Interaction>&
   ),

   // handle symetry automatically
   // (if the dispatcher receives Sphere+Facet,
   // the dispatcher might call functor for Facet+Sphere,
   // reversing the arguments)
   false
>
{ /* … */ }
```

Functor derived from InteractionGeometryFunctor must then

- override the ::go method with appropriate arguments (they must match exactly types given to **TYPELIST_*** macro);
- declare what types they should be dispatched for, and in what order if they are not the same.

```
class Ig2_Facet_Sphere_Dem3DofGeom: public InteractionGeometryFunctor{
   public:

   // override the InteractionGeometryFunctor::go
   //   (it is really inherited from FunctorWrapper template,
   //    therefore not declare explicitly in the
   //    InteractionGeometryFunctor declaration as such)
   // since dispatcher dispatches only for declared types
   //   (or types derived from them), we can do
   //   static_cast<Facet>(shape1) and static_cast<Sphere>(shape2)
   //   in the ::go body, without worrying about types being wrong.
   virtual bool go(
      // objects for dispatch
      const shared_ptr<Shape>& shape1, const shared_ptr<Shape>& shape2,
      // other arguments
      const State& state1, const State& state2, const Vector3r& shift2,
      const bool& force, const shared_ptr<Interaction>& c
   );
   /* … */

   // this declares the type we want to be dispatched for, matching
   //   first 2 arguments to ::go and first 2 classes in TYPELIST_7 above
```

```
   //   shape1 is a Facet and shape2 is a Sphere
   //   (or vice versa, see lines below)
   FUNCTOR2D(Facet,Sphere);

   // declare how to swap the arguments
   //   so that we can receive those as well
   DEFINE_FUNCTOR_ORDER_2D(Facet,Sphere);
   /* … */
};
```

7.3.6.2. Dispatch resolution

The dispatcher doesn't always have functors that exactly match the actual types it receives. In the same way as virtual methods, it tries to find the closest match in such way that:

1. the actual instances are derived types of those the functor accepts, or exactly the accepted types;
2. sum of distances from actual to accepted types is sharp-minimized (each step up in the class hierarchy counts as 1)

If no functor is able to accept given types (first condition violated) or multiple functors have the same distance (in condition 2), an exception is thrown.

This resolution mechanism makes it possible, for instance, to have a hierarchy of Dem3DofGeom classes (for different combination of shapes: Dem3DofGeom_SphereSphere, Dem3DofGeom_FacetSphere, Dem3DofGeom_WallSphere), but only provide a LawFunctor accepting **Dem3DofGeom**, rather than having different laws for each shape combination.

Note: Performance implications of dispatch resolution are relatively low. The dispatcher lookup is only done once, and uses fast lookup matrix (1D or 2D); then, the functor found for this type(s) is cached within the **Interaction** (or **Body**) instance. Thus, regular functor call costs the same as dereferencing pointer and calling virtual method. There is blueprint to avoid virtual function call as well.

Note: At the beginning, the dispatch matrix contains just entries exactly matching given functors. Only when necessary (by passing other types), appropriate entries are filled in as well.

7.3.6.3. Indexing dispatch types

Classes entering the dispatch mechanism must provide for fast identification of themselves and of their parent class. [5] This is called class indexing and all such classes derive from Indexable. There are **top-level** Indexables (types that the dispatchers accept) and each derived class registers its index related to this top-level Indexable. Currently, there are:

Top-level Indexable	used by
Shape	BoundFunctor, InteractionGeometryDispatcher
Material	InteractionPhysicsDispatcher
InteractionPhysics	LawDispatcher
InteractionGeometry	LawDispatcher

The top-level Indexable must use the REGISTER_INDEX_COUNTER macro, which sets up the machinery for identifying types of derived classes; they must then use the **REGISTER_CLASS_INDEX** macro *and* call **createIndex()** in their constructor. For instance, taking the Shape class (which is a top-level Indexable):

[5] The functionality described in *Run-time type identification (RTTI)* serves a different purpose (serialization) and would hurt the performance here. For this reason, classes provide numbers (indices) in addition to strings.

```
// derive from Indexable
class Shape: public Serializable, public Indexable {
   // never call createIndex() in the top-level Indexable ctor!
   /* … */

   // allow index registration for classes deriving from ``Shape``
   REGISTER_INDEX_COUNTER(Shape);
};
```

Now, all derived classes (such as Sphere or Facet) use this:

```
class Sphere: public Shape{
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Sphere,Shape,"docstring",
      ((Type1,attr1,default1,"docstring1"))
      /* … */,
      // this is the CTOR argument
         // important; assigns index to the class at runtime
         createIndex();
   );
   // register index for this class, and give name of the immediate parent class
   //    (i.e. if there were a class deriving from Sphere, it would use
   //     REGISTER_CLASS_INDEX(SpecialSphere,Sphere),
   //     not REGISTER_CLASS_INDEX(SpecialSphere,Shape)!)
   REGISTER_CLASS_INDEX(Sphere,Shape);
};
```

At runtime, each class within the top-level Indexable hierarchy has its own unique numerical index. These indices serve to build the dispatch matrix for each dispatcher.

7.3.6.4. Inspecting dispatch in python

If there is a need to debug/study multiple dispatch, python provides convenient interface for this low-level functionality.

We can inspect indices with the **dispIndex** property (note that the top-level indexable **Shape** has negative (invalid) class index; we purposively didn't call **createIndex** in its constructor):

```
Yade [78]: Sphere().dispIndex, Facet().dispIndex, Wall().dispIndex
 ->  [78]: (2, 1, 4)

Yade [79]: Shape().dispIndex                    # top-level indexable
 ->  [79]: -1
```

Dispatch hierarchy for a particular class can be shown with the **dispHierarchy()** function, returning list of class names: 0th element is the instance itself, last element is the top-level indexable (again, with invalid index); for instance:

```
Yade [80]: Dem3DofGeom().dispHierarchy()       # parent class of all other Dem3DofGeom_ classes
 ->  [80]: ['Dem3DofGeom', 'InteractionGeometry']

Yade [81]: Dem3DofGeom_SphereSphere().dispHierarchy(), Dem3DofGeom_FacetSphere().dispHierarchy(), Dem3DofGeom_Wa
 ->  [81]:
(['Dem3DofGeom_SphereSphere', 'Dem3DofGeom', 'InteractionGeometry'],
 ['Dem3DofGeom_FacetSphere', 'Dem3DofGeom', 'InteractionGeometry'],
 ['Dem3DofGeom_WallSphere', 'Dem3DofGeom', 'InteractionGeometry'])

Yade [82]: Dem3DofGeom_WallSphere().dispHierarchy(names=False)   # show numeric indices instead
 ->  [82]: [4, 1, -1]
```

Dispatchers can also be inspected, using the .dispMatrix() method:

```
Yade [83]: ig=InteractionGeometryDispatcher([
     ....:    Ig2_Sphere_Sphere_Dem3DofGeom(),
     ....:    Ig2_Facet_Sphere_Dem3DofGeom(),
     ....:    Ig2_Wall_Sphere_Dem3DofGeom()
     ....: ])

Yade [88]: ig.dispMatrix()
 ->  [88]:
{('Facet', 'Sphere'): 'Ig2_Facet_Sphere_Dem3DofGeom',
 ('Sphere', 'Facet'): 'Ig2_Facet_Sphere_Dem3DofGeom',
 ('Sphere', 'Sphere'): 'Ig2_Sphere_Sphere_Dem3DofGeom',
 ('Sphere', 'Wall'): 'Ig2_Wall_Sphere_Dem3DofGeom',
 ('Wall', 'Sphere'): 'Ig2_Wall_Sphere_Dem3DofGeom'}

Yade [89]: ig.dispMatrix(False)          # don't convert to class names
 ->  [89]:
{(1, 2): 'Ig2_Facet_Sphere_Dem3DofGeom',
 (2, 1): 'Ig2_Facet_Sphere_Dem3DofGeom',
 (2, 2): 'Ig2_Sphere_Sphere_Dem3DofGeom',
 (2, 4): 'Ig2_Wall_Sphere_Dem3DofGeom',
 (4, 2): 'Ig2_Wall_Sphere_Dem3DofGeom'}
```

We can see that functors make use of symmetry (i.e. that Sphere+Wall are dispatched to the same functor as Wall+Sphere).

Finally, dispatcher can be asked to return functor suitable for given argument(s):

```
Yade [90]: ld=LawDispatcher([Law2_Dem3DofGeom_CpmPhys_Cpm()])

Yade [91]: ld.dispMatrix()
 ->  [91]: {('Dem3DofGeom', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm'}

# see how the entry for Dem3DofGeom_SphereSphere will be filled after this request
Yade [93]: ld.dispFunctor(Dem3DofGeom_SphereSphere(),CpmPhys())
 ->  [93]: <Law2_Dem3DofGeom_CpmPhys_Cpm instance at 0x2cc8170>

Yade [94]: ld.dispMatrix()
 ->  [94]:
{('Dem3DofGeom', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm',
 ('Dem3DofGeom_SphereSphere', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm'}
```

7.3.6.5. OpenGL functors

OpenGL rendering is being done also by 1D functors (dispatched for the type to be rendered). Since it is sufficient to have exactly one class for each rendered type, the functors are found automatically. Their base functor types are **GlShapeFunctor**, **GlBoundFunctor**, **GlInteractionGeometryFunctor** and so on. These classes register the type they render using the RENDERS macro:

```
class Gl1_Sphere: public GlShapeFunctor {
   public :
      virtual void go(const shared_ptr<Shape>&,
         const shared_ptr<State>&,
         bool wire,
         const GLViewInfo&
      );
   RENDERS(Sphere);
   YADE_CLASS_BASE_DOC_STATICATTRS(Gl1_Sphere,GlShapeFunctor,"docstring",
      ((Type1,staticAttr1,informativeDefault,"docstring"))
      /* … */
   );
};
```

```
REGISTER_SERIALIZABLE(Gl1_Sphere);
```

You can list available functors of a particular type by querying child classes of the base functor:

```
Yade [96]: yade.system.childClasses('GlShapeFunctor')
 ->  [96]: set(['Gl1_Box', 'Gl1_Facet', 'Gl1_Sphere', 'Gl1_Tetra', 'Gl1_Wall'])
```

Note: OpenGL functors may disappear in the future, being replaced by virtual functions of each class that can be rendered.

7.3.7. Parallel execution

Yade was originally not designed with parallel computation in mind, but rather with maximum flexibility (for good or for bad). Parallel execution was added later; in order to not have to rewrite whole Yade from scratch, relatively non-instrusive way of parallelizing was used: [OpenMP]. OpenMP is standartized shared-memory parallel execution environment, where parallel sections are marked by special **#pragma** in the code (which means that they can compile with compiler that doesn't support OpenMP) and a few functions to query/manipulate OpenMP runtime if necessary.

There is parallelism at 3 levels:

- Computation, interaction (python, GUI) and rendering threads are separate. This is done via regular threads (boost::threads) and is not related to OpenMP.
- ParallelEngine can run multiple engine groups (which are themselves run serially) in parallel; it rarely finds use in regular simulations, but it could be used for example when coupling with an independent expensive computation:

  ```
  ParallelEngine([
          [Engine1(),Engine2()],    # Engine1 will run before Engine2
          [Engine3()]               # Engine3() will run in parallel with the group [Engine1(),En
                                    # arbitrary number of groups can be used
  ])
  ```

  Engine2 will be run after **Engine1**, but in parallel with **Engine3**.

  > **Warning:** It is your reponsibility to avoid concurrent access to data when using ParallelEngine. Make sure you understand *very well* what the engines run in parallel do.

- Parallelism inside Engines. Some loops over bodies or interactions are parallelized (notably InteractionDispatchers and NewtonIntegrator, which are treated in detail later (FIXME: link)):

  ```
  #pragma omp parallel for
  for(long id=0; id<size; id++){
     const shared_ptr<Body>& b(scene->bodies[id]);
     /* … */
  }
  ```

  Note: OpenMP requires loops over contiguous range of integers (OpenMP 3 also accepts containers with random-access iterators).

  If you consider running parallelized loop in your engine, always evalue its benefits. OpenMP has some overhead fo creating threads and distributing workload, which is proportionally more expensive if the loop body execution is fast. The results are highly hardware-dependent (CPU caches, RAM controller).

Maximum number of OpenMP threads is determined by the **OMP_NUM_THREADS** environment variable and is constant throughout the program run. Yade main program also sets this variable (before loading OpenMP libraries) if you use the **-j**/**--threads** option. It can be queried at runtime with the

omp_get_max_threads function.

At places which are susceptible of being accessed concurrently from multiple threads, Yade provides some mutual exclusion mechanisms, discussed elsewhere (FIXME):

- simultaneously writeable container for *ForceContainer*,
- mutex for Body::State.

7.3.8. Logging

Regardless of whether the *optional-libraries* log4cxx is used or not, yade provides logging macros. [6] If log4cxx is enabled, these macros internally operate on the local logger instance (named **logger**, but that is hidden for the user); if log4cxx is disabled, they send their arguments to standard error output (**cerr**).

Every class using logging should create logger using these 2 macros (they expand to nothing if log4cxx is not used):

DECLARE_LOGGER; in class declaration body (in the **.hpp** file); this declares static variable **logger**;

CREATE_LOGGER(ClassName); in the implementation file; it creates and initializes that static variable. The logger will be named **yade.ClassName**.

The logging macros are the following:

- **LOG_TRACE**, **LOG_DEBUG**, **LOG_INFO**, **LOG_WARN**, **LOG_ERROR**, **LOG_FATAL** (increasing severity); their argument is fed to the logger stream, hence can contain the << operation:

  ```
  LOG_WARN("Exceeded "<<maxSteps<<" steps in attempts to converge, the result returned will not
  ```

  Every log message is prepended filename, line number and function name; the final message that will appear will look like this:

  ```
  237763 WARN  yade.ViscosityIterSolver /tmp/yade/trunk/extra/ViscosityIterSolver.cpp:316 newton
  ```

  The **237763 WARN yade.ViscosityIterSolver** (microseconds from start, severity, logger name) is added by log4cxx and is completely configurable, either programatically, or by using file **~/.yade-$SUFFIX/logging.conf**, which is loaded at startup, if present (FIXME: see more etc user's guide)

- special tracing macros **TRVAR1**, **TRVAR2**, ... **TRVAR6**, which show both variable name and its value (there are several more macros defined inside **/lib/base/Logging.hpp**, but they are not generally in use):

  ```
  TRVAR3(var1,var2,var3);
  // will be expanded to:
  LOG_TRACE("var1="<<var1<<"; var2="<<var2<<"; var3="<<var3);
  ```

Note: For performance reasons, optimized builds eliminate **LOG_TRACE** and **LOG_DEBUG** from the code at preprocessor level.

Note: Builds without log4cxx (even in debug mode) eliminate **LOG_TRACE** and **LOG_DEBUG**. As there is no way to enable/disable them selectively, the log amount would be huge.

Python provides rudimentary control for the logging system in **yade.log** module (FIXME: ref to docs):

```
Yade [97]: from yade import log

Yade [98]: log.setLevel('InsertionSortCollider',log.DEBUG)  # sets logging level of the yade.InsertionSortColli
```

[6] Because of (seemingly?) no upstream development of log4cxx and a few problems it has, Yade will very likely move to the hypothetical **boost::logging** library once it exists. The logging code will not have to be changed, however, as the log4cxx logic is hidden behind these macros.

```
Yade [99]: log.setLevel('',log.WARN)                    # sets logging level of all yade.* loggers (they in
```

As of now, there is no python interface for performing logging into lo4cxx loggers themselves.

7.3.9. Timing

Yade provides 2 services for measuring time spent in different pars of the code. One has the granularity of engine and can be enabled at runtime. The other one is finer, but requires adjusting and recompiling the code being measured.

7.3.9.1. Per-engine timing

The coarser timing works by merely accumulating numebr of invocations and time (with the precision of the **clock_gettime** function) spent in each engine, which can be then post-processed by associated Python module **yade.timing**. There is a static bool variable controlling whether such measurements take place (disabled by default), which you can change

```
TimingInfo::enabled=True;            // in c++
```

```
O.timingEnabled=True                 ## in python
```

After running the simulation, **yade.timing.stats()** function will show table with the results and percentages:

```
Yade [100]: TriaxialTest(numberOfGrains=100).load()

Yade [101]: O.engines[0].label='firstEngine'   ## labeled engines will show by labels in the stats table

Yade [102]: import yade.timing;
Yade [103]: O.timingEnabled=True

Yade [104]: yade.timing.reset()                   ## not necessary if used for the first time

Yade [105]: O.run(50); O.wait()

Yade [106]: yade.timing.stats()
Name                                                    Count                Time            Rel. time
-------------------------------------------------------------------------------------------------------
"firstEngine"                                              50                63us                0.19%
BoundDispatcher                                             2                23us                0.07%
InsertionSortCollider                                      50              2245us                6.67%
InteractionDispatchers                                     50              1313us                3.90%
GlobalStiffnessTimeStepper                                  2                25us                0.08%
TriaxialCompressionEngine                                  50               367us                1.09%
TriaxialStateRecorder                                       3             29161us               86.61%
NewtonIntegrator                                           50               470us                1.40%
TOTAL                                                                      33669us              100.00%
```

Exec count and time can be accessed and manipulated through **Engine::timingInfo** from c++ or **Engine().execCount** and **Engine().execTime** properties in Python.

7.3.9.2. In-engine and in-functor timing

Timing within engines (and functors) is based on TimingDeltas class. It is made for timing loops (functors' loop is in their respective dispatcher) and stores cummulatively time differences between *checkpoints*.

Note: Fine timing with **TimingDeltas** will only work if timing is enabled globally (see previous section). The code would still run, but giving zero times and exec counts.

1. Engine::timingDeltas must point to an instance of TimingDeltas (prefferably instantiate TimingDeltas in the constructor):

```
// header file
class Law2_Dem3DofGeom_CpmPhys_Cpm: public LawFunctor {
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Law2_Dem3DofGeom_CpmPhys_Cpm,LawFunctor,"docstring",
      /* attrs */,
      /* constructor */
      timingDeltas=shared_ptr<TimingDeltas>(new TimingDeltas);
   );
   // ...
};
```

2. Inside the loop, start the timing by calling **timingDeltas->start();**

3. At places of interest, call **timingDeltas->checkpoint("label")**. The label is used only for post-processing, data are stored based on the checkpoint position, not the label.

> **Warning:** Checkpoints must be always reached in the same order, otherwise the timing data will be garbage. Your code can still branch, but you have to put checkpoints to places which are in common.

```
void Law2_Dem3DofGeom_CpmPhys_Cpm::go(shared_ptr<InteractionGeometry>& _geom,
                                      shared_ptr<InteractionPhysics>& _phys,
                                      Interaction* I,
                                      Scene* scene)
{
   timingDeltas->start();                          // the point at which the first timing starts
   // prepare some variables etc here
   timingDeltas->checkpoint("setup");
   // find geometrical data (deformations) here
   timingDeltas->checkpoint("geom");
   // compute forces here
   timingDeltas->checkpoint("material");
   // apply forces, cleanup here
   timingDeltas->checkpoint("rest");
}
```

The output might look like this (note that functors are nested inside dispatchers and **TimingDeltas** inside their engine/functor):

```
Name                                 Count                 Time          Rel. time
-------------------------------------------------------------------------------
ForceReseter                           400                 9449µs              0.01%
BoundDispatcher                        400              1171770µs              1.15%
InsertionSortCollider                  400              9433093µs              9.24%
InteractionGeometryDispatcher          400             15177607µs             14.87%
InteractionPhysicsDispatcher           400              9518738µs              9.33%
LawDispatcher                          400             64810867µs             63.49%
  Law2_Dem3DofGeom_CpmPhys_Cpm
    setup                          4926145              7649131µs             15.25%
    geom                           4926145             23216292µs             46.28%
    material                       4926145              8595686µs             17.14%
    rest                           4926145             10700007µs             21.33%
    TOTAL                                              50161117µs            100.00%
NewtonIntegrator                       400              1866816µs              1.83%
"strainer"                             400                21589µs              0.02%
"plotDataCollector"                    160                64284µs              0.06%
```

```
"damageChecker"                    9          3272µs          0.00%
TOTAL                                     102077490µs        100.00%
```

> **Warning:** Do not use TimingDeltas in parallel sections, results might not be meaningful. In particular, avoid timing functors inside InteractionDispatchers when running with multiple OpenMP threads.

TimingDeltas data are accessible from Python as list of (*label*,*time*,*count*) tuples, one tuple representing each checkpoint:

```
deltas=someEngineOrFunctor.timingDeltas.data()
deltas[0][0] # 0th checkpoint label
deltas[0][1] # 0th checkpoint time in nanoseconds
deltas[0][2] # 0th checkpoint execution count
deltas[1][0] # 1st checkpoint label
             # …
deltas.reset()
```

7.3.9.3. Timing overhead

The overhead of the coarser, per-engine timing, is very small. For simulations with at least several hundreds of elements, they are below the usual time variance (a few percent).

The finer TimingDeltas timing can have major performance impact and should be only used during debugging and performance-tuning phase. The parts that are file-timed will take disproportionally longer time that the rest of engine; in the output presented above, LawDispatcher takes almost ☒ of total simulation time in average, but the number would be twice of thrice lower typically (note that each checkpoint was timed almost 5 million times in this particular case).

7.3.10. OpenGL Rendering

Yade provides 3d rendering based on [QGLViewer]. It is not meant to be full-featured rendering and post-processing, but rather a way to quickly check that scene is as intended or that simulation behaves sanely.

Note: Although 3d rendering runs in a separate thread, it has performance impact on the computation itself, since interaction container requires mutual exclusion for interaction creation/deletion. The **InteractionContainer::drawloopmutex** is either held by the renderer (OpenGLRenderingEngine) or by the insertion/deletion routine.

> **Warning:** There are 2 possible causes of crash, which are not prevented because of serious performance penalty that would result:
> 1. access to BodyContainer, in particular deleting bodies from simulation; this is a rare operation, though.
> 2. deleting Interaction::interactionPhysics or Interaction::interactionGeometry.

Renderable entities (Shape, State, Bound, InteractionGeometry, InteractionPhysics) have their associated OpenGL functors. An entity is rendered if

1. Rendering such entities is enabled by appropriate attribute in OpenGLRenderingEngine
2. Functor for that particular entity type is found via the *dispatch mechanism*.

Gl1_* functors operating on Body's attributes (Shape, State, Bound) are called with the OpenGL context translated and rotated according to State::pos and State::ori. Interaction functors work in global coordinates.

7.4. Simulation framework

Besides the support framework mentioned in the previous section, some functionality pertaining to simulation itself is also provided.

There are special containers for storing bodies, interactions and (generalized) forces. Their internal functioning is normally opaque to the programmer, but should be understood as it can influence performance.

7.4.1. Scene

Scene is the object containing the whole simulation. Although multiple scenes can be present in the memory, only one of them is active. Saving and loading (serializing and deserializing) the Scene object should make the simulation run from the point where it left off.

Note: All Engines and functors have interally a **Scene* scene** pointer which is updated regularly by engine/functor callers; this ensures that the current scene can be accessed from within user code.

For outside functions (such as those called from python, or static functions in **Shop**), you can use **Omega::instance().getScene()** to retrieve a **shared_ptr<Scene>** of the current scene.

7.4.2. Body container

Body container is linear storage of bodies. Each body in the simulation has its unique id, under which it must be found in the BodyContainer. Body that is not yet part of the simulation typically has id equal to invalid value **Body::ID_NONE**, and will have its **id** assigned upon insertion into the container. The requirements on BodyContainer are

- O(1) access to elements,
- linear-addressability (0...n indexability),
- store **shared_ptr**, not objects themselves,
- *no* mutual exclusion for insertion/removal (this must be assured by the called, if desired),
- intelligent allocation of **id** for new bodies (tracking removed bodies),
- easy iteration over all bodies.

Note: Currently, there is "abstract" class **BodyContainer**, from which derive concrete implementations; the initial idea was the ability to select at runtime which implementation to use (to find one that performs the best for given simulation). This incurs the penalty of many virtual function calls, and will probably change in the future. All implementations of BodyContainer were removed in the meantime, except **BodyVector** (internally a **vector<shared_ptr<Body>** > plus a few methods around), which is the fastest.

7.4.2.1. Insertion/deletion

Body insertion is typically used in FileGenerator's:

```
shared_ptr<Body> body(new Body);
// … (body setup)
scene->bodies->insert(body); // assigns the id
```

Bodies are deleted only rarely:

```
scene->bodies->erase(id);
```

Warning: Since mutual exclusion is not assured, never insert/erase bodies from parallel sections, unless you explicitly assure there will be no concurrent access.

7.4.2.2. Iteration

The container can be iterated over using **FOREACH** macro (shorthand for **BOOST_FOREACH**):

```
FOREACH(const shared_ptr<Body>& b, *scene->bodies){
   if(!b) continue;                        // skip deleted bodies
   /* do something here */
}
```

Note a few important things:

1. Always use **const shared_ptr<Body>&** (const reference); that avoids incrementing and decrementing the reference count on each **shared_ptr**.
2. Take care to skip NULL bodies (**if(!b) continue**): deleted bodies are deallocated from the container, but since body id's must be persistent, their place is simply held by an empty **shared_ptr<Body>()** object, which is implicitly convertible to **false**.

In python, the BodyContainer wrapper also has iteration capabilities; for convenience (which is different from the c++ iterator), NULL bodies as silently skipped:

```
Yade [108]: O.bodies.append([Body(),Body(),Body()])
 ->  [108]: [0, 1, 2]

Yade [109]: O.bodies.erase(1)
 ->  [109]: True

Yade [110]: [b.id for b in O.bodies]
 ->  [110]: [0, 2]
```

In loops parallelized using OpenMP, the loop must traverse integer interval (rather than using iterators):

```
const long size=(long)bodies.size();       // store this value, since it doesn't change during the loop
#pragma omp parallel for
for(long _id=0; _id<size; _id++){
   const shared_ptr<Body>& b(bodies[_id]);
   if(!b) continue;
   /* … */
}
```

7.4.3. InteractionContainer

Interactions are stored in special container, and each interaction must be uniquely identified by pair of ids (id1,id2).

- O(1) access to elements,
- linear-addressability (0…n indexability),
- store **shared_ptr**, not objects themselves,
- mutual exclusion for insertion/removal,
- easy iteration over all interactions,

- addressing symmetry, i.e. interaction(id1,id2)⊠interaction(id2,id1)

Note: As with BodyContainer, there is "abstract" class InteractionContainer, and then its concrete implementations. Currently, only InteractionVecMap implementation is used and all the other were removed. Therefore, the abstract InteractionContainer class may disappear in the future, to avoid unnecessary virtual calls.

Further, there is a blueprint for storing interactions inside bodies, as that would give extra advantage of quickly getting all interactions of one particular body (currently, this necessitates loop over all interactions); in that case, InteractionContainer would disappear.

7.4.3.1. Insert/erase

Creating new interactions and deleting them is delicate topic, since many eleents of simulation must be synchronized; the exact workflow is described in *Handling interactions*. You will almost certainly never need to insert/delete an interaction manually from the container; if you do, consider designing your code differently.

```
// both insertion and erase are internally protected by a mutex,
// and can be done from parallel sections safely
scene->interactions->insert(shared_ptr<Interaction>(new Interactions(id1,id2)));
scene->interactions->erase(id1,id2);
```

7.4.3.2. Iteration

As with BodyContainer, iteration over interactions should use the **FOREACH** macro:

```
FOREACH(const shared_ptr<Interaction>& i, *scene->interactions){
   if(!i->isReal()) continue;
   /* … */
}
```

Again, note the usage const reference for **i**. The check **if(!i->isReal())** filters away interactions that exist only *potentially*, i.e. there is only Bound overlap of the two bodies, but not (yet) overlap of bodies themselves. The **i->isReal()** function is equivalent to **i->interactionGeometry && i->interactionPhysics**. Details are again explained in *Handling interactions*.

In some cases, such as OpenMP-loops requiring integral index (OpenMP >= 3.0 allows parallelization using random-access iterator as well), you need to iterate over interaction indices instead:

```
inr nIntr=(int)scene->interactions->size(); // hoist container size
#pragma omp parallel for
for(int j=0; j<nIntr, j++){
   const shared_ptr<Interaction>& i(scene->interactions[j]);
   if(!->isReal()) continue;
   /* … */
}
```

7.4.4. ForceContainer

ForceContainer holds "generalized forces", i.e. forces, torques, (explicit) dispalcements and rotations for each body.

During each computation step, there are typically 3 phases pertaining to forces:

1. Resetting forces to zero (usually done by the ForceResetter engine)
2. Incrementing forces from parallel sections (solving interactions – from LawFunctor)

3. Reading absolute force values sequentially for each body: forces applied from different interactions are summed together to give overall force applied on that body (NewtonIntegrator, but also various other engine that read forces)

This scenario leads to special design, which allows fast parallel write access:

- each thread has its own storage (zeroed upon request), and only write to its own storage; this avoids concurrency issues. Each thread identifies itself by the omp_get_thread_num() function provided by the OpenMP runtime.
- before reading absolute values, the container must be synchronized, i.e. values from all threads are summed up and stored separately. This is a relatively slow operation and we provide ForceContainer::syncCount that you might check to find cummulative number of synchronizations and compare it agains number of steps. Ideally, ForceContainer is only synchronized once at each step.
- the container is resized whenever an element outside the current range is read/written to (the read returns zero in that case); this avoids the necessity of tracking number of bodies, but also is potential danger (such as **scene->forces.getForce(1000000000)**, which will probably exhaust your RAM). Unlike c++, Python does check given id against number of bodies.

```
// resetting forces (inside ForceResetter)
scene->forces.reset()

// in a parallel section
scene->forces.addForce(id,force); // add force

// container is not synced after we wrote to it, sync before reading
scene->forces.sync();
const Vector3r& f=scene->forces.getForce(id);
```

Synchronization is handled automatically if values are read from python:

```
Yade [111]: O.bodies.append(Body())
 ->  [111]: 1

Yade [112]: O.forces.addF(0,Vector3(1,2,3))

Yade [113]: O.forces.f(0)
 ->  [113]: Vector3(1,2,3)

Yade [114]: O.forces.f(100)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/home/vaclav/ydoc/<ipython console> in <module>()

IndexError: Body id out of range.
```

7.4.5. Handling interactions

Creating and removing interactions is a rather delicate topic and number of components must cooperate so that the whole behaves as expected.

Terminologically, we distinguish

potential interactions, having neither geometry nor physics. Interaction.real can be used to query the status (**Interaction::isReal()** in c++).

real interactions, having both geometry and physics. Below, we shall discuss the possibility of interactions that only have geometry but no physics.

During each step in the simulation, the following operations are performed on interactions in a typical

simulation:

1. Collider creates potential interactions based on spatial proximity. Not all pairs of bodies are susceptible of entering interaction; the decision is done in Collider::mayCollide:

   - clumps may not enter interactions (only their members can)
   - clump members may not interact if they belong to the same clump
   - bitwise AND on both bodies' masks must be non-zero (i.e. there must be at least one bit set in common)

2. Collider erases interactions that were requested for being erased (see below).

3. InteractionDispatchers (via InteractionGeometryDispatcher) calls appropriate InteractionGeometryFunctor based on Shape combination of both bodies, if such functor exists. For real interactions, the functor updates associated InteractionGeometry. For potential interactions, the functor returns

   false if there is no geometrical overlap, and the interaction will stillremain potential-only

   true if there is geometrical overlap; the functor will have created an InteractionGeometry in such case.

   Note: For *real* interactions, the functor *must* return **true**, even if there is no more spatial overlap between bodies. If you wish to delete an interaction without geometrical overlap, you have to do this in the LawFunctor.

   This behavior is deliberate, since different laws have different requirements, though ideally using relatively small number of generally useful geometry functors.

   Note: If there is no functor suitable to handle given combination of shapes, the interaction will be left in potential state, without raising any error.

4. For real interactions (already existing or jsut created in last step), InteractionDispatchers (via InteractionPhysicsDispatcher) calls appropriate InteractionPhysicsFunctor based on Material combination of both bodies. The functor *must* update (or create, if it doesn't exist yet) associated InteractionPhysics instance. It is an error if no suitable functor is found, and an exception will be thrown.

5. For real interactions, InteractionDispatchers (via LawDispatcher) calls appropriate LawFunctor based on combintation of InteractionGeometry and InteractionPhysics of the interaction. Again, it is an error if no functor capable of handling it is found.

6. LawDispatcher can decide that an interaction should be removed (such as if bodies get too far apart for non-cohesive laws; or in case of complete damage for damage models). This is done by calling

   ```
   InteractionContainer::requestErase(id1,id2)
   ```

   Such interaction will not be deleted immediately, but will be reset to potential state. At next step, the collider will call **InteractionContainer::erasePending**, which will only completely erase interactions the collider indicates; the rest will be kept in potential state.

7.4.5.1. Creating interactions explicitly

Interactions may still be created explicitly with utils.createInteraction, without any spatial requirements. This function searches current engines for dispatchers and uses them. InteractionGeometryFunctor is called with the **force** parameter, obliging it to return **true** even if there is no spatial overlap.

7.5. Runtime structure

7.5.1. Startup sequence

Yade's main program is python script in core/main/main.py.in; the build system replaces a few **${variables}** in that file before copying it to its install location. It does the following:

1. Process command-line options, set environment variables based on those options.
2. Import main yade module (**import yade**), residing in py/__init__.py.in. This module locates plugins (recursive search for files **lib*.so** in the **lib** installation directory). yade.boot module is used to setup logging, temporary directory, ... and, most importantly, loads plugins.
3. Manage further actions, such as running scripts given at command line, opening qt.Controller (if desired), launching the **ipython** prompt.

7.5.2. Singletons

There are several "global variables" that are always accessible from c++ code; properly speaking, they are Singletons, classes of which exactly one instance always exists. The interest is to have some general functionality acessible from anywhere in the code, without the necessity of passing pointers to such objects everywhere. The instance is created at startup and can be always retrieved (as non-const reference) using the **instance()** static method (e.g. **Omega::instance().getScene()**).

There are 3 singletons:

SerializableSingleton Handles serialization/deserialization; it is not used anywhere except for the serialization code proper.

ClassFactory Registers classes from plugins and able to factor instance of a class given its name as string (the class must derive from **Factorable**). Not exposed to python.

Omega Access to simulation(s); deserves separate section due to its importance.

7.5.2.1. Omega

The Omega class handles all simulation-related functionality: loading/saving, running, pausing.

In python, the wrapper class to the singleton is instantiated [7] as global variable **O**. Because there is no separate **Scene** class in python, Omega is used to access its contents from python. Although multiple **Scene** objects may be instantiated in c++, it is always the current scene that Omega represents.

The correspondence of data is literal: Omega.materials corresponds to **Scene::materials** of the current scene; likewise for materials, bodies, interactions, tags, cell, engines, initializers, miscParams.

Some variables do not correspond literally, for historical reasons (which should be fixed):

Python	c++
Omega.iter	**Scene::currentIteration**
Omega.dt	**Scene::dt**
Omega.time	**Scene::simulationTime**
Omega.realtime	**Omega::getComputationTime()**
Omega.stopAtIter	**Scene::stopAtIteration**

Omega in c++ contains pointer to the current scene (**Omega::scene**, retrieved by **Omega::instance().getScene()**). Using Omega.switchScene, it is possible to swap this pointer with **Omega::sceneAnother**, a completely independent simulation. This can be useful for example (and

[7] It is understood that instantiating **Omega()** in python only instantiated the wrapper class, not the singleton itself.

this motivated this functionality) if while constructing simulation, another simulation has to be run to dynamically generate (i.e. by running simulation) packing of spheres.

7.5.3. Engine loop

Running simulation consists in looping over Engines and calling them in sequence. This loop is defined in **Scene::moveToNextTimeStep** function in core/Scene.cpp. Before the loop starts, O.initializers are called; they are only run once. The engine loop does the following in each iteration over O.engines:

1. set **Engine::scene** pointer to point to the current **Scene**.
2. Call **Engine::isActivated()**; if it returns **false**, the engine is skipped.
3. Call **Engine::action()**
4. If O.timingEnabled, increment Engine::execTime‘ by difference from last time reading (either after the previous engine was run, or immediately before the loop started, if this engine comes first). Increment Engine::execCount by 1.

After engines are processed, virtual time is incremented by timestep and iteraction number is incremented by 1.

7.5.3.1. Background execution

The engine loop is (normally) executed in background thread (handled by SimulationFlow class), leaving forground thread free to manage user interaction or running python script. The background thread is managed by O.run() and O.pause() commands. Foreground thread can be blocked until the loop finishes using O.wait().

Single iteration can be run without spawning additional thread using O.step().

7.6. Python framework

7.6.1. Wrapping c++ classes

Each class deriving from Serializable is automatically exposed to python, with access to its (registered) attributes. This is achieved via *YADE_CLASS_BASE_DOC_* macro family*. All classes registered in class factory are default-constructed in **Omega::buildDynlibDatabase**. Then, each serializable class calls **Serializable::pyRegisterClass** virtual method, which injects the class wrapper into (initially empty) **yade.wrapper** module. **pyRegisterClass** is defined by **YADE_CLASS_BASE_DOC** and knows about class, base class, docstring, attributes, which subsequently all appear in boost::python class definition.

Wrapped classes define special constructor taking keyword arguments corresponding to class attributes; therefore, it is the same to write:

```
Yade [115]: f1=ForceEngine()

Yade [116]: f1.subscribedBodies=[0,4,5]

Yade [117]: f1.force=Vector3(0,-1,-2)
```

and

```
Yade [118]: f2=ForceEngine(subscribedBodies=[0,4,5],force=Vector3(0,-1,-2))

Yade [119]: print f1.dict()
```

```
{'subscribedBodies': [0, 4, 5], 'force': Vector3(0,-1,-2), 'label': ''}

Yade [120]: print f2.dict()
{'subscribedBodies': [0, 4, 5], 'force': Vector3(0,-1,-2), 'label': ''}
```

Wrapped classes also inherit from Serializable several special virtual methods: dict() returning all registered class attributes as dictionary (shown above), clone() returning copy of instance (by copying attribute values), updateAttrs() and updateExistingAttrs() assigning attributes from given dictionary (the former thrown for unknown attribute, the latter doesn't).

Read-only property **name** wraps c++ method **getClassName()** returning class name as string. (Since c++ class and the wrapper class always have the same name, getting python type using **___class___** and its property **___name___** will give the same value).

```
Yade [121]: s=Sphere()

Yade [122]: s.name, s.__class__.__name__
 ->  [122]: ('Sphere', 'Sphere')
```

7.6.2. Subclassing c++ types in python

In some (rare) cases, it can be useful to derive new class from wrapped c++ type in pure python. This is done in the *yade.pack* module: Predicate is c++ base class; from this class, several c++ classes are derived (such as inGtsSurface), but also python classes (such as the trivial inSpace predicate). **inSpace** derives from python class **Predicate**; it is, however, not direct wrapper of the c++ **Predicate** class, since virtual methods would not work.

boost::python provides special **boost::python::wrapper** template for such cases, where each overridable virtual method has to be declared explicitly, requesting python override of that method, if present. See Overridable virtual functions for more details.

7.6.3. Reference counting

Python internally uses reference counting on all its objects, which is not visible to casual user. It has to be handled explicitly if using pure Python/C API with **Py_INCREF** and similar functions.

boost::python used in Yade fortunately handles reference counting internally. Additionally, it automatically integrates reference counting for **shared_ptr** and python objects, if class **A** is wrapped as **boost::python::class_<A,shared_ptr<A>>**. Since *all* Yade classes wrapped using *YADE_CLASS_BASE_DOC_* macro family* are wrapped in this way, returning **shared_ptr<...>** objects from is the preffered way of passing objects from c++ to python.

Returning **shared_ptr** is much more efficient, since only one pointer is returned and reference count internally incremented. Modifying the object from python will modify the (same) object in c++ and vice versa. It also makes sure that the c++ object will not be deleted as long as it is used somewhere in python, preventing (important) source of crashes.

7.6.4. Custom converters

When an object is passed from c++ to python or vice versa, then either

1. the type is basic type which is transparently passed between c++ and python (int, bool, std::string etc)
2. the type is wrapped by boost::python (such as Yade classes, **Vector3** and so on), in which case

wrapped object is returned; [8]

Other classes, including template containers such as **std::vector** must have their custom converters written separately. Some of them are provided in py/yadeWrapper/customConverters.cpp, notably converters between python (homogeneous, i.e. with all elements of the same type) sequences and c++ **std::vector** of corresponding type; look in that source file to add your own converter or for inspiration.

When an object is crossing c++/python boundary, boost::python's global "converters registry" is searched for class that can perform conversion between corresponding c++ and python types. The "converters registry" is common for the whole program instance: there is no need to register converters in each script (by importing **_customConverters**, for instance), as that is done by yade at startup already.

Note: Custom converters only work for value that are passed by value to python (not "by reference"): some attributes defined using *YADE_CLASS_BASE_DOC_* macro family* are passed by value, but if you define your own, make sure that you read and understand Why is my automatic to-python conversion not being found?.

In short, the default for **def_readwrite** and **def_readonly** is to return references to underlying c++ objects, which avoids performing conversion on them. For that reason, return value policy must be set to **return_by_value** explicitly, using slighly more complicated **add_property** syntax, as explained at the page referenced.

7.7. Maintaining compatibility

In Yade development, we identified compatibility to be very strong desire of users. Compatibility concerns python scripts, *not* simulations saved in XML or old c++ code.

7.7.1. Renaming class

Script scripts/rename-class.py should be used to rename class in **c++** code. It takes 2 parameters (old name and new name) and must be run from top-level source directory:

```
$ scripts/rename-class.py OldClassName NewClassName
Replaced 4 occurences, moved 0 files and 0 directories
Update python scripts (if wanted) by running: perl -pi -e 's/\bOldClassName\b/NewClassName/g' `ls **/*.py |grep
```

This has the following effects:

1. If file or directory has basename **OldClassName** (plus extension), it will be renamed using **bzr**.
2. All occurences of whole word **OldClassName** will be replaced by **NewClassName** in c++ sources.
3. An extry is added to py/system.py, which contains map of deprecated class names. At yade startup, proxy class with **OldClassName** will be created, which issues a **DeprecationWarning** when being instantiated, informing you of the new name you should use; it creates an instance of **NewClassName**, hence not disruting your script's functioning:

   ```
   Yade [3]: SimpleViscoelasticMat()
   /usr/local/lib/yade-trunk/py/yade/__init__.py:1: DeprecationWarning: Class `SimpleViscoelasticMat' was r
   ->  [3]: <ViscElMat instance at 0x2d06770>
   ```

As you have just been informed, you can run **yade --update** to all old names with their new names in scripts you provide:

[8] Wrapped classes are automatically registered when the class wrapper is created. If wrapped class derives from another wrapped class (and if this dependency is declared with the **boost::python::bases** template, which Yade's classes do automatically), parent class must be registered before derived class, however. (This is handled via loop in **Omega::buildDynlibDatabase**, which reiterates over classes, skipping failures, until they all successfully register) Math classes (Vector3, Matrix3, Quaternion) are wrapped by hand, to be found in py/mathWrap/miniEigen.cpp; this module is imported at startup.

```
$ yade-trunk --update script1.py some/where/script2.py
```

This gives you enough freedom to make your class name descriptive and intuitive.

7.7.2. Renaming class attribute

Renaming class attribute is handled from c++ code. You have the choice of merely warning at accessing old attribute (giving the new name), or of throwing exception in addition, both with provided explanation. See **deprec** parameter to *YADE_CLASS_BASE_DOC_* macro family* for details.

7.8. Debian packaging instructions

In order to make parallel installation of several Yade version possible, we adopted similar strategy as e.g. **gcc** packagers in Debian did:

1. Real Yade packages are named **yade-0.30** (for stable versions) or **yade-bzr2341** (for snapshots).
2. They provide **yade** or **yade-snapshot** virtual packages respectively.
3. Each source package creates several installable packages (using **bzr2341** as example version):
   a) **yade-bzr2341** with the optimized binaries; the executable binary is **yade-bzr2341** (**yade-bzr2341-multi**, ...)
   b) **yade-bzr2341-dbg** with debug binaries (debugging symbols, non-optimized, and with crash handlers); the executable binary is **yade-bzr2341-dbg**
   c) **yade-bzr2341-doc** with sample scripts and some documentation (see bug #398176 however)
   d) (future?) **yade-bzr2341-reference** with reference documentation (see bug #401004)
4. Using Debian alternatives, the highest installed package provides additionally commands without the version specification like **yade**, **yade-multi**, ... as aliases to that version's binaries. (**yade-dbg**, ... for the debuggin packages). The exact rule is:
   a) Stable releases have always higher priority than snapshots
   b) Higher versions/revisions have higher pripority than lower versions/revisions.

7.8.1. Prepare source package

Debian packaging files are located in debian/ directory. They contain build recipe debian/rules, dependecy and package declarations debian/control and maintainer scripts. Some of those files are only provided as templates, where some variables (such as version number) are replaced by special script.

The script scripts/debian-prep processes templates in debian/ and creates files which can be used by debian packaging system. Before running this script:

1. If you are releasing stable version, make sure there is file named **RELEASE** containing single line with version number (such as **0.30**). This will make scripts/debian-prep create release packages. In absence of this file, snapshots packaging will be created instead. Release or revision number (as detected by running **bzr revno** in the source tree) is stored in **VERSION** file, where it is picked up during package build and embedded in the binary.
2. Find out for which debian/ubuntu series your package will be built. This is the name that will appear on the top of (newly created) **debian/changelog** file. This name will be usually **unstable**,

testing or **stable** for debian and **karmic**, **lucid** etc for ubuntu. WHen package is uploaded to Launchpad's build service, the package will be built for this specified release.

Then run the script from the top-level directory, giving series name as its first (only) argument:

```
$ scripts/debian-prep lucid
```

After this, signed debian source package can be created:

```
$ debuild -S -sa -k62A21250 -I -Iattic
```

(**-k** gives GPG key identifier, **-I** skips **.bzr** and similar directories, **-Iattic** will skip the useless **attic** directory).

7.8.2. Create binary package

Local in-tree build Once files in **debian/** are prepared, packages can be build by issuing:: $ fakeroot debian/rules binary

Clean system build Using **pbuilder** system, package can be built in a chroot containing clean debian/ubuntu system, as if freshly installed. Package dependencies are automatically installed and package build attempted. This is a good way of testing packaging before having the package built remotely at Launchpad. Details are provided at wiki page.

Launchpad build service Launchpad provides service to compile package for different ubuntu releases (series), for all supported architectures, and host archive of those packages for download via APT. Having appropriate permissions at Launchpad (verified GPG key), source package can be uploaded to yade's archive by:

```
$ dput ppa:yade-users/ppa ../yade-bzr2341_1_source.changes
```

After several hours (depending on load of Launchpad build farm), new binary packages will be published at https://launchpad.net/~yade-users/+archive/ppa.

This process is well documented at https://help.launchpad.net/Packaging/PPA.

8. Conclusion

The Discrete Element Method (as implemented in Yade) was presented mathematically in some detail, in the context of other discrete methods. While some topics were not treated exhaustively, we hope to have created a solid basis for complete documentation of the theoretical basis of Yade.

A new particle model of concrete was developed within the DEM framework, based on continuum formulations. It was tested on standard setups and calibration procedures are described in detail. Because the formulation is rather general, the model itself is suitable for use for other cohesive materials by changing numerical parameters. For confidentiality reasons, we did not show any applications of the model. It was shown that particle-based model, although only formulated locally on one-dimensional contact, can capture behavior found in experiments; although there is still long journey towards complete description of continuum behavior, the basis has shown to be solid and worth confidence. Lot of meaningful work can also be done in the analytical description of macroscopic properties based on local parameters, which we only touched lightly; this should permit to better establish mathematical relationship with continuum-based models.

The description of Yade from user and programmer perspective are the first comprehensive documents of this kind. Yade API documentation in the Appendix is unique in the sense that it is identical for c++ and Python, the languages used in Yade. Yade as a platform for DEM has seen important rise in usage at several universities during last 3 years. We attribute it mostly to documentation, consistent API and quality of the core code.

There are many possible ways for future development. On the cohesive particle model side, lot of work can be done on analytical (statistical) description of the relationship between dense interaction network and continuum, including transitions between particle-based and continuum-based models; this was actually part of this project, but was only finished in part and the partial results will be handed down to the posterity separately. Concerning Yade as software, the most challenging is to steer individual developers away from the desire for fast results towards a more responsible attitude of community development. Community development is sustainable, while individual hacking is not. Aesthetics (not in any high sense) is an essential part of sustainable development. A beautiful program (from which Yade is still very far) can be likened to some music or to a beautiful mathematical theory: it has an inner logic from which, once grasped, all the details flow seamlessly. I wish Yade a bright future.

The entropy always grows, there is no perfection on the Earth and it is perfect just like that.

Part III.

Appendices

A. Object-oriented programming paradigm

A.1. Key concepts

Object-orientated programming represents any given problem by using hierarchy of types (of which tradition goes back to Aristotle's hierarchy of being and is used e.g. in Linnée biology nomenclature) and objects of those types. Hierarchy of types defines subordination relationships ("inheritance"). Objects in themselves are opaque to outer objects ("encapsulation"), but external manipulation is possible without internals' knowledge using interface ("methods") defined on each type; such interfaces are independent on internal structure of the object ("data abstraction"). We will try to illustrate basic features of the object-oriented paradigm in the following paragrahs.

Inheritance of types improves expressibility of the language, as programming can proceed on a more generic level. For instance, let us define a geometry type hierarchy, shown at fig. A.1. The subordination (shown by arrows) declares that, for instance, **Quadrilateral** is a **Polygon**, and thus is indirectly also a **ClosedShape2d**.

Encapsulation. Suppose that **ClosedShape2d** will define a *property*[1] **position**, holding spatial position of that shape; note that we put such information inside the object ("encapsulation") and the object will know its own position, if asked. By virtue of inheritance, all types inheriting from **ClosedShape2d** will also have the **position** property, without having to change their code in any way.

Data abstraction. An algorithm changing **position** of a **ClosedShape2d** doesn't have to know particular type of the object it operates on, since the property is inherited by all subtypes. In another words, the algorithm abstracts from particular type of an object, operating on the abstract level of **ClosedShape2d**.

[1] Properties are usually called "member data" in c++.

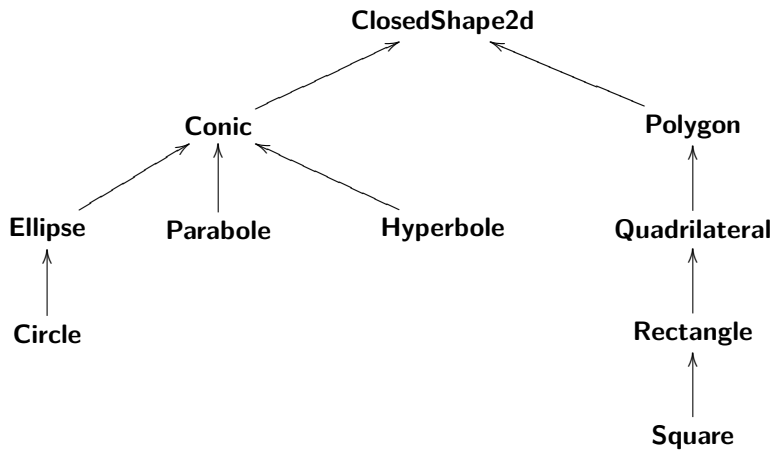


Figure A.1.: Sample type hierarchy of closed planar shapes.

Polymorphism. If the user wanted to know a **ClosedShape2d** to return its area, we no longer can stay on the abstract level, since area computation will be different for subtypes of **ClosedShape2d**. An algorithm interested in area would have to know about each subtype's internals and, depending on the subtype, have different branches to compute area from those internals; this would, however, break encapsulation and the algorithm would have to be modified for each type added to the hierarchy.

Instead, only abstract *interface* will be declared in the **ClosedShape2d** class, a method[2] **area()** computing area; implementation of this interface will have to be done in each derived class, but the interface itself will be the same for all subtypes. Then, when **area()** method of an object is called, the object will internally decide, based on its own type, which implementation to call, so that a **Square** calls **Square**'s implementation. Such decision based on subtype is called *virtual dispatch* and has to be done at runtime (so called "late binding"); the corresponding method is a *virtual* method. This allows a **Square** to appear as a **ClosedShape2d** on the interface level, but still function as **Square** ("polymorphism").

area() dispatches based only on one subtype, of the instance itself. In some cases, it makes sense to dispatch based on multiple types. For instance, computing overlap area between 2 **ClosedShape2d**s will require knowledge about both types; the **overlap(other)** method would be dispatched based on combination of type of the object as well as of **other**. Such behavior is called *multiple dispatch* or *multimethod*.

A.2. Language support and performance

Different programing languages implement the object-oriented pattern differently and we should point out implementations in languages used in Yade, i. e. c++ and Python.

C++ is strongly-typed language: types of objects must be known at compile-time and function calls are resolved at compile-time (early binding, static binding) with the exception of virtual functions, which are by definition resolved at runtime (late binding). This design decision allows generation of code that suffers no performance penalty compared to plain c code, with the exception of virtual functions.

Data encapsulation can be enforced by imposing access level on member data/methods:

private, where only object of this class access the data (or call the methods),

protected, where objects of this class and its derived (inheriting) subclasses can access the data),

public, where any object can access the data.

Virtual functions infer performance penaly (see Driesen and Hölzle [13] for detailed implementation analysis and measurements), as instead of direct function call,[3] the address of the function to call is determined by lookup in virtual function table[4].

Multivirtual functions are not part of c++. The author of c++ published paper Pirkelbauer et al. [45], but their future addition to c++ is uncertain. To-day, they must be emulated with a hand-made dispatch mechanism, such as Smith [62] or Shopyrin [60], making use of simple virtual dispatch.

Python is dynamic language, said to be "duck-typed" – appropriate types are checked by existence of properties that are asked for; that provides high flexibility, allowing, for example, to pass an unrelated object where a **file** is required, as long as this object's type has all properties that are used.

In Python, every attribute access and method call results in dictionary (string → object map) lookup, which proceeds from class of the object itself and climbs up the class hierarchy, until found. This

[2] Called "member function" in c++.

[3] Member functions (members) are generated as regular functions with first hidden parameter carrying pointer to the object's instance.

[4] Implementation details of virtual dispatch are not mandated by the c++ standard, but the vtable approach is used by major c++ compilers (the GNU compiler in particular)

means that *all methods in Python are virtual*. There are no access levels, hence all data members are "public" (using the c++ terminology). The distinction between member data/function is made only for convenience, since functions are also objects (defining the special __**call**__ attribute).

Multimethods can be emulated quite easily in Python (e.g. van Rossum [64]); for our (Yade) purposes, multimethods are only interesting for the c++ part.

B. Quaternions

Quaternions[51] are hypercomplex numbers with 3 imaginary components, invented by William Hamilton [18]. They take the form $w + x\hat{i} + y\hat{j} + z\hat{k}$, being points in $\mathbb{R}^4$ space with base $(1, \hat{i}, \hat{j}, \hat{k})$; the relation between the bases is such that

$$\hat{i}^2 = \hat{j}^2 = \hat{k}^2 = \hat{i}\hat{j}\hat{k} = -1. \tag{B.1}$$

This equation allows to establish all products of the base elements, e.g. $\hat{i}\hat{j} = \hat{k}$ and $\hat{j}\hat{i} = -\hat{k}$. Note that multiplication is *non-cummutative*. There are several operations defined on quaternions we should mention.

Quaternion product is analogous to complex number product, using distribubtion law, and additionally uses rules for base products:

$$\begin{aligned} pq &= (w_1 + x_1\hat{i} + x_2\hat{j} + y_2\hat{k})(w_2 + x_2\hat{i} + y_2\hat{j} + z_2\hat{k}) = \\ &= w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 + \\ &+ (w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2)\hat{i} + \\ &+ (w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2)\hat{j} + \\ &+ (w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2)\hat{k} \end{aligned} \tag{B.2}$$

The product is obviously not commutative, due to non-commutativity of base products.

Quaternion conjugate q^* of a $q = w + x\hat{i} + y\hat{j} + z\hat{k}$ is defined

$$q^* = w - x\hat{i} - y\hat{j} - z\hat{k}. \tag{B.3}$$

Note that $(q^*)^* = q$ (involution); conjugation of product reverses the order, so that

$$(pq)^* = q^*p^*. \tag{B.4}$$

Quaternion norm $\|q\|$ is defined as $\sqrt{qq^*}$,

$$\|q\| = \sqrt{qq^*} = \sqrt{q^*q} = \sqrt{w^2 + x^2 + y^2 + z^2}, \tag{B.5}$$

same as euclidean norm in the corresponding $\mathbb{R}^4$ space.

Quaternion inverse q^{-1} is such that $qq^{-1} = 1$. It can be shown that it is satisfied by

$$q^{-1} = \frac{q^*}{\|q\|^2}, \tag{B.6}$$

leading to identity of inverse and conjugate $q^{-1} = q^*$ for unit quaternions.

B.1. Unit quaternions as spatial rotations

Unit quaternions can represent spatial rotations [52]; in this section, unit length of all quaternions is tacitly assumed.

In geometry, quaternions coefficients are usually written as scalar in $\mathbb{R}$ (the real part) plus vector in $\mathbb{R}^3$ (the imaginary part), so that

$$\mathrm{q} = w + x\hat{\mathrm{i}} + y\hat{\mathrm{j}} + z\hat{\mathrm{k}} = w + \mathbf{u} \tag{B.7}$$

algebraic rules are then formally enriched by vector multiplication

$$\mathbf{mn} = \mathbf{m} \times \mathbf{n} - \mathbf{m} \cdot \mathbf{n} \tag{B.8}$$

defined in terms of regular scalar and vector product of vectors.

Vector rotation. It can be shown [52] that rotating $\mathbb{R}^3$ vector $\mathbf{a}$ around normalized axis $\mathbf{u}$ by angle ϑ, yielding transformed $\mathbf{a}'$, can be expressed by quaternion product

$$\mathrm{q} = \cos\frac{\vartheta}{2} + \mathbf{u}\sin\frac{\vartheta}{2} \tag{B.9}$$

$$\mathbf{a}' = \mathrm{q}\mathbf{a}\mathrm{q}^{-1} \tag{B.10}$$

$$= \mathrm{q}\mathbf{a}\mathrm{q}^{*} \quad (\text{since } \|\mathrm{q}\| = 1) \tag{B.11}$$

Conversion to axis-angle representation. Relationship inverse to (B.9) gives decomposition of quaternion to its axis and angle components. Using the (B.7) notation, if $|\mathrm{u}| > 0$, then

$$\mathrm{q}_\vartheta = 2\arccos w, \qquad \mathrm{q}_{\mathbf{u}} = \hat{\mathrm{u}}. \tag{B.12}$$

The case $|\mathrm{u}| = 0$ implies $\cos\vartheta/2 = 1$, therefore $\vartheta = 2n\pi$. It follows that $\mathrm{q}_\vartheta = 0$ and $\mathrm{q}_{\mathbf{u}}$ is arbitrary unit vector.

Conversion to and from rotation vector. Quaternion q can be expressed as rotation vector $\mathbf{a}$ simply as $\mathbf{a} = \mathrm{q}_\vartheta \mathrm{q}_{\mathbf{u}}$. In the other sense, rotation vector $\mathbf{a}$ can be expressed as quaternion, using the (B.7) form,

$$\mathrm{q} = \cos\frac{|\mathbf{a}|}{2} + \hat{\mathbf{a}}\sin\frac{|\mathbf{a}|}{2}. \tag{B.13}$$

Aligning 2 vectors. Given 2 vectors $\mathbf{a}$, $\mathbf{b}$, we want to find quaternion $\mathrm{q} = \mathrm{Align}(\mathbf{a}, \mathbf{b})$ such that $\mathbf{b} = \mathrm{q}\mathbf{a}\mathrm{q}^{*}$. Computing auxiliary unit bisector $\mathbf{c} = \widehat{\mathbf{a} + \mathbf{b}}$, we can express (from scalar product geometrical interpretation)

$$\cos\frac{\vartheta}{2} = \frac{\mathbf{a} \cdot \mathbf{c}}{|\mathbf{a}|}. \tag{B.14}$$

The angle part, $\mathbf{u}\sin\vartheta/2$ from (B.7), can be found as

$$\frac{\mathbf{a} \times \mathbf{c}}{|\mathbf{a}|} = \frac{|\mathbf{a}|\sin\frac{\vartheta}{2}\mathbf{u}}{|\mathbf{a}|} = \mathbf{u}\sin\frac{\vartheta}{2}. \tag{B.15}$$

Finally, we come to the solution

$$\mathrm{q} = \frac{\mathbf{a} \cdot \mathbf{c}}{|\mathbf{a}|} + \frac{\mathbf{a} \times \mathbf{c}}{|\mathbf{a}|} = \frac{\mathbf{a} \cdot \widehat{\mathbf{a} + \mathbf{b}}}{|\mathbf{a}|} + \frac{\mathbf{a} \times \widehat{\mathbf{a} + \mathbf{b}}}{|\mathbf{a}|} = \mathrm{Align}(\mathbf{a}, \mathbf{b}). \tag{B.16}$$

The operation is obviously meaningful only if $|\mathbf{a}| \neq 0$.

Rotation composition Applying two consecutive rotations q and p on the vector shows that it is equivalent to applying composed rotation in the form of quaternion product pq (note the reverse order):

$$q(p\mathbf{a}p^*)q^* = qp\mathbf{a}p^*q^* = (qp)\mathbf{a}(qp)^* \tag{B.17}$$

As a special case, this shows that conjugate quaternion q^* is rotation in the opposite sense, since $q^*(q\mathbf{a}q^*)q^{**} = \mathbf{a}$. Rotations can be composed arbitrarily, in general

$$q_n(\cdots(q_2(q_1\mathbf{a}q_1^*)q_2^*)\cdots)q_n^* = (q_n\cdots q_2q_1)\mathbf{a}(q_n\cdots q_2q_1)^* \tag{B.18}$$

Rotation interpolation Quaternions allow for easy and efficient interpolation between two quaternions p and q, parametrized on $t \in \{0\ldots1\}$ [61]. Since rotations are points on unit 4-dimensional hypersphere (quaternions representing rotations satisfy $\|q\| = 1$), this interpolation is called *spherical linear interpolation* ("slerp"). It follows from series expansion and quaternion unit length that

$$q^t = \cos t\frac{\vartheta}{2} + \boldsymbol{\nu}\sin t\frac{\vartheta}{2} \tag{B.19}$$

The interpolation over $t \in \langle 0\ldots1\rangle$

$$\mathrm{Slerp}(p, q; t) = p(p^*q)^t \tag{B.20}$$

is shortest path (geodesic on the hypersphere) between p and q with constant 3d angular velocity vector in the sense of the $\boldsymbol{\nu}$ unit vector representing rotation axis. This interpolation is frequently used in computer graphics.

B.2. Comparison of spatial rotation representations

There are 4 widespread ways to represent rotations:

Rotation matrix (direction cosine matrix, DCM), 3×3 matrix of which 3 columns are transformed base vectors $\hat{x}'$, $\hat{y}'$, $\hat{z}'$. Rotation composition is done by matrix multiplication in the reverse order. Since spatial rotations space is 3-dimensional, matrix elements are not independent; transformed base vectors must be orthonormal.

In numerical simulations, rotation matrices must be (frequently) renormalized, as accumulating numerical errors leads to loss of orthonormality, introducing distorsion. The cost of re-normalization is evaluation of 3 square roots (when using the Gram-Schmidt algorithm), compared to signle square root evaluation involved in quaternion normalization.

Euler angles, 3 angles of independent rotations around axes appiled consecutively [15]. There are different conventions which rotation axes are used and in what order; it can be any non-homogeneous triple of x, y, z, which are originally coincident with frame of reference axis. Two of 24 possible conventions are widely used, **z-x-z** and **z-y-x** (known as yaw-roll-pitch).

The convention called **z-x-z** rotates around z (coincident with z of the frame of reference), then around now-rotated x and finally around rotated z. This convention is usually meant when "Euler angles" is used without further specification.

The "yaw-pitch-roll" convention is used in aeronautics rotates the frame of reference consecutively around z, y (once rotated) and x (twice rotated) axes and has intuitive meaning for spacecraft orientation, whence the "yaw-roll-pitch" name.

Rotation composition can be done by matrix multiplication of corresponding rotation matrices, which are themselves product of 3 matrices representing each rotation separately and multiplied in proper order.

The chief disadvantage of using any of the Euler angles is their singularity at poles, known as *gimbal lock*; for instance, the pitch of $\pi/2$ leads to infinite possible combinations of yaw and roll angles to arrive at given configuration (e.g. roll angle can be zero, while only yaw angle changes).

Angle/axis (angle to turn around normalized axis) with clear geometrical meaning. The disadvatage is that rotation composition is possible only indirectly, by performing conversion to another representation first. Multiplying axis by angle, rotation vector is obtained.

Quaternions were introduced above; among their advantages is easy rotation composition and lack of singularities (in contrast to Euler angles). Like rotation matrices, quaternions must be renormalized to avoid accumulation of numerical errros; however, quaternion normalization involves only evaluation of one square root.

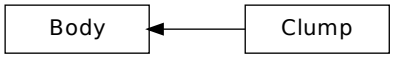
One disadvatage is that rotating vector by quaternion performs 39 multiples and adds, whereas rotating by rotation matrix needs only 15 such operations. For that reason, some libraries create rotation matrix from the quaternion internally and multiply by that matrix instead; the (orthonormal) matrix R can be directly computed from $q = w + x\hat{i} + y\hat{j} + z\hat{k}$

$$R = \begin{pmatrix} w^2 + x^2 - y^2 - z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & w^2 - x^2 + y^2 - z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & w^2 - x^2 - y^2 + z^2 \end{pmatrix}. \tag{B.21}$$

C. Class reference (yade.wrapper module)

C.1. Bodies

C.1.1. Body



class Body(*inherits Serializable*)

A particle, basic element of simulation; interacts with other bodies.

bound(*=uninitalized*)

Bound, approximating volume for the purposes of collision detection.

clumpId

Id of clump this body makes part of; invalid number if not part of clump; see Body::isStandalone, Body::isClump, Body::isClumpMember properties.

This property is not meant to be modified directly from Python, use O.bodies.appendClumped instead.

dynamic

Shorthand for Body::isDynamic

groupMask(*=1*)

Bitmask for determining interactions.

id

Unique id of this body

isClump

True if this body is clump itself, false otherwise.

isClumpMember

True if this body is clump member, false otherwise.

isDynamic(*=true*)

Whether this body will be moved by forces.

isStandalone

True if this body is neither clump, nor clump member; false otherwise.

mask

Shorthand for Body::groupMask

mat

Shorthand for Body::material

material(*=uninitalized*)

Material instance associated with this body.

shape(*=uninitalized*)

Geometrical Shape.

state(*=new State*)

Physical state.

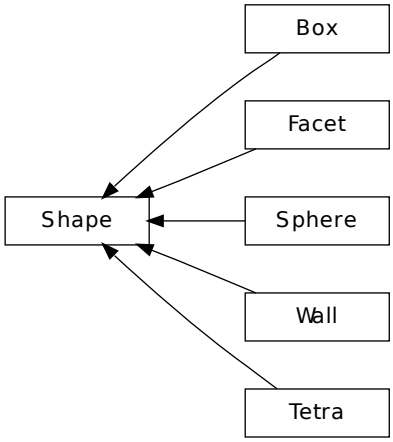
class Clump(*inherits Body → Serializable*)

Rigid aggregate of bodies

members(*=uninitalized*)

Ids and relative positions+orientations of members of the clump (should not be accessed directly)

C.1.2. Shape



class Shape(*inherits Serializable*)

Geometry of a body

color(=*Vector3r(1, 1, 1)*)

Color for rendering (normalized RGB).

dispHierarchy([*(bool)names=True*]) → list

Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex

Return class index of this instance.

highlight(=*false*)

Whether this Shape will be highlighted when rendered.

wire(=*false*)

Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class Box(*inherits Shape → Serializable*)

Box (cuboid) particle geometry. (Avoid using in new code, prefer Facet instead.

extents(=*uninitalized*)

Half-size of the cuboid

class Facet(*inherits Shape → Serializable*)

Facet (triangular particle) geometry.

edgeAdjHalfAngle(=*vector<Real>(3, 0)*)

half angle between normals of this facet and the adjacent facet [experimental]

edgeAdjIds(=*vector<body_id_t>(3, Body::ID_NONE)*)

Facet id's that are adjacent to respective edges [experimental]

vertices

Vertex positions in local coordinates.

class Sphere(*inherits Shape → Serializable*)

Geometry of spherical particle.

radius(=*NaN*)

Radius [m]

class Tetra(*inherits Shape → Serializable*)

Tetrahedron geometry.

v(=*std::vector<Vector3r>(4)*)

Tetrahedron vertices in global coordinate system.

class Wall(*inherits Shape → Serializable*)

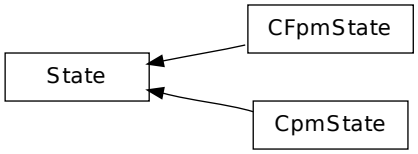
Object representing infinite plane aligned with the coordinate system (axis-aligned wall).

axis(=*0*)

Axis of the normal; can be 0,1,2 for +x, +y, +z respectively (Body's orientation is disregarded for walls)

sense(=*0*)
Which side of the wall interacts: -1 for negative only, 0 for both, +1 for positive only

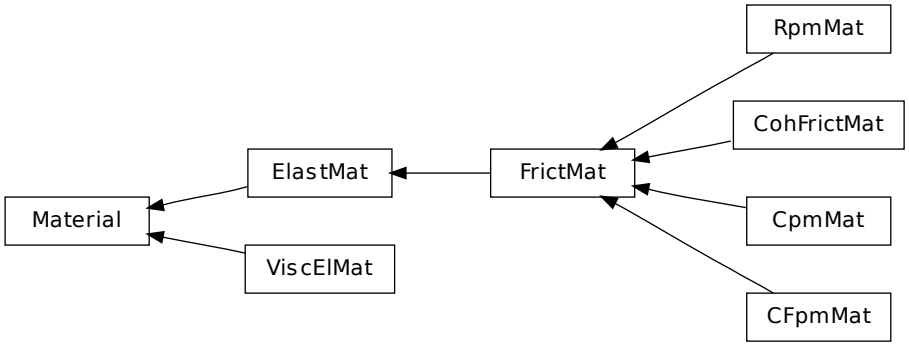
C.1.3. State



class State(*inherits Serializable*)
State of a body (spatial configuration, internal variables).
accel(=*Vector3r::Zero()*)
Current acceleration.
angAccel(=*Vector3r::Zero()*)
Current angular acceleration
angMom(=*Vector3r::Zero()*)
Current angular momentum
angVel(=*Vector3r::Zero()*)
Current angular velocity
blockedDOFs
Degress of freedom where linear/angular velocity will be always zero, regardless of applied force/torque. List of any combination of 'x','y','z','rx','ry','rz'.
inertia(=*Vector3r::Zero()*)
Inertia of associated body, in local coordinate system.
mass(=*0*)
Mass of this body
ori
Current orientation.
pos
Current position.
refOri(=*Quaternionr::Identity()*)
Reference orientation
refPos(=*Vector3r::Zero()*)
Reference position
se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.
vel(=*Vector3r::Zero()*)
Current linear velocity.
class CFpmState(*inherits State → Serializable*)
CFpm state information about each body.
None of that is used for computation (at least not now), only for post-processing.
numBrokenCohesive(=*0*)
Number of (cohesive) contacts that damaged completely
class CpmState(*inherits State → Serializable*)
State information about body use by *cpm-model*.
None of that is used for computation (at least not now), only for post-processing.
epsPlBroken(=*0*)
Plastic strain on contacts already deleted (bogus values)
epsVolumetric(=*0*)
Volumetric strain around this body (unused for now)
normDmg(=*0*)
Average damage including already deleted contacts (it is really not damage, but 1-relResidualStrength now)
normEpsPl(=*0*)
Sum of plastic strains normalized by number of contacts (bogus values)

numBrokenCohesive(=*0*)
Number of (cohesive) contacts that damaged completely
numContacts(=*0*)
Number of contacts with this body
sigma(=*Vector3r::Zero()*)
Normal stresses on the particle
tau(=*Vector3r::Zero()*)
Shear stresses on the particle.

C.1.4. Material



class Material(*inherits Serializable*)
Material properties of a body.
density(=*1000*)
Density of the material [kg/m³]
dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.
dispIndex
Return class index of this instance.
id
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via O.materials.append. (This id is necessary since yade::serialization doesn't track shared pointers, but might disappear in the future)
label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.
newAssocState() → State
Return new State instance, which is associated with this Material. Some materials have special requirement on Body::state type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, ... functions from python).
class CFpmMat(*inherits FrictMat → ElastMat → Material → Serializable*)
cohesive frictional material, for use with other CFpm classes
type(=*0*)
Type of the particle. If particles of two different types interact, it will be with friction only (no cohesion).[-]
class CohFrictMat(*inherits FrictMat → ElastMat → Material → Serializable*)

isBroken(=*true*)
isCohesive(=*true*)
class CpmMat(*inherits FrictMat → ElastMat → Material → Serializable*)
Concrete material, for use with other Cpm classes.

Note: Density is initialized to 4800 kgm☒³automatically, which gives approximate 2800 kgm☒³ on 0.5 density packing.
The model is contained in externally defined macro CPM_MATERIAL_MODEL, which features damage in tension, plasticity in shear and compression and rate-dependence. For commercial reasons, rate-dependence and compression-plasticity is not present in reduced version of the model, used when CPM_MATERIAL_MODEL is not defined. The full model will be described in detail in my (Václav Šmilauer) thesis along with calibration procedures (rigidity, poisson's ratio, compressive/tensile strength ratio, fracture energy, behavior under confinement, rate-dependent behavior).
Even the public model is useful enough to run simulation on concrete samples, such as uniaxial tension-compression test.

G_over_E(=*NaN*)
Ratio of normal/shear stiffness at interaction level [-]

dmgRateExp(=*0*)
Exponent for normal viscosity function. [-]

dmgTau(=*-1*, *deactivated if negative*)
Characteristic time for normal viscosity. [s]

epsCrackOnset(=*NaN*)
Limit elastic strain [-]

isoPrestress(=*0*)
Isotropic prestress of the whole specimen. [Pa]

neverDamage(=*false*)
If true, no damage will occur (for testing only).

plRateExp(=*0*)
Exponent for visco-plasticity function. [-]

plTau(=*-1*, *deactivated if negative*)
Characteristic time for visco-plasticity. [s]

relDuctility(=*NaN*)
Relative ductility, for damage evolution law peak right-tangent. [-]

sigmaT(=*NaN*)
Initial cohesion [Pa]

class ElastMat(*inherits Material → Serializable***)**
Purely elastic material.

poisson(=*.25*)
Poisson's ratio [-]

young(=*1e9*)
Young's modulus [Pa]

class FrictMat(*inherits ElastMat → Material → Serializable***)**
Material with internal friction.

frictionAngle(=*.5*)
Internal friction angle (in radians) [-]

class RpmMat(*inherits FrictMat → ElastMat → Material → Serializable***)**
Rock material, for use with other Rpm classes.

Brittleness(=*0*)
One of destruction parameters. [-] //(Needs to be reworked)

G_over_E(=*1*)
Ratio of normal/shear stiffness at interaction level. [-]

exampleNumber(=*0*)
Number of the specimen. This value is equal for all particles of one specimen. [-]

initCohesive(=*false*)
The flag shows, whether particles of this material can be cohesive. [-]

stressCompressMax(=*0*)
Maximal strength for compression. The main destruction parameter. [Pa] //(Needs to be reworked)

class ViscElMat(*inherits Material → Serializable***)**
Material for simple viscoelastic model of contact.
Note: **Shop::getViscoelasticFromSpheresInteraction** (and utils.getViscoelasticFromSpheresInteraction in python) compute kn, cn, ks, cs from analyti-

cal solution of a pair spheres interaction problem.

cn(=*NaN*)
: Normal viscous constant

cs(=*NaN*)
: Shear viscous constant

frictionAngle(=*NaN*)
: Friction angle [rad]

kn(=*NaN*)
: Normal elastic stiffness

ks(=*NaN*)
: Shear elastic stiffness

C.1.5. Bound

class Bound(*inherits Serializable*)
: Object bounding part of space taken by associated body; might be larger, used to optimalize collision detection

diffuseColor(=*Vector3r(1, 1, 1)*)
: Color for rendering this object

dispHierarchy([*(bool)names=True*]) → list
: Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
: Return class index of this instance.

max
: Upper corner of box containing this bound (and the Body as well)

min
: Lower corner of box containing this bound (and the Body as well)

class Aabb(*inherits Bound → Serializable*)
: Axis-aligned bounding box, for use with InsertionSortCollider. (This class is quasi-redundant since min,max are already contained in Bound itself. That might change at some point, though.)

C.2. Interactions

C.2.1. Interaction

class Interaction(*inherits Serializable*)
: Interaction between pair of bodies.

cellDist
: Distance of bodies in cell size units, if using periodic boundary conditions; id2 is shifted by this number of cells from its State::pos coordinates for this interaction to exist. Assigned by the collider.

> **Warning:** (internal) cellDist must survive Interaction::reset(), it is only initialized in ctor. Interaction that was cancelled by the constitutive law, was reset() and became only potential must have the priod information if the geometric functor again makes it real. Good to know after few days of debugging that :-)

geom
: Shorthand for Interaction::interactionGeometry

id1
: Id of the first body in this interaction.

id2
Id of the second body in this interaction.

interactionGeometry(=*uninitalized*)
Geometry part of the interaction.

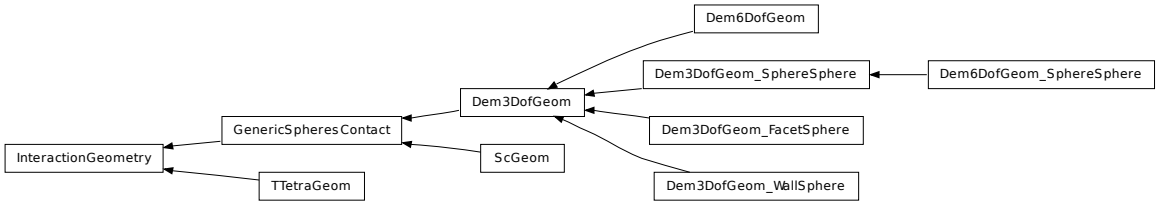
interactionPhysics(=*uninitalized*)
Physical (material) part of the interaction.

isReal
True if this interaction has both geom and phys; False otherwise.

iterMadeReal(=*-1*)
Step number at which the interaction was fully (in the sense of interactionGeometry and interactionPhysics) created. (Should be touched only by InteractionPhysicsDispatcher and InteractionDispatchers, therefore they are made friends of Interaction

phys
Shorthand for Interaction::interactionPhysics

C.2.2. InteractionGeometry



class InteractionGeometry(*inherits Serializable***)**
Geometrical configuration of interaction

dispHierarchy([*(bool)names=True*]**)** → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

class Dem3DofGeom(*inherits GenericSpheresContact → InteractionGeometry → Serializable***)**
Abstract base class for representing contact geometry of 2 elements that has 3 degrees of freedom: normal (1 component) and shear (Vector3r, but in plane perpendicular to the normal).

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

logCompression(=*false*)
make strain go to -∞ for length going to zero (false by default).

refLength(=*uninitalized*)
some length used to convert displacements to strains. *(auto-computed)*

se31(=*uninitalized*)
Copy of body #1 se3 (needed to compute torque from the contact, strains etc). *(auto-updated)*

se32(=*uninitalized*)
Copy of body #2 se3. *(auto-updated)*

class Dem3DofGeom_FacetSphere(*inherits Dem3DofGeom → GenericSpheresContact → InteractionGeometry → Serializable***)**
Class representing facet+sphere in contact which computes 3 degrees of freedom (normal and shear deformation).

cp1pt(=*uninitalized*)
Reference contact point on the facet in facet-local coords.

cp2rel(=*uninitalized*)
Orientation between +x and the reference contact point (on the sphere) in sphere-local coords

effR2(=*uninitalized*)
Effective radius of sphere

localFacetNormal(=*uninitalized*)

Unit normal of the facet plane in facet-local coordinates

class Dem3DofGeom_SphereSphere(*inherits Dem3DofGeom → GenericSpheresContact → InteractionGeometry → Serializable***)**

Class representing 2 spheres in contact which computes 3 degrees of freedom (normal and shear deformation).

cp1rel(*=uninitalized***)**

Sphere's #1 relative orientation of the contact point with regards to sphere-local +x axis (quasi-constant)

cp2rel(*=uninitalized***)**

Same as cp1rel, but for sphere #2.

effR1(*=uninitalized***)**

Effective radius of sphere #1; can be smaller/larger than refR1 (the actual radius), but quasi-constant throughout interaction life

effR2(*=uninitalized***)**

Same as effR1, but for sphere #2.

class Dem3DofGeom_WallSphere(*inherits Dem3DofGeom → GenericSpheresContact → InteractionGeometry → Serializable***)**

Representation of contact between wall and sphere, based on Dem3DofGeom.

cp1pt(*=uninitalized***)**

initial contact point on the wall, relative to the current contact point

cp2rel(*=uninitalized***)**

orientation between +x and the reference contact point (on the sphere) in sphere-local coords

effR2(*=uninitalized***)**

effective radius of sphere

class Dem6DofGeom(*inherits Dem3DofGeom → GenericSpheresContact → InteractionGeometry → Serializable***)**

Abstract class for providing torsion and bending, in addition to inherited normal and shear strains.

class Dem6DofGeom_SphereSphere(*inherits Dem3DofGeom_SphereSphere → Dem3DofGeom → GenericSpheresContact → InteractionGeometry → Serializable***)**

Class representing 2 sphere in contact which computes 6 degrees of freedom (normal, shear, bending and twisting deformation)

initRelOri12(*=uninitalized***)**

Initial relative orientation of spheres, used for bending and twisting computation.

class GenericSpheresContact(*inherits InteractionGeometry → Serializable***)**

Class uniting ScGeom and Dem3DofGeom, for the purposes of GlobalStiffnessTimeStepper. (It might be removed inthe future). Do not use this class directly.

normal(*=uninitalized***)**

Unit vector oriented along the interaction. *(auto-updated)*

refR1(*=uninitalized***)**

Reference radius of particle #1. *(auto-computed)*

refR2(*=uninitalized***)**

Reference radius of particle #2. *(auto-computed)*

class ScGeom(*inherits GenericSpheresContact → InteractionGeometry → Serializable***)**

Class representing geometry of two spheres in contact. The contact has 3 DOFs (normal and 2×shear) and uses incremental algorithm for updating shear. (For shear formulated in total displacements and rotations, see Dem3DofGeom and related classes).

We use symbols $\mathbf{x}$, $\mathbf{v}$, $\boldsymbol{\omega}$ respectively for position, linear and angular velocities (all in global coordinates) and r for particles radii; subscripted with 1 or 2 to distinguish 2 spheres in contact. Then we compute unit contact normal

$$\mathbf{n} = \frac{\mathbf{x}_2 - \mathbf{x}_1}{\|\mathbf{x}_2 - \mathbf{x}_1\|}$$

Relative velocity of spheres is then

$$\mathbf{v}_{12} = (\mathbf{v}_2 + \boldsymbol{\omega}_2 \times (-r_2\mathbf{n})) - (\mathbf{v}_1 + \boldsymbol{\omega}_1 \times (r_1\mathbf{n}))$$

and its shear component

$$\Delta\mathbf{v}^s_{12} = \mathbf{v}_{12} - (\mathbf{n} \cdot \mathbf{v}_{12})\mathbf{n}.$$

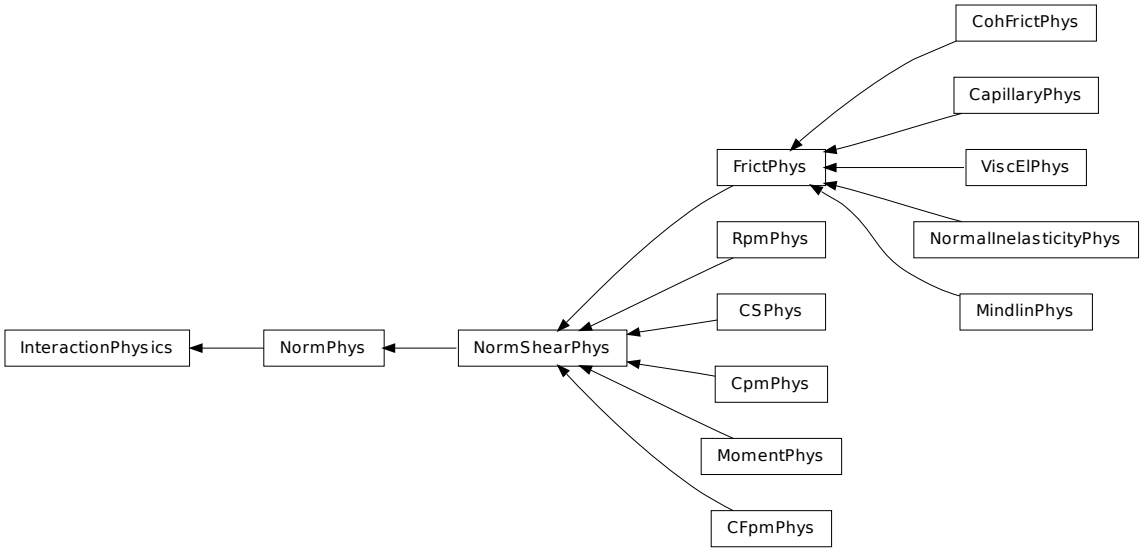
Tangential displacement increment over last step then reads

$$\mathbf{x}_{12}^{s} = \Delta t \mathbf{v}_{12}^{s}.$$

contactPoint(=*Vector3r::Zero()*)
Reference point of the contact. *(auto-computed)*
penetrationDepth
documentation
prevNormal(=*Vector3r::Zero()*)
Normal of the contact in the previous step. *(auto-computed)*
shear(=*Vector3r::Zero()*)
Total value of the current shear. Update the value using ScGeom::updateShear. *(auto-computed)*

class TTetraGeom(*inherits InteractionGeometry → Serializable*)
Geometry of interaction between 2 tetrahedra, including volumetric characteristics
contactPoint(=*uninitalized*)
Contact point (global coords)
equivalentCrossSection(=*NaN*)
Cross-section of the overlap (perpendicular to the axis of least inertia
equivalentPenetrationDepth(=*NaN*)
??
maxPenetrationDepthA(=*NaN*)
??
maxPenetrationDepthB(=*NaN*)
??
normal(=*uninitalized*)
Normal of the interaction, directed in the sense of least inertia of the overlap volume
penetrationVolume(=*NaN*)
Volume of overlap [m³]

C.2.3. InteractionPhysics



class InteractionPhysics(*inherits Serializable*)
Physical (material) properties of interaction.
dispHierarchy([*(bool)names=True*]**)** → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

class CFpmPhys(*inherits NormShearPhys → NormPhys → InteractionPhysics → Serializable*)
Representation of a single interaction of the CFpm type, storage for relevant parameters

FnMax(*=0*)
Defines the maximum admissible normal force in traction FnMax=tensileStrength*crossSection, with crossSection=pi*Rmin^2. [Pa]

FsMax(*=0*)
Defines the maximum admissible tangential force in shear FsMax=cohesion*FnMax, with crossSection=pi*Rmin^2. [Pa]

cumulativeRotation(*=0*)
Cumulated rotation... [-]

frictionAngle(*=0*)
defines Coulomb friction. [deg]

initD(*=0*)
equilibrium distance for particles. Computed as the initial interparticular distance when bonded particle interact. initD=0 for non cohesive interactions.

initialOrientation1(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Used for moment computation.

initialOrientation2(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Used for moment computation.

isCohesive(*=false*)
If false, particles interact in a frictional way. If true, particles are bonded regarding the given cohesion and tensileStrength.

kr(*=0*)
Defines the stiffness to compute the resistive moment in rotation. [-]

maxBend(*=0*)
Defines the maximum admissible resistive moment in rotation Mtmax=maxBend*Fn, maxBend=eta*meanRadius. [m]

moment_bending(*=Vector3r::Zero()*)
[N.m]

moment_twist(*=Vector3r::Zero()*)
[N.m]

prevNormal(*=Vector3r::Zero()*)
Normal to the contact at previous time step.

strengthSoftening(*=0*)
Defines the softening when Dtensile is reached to avoid explosion. Typically, when D > Dtensile, Fn=FnMax - (kn/strengthSoftening)*(Dtensile-D). [-]

tanFrictionAngle(*=0*)
Tangent of frictionAngle. [-]

class CSPhys(*inherits NormShearPhys → NormPhys → InteractionPhysics → Serializable*)
Physical properties for Cundall&Strack constitutive law, created by Ip2_2xFrictMat_CSPhys.

frictionAngle(*=NaN*)
Friction angle of the interaction. *(auto-computed)*

tanFrictionAngle(*=NaN*)
Precomputed tangent of CSPhys::frictionAngle. *(auto-computed)*

class CapillaryPhys(*inherits FrictPhys → NormShearPhys → NormPhys → InteractionPhysics → Serializable*)
Physics (of interaction) for Law2_ScGeom_CapillaryPhys_Capillarity. Rk: deprecated -> needs some work to be conform with the new formalism!

CapillaryPressure(*=0.*)
Value of the capillary pressure Uc defines as Ugas-Uliquid

Delta1(*=0.*)
Defines the surface area wetted by the meniscus on the smallest grains of radius R1 (R1<R2)

Delta2(*=0.*)
Defines the surface area wetted by the meniscus on the biggest grains of radius R2 (R1<R2)

Fcap(*=Vector3r::Zero()*)
Capillary Force produces by the presence of the meniscus

Vmeniscus(=*0.*)
Volume of the menicus
fusionNumber(=*0.*)
Indicates the number of meniscii that overlap with this one
meniscus(=*false*)
Presence of a meniscus if true

class CohFrictPhys(*inherits FrictPhys → NormShearPhys → NormPhys → InteractionPhysics → Serializable*)

cohesionBroken(=*true*)
is cohesion active? will be set false when a fragile contact is broken
cohesionDisablesFriction(=*false*)
is shear strength the sum of friction and adhesion or only adhesion?
currentContactOrientation(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
fragile(=*true*)
do cohesion disapear when contact strength is exceeded?
initialContactOrientation(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
initialOrientation1(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
initialOrientation2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
initialPosition1(=*Vector3r(0, 0, 0)*)
initialPosition2(=*Vector3r(0, 0, 0)*)
kr(=*0*)
rotational stiffness [N.m/rad]
moment_bending(=*Vector3r(0, 0, 0)*)
moment_twist(=*Vector3r(0, 0, 0)*)
normalAdhesion(=*0*)
tensile strength
orientationToContact1(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
orientationToContact2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
shearAdhesion(=*0*)
cohesive part of the shear strength (a frictional term might be added depending on Law2_ScGeom_CohFrictPhys_ElasticPlastic::cohesionDisablesFriction)
twistCreep(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)

class CpmPhys(*inherits NormShearPhys → NormPhys → InteractionPhysics → Serializable*)

Representation of a single interaction of the Cpm type: storage for relevant parameters.
Evolution of the contact is governed by Law2_Dem3DofGeom_CpmPhys_Cpm, that includes damage effects and chages of parameters inside CpmPhys. See *cpm-model* for details.

E(=*NaN*)
normal modulus (stiffness / crossSection) [Pa]
Fn
Magnitude of normal force.
Fs
Magnitude of shear force
G(=*NaN*)
shear modulus [Pa]
crossSection(=*NaN*)
equivalent cross-section associated with this contact [m²]
dmgOverstress(=*0*)
damage viscous overstress (at previous step or at current step)
dmgRateExp(=*0*)
exponent in the rate-dependent damage evolution
dmgStrain(=*0*)
damage strain (at previous or current step)
dmgTau(=*-1*)
characteristic time for damage (if non-positive, the law without rate-dependence is used)
epsCrackOnset(=*NaN*)
strain at which the material starts to behave non-linearly
epsFracture(=*NaN*)
strain where the damage-evolution law tangent from the top (epsCrackOnset) touches the axis;

since the softening law is exponential, this doesn't mean that the contact is fully damaged at this point, that happens only asymptotically

epsN
Current normal strain

epsNPl(=*0*)
normal plastic strain (initially zero)

epsPlSum(=*0*)
cummulative shear plastic strain measure (scalar) on this contact

epsT
Transversal strain (not used)

epsTrans(=*0*)
Transversal strain (perpendicular to the contact axis)

isCohesive(=*false*)
if not cohesive, interaction is deleted when distance is greater than zero.

isoPrestress(=*0*)
"prestress" of this link (used to simulate isotropic stress)

kappaD(=*0*)
Up to now maximum normal strain (semi-norm), non-decreasing in time.

neverDamage(=*false*)
the damage evolution function will always return virgin state

omega
Damage internal variable

plRateExp(=*0*)
exponent in the rate-dependent viscoplasticity

plTau(=*-1*)
characteristic time for viscoplasticity (if non-positive, no rate-dependence for shear)

relResidualStrength
Relative residual strength

sigmaN
Current normal stress

sigmaT
Current shear stress

tanFrictionAngle(=*NaN*)
tangens of internal friction angle [-]

undamagedCohesion(=*NaN*)
virgin material cohesion [Pa]

class FrictPhys(*inherits NormShearPhys → NormPhys → InteractionPhysics → Serializable***)**
Interaction with friction

prevNormal(=*Vector3r::Zero()*)
unit normal of the contact plane in previous step

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

class MindlinPhys(*inherits FrictPhys → NormShearPhys → NormPhys → InteractionPhysics → Serializable***)**
Representation of an interaction of the Mindlin type.

kno(=*0.0*)
Constant value in the formulation of the normal stiffness

kso(=*0.0*)
Constant value in the formulation of the tangential stiffness

class MomentPhys(*inherits NormShearPhys → NormPhys → InteractionPhysics → Serializable***)**
Physical interaction properties for use with Law2_SCG_MomentPhys_CohesionlessMomentRotation, created by Ip2_MomentMat_MomentMat_MomentPhys.

Eta(=*0*)
??

cumulativeRotation(=*0*)
??

frictionAngle(=*0*)

Friction angle [rad]

initialOrientation1(=*Quaternionr::Identity()*)

??

initialOrientation2(=*Quaternionr::Identity()*)

??

kr(=*0*)

rolling stiffness

moment_bending(=*Vector3r::Zero()*)

??

moment_twist(=*Vector3r::Zero()*)

??

prevNormal(=*Vector3r::Zero()*)

Normal in the previous step.

shear(=*Vector3r::Zero()*)

??

tanFrictionAngle(=*0*)

Tangent of friction angle

class NormPhys(*inherits InteractionPhysics → Serializable***)**

Abstract class for interactions that have normal stiffness.

kn(=*NaN*)

Normal stiffness

normalForce(=*Vector3r::Zero()*)

Normal force after previous step (in global coordinates).

class NormShearPhys(*inherits NormPhys → InteractionPhysics → Serializable***)**

Abstract class for interactions that have shear stiffnesses, in addition to normal stiffness. This class is used in the PFC3d-style stiffness timestepper.

ks(=*NaN*)

Shear stiffness

shearForce(=*Vector3r::Zero()*)

Shear force after previous step (in global coordinates).

class NormalInelasticityPhys(*inherits FrictPhys → NormShearPhys → NormPhys → InteractionPhysics → Serializable***)**

Physics (of interaction) for using Law2_ScGeom_NormalInelasticityPhys_NormalInelasticity

currentContactOrientation(=*Quaternionr::Identity()*)

forMaxMoment(=*1.0*)

parameter stored for each interaction, and allowing to compute the maximum value of the exchanged torque : TorqueMax= forMaxMoment * NormalForce

initialContactOrientation(=*Quaternionr::Identity()*)

initialOrientation1(=*Quaternionr::Identity()*)

initialOrientation2(=*Quaternionr::Identity()*)

initialPosition1(=*Vector3r::Zero()*)

initialPosition2(=*Vector3r::Zero()*)

kr(=*0.0*)

the rolling stiffness of the rigidity

orientationToContact1(=*Quaternionr::Identity()*)

orientationToContact2(=*Quaternionr::Identity()*)

previousFn(=*0.0*)

the value of the normal force at the last time step

previousun(=*0.0*)

the value of this un at the last time step

unMax(=*0.0*)

the maximum value of penetration depth of the history of this interaction

class RpmPhys(*inherits NormShearPhys → NormPhys → InteractionPhysics → Serializable***)**

Representation of a single interaction of the Cpm type: storage for relevant parameters.
Evolution of the contact is governed by Law2_Dem3DofGeom_CpmPhys_Cpm, that includes damage effects and chages of parameters inside CpmPhys

E(=*NaN*)

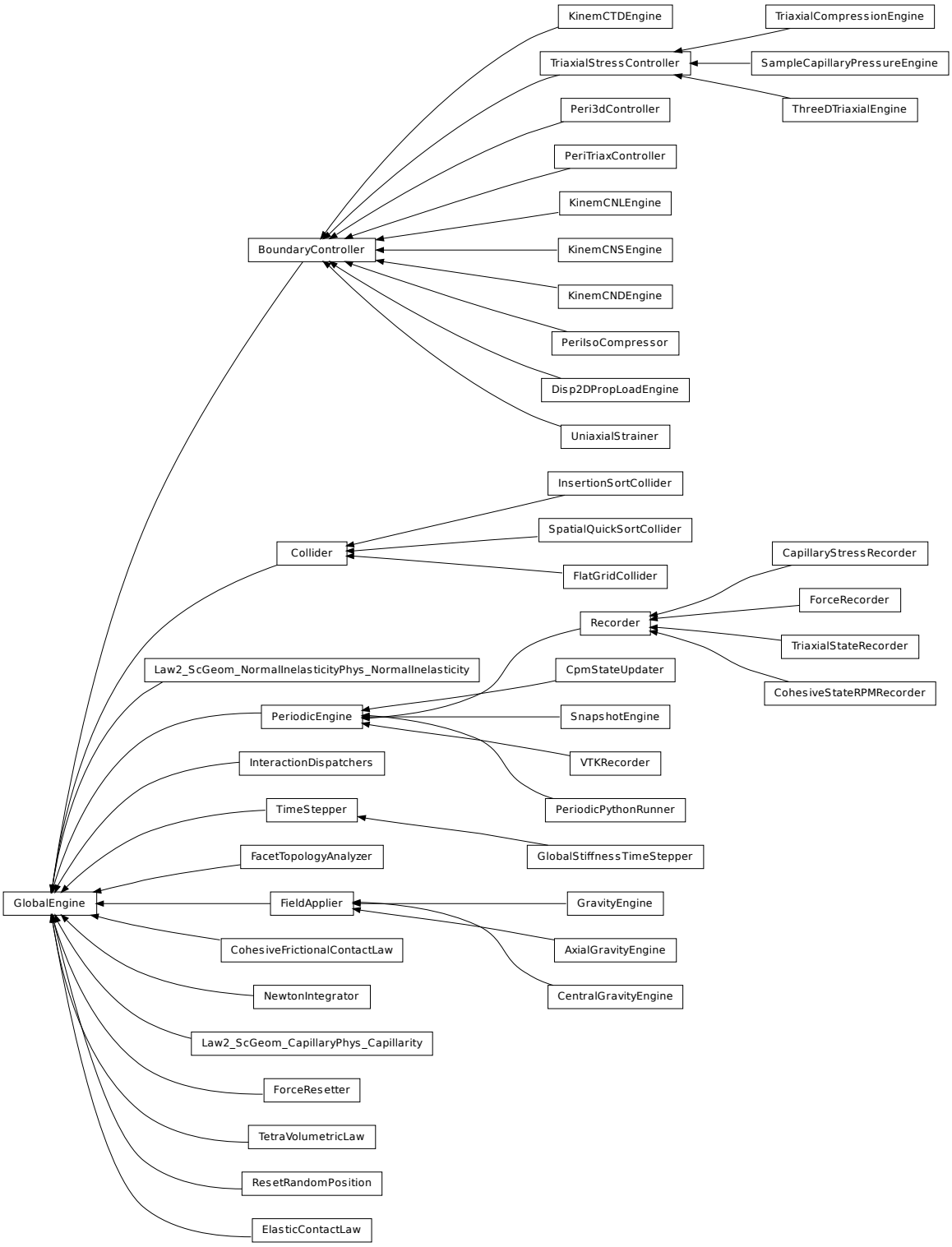
normal modulus (stiffness / crossSection) [Pa]

G(=*NaN*)
shear modulus [Pa]
crossSection(=*0*)
equivalent cross-section associated with this contact [m²]
isCohesive(=*false*)
if not cohesive, interaction is deleted when distance is greater than lengthMaxTension or less than lengthMaxCompression.
lengthMaxCompression(=*0*)
Maximal penetration of particles during compression. If it is more, the interaction is deleted [m]
lengthMaxTension(=*0*)
Maximal distance between particles during tension. If it is more, the interaction is deleted [m]
tanFrictionAngle(=*NaN*)
tangens of internal friction angle [-]

class ViscElPhys(*inherits FrictPhys → NormShearPhys → NormPhys → InteractionPhysics → Serializable***)**

InteractionPhysics created from ViscElMat, for use with Law2_ScGeom_ViscElPhys_Basic.
cn(=*NaN*)
Normal viscous constant
cs(=*NaN*)
Shear viscous constant

C.3. Global engines



class GlobalEngine(*inherits Engine → Serializable***)**
Engine that will generally affect the whole simulation (contrary to PartialEngine).

class AxialGravityEngine(*inherits FieldApplier → GlobalEngine → Engine → Serializable***)**
Apply acceleration (independent of distance) directed towards an axis.

acceleration(*=0***)**
Acceleration magnitude [kgms⊠²]

axisDirection(*=Vector3r::UnitX()***)**

direction of the gravity axis (will be normalized automatically)

axisPoint(=*Vector3r::Zero()*)

Point through which the axis is passing.

class BoundaryController(*inherits GlobalEngine → Engine → Serializable*)

Base for engines controlling boundary conditions of simulations. Not to be used directly.

class CapillaryStressRecorder(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Records informations from capillary meniscii on samples submitted to triaxial compressions. -> New formalism needs to be tested!!!

class CentralGravityEngine(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)

Engine applying acceleration to all bodies, towards a central body.

accel(=*0*)

Acceleration magnitude [kgms⊠²]

centralBody(=*Body::ID_NONE*)

The body towards which all other bodies are attracted.

reciprocal(=*false*)

If true, acceleration will be applied on the central body as well.

class CohesiveFrictionalContactLaw(*inherits GlobalEngine → Engine → Serializable*)

[DEPRECATED] Loop over interactions applying Law2_ScGeom_CohFrictPhys_ElasticPlastic on all interactions.

Note: Use InteractionDispatchers and Law2_ScGeom_CohFrictPhys_ElasticPlastic instead of this class for performance reasons.

always_use_moment_law(=*false*)

If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

creep_viscosity(=*false*)

creep viscosity [Pa.s/m]. probably should be moved to Ip2_2xCohFrictMat_CohFrictPhys...

detectBrokenBodies(=*false*)

erosionActivated(=*false*)

momentRotationLaw(=*false*)

use bending/twisting moment at contacts. See CohesiveFrictionalContactLaw::always_use_moment_law for details.

neverErase(=*false*)

Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

shear_creep(=*false*)

activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

twist_creep(=*false*)

activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

class CohesiveStateRPMRecorder(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Store number of cohesive contacts in RPM model to file.

numberCohesiveContacts(=*0*)

Number of cohesive contacts found at last run. [-]

class Collider(*inherits GlobalEngine → Engine → Serializable*)

Abstract class for finding spatial collisions between bodies.

class CpmStateUpdater(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Update CpmState of bodies based on state variables in CpmPhys of interactions with this bod. In particular, bodies' colors and CpmState::normDmg depending on average damage of their interactions and number of interactions that were already fully broken and have disappeared is updated. This engine contains its own loop (2 loops, more precisely) over all bodies and should be run periodically to update colors during the simulation, if desired.

avgRelResidual(=*NaN*)

Average residual strength at last run.

maxOmega(=*NaN*)

Globally maximum damage parameter at last run.

class Disp2DPropLoadEngine(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

Disturbs a simple shear sample in a given displacement direction

This engine allows to apply, on a simple shear sample, a loading controlled by du/dgamma = cste, which is equivalent to du + cste' * dgamma = 0 (proportionnal path loadings). To do so, the upper plate of the simple shear box is moved in a given direction (corresponding to a given du/dgamma), whereas lateral plates are moved so that the box remains closed. This engine can easily be used to perform directionnal probes, with a python script launching successivly the same .xml which contains this engine, after having modified the direction of loading (see *theta* attribute). That's why this Engine contains a *saveData* procedure which can save data on the state of the sample at the end of the loading (in case of successive loadings - for successive directions - through a python script, each line would correspond to one direction of loading).

Key(=*'"'*)
: string to add at the names of the saved files, and of the output file filled by *saveData*

LOG(=*false*)
: boolean controling the output of messages on the screen

id_boxback(=*4*)
: the id of the wall at the back of the sample

id_boxbas(=*1*)
: the id of the lower wall

id_boxfront(=*5*)
: the id of the wall in front of the sample

id_boxleft(=*0*)
: the id of the left wall

id_boxright(=*2*)
: the id of the right wall

id_topbox(=*3*)
: the id of the upper wall

nbre_iter(=*0*)
: the number of iterations of loading to perform

theta(=*0.0*)
: the angle, in a (gamma,h=-u) plane from the gamma - axis to the perturbation vector (trigo wise) [degrees]

v(=*0.0*)
: the speed at which the perturbation is imposed. In case of samples which are more sensitive to normal loadings than tangential ones, one possibility is to take v = V_shear - | (V_shear-V_comp)*sin(theta) | => v=V_shear in shear; V_comp in compression [m/s]

class ElasticContactLaw(*inherits GlobalEngine → Engine → Serializable***)**

[DEPRECATED] Loop over interactions applying Law2_ScGeom_FrictPhys_Basic on all interactions.

Note: Use InteractionDispatchers and Law2_ScGeom_FrictPhys_Basic instead of this class for performance reasons.

neverErase(=*false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

useShear(=*false*)
: Use ScGeom::updateShear rather than ScGeom::rotateAndGetShear for shear force computation.

class FacetTopologyAnalyzer(*inherits GlobalEngine → Engine → Serializable***)**

Initializer for filling adjacency geometry data for facets.

Common vertices and common edges are identified and mutual angle between facet faces is written to Facet instances. If facets don't move with respect to each other, this must be done only at the beginng.

commonEdgesFound(=*0*)
: how many common edges were identified during last run. *(auto-updated)*

commonVerticesFound(=*0*)
: how many common vertices were identified during last run. *(auto-updated)*

projectionAxis(=*Vector3r::UnitX()*)
: Axis along which to do the initial vertex sort

relTolerance(=*1e-4*)
: maximum distance of 'identical' vertices, relative to minimum facet size

class FieldApplier(*inherits GlobalEngine → Engine → Serializable***)**
Base for engines controlling boundary conditions of simulations. Not to be used directly.

class FlatGridCollider(*inherits Collider → GlobalEngine → Engine → Serializable***)**
Non-optimized grid collider, storing grid as dense flat array. Each body is assigned to (possibly multiple) cells, which are arranged in regular grid between *aabbMin* and *aabbMax*, with cell size *step* (same in all directions). Bodies outsize (*aabbMin*, *aabbMax*) are handled gracefully, assigned to closest cells (this will create spurious potential interactions). *verletDist* determines how much is each body enlarged to avoid collision detection at every step.
Note: This collider keeps all cells in linear memory array, therefore will be memory-inefficient for sparse simulations.

> **Warning:** Body::bound objects are not used, BoundFunctors are not used either: assigning cells to bodies is hard-coded internally. Currently handles Shapes are: Sphere.

Note: Periodic boundary is not handled (yet).

aabbMax(*=Vector3r::Zero()***)**
Upper corner of grid (approximate, might be rouded up to *minStep*.

aabbMin(*=Vector3r::Zero()***)**
Lower corner of grid.

step(*=0***)**
Step in the grid (cell size)

verletDist(*=0***)**
Length by which enlarge space occupied by each particle; avoids running collision detection at every step.

class ForceRecorder(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable***)**
Engine saves the resulting force affecting to Subscribed bodies. For instance, can be useful for defining the forces, which affect to _buldozer_ during its work.

subscribedBodies(*=uninitalized***)**
Lists of bodies whose state will be measured

class ForceResetter(*inherits GlobalEngine → Engine → Serializable***)**
Reset all forces stored in Scene::forces (**O.forces** in python). Typically, this is the first engine to be run at every step.

class GlobalStiffnessTimeStepper(*inherits TimeStepper → GlobalEngine → Engine → Serializable***)**
An engine assigning the time-step as a fraction of the minimum eigen-period in the problem

defaultDt(*=1***)**
used as default AND as max value of the timestep

previousDt(*=1***)**
last computed dt *(auto-updated)*

timestepSafetyCoefficient(*=0.8***)**
safety factor between the minimum eigen-period and the final assigned dt (less than 1))

class GravityEngine(*inherits FieldApplier → GlobalEngine → Engine → Serializable***)**
Engine applying constant acceleration to all bodies.

gravity(*=Vector3r::Zero()***)**
Acceleration [kgms⊠²]

class InsertionSortCollider(*inherits Collider → GlobalEngine → Engine → Serializable***)**
Collider with O(n log(n)) complexity, using Aabb for bounds.
At the initial step, Bodies' bounds (along sortAxis) are first std::sort'ed along one axis (sortAxis), then collided. The initial sort has $O(n^2)$ complexity, see Colliders' performance for some information (There are scripts in examples/collider-perf for measurements).
Insertion sort is used for sorting the bound list that is already pre-sorted from last iteration, where each inversion calls checkOverlap which then handles either overlap (by creating interaction if necessary) or its absence (by deleting interaction if it is only potential).
Bodies without bounding volume (such as clumps) are handled gracefully and never collide. Deleted bodies are handled gracefully as well.
This collider handles periodic boundary conditions. There are some limitations, notably:

1. No body can have Aabb larger than cell's half size in that respective dimension. You get exception it it does and gets in interaction.

2.No body can travel more than cell's distance in one step; this would mean that the simulation is numerically exploding, and it is only detected in some cases.

Stride can be used to avoid running collider at every step by enlarging the particle's bounds, tracking their velocities and only re-run if they might have gone out of that bounds (see Verlet list for brief description and background) . This requires cooperation from NewtonIntegrator as well as BoundDispatcher, which will be found among engines automatically (exception is thrown if they are not found).

If you wish to use strides, set **sweepLength** (length by which bounds will be enlarged in all directions) to some value, e.g. 0.05 × typical particle radius. This parameter expresses the tradeoff between many potential interactions (running collider rarely, but with longer exact interaction resolution phase) and few potential interactions (running collider more frequently, but with less exact resolutions of interactions); it depends mainly on packing density and particle radius distribution. If you additionally set **nBins** to >=1, not all particles will have their bound enlarged by **sweepLength**; instead, they will be put to bins (in the statistical sense) based on magnitude of their velocity; **sweepLength** will only be used for particles in the fastest bin, whereas only proportionally smaller length will be used for slower particles; The coefficient between bin's velocities is given by **binCoeff**.

binCoeff(=*5*)

Coefficient of bins for velocities, i.e. if **binCoeff==5**, successive bins have 5 × smaller velocity peak than the previous one. (Passed to VelocityBins)

binOverlap(=*0.8*)

Relative bins hysteresis, to avoid moving body back and forth if its velocity is around the border value. (Passed to VelocityBins)

fastestBodyMaxDist(=*-1*)

Maximum displacement of the fastest body since last run; if >= sweepLength, we could get out of bboxes and will trigger full run. DEPRECATED, was only used without bins. *(auto-updated)*

histInterval(=*100*)

How often to show velocity bins graphically, if debug logging is enabled for VelocityBins.

maxRefRelStep(=*.3*)

(Passed to VelocityBins)

nBins(=*0*)

Number of velocity bins for striding. If <=0, bin-less strigin is used (this is however DEPRECATED).

periodic

Whether the collider is in periodic mode (read-only; for debugging) *(auto-updated)*

sortAxis(=*0*)

Axis for the initial contact detection.

sortThenCollide(=*false*)

Separate sorting and colliding phase; it is MUCH slower, but all interactions are processed at every step; this effectively makes the collider non-persistent, not remembering last state. (The default behavior relies on the fact that inversions during insertion sort are overlaps of bounding boxes that just started/ceased to exist, and only processes those; this makes the collider much more efficient.)

strideActive

Whether striding is active (read-only; for debugging). *(auto-updated)*

sweepFactor(=*1.05*)

Overestimation factor for the sweep velocity; must be >=1.0. Has no influence on sweepLength, only on the computed stride. [DEPRECATED, is used only when bins are not used].

sweepLength(=*-1*, *Stride deactivated*)

Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if negative.

class **InteractionDispatchers(***inherits GlobalEngine → Engine → Serializable*)

Unified dispatcher for handling interaction loop at every step, for parallel performance reasons.

___init___() → None

object __init__(tuple args, dict kwds)

___init___((object)arg2, (object)arg3, (object)arg4) → **object :** Construct from lists Ig2,

Ip2, Law functors respectively; they will be passed to interal dispatchers, which you might retrieve. (NOT YET DONE: Optionally, list of IntrCallbacks can be provided as fourth argument.)

callbacks(=*uninitalized*)
Callbacks which will be called for every Interaction, if activated.

geomDispatcher
InteractionGeometryDispatcher object that is used for dispatch.

lawDispatcher
LawDispatcher object used for dispatch.

physDispatcher
InteractionPhysicsDispatcher object used for dispatch.

class KinemCNDEngine(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**

To apply a constant normal displacement shear for a parallelogram box
This engine, designed for simulations implying a simple shear box (SimpleShear Preprocessor), allows to perform a constant normal displacement shear, by translating horizontally the upper plate, while the lateral ones rotate so that they always keep contact with the lower and upper walls.

Key(=*'"'*)
string to add at the names of the saved files

gamma(=*0.0*)
the current value of the tangential displacement

gamma_save(=*uninitalized*)
vector with the values of gamma at which a save of the simulation is performed [m]

gammalim(=*0.0*)
the value of the tangential displacement at wich the displacement is stopped [m]

id_boxleft(=*0*)
the id of the left wall

id_boxright(=*2*)
the id of the right wall

id_topbox(=*3*)
the id of the upper wall

shearSpeed(=*0.0*)
the speed at which the shear is performed : speed of the upper plate [m/s]

temoin_save(=*uninitalized*)
vector (same length as 'gamma_save'), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0)

class KinemCNLEngine(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**

To apply a constant normal stress shear for a parallelogram box (simple shear)
This engine, used in simulations issued from SimpleShear Preprocessor, allows to translate horizontally the upper plate while the lateral ones rotate so that they always keep contact with the lower and upper walls.
In fact the upper plate can move not only horizontally but also vertically, so that the normal stress acting on it remains constant (this constant value is not choosen by the user but is the one that exists at the beginning of the simulation)
The right vertical displacements which will be allowed are computed from the rigidity Kn of the sample over the wall (so to cancel a deltaSigma, a normal dplt deltaSigma*S/(Kn) is set)
The movement is moreover controlled by the user via a *shearSpeed* which will be the speed of the upper wall, and by a maximum value of horizontal displacement *gammalim*, after which the shear stops.
Note: Not only the positions of walls are updated but also their speeds, which is all but useless considering the fact that in the contact laws these velocities of bodies are used to compute values of tangential relative displacements.

> **Warning:** Because of this last point, if you want to use later saves of simulations executed with this Engine, but without that stopMovement was executed, your boxes will keep their speeds => you will have to cancel them 'by hand' in the .xml.

F_0(=*0.0*)
: the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). All control will be performed in order to keep this value of F_0 [N]

Key(=*'"'*)
: string to add at the names of the saved files

LOG(=*false*)
: boolean controling the output of messages on the screen

coeff_dech(=*1.0*)
: in the case of the use of 'Law2_ScGeom_NormalInelasticityPhys_NormalInelasticity' for ex, where kn(unload)#kn(load). The engine cares to find the value at the first run

firstRun(=*true*)
: boolean set to false as soon as the engine has done its job one time : usefull to know if the force acting on the plate is known or not (and if F_0 has to be initialized)

gamma(=*0.0*)
: current value of tangential displacement [m]

gamma_save(=*uninitalized*)
: vector with the values of gamma at which a save of the simulation is performed [m]

gammalim(=*0.0*)
: the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

id_boxback(=*4*)
: the id of the wall at the back of the sample

id_boxbas(=*1*)
: the id of the lower wall

id_boxfront(=*5*)
: the id of the wall in front of the sample

id_boxleft(=*0*)
: the id of the left wall

id_boxright(=*2*)
: the id of the right wall

id_topbox(=*3*)
: the id of the upper wall

max_vel(=*1.0*)
: to limit the speed of the vertical displacements done to maintain F equal to F_0 [m/s]

shearSpeed(=*0.0*)
: the speed at wich the shearing is performed : speed of the upper plate [m/s]

temoin_save(=*uninitalized*)
: vector (same length as 'gamma_save'), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0)

wallDamping(=*0.2*)
: the vertical displacements done to maintain F equal to F_0 are in fact damped, through this wallDamping

class KinemCNSEngine(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**

To apply a constant normal rigidity shear for a parallelogram box (simple shear)
This engine, useable in simulations implying one deformable parallelepipedic box (e.g. SimpleShear Preprocessor), allows to translate horizontally the upper plate while the lateral ones rotate so that they always keep contact with the lower and upper walls. The upper plate can move not only horizontally but also vertically, so that the normal rigidity defined by DeltaF(upper plate)/DeltaU(upper plate) = constant (= KnC defined by the user).
The movement is moreover controlled by the user via a *shearSpeed* which is the horizontal speed of the upper wall, and by a maximum value of horizontal displacement *gammalim* (of the upper plate), after which the shear stops.
Note: not only the positions of walls are updated but also their speeds, which is all but useless considering the fact that in the contact laws these velocities of bodies are used to compute values of tangential relative displacements.

Warning: But, because of this last point, if you want to use later saves of simulations executed with this Engine, but without that stopMovement was executed, your boxes will keep their speeds => you will have to cancel them by hand in the .xml

F_0(=*0.0*)
the (vertical) force acting on the upper plate on the very first time step (determined by the Engine) [N]

Key(=*"*")
string to add at the names of the saved files

KnC(=*10.0e6*)
the normal rigidity choosen by the user [MPa/mm]

LOG(=*false*)
boolean controling the output of messages on the screen

Y0(=*0.0*)
the height of the upper plate at the very first time step : the engine finds its value

coeff_dech(=*1.0*)
in the case of the use of 'Law2_ScGeom_NormalInelasticityPhys_NormalInelasticity' for ex, where kn(unload)#kn(load). The engine cares to find the value at the first run

firstRun(=*true*)
boolean set to false as soon as the engine has done its job one time : usefull to know if the force acting on the plate is known or not (and if F_0 has to be initialized)

gamma(=*0.0*)
current value of tangential displacement [m]

gammalim(=*0.0*)
the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

id_boxback(=*4*)
the id of the wall at the back of the sample

id_boxbas(=*1*)
the id of the lower wall

id_boxfront(=*5*)
the id of the wall in front of the sample

id_boxleft(=*0*)
the id of the left wall

id_boxright(=*2*)
the id of the right wall

id_topbox(=*3*)
the id of the upper wall

max_vel(=*1.0*)
to limit the speed of the vertical displacements applied to control upper plate [m/s]

shearSpeed(=*0.0*)
the speed at wich the shearing is performed : speed of the upper plate [m/s]

wallDamping(=*0.2*)
the vertical displacements done to maintain F equal to F_0 are in fact damped, through this wallDamping

class KinemCTDEngine(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**
To compress a simple shear sample by moving the upper box in a vertical way only, so that the tangential displacement (defined by the horizontal gap between the upper and lower boxes) remains constant. The lateral boxes move also to keep always contact. All that until this box is submitted to a given stress (=*target_sigma*). Moreover saves are executed at each value of stresses stored in the vector *sigma_save*, and at *target_sigma*

Key(=*"*")
string to add at the names of the saved files

compSpeed(=*0.0*)
(vertical) speed of the upper box : >0 for real compression, <0 for unloading [m/s]

id_boxback(=*4*)
the id of the wall at the back of the sample

id_boxbas(=*1*)

the id of the lower wall

id_boxfront(=*5*)
the id of the wall in front of the sample

id_boxleft(=*0*)
the id of the left wall

id_boxright(=*2*)
the id of the right wall

id_topbox(=*3*)
the id of the upper wall

sigma_save(=*uninitalized*)
vector with the values of sigma at which a save of the simulation should be performed [kPa]

target_sigma(=*0.0*)
the value of sigma at which the compression should stop [kPa]

temoin_save(=*uninitalized*)
vector (same length as 'sigma_save'), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0)

class Law2_ScGeom_CapillaryPhys_Capillarity(*inherits GlobalEngine → Engine → Serializable*)

This law allows to take into account capillary forces/effects between spheres coming from the presence of interparticular liquid bridges (menisci).

refs:

1. in french [57] (lot of documentation)
2. in english [56] (less documentation), pg. 64-75.

The law needs ascii files M(r=i) with i=R1/R2 to work (see https://yade-dem.org/index.php/CapillaryTriaxialTest). These ASCII files contain a set of results from the resolution of the Laplace-Young equation for different configurations of the interacting geometry.

The control parameter is the capillary pressure (or suction) Uc = ugas - Uliquid. Liquid bridges properties (volume V, extent over interacting grains delta1 and delta2) are computed as a result of the defined capillary pressure and of the interacting geometry (spheres radii and interparticular distance).

CapillaryPressure(=*0.*)
Value of the capillary pressure Uc defines as Uc=Ugas-Uliquid

binaryFusion(=*true*)
If true, capillary forces are set to zero as soon as, at least, 1 overlap (menisci fusion) is detected

fusionDetection(=*false*)
If true potential menisci overlaps are checked

class Law2_ScGeom_NormalInelasticityPhys_NormalInelasticity(*inherits GlobalEngine → Engine → Serializable*)

Contact law including cohesion, moment transfer and inelastic compression behaviour

This contact Law is inspired by CohesiveFrictionalContactLaw (inspired itselve directly from the work of Plassiard & Belheine, see the corresponding articles in (Annual Report 2006) in http://geo.hmg.inpg.fr/frederic/Discrete_Element_Group_FVD.html for example).

It allows so to set moments, cohesion, tension limit and (that's the difference) inelastic unloadings in compression between bodies. All that concerned brokenBodies (this flag and the erosionactivated one) and the useless 'iter' has been suppressed.

The Relationsships corresponding are Ip2_2xCohFrictMat_NormalInelasticityPhys, where the rigidities, the friction angles (with their tan()), and the orientations of the interactions are calculated. No more cohesion and tension limits are computed for all the interactions.

To use it you should also use :

- CohFrictMat for bodies, with *isCohesive* = 1 (A verifier ce dernier point)
- Ip2_2xCohFrictMat_NormalInelasticityPhys (=> which involves interactions of NormalInelasticityPhys type).

The effect of this law on the normal force are illustrated in scripts/NormalInelasticityTest.py

coeff_dech(=*1.0*)

=kn(unload) / kn(load)

momentAlwaysElastic(=*false*)
boolean, true=> the torque (computed only if momentRotationLaw !!) is not limited by a plastic threshold

momentRotationLaw(=*true*)
boolean, true=> computation of a torque (against relative rotation) exchanged between particles

class NewtonIntegrator(*inherits GlobalEngine → Engine → Serializable***)**
Engine integrating newtonian motion equations.

callbacks(=*uninitalized*)
List (std::vector in c++) of BodyCallbacks which will be called for each body as it is being processed.

damping(=*0.2*)
damping coefficient for Cundall's non viscous damping (see [7]) [-]

exactAsphericalRot(=*true*)
Enable more exact body rotation integrator for aspherical bodies *only*, using formulation from [1], pg. 89.

homotheticCellResize(=*false*)
Enable artificially moving all bodies with the periodic cell, such that its resizes are homogeneous. The move is reflecting changes in Cell::velGrad, using NewtonIntegrator::prevVelGrad.

maxVelocitySq(=*NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

prevVelGrad(=*Matrix3r::Zero()*)
Store previous velocity gradient (Cell::velGrad) to track acceleration. *(auto-updated)*

class Peri3dController(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**
Experimental controller of full strain/stress tensors on periodic cell. Stress and strain tensors are computed using formulas derived in [31], in particular equations (33) and (35).

goal(=*Matrix3r::Zero()*)
Goal state.

maxStrainRate(=*1*)
Maximum absolute value of strain rate (both normal and shear components of Cell.velGrad)

strain(=*Matrix3r::Zero()*)
Current deformation tensor *(auto-updated)*

stress(=*Matrix3r::Zero()*)
Current stress tensor *(auto-updated)*

stressMask(=*0*, *all strains*)
mask determining whether components of goal are strain (0) or stress (1). The order is 00,11,22,12,02,01 from the least significant bit. (e.g. 0b000011 is stress 00 and stress 11).

class PeriIsoCompressor(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**
Compress/decompress cloud of spheres by controlling periodic cell size until it reaches prescribed average stress, then moving to next stress value in given stress series.

charLen(=*-1.*)
Characteristic length, should be something like mean particle diameter (default -1=invalid value))

currUnbalanced
Current value of unbalanced force

doneHook(=*"'*)
Python command to be run when reaching the last specified stress

globalUpdateInt(=*20*)
how often to recompute average stress, stiffness and unbalanced force

keepProportions(=*true*)
Exactly keep proportions of the cell (stress is controlled based on average, not its components

maxSpan(=*-1.*)
Maximum body span in terms of bbox, to prevent periodic cell getting too small. *(auto-computed)*

maxUnbalanced(=*1e-4*)
if actual unbalanced force is smaller than this number, the packing is considered stable,
sigma
Current stress value
state(=*0*)
Where are we at in the stress series
stresses(=*uninitalized*)
Stresses that should be reached, one after another

class **PeriTriaxController(***inherits BoundaryController → GlobalEngine → Engine → Serializable*)

Engine for independently controlling stress or strain in periodic simulations.
strainStress contains absolute values for the controlled quantity, and **stressMask** determines meaning of those values (0 for strain, 1 for stress): e.g. **(1<<0 | 1<<2) = 1 | 4 = 5** means that **strainStress[0]** and **strainStress[2]** are stress values, and **strainStress[1]** is strain.
See scripts/test/periodic-triax.py for a simple example.
absStressTol(=*1e3*)
Absolute stress tolerance
currUnbalanced(=*NaN*)
current unbalanced force (updated every globUpdate) *(auto-updated)*
doneHook(=*uninitalized*)
python command to be run when the desired state is reached
dynCell(=*false*)
Imposed stress can be controlled using the packing stiffness or by applying the laws of dynamic (dynCell=true). Don't forget to assign a mass to the cell (PeriTriaxController->mass).
globUpdate(=*5*)
How often to recompute average stress, stiffness and unbalaced force.
goal(=*Vector3r::Zero()*)
Desired stress or strain values (depending on stressMask), strains defined as **strain(i)=log(Fii)**
growDamping(=*.25*)
Damping of cell resizing (0=perfect control, 1=no control at all); see also **wallDamping** in TriaxialStressController.
mass(=*NaN*)
mass of the cell (user set)
maxBodySpan(=*Vector3r::Zero()*)
maximum body dimension *(auto-computed)*
maxStrainRate(=*Vector3r(1, 1, 1)*)
Maximum strain rate of the periodic cell.
maxUnbalanced(=*1e-4*)
maximum unbalanced force.
prevGrow(=*Vector3r::Zero()*)
previous cell grow
relStressTol(=*3e-5*)
Relative stress tolerance
reversedForces(=*false*)
For broken constitutive laws, normalForce and shearForce on interactions are in the reverse sense. see bugreport
stiff(=*Vector3r::Zero()*)
average stiffness (only every globUpdate steps recomputed from interactions) *(auto-updated)*
strain(=*Vector3r::Zero()*)
cell strain *(auto-updated)*
strainRate(=*Vector3r::Zero()*)
cell strain rate *(auto-updated)*
stress(=*Vector3r::Zero()*)
diagonal terms of the stress tensor
stressMask(=*0, all strains*)
mask determining strain/stress (0/1) meaning for goal components
stressTensor(=*Matrix3r::Zero()*)
average stresses, updated at every step (only every globUpdate steps recomputed from inter-

actions if !dynCell)

class PeriodicEngine(*inherits GlobalEngine → Engine → Serializable***)**

Run Engine::action with given fixed periodicity real time (=wall clock time, computation time), virtual time (simulation time), iteration number), by setting any of those criteria (virtPeriod, realPeriod, iterPeriod) to a positive value. They are all negative (inactive) by default.

The number of times this engine is activated can be limited by setting nDo>0. If the number of activations will have been already reached, no action will be called even if an active period has elapsed.

If initRun is set (false by default), the engine will run when called for the first time; otherwise it will only start counting period (realLast etc interal variables) from that point, but without actually running, and will run only once a period has elapsed since the initial run.

This class should be used directly; rather, derive your own engine which you want to be run periodically.

Derived engines should override Engine::action(), which will be called periodically. If the derived Engine overrides also Engine::isActivated, it should also take in account return value from PeriodicEngine::isActivated, since otherwise the periodicity will not be functional.

Example with PeriodicPythonRunner, which derives from PeriodicEngine; likely to be encountered in python scripts):

```
PeriodicPythonRunner(realPeriod=5,iterPeriod=10000,command='print 0.iter')
```

will print iteration number every 10000 iterations or every 5 seconds of wall clock time, whiever comes first since it was last run.

initRun(*=false***)**
: Run the first time we are called as well.

iterLast(*=0***)**
: Tracks step number of last run *(auto-updated)*.

iterPeriod(*=0, deactivated***)**
: Periodicity criterion using step number (deactivated if <= 0)

nDo(*=-1, deactivated***)**
: Limit number of executions by this number (deactivated if negative)

nDone(*=0***)**
: Track number of executions (cummulative) *(auto-updated)*.

realLast(*=0***)**
: Tracks real time of last run *(auto-updated)*.

realPeriod(*=0, deactivated***)**
: Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

virtLast(*=0***)**
: Tracks virtual time of last run *(auto-updated)*.

virtPeriod(*=0, deactivated***)**
: Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class PeriodicPythonRunner(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable***)**

Execute a python command periodically, with defined (and adjustable) periodicity. See PeriodicEngine documentation for details.

command(*=""***)**
: Command to be run by python interpreter. Not run if empty.

class Recorder(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable***)**

Engine periodically storing some data to (one) external file. In addition PeriodicEngine, it handles opening the file as needed. See PeriodicEngine for controlling periodicity.

file(*=uninitalized***)**
: Name of file to save to; must not be empty.

truncate(*=false***)**
: Whether to delete current file contents, if any, when opening (false by default)

class ResetRandomPosition(*inherits GlobalEngine → Engine → Serializable***)**

Creates spheres during simulation, placing them at random positions. Every time called, one new sphere will be created and inserted in the simulation.

angularVelocity(*=Vector3r::Zero()***)**
: Mean angularVelocity of spheres.

angularVelocityRange(=*Vector3r::Zero()*)
Half size of a angularVelocity distribution interval. New sphere will have random angularVelocity within the range angularVelocity±angularVelocityRange.
factoryFacets(=*uninitalized*)
The geometry of the section where spheres will be placed; they will be placed on facets or in volume between them depending on *volumeSection* flag.
maxAttempts(=*20*)
Max attemps to place sphere. If placing the sphere in certain random position would cause an overlap with any other physical body in the model, SpheresFactory will try to find another position.
normal(=*Vector3r(0, 1, 0)*)
??
point(=*Vector3r::Zero()*)
??
subscribedBodies(=*uninitalized*)
Affected bodies.
velocity(=*Vector3r::Zero()*)
Mean velocity of spheres.
velocityRange(=*Vector3r::Zero()*)
Half size of a velocities distribution interval. New sphere will have random velocity within the range velocity±velocityRange.
volumeSection(=*false*, *define factory by facets.*)
Create new spheres inside factory volume rather than on its surface.

class **SampleCapillaryPressureEngine(***inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)
Rk: this engine has to be tested withthe new formalism. It produces the isotropic compaction of an assembly and allows to controlled the capillary pressure inside (uses Law2_ScGeom_CapillaryPhys_Capillarity).
Pressure(=*0*)
Value of the capillary pressure Uc=Ugas-Uliquid (see Law2_ScGeom_CapillaryPhys_Capillarity). [Pa]
PressureVariation(=*0*)
Variation of the capillary pressure (each iteration). [Pa]
SigmaPrecision(=*0.001*)
tolerance in terms of mean stress to consider the packing as stable
StabilityCriterion(=*0.01*)
tolerance in terms of :yref:'TriaxialCompressionEngine::UnbalancedForce' to consider the packing as stable
UnbalancedForce(=*1*)
mean resultant forces divided by mean contact force
binaryFusion(=*1*)
If yes, capillary force are set to 0 when, at least, 1 overlap is detected for a meniscus. If no, capillary force is divided by the number of overlaps.
fusionDetection(=*1*)
Is the detection of menisci overlapping activated?
pressureVariationActivated(=*1*)
Is the capillary pressure varying?

class **SnapshotEngine(***inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)
Periodically save snapshots of GLView(s) as .png files. Files are named *fileBase*+*counter*+'.png' (counter is left-padded by 0s, i.e. snap0004.png)
counter(=*0*)
Number appended to fileBase *(auto-updated)*
fileBase(=*"“*)
Basename for snapshots
format(=*"PNG"*)
Format of snapshots (one of JPEG, PNG, EPS, PS, PPM, BMP) QGLViewer documentation. File extension will be lowercased *format*. Validity of format is not checked.
ignoreErrors(=*true*)

Silently return if selected view doesn't exist

msecSleep(=*0*)

number of msec to sleep after snapshot (to prevent 3d hw problems) [ms]

savedSnapshots(=*uninitalized*)

Files that have been created so far

viewNo(=*0*, *primary view*)

The GLView number that we save.

class SpatialQuickSortCollider(*inherits Collider → GlobalEngine → Engine → Serializable*)

Collider using quicksort along axes at each step, using Aabb bounds.

Its performance is lower than that of InsertionSortCollider (see Colliders' performance), but the algorithm is simple enought to make it good for checking other collider's correctness.

class TetraVolumetricLaw(*inherits GlobalEngine → Engine → Serializable*)

Calculate physical response of 2 tetrahedra in interaction, based on penetration configuration given by TTetraGeom.

class ThreeDTriaxialEngine(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)

The engine perform a triaxial compression with a control in direction 'i' in stress (if stressControl_i) else in strain.

For a stress control the imposed stress is specified by 'sigma_i' with a 'max_veli' depending on 'strainRatei'. To obtain the same strain rate in stress control than in strain control you need to set 'wallDamping = 0.8'. For a strain control the imposed strain is specified by 'strainRatei'. With this engine you can also perform internal compaction by growing the size of particles by using TriaxialStressController::controlInternalStress. For that, just switch on 'internalCompaction=1' and fix sigma_iso=value of mean pressure that you want at the end of the internal compaction.

Key(=*'"'*)

A string appended at the end of all files, use it to name simulations.

UnbalancedForce(=*1*)

mean resultant forces divided by mean contact force

currentStrainRate1(=*0*)

current strain rate in direction 1 - converging to :yref:'ThreeDTriaxialEngine::strainRate1' (./s)

currentStrainRate2(=*0*)

current strain rate in direction 2 - converging to :yref:'ThreeDTriaxialEngine::strainRate2' (./s)

currentStrainRate3(=*0*)

current strain rate in direction 3 - converging to :yref:'ThreeDTriaxialEngine::strainRate3' (./s)

frictionAngleDegree(=*-1*)

Value of friction used in the simulation if (updateFrictionAngle)

setContactProperties(*(float)arg2*) → None

Assign a new friction angle (degrees) to dynamic bodies and relative interactions

strainRate1(=*0*)

target strain rate in direction 1 (./s)

strainRate2(=*0*)

target strain rate in direction 2 (./s)

strainRate3(=*0*)

target strain rate in direction 3 (./s)

stressControl_1(=*true*)

Switch to choose a stress or a strain control in directions 1

stressControl_2(=*true*)

Switch to choose a stress or a strain control in directions 2

stressControl_3(=*true*)

Switch to choose a stress or a strain control in directions 3

updateFrictionAngle(=*false*)

Switch to activate the update of the intergranular frictionto the value :yref:'ThreeDTriaxialEngine::frictionAngleDegree

class TimeStepper(*inherits GlobalEngine → Engine → Serializable*)

Engine defining time-step (fundamental class)

active(=*true*)
is the engine active?
timeStepUpdateInterval(=*1*)
dt update interval

class TriaxialCompressionEngine(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable***)**

The engine is a state machine with the following states; transitions my be automatic, see below.

1. STATE_ISO_COMPACTION: isotropic compaction (compression) until the prescribed mean pressue sigmaIsoCompaction is reached and the packing is stable. The compaction happens either by straining the walls (!internalCompaction) or by growing size of grains (internalCompaction).
2. STATE_ISO_UNLOADING: isotropic unloading from the previously reached state, until the mean pressure sigmaLateralConfinement is reached (and stabilizes).
   Note: this state will be skipped if sigmaLateralConfinement == sigmaIsoCompaction.
3. STATE_TRIAX_LOADING: confined uniaxial compression: constant sigmaLateralConfinement is kept at lateral walls (left, right, front, back), while top and bottom walls load the packing in their axis (by straining), until the value of epsilonMax (deformation along the loading axis) is reached. At this point, the simulation is stopped.
4. STATE_FIXED_POROSITY_COMPACTION: isotropic compaction (compression) until a chosen porosity value (parameter:fixedPorosity). The six walls move with a chosen translation speed (parameter StrainRate).
5. STATE_TRIAX_LIMBO: currently unused, since simulation is hard-stopped in the previous state.

Transition from COMPACTION to UNLOADING is done automatically if autoUnload==true;
Transition from (UNLOADING to LOADING) or from (COMPACTION to LOADING: if UNLOADING is skipped) is done automatically if autoCompressionActivation=true;
Both autoUnload and autoCompressionActivation are true by default.

Note: This engine handles many different manipulations, including some save/reload with attributes modified manually in between. Please don't modify the algorithms, even if they look strange (especially test sequences) without notifying me and getting explicit approval. A typical situation is somebody generates a sample with !autoCompressionActivation and run : he wants a saved simulation at the end. He then reload the saved state, modify some parameters, set autoCompressionActivation=true, and run. He should get the compression test done.

Key(=*"*)
A string appended at the end of all files, use it to name simulations.
StabilityCriterion(=*0.001*)
tolerance in terms of TriaxialCompressionEngine::UnbalancedForce to consider the packing is stable
UnbalancedForce(=*1*)
mean resultant forces divided by mean contact force
autoCompressionActivation(=*true*)
Auto-switch from isotropic compaction (or unloading state if sigmaLateralConfinement<sigmaIsoCompaction) to deviatoric loading
autoStopSimulation(=*true*)
Stop the simulation when the sample reach STATE_LIMBO, or keep running
autoUnload(=*true*)
Auto-switch from isotropic compaction to unloading
currentState(=*1*)
currentStrainRate(=*0*)
current strain rate - converging to TriaxialCompressionEngine::strainRate (./s)
epsilonMax(=*0.5*)
Value of axial deformation for which the loading must stop
fixedPoroCompaction(=*false*)
A special type of compaction with imposed final porosity TriaxialCompressionEngine::fixedPorosity (WARNING : can give unrealistic results!)
fixedPorosity(=*0*)
Value of porosity chosen by the user

frictionAngleDegree(=*-1*)
Value of friction assigned just before the deviatoric loading
maxStress(=*0*)
Max value of stress during the simulation (for post-processing)
noFiles(=*false*)
If true, no files will be generated (*.xml, *.spheres,...)
previousSigmaIso(=*1*)
Previous value of inherited sigma_iso (used to detect manual changes of the confining pressure)
previousState(=*1*)
Previous state (used to detect manual changes of the state in .xml)
setContactProperties(*(float)arg2*) → None
Assign a new friction angle (degrees) to dynamic bodies and relative interactions
sigmaIsoCompaction(=*1*)
Prescribed isotropic pressure during the compaction phase
sigmaLateralConfinement(=*1*)
Prescribed confining pressure in the deviatoric loading; might be different from TriaxialCompressionEngine::sigmaIsoCompaction
spheresVolume(=*1*)
strainRate(=*0*)
target strain rate (./s)
testEquilibriumInterval(=*20*)
interval of checks for transition between phases, higher than 1 saves computation time.
translationAxis(=*TriaxialStressController::normal*[, *wall_bottom_id*]**)**
compression axis
uniaxialEpsilonCurr(=*1*)
Current value of axial deformation during confined loading (is reference to strain[1])

class **TriaxialStateRecorder(***inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable***)**
Engine recording triaxial variables (see the variables list in the first line of the output file). This recorder needs TriaxialCompressionEngine or ThreeDTriaxialEngine present in the simulation).
porosity(=*1*)
porosity of the packing [-]

class **TriaxialStressController(***inherits BoundaryController → GlobalEngine → Engine → Serializable***)**
An engine maintaining constant stresses on some boundaries of a parallepipedic packing.
boxVolume
Total packing volume.
computeStressStrainInterval(=*10*)
depth(=*0*)
depth0(=*0*)
finalMaxMultiplier(=*1.00001*)
max multiplier of diameters during internal compaction (secondary precise adjustment - TriaxialStressController::maxMultiplier is used in the initial stage)
height(=*0*)
height0(=*0*)
internalCompaction(=*true*)
Switch between 'external' (walls) and 'internal' (growth of particles) compaction.
isAxisymetric(=*true*)
if true, sigma_iso is assigned to sigma1, 2 and 3
maxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (initial fast increase - TriaxialStressController::finalMaxMultiplier is used in a second stage)
max_vel(=*0.001*)
Maximum allowed walls velocity [m/s]. This value superseeds the one assigned by the stress controller if the later is higher. max_vel can be set to infinity in many cases, but sometimes helps stabilizing packings. Based on this value, different maxima are computed for each axis based on the dimensions of the sample, so that if each boundary moves at its maximum velocity, the strain rate will be isotropic (see e.g. TriaxialStressController::max_vel1).

max_vel1
see TriaxialStressController::max_vel *(auto-computed)*
max_vel2
see TriaxialStressController::max_vel *(auto-computed)*
max_vel3
see TriaxialStressController::max_vel *(auto-computed)*
meanStress(=*0*)
Mean stress in the packing.
porosity
Pososity of the packing.
previousMultiplier(=*1*)
previousStress(=*0*)
radiusControlInterval(=*10*)
sigma1(=*0*)
applied stress on axis 1 (see TriaxialStressController::isAxisymetric)
sigma2(=*0*)
applied stress on axis 2 (see TriaxialStressController::isAxisymetric)
sigma3(=*0*)
applied stress on axis 3 (see TriaxialStressController::isAxisymetric)
sigma_iso(=*0*)
applied confining stress (see TriaxialStressController::isAxisymetric)
stiffnessUpdateInterval(=*10*)
target strain rate (./s)
strain
Current strain (logarithmic).
thickness(=*-1*)
volumetricStrain(=*0*)
Volumetric strain (see TriaxialStressController::strain).
wallDamping(=*0.25*)
wallDamping coefficient - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement
wall_back_activated(=*true*)
wall_back_id(=*0*)
id of boundary ; coordinate 2-
wall_bottom_activated(=*true*)
wall_bottom_id(=*0*)
id of boundary ; coordinate 1-
wall_front_activated(=*true*)
wall_front_id(=*0*)
id of boundary ; coordinate 2+
wall_left_activated(=*true*)
wall_left_id(=*0*)
id of boundary ; coordinate 0-
wall_right_activated(=*true*)
wall_right_id(=*0*)
id of boundary ; coordinate 0+
wall_top_activated(=*true*)
wall_top_id(=*0*)
id of boundary ; coordinate 1+
width(=*0*)
width0(=*0*)

class UniaxialStrainer(*inherits BoundaryController → GlobalEngine → Engine → Serializable***)**

Axial displacing two groups of bodies in the opposite direction with given strain rate.

absSpeed(=*NaN*)
alternatively, absolute speed of boundary motion can be specified; this is effective only at the beginning and if strainRate is not set; changing absSpeed directly during simulation wil have no effect. [ms☒¹]

active(=*true*)
Whether this engine is activated
asymmetry(=*0*, *symmetric*)
If 0, straining is symmetric for negIds and posIds; for 1 (or -1), only posIds are strained and negIds don't move (or vice versa)
avgStress(=*0*)
Current average stress *(auto-updated)* [Pa]
axis(=*2*)
The axis which is strained (0,1,2 for x,y,z)
blockDisplacements(=*false*)
Whether displacement of boundary bodies perpendicular to the strained axis are blocked of are free
blockRotations(=*false*)
Whether rotations of boundary bodies are blocked.
crossSectionArea(=*NaN*)
crossSection perpendicular to he strained axis, computed from Aabb of Scene, or given explicitly [m²]
currentStrainRate(=*NaN*)
Current strain rate (update automatically). *(auto-updated)*
idleIterations(=*0*)
Number of iterations that will pass without straining activity after stopStrain has been reached
initAccelTime(=*-200*)
Time for strain reaching the requested value (linear interpolation). If negative, the time is dt*(-initAccelTime), where dt is the timestep at the first iteration. [s]
limitStrain(=*0*, *disabled*)
Invert the sense of straining (sharply, without transition) one this value of strain is reached. Not effective if 0.
negIds(=*uninitalized*)
Bodies on which strain will be applied (on the negative end along the axis)
notYetReversed(=*true*)
Flag whether the sense of straining has already been reversed (only used internally).
originalLength(=*NaN*)
Distance of reference bodies in the direction of axis before straining started (computed automatically) [m]
posIds(=*uninitalized*)
Bodies on which strain will be applied (on the positive end along the axis)
setSpeeds(=*false*)
should we set speeds at the beginning directly, instead of increasing strain rate progressively?
stopStrain(=*NaN*)
Strain at which we will pause simulation; inactive (nan) by default; must be reached from below (in absolute value)
strain(=*0*)
Current strain value, elongation/originalLength *(auto-updated)* [-]
strainRate(=*NaN*)
Rate of strain, starting at 0, linearly raising to strainRate. [-]

class VTKRecorder(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable***)**
Engine recording snapshots of simulation into series of *.vtu files, readable by VTK-based post-processing programs such as Paraview. Both bodies (spheres and facets) and interactions can be recorded, with various vector/scalar quantities that are defined on them.
PeriodicEngine.initRun is initialized to **True** automatically.
compress(=*false*)
Compress output XML files [experimental].
fileName(=*""*)
Base file name; it will be appended with {spheres,intrs,facets}-243100.vtu depending on active recorders and step number (243100 in this case). It can contain slashes, but the directory must exist already.
mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be exported. If 0 - all bodies

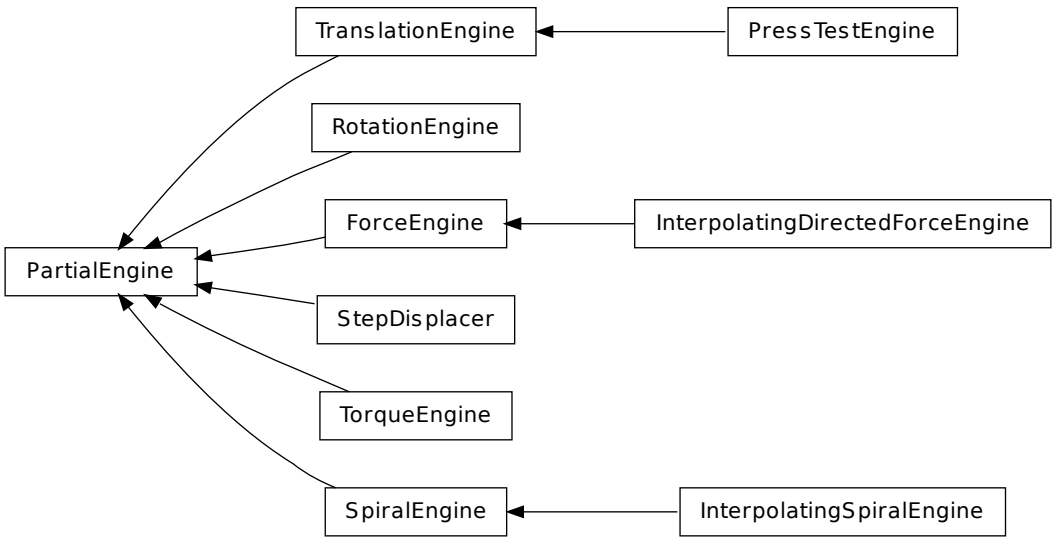
will be exported.

recorders
List of active recorders (as strings). Accepted recorders are:
all Saves all possible parameters, except of specific. Default value.
spheres Saves positions and radii (**radii**) of spherical particles.
id Saves id's (field **id**) of spheres; active only if **spheres** is active.
clumpId Saves id's of clumps to which each sphere belongs (field **clumpId**); active only if **spheres** is active.
colors Saves colors of spheres and of facets (field **color**); only active if **spheres** or **facets** are activated.
mask Saves groupMasks of spheres and of facets (field **mask**); only active if **spheres** or **facets** are activated.
materialId Saves materialID of spheres and of facets; only active if **spheres** or **facets** are activated.
velocity Saves linear and angular velocities of spherical particles as Vector3 and length(fields **linVelVec**, **linVelLen** and **angVelVec**, **angVelLen** respectively''); only effective with **spheres**.
facets Save facets positions (vertices).
stress Saves stresses of spheres and of facets as Vector3 and length; only active if **spheres** or **facets** are activated.
cpm Saves data pertaining to the concrete model: **cpmDamage** (normalized residual strength averaged on particle), **cpmSigma** (stress on particle, normal components), **cpmTau** (shear components of stress on particle), **cpmSigmaM** (mean stress around particle); **intr** is activated automatically by **cpm**.
intr When **cpm** is used, it saves magnitude of normal (**forceN**) and shear (**absForceT**) forces. Without **cpm**, saves [TODO]

skipFacetIntr(=*true*)
Skip interactions with facets, when saving interactions

skipNondynamic(=*false*)
Skip non-dynamic spheres (but not facets).

C.4. Partial engines



class PartialEngine(*inherits Engine → Serializable***)**
Engine affecting only particular bodies in the simulation, defined by *subscribedBodies*.

subscribedBodies(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

class ForceEngine(*inherits PartialEngine → Engine → Serializable***)**

Apply contact force on some particles at each step.

force(=*Vector3r::Zero()*)
: Force to apply.

class InterpolatingDirectedForceEngine(*inherits ForceEngine → PartialEngine → Engine → Serializable*)

Engine for applying force of varying magnitude but constant direction on subscribed bodies. times and magnitudes must have the same length, direction (normalized automatically) gives the orientation.

As usual with interpolating engines: the first magnitude is used before the first time point, last magnitude is used after the last time point. Wrap specifies whether time wraps around the last time point to the first time point.

direction(=*Vector3r::UnitX()*)
: Contact force direction (normalized automatically)

magnitudes(=*uninitalized*)
: Force magnitudes readings [N]

times(=*uninitalized*)
: Time readings [s]

wrap(=*false*)
: wrap to the beginning of the sequence if beyond the last time point

class InterpolatingSpiralEngine(*inherits SpiralEngine → PartialEngine → Engine → Serializable*)

Engine applying spiral motion, finding current angular velocity by linearly interpolating in times and velocities and translation by using slope parameter.

The interpolation assumes the margin value before the first time point and last value after the last time point. If wrap is specified, time will wrap around the last times value to the first one (note that no interpolation between last and first values is done).

angularVelocities(=*uninitalized*)
: List of angular velocities; manadatorily of same length as times. [rad/s]

slope(=*0*)
: Axial translation per radian turn (can be negative) [m/rad]

times(=*uninitalized*)
: List of time points at which velocities are given; must be increasing [s]

wrap(=*false*)
: Wrap t if t>times_n, i.e. t_wrapped=t-N*(times_n-times_0)

class PressTestEngine(*inherits TranslationEngine → PartialEngine → Engine → Serializable*)

This class simulates the simple press work When the press cracks the solid brittle material, it returns back to the initial position and stops the simulation loop.

numberIterationAfterDestruction(=*0*)
: The number of iterations, which will be carry out after destruction [-]

predictedForce(=*0*)
: The minimal force, after what the engine will look for a destruction [N]

riseUpPressHigher(=*1*)
: After destruction press rises up. This is the relationship between initial press velocity and velocity for going *back* [-]

class RotationEngine(*inherits PartialEngine → Engine → Serializable*)

Engine applying rotation (by setting angular velocity) to subscribed bodies. If rotateAroundZero is set, then each body is also displaced around zeroPoint.

angularVelocity(=*0*)
: Angular velocity. [rad/s]

rotateAroundZero(=*false*)
: If True, bodies will not rotate around their centroids, but rather around **zeroPoint**.

rotationAxis(=*Vector3r::UnitX()*)
: Axis of rotation (direction); will be normalized automatically.

zeroPoint(=*Vector3r::Zero()*)
: Point around which bodies will rotate if **rotateAroundZero** is True

class SpiralEngine(*inherits PartialEngine → Engine → Serializable*)

Engine applying both rotation and translation, along the same axis, whence the name SpiralEngine

angleTurned(*=0* **)**
How much have we turned so far. *(auto-updated)* [rad]
angularVelocity(*=0* **)**
Angular velocity [rad/s]
axis(*=Vector3r::UnitX()* **)**
Axis of translation and rotation; will be normalized by the engine.
axisPt(*=Vector3r::Zero()* **)**
A point on the axis, to position it in space properly.
linearVelocity(*=0* **)**
Linear velocity [m/s]

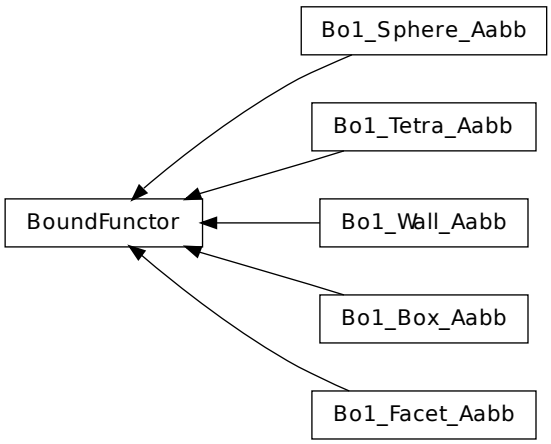
class StepDisplacer(*inherits PartialEngine → Engine → Serializable* **)**
Apply generalized displacement (displacement or rotation) stewise on subscribed bodies.
deltaSe3(*=Se3r(Vector3r::Zero(), Quaternionr::Identity())* **)**
Difference of position/orientation that will be added. Position is added to current State::pos using vector addition, orientation to State::ori using quaternion multiplication (rotation composition).
setVelocities(*=false* **)**
If true, velocity and angularVelocity are modified in such a way that over one iteration (dt), the body will have prescribed se3 jump. In this case, se3 itself is not updated for dynamic bodies, since they would have the delta applied twice (here and in the integrator). For non-dynamic bodies however, se3 *is* still updated.

class TorqueEngine(*inherits PartialEngine → Engine → Serializable* **)**
Apply given torque (momentum) value at every subscribed particle, at every step.
moment(*=Vector3r::Zero()* **)**
Torque value to be applied.

class TranslationEngine(*inherits PartialEngine → Engine → Serializable* **)**
This engine is the base class for different engines, which require any kind of motion.
translationAxis(*=uninitalized* **)**
Direction [Vector3]
velocity(*=uninitalized* **)**
Velocity [m/s]

C.5. Bounding volume creation

C.5.1. BoundFunctor



class BoundFunctor(*inherits Functor → Serializable* **)**
Functor for creating/updating Body::bound.

class Bo1_Box_Aabb(*inherits BoundFunctor → Functor → Serializable* **)**
Create/update an Aabb of a Box.

class Bo1_Facet_Aabb(*inherits BoundFunctor → Functor → Serializable* **)**

Creates/updates an Aabb of a facet.

class Bo1_Sphere_Aabb(*inherits BoundFunctor → Functor → Serializable***)**

Functor creating Aabb from Sphere.

aabbEnlargeFactor

Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an InteractionGeometryFunctor which will not simply discard such interactions: Ig2_Sphere_Sphere_Dem3DofGeom::distFactor / Ig2_Sphere_Sphere_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class Bo1_Tetra_Aabb(*inherits BoundFunctor → Functor → Serializable***)**

Create/update Aabb of a Tetra

class Bo1_Wall_Aabb(*inherits BoundFunctor → Functor → Serializable***)**

Creates/updates an Aabb of a Wall

C.5.2. BoundDispatcher

class BoundDispatcher(*inherits Dispatcher → Engine → Serializable***)**

Dispatcher for creating/updating Body::bound objects.

__init__() → None

object __init__(tuple args, dict kwds)

__init__((object)arg2) → object : Construct with list of associated functors.

activated(*=true***)**

Whether the engine is activated (only should be changed by the collider)

dispFunctor(*(Shape)arg2***, *(Bound)arg3***) →** BoundFunctor

Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]**) →** dict

Return dictionary with contents of the dispatch matrix.

functors

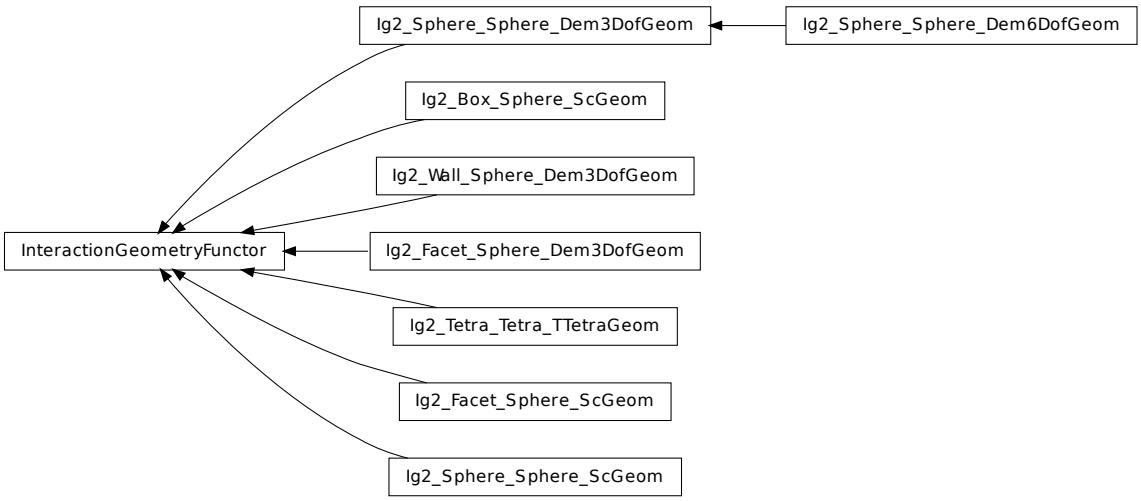
Functors objects associated with this dispatcher.

sweepDist(*=0***)**

Distance by which enlarge all bounding boxes, to prevent collider from being run at every step (only should be changed by the collider).

C.6. Interaction Geometry creation

C.6.1. InteractionGeometryFunctor



class InteractionGeometryFunctor(*inherits Functor → Serializable***)**

Functor for creating/updating Interaction::interactionGeometry objects.

class Ig2_Box_Sphere_ScGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Create an interaction geometry ScGeom from Box and Sphere

class Ig2_Facet_Sphere_Dem3DofGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Compute geometry of facet-sphere contact with normal and shear DOFs. As in all other Dem3DofGeom-related classes, total formulation of both shear and normal deformations is used. See Dem3DofGeom_FacetSphere for more information.

class Ig2_Facet_Sphere_ScGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Create/update a ScGeom instance representing intersection of Facet and Sphere.

shrinkFactor(*=0, no shrinking***)**
The radius of the inscribed circle of the facet is decreased by the value of the sphere's radius multipled by *shrinkFactor*. From the definition of contact point on the surface made of facets, the given surface is not continuous and becomes in effect surface covered with triangular tiles, with gap between the separate tiles equal to the sphere's radius multiplied by 2×*shrinkFactor*. If zero, no shrinking is done.

class Ig2_Sphere_Sphere_Dem3DofGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Functor handling contact of 2 spheres, producing Dem3DofGeom instance

distFactor(*=-1***)**
Factor of sphere radius such that sphere "touch" if their centers are not further than distFactor*(r1+r2); if negative, equilibrium distance is the sum of the sphere's radii.

class Ig2_Sphere_Sphere_Dem6DofGeom(*inherits Ig2_Sphere_Sphere_Dem3DofGeom → InteractionGeometryFunctor → Functor → Serializable***)**
Create/update contact of 2 spheres with 6 DOFs (Dem6DofGeom_SphereSphere instance) [experimental]

class Ig2_Sphere_Sphere_ScGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Create/update a ScGeom instance representing intersection of two Spheres.

interactionDetectionFactor
Enlarge both radii by this factor (if >1), to permit creation of distant interactions. InteractionGeometry will be computed when interactionDetectionFactor*(rad1+rad2) > distance.

Note: This parameter is functionally coupled with Bo1_Sphere_Aabb::aabbEnlargeFactor, which will create larger bounding boxes and should be of the same value.

> **Warning:** Functionally equal class Ig2_Sphere_Sphere_Dem3DofGeom (which creates Dem3DofGeom rather than ScGeom) calls this parameter distFactor, but its semantics is *different* in some aspects.

class Ig2_Tetra_Tetra_TTetraGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Create/update geometry of collision between 2 tetrahedra (TTetraGeom instance)

class Ig2_Wall_Sphere_Dem3DofGeom(*inherits InteractionGeometryFunctor → Functor → Serializable***)**
Create/update contact of Wall and Sphere (Dem3DofGeom_WallSphere instance)

C.6.2. InteractionGeometryDispatcher

class InteractionGeometryDispatcher(*inherits Dispatcher → Engine → Serializable***)**
Dispatcher for creating/updating Interaction::interactionGeometry objects

__init__() → None
object __init__(tuple args, dict kwds)

__init__((object)arg2) → **object :** Construct with list of associated functors.

dispFunctor(*(Shape)arg2, (Shape)arg3***)** → InteractionGeometryFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]**)** → dict

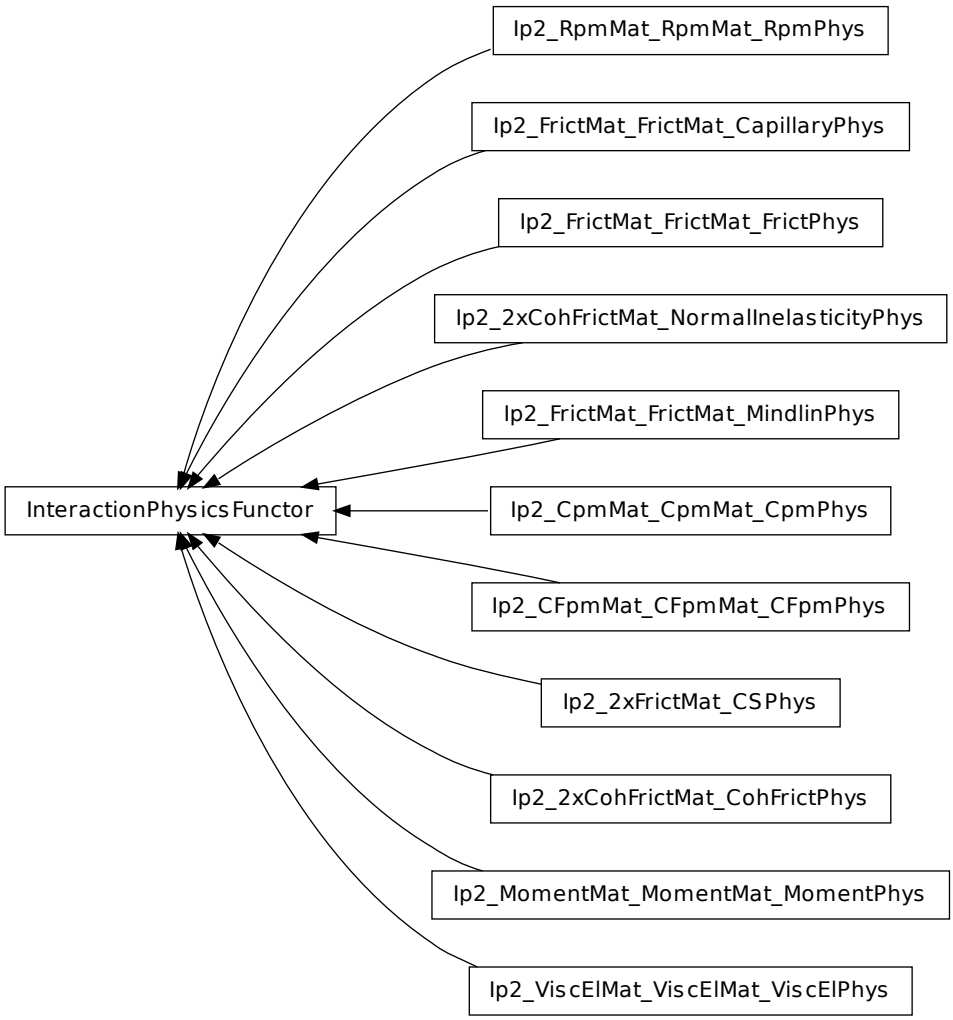
Return dictionary with contents of the dispatch matrix.

functors

Functors objects associated with this dispatcher.

C.7. Interaction Physics creation

C.7.1. InteractionPhysicsFunctor



class InteractionPhysicsFunctor(*inherits Functor → Serializable***)**

Functor for creating/updating Interaction::interactionPhysics objects.

class Ip2_2xCohFrictMat_CohFrictPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable***)**

Generates cohesive-frictional interactions with moments. Used in the contact law Law2_ScGeom_CohFrictPhys_ElasticPlastic.

normalCohesion(*=10000000***)**

setCohesionNow(*=false***)**

setCohesionOnNewContacts(*=false***)**

shearCohesion(*=10000000***)**

class Ip2_2xCohFrictMat_NormalInelasticityPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable***)**

The RelationShips for using Law2_ScGeom_NormalInelasticityPhys_NormalInelasticity

In these RelationShips all the attributes of the interactions (which are of NormalInelasticityPhys type) are computed.

Warning: as in the others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

betaR(=*0.12*)
Parameter for computing the torque-stifness : T-stifness = betaR * Rmoy^2

setCohesionNow(=*false*)

setCohesionOnNewContacts(=*false*)

class Ip2_2xFrictMat_CSPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable*)
Functor creating CSPhys from two FrictMat. See Law2_Dem3Dof_CSPhys_CundallStrack for details.

class Ip2_CFpmMat_CFpmMat_CFpmPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable*)
Converts 2 CFpmmat instances to CFpmPhys with corresponding parameters.

Alpha(=*0*)
Defines the ratio ks/kn.

Beta(=*0*)
Defines the ratio kr/(ks*meanRadius^2) to compute the resistive moment in rotation. [-]

cohesion(=*0*)
Defines the maximum admissible tangential force in shear FsMax=cohesion*crossSection. [Pa]

cohesiveTresholdIteration(=*1*)
Should new contacts be cohesive? They will before this iter, they won't afterward.

eta(=*0*)
Defines the maximum admissible resistive moment in rotation MtMax=eta*meanRadius*Fn. [-]

strengthSoftening(=*0*)
Defines the softening when Dtensile is reached to avoid explosion of the contact. Typically, when D > Dtensile, Fn=FnMax - (kn/strengthSoftening)*(Dtensile-D). [-]

tensileStrength(=*0*)
Defines the maximum admissible normal force in traction FnMax=tensileStrength*crossSection. [Pa]

useAlphaBeta(=*false*)
If true, stiffnesses are computed based on Alpha and Beta.

class Ip2_CpmMat_CpmMat_CpmPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable*)
Convert 2 CpmMat instances to CpmPhys with corresponding parameters. Uses simple (arithmetic) averages if material are different. Simple copy of parameters is performed if the material is shared between both particles. See *cpm-model* for detals.

cohesiveThresholdIter(=*10*)
Should new contacts be cohesive? They will before this iter#, they will not be afterwards. If 0, they will never be. If negative, they will always be created as cohesive (10 by default).

class Ip2_FrictMat_FrictMat_CapillaryPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable*)
RelationShips to use with Law2_ScGeom_CapillaryPhys_Capillarity
In these RelationShips all the interaction attributes are computed.

Warning: as in the others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

class Ip2_FrictMat_FrictMat_FrictPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable*)
Create a FrictPhys from two FrictMats. The compliance of one sphere under symetric point loads is defined here as 1/(E.r), with E the stiffness of the sphere and r its radius, and corresponds to a compliance 1/(2.E.r)=1/(E.D) from each contact point. The compliance of the contact itself will be the sum of compliances from each sphere, i.e. 1/(E.D1)+1/(E.D2) in the general case, or 1/(E.r) in the special case of equal sizes. Note that summing compliances corresponds to an harmonic average of stiffnesss, which is how kn is actually computed in the Ip2_FrictMat_FrictMat_FrictPhys functor.
The shear stiffness ks of one sphere is defined via the material parameter FrictPhys::poisson, as ks=poisson*kn, and the resulting shear stiffness of the interaction will

be also an harmonic average.

The friction angle of the contact is defined as the minimum angle of the two materials in contact.

Only interactions with ScGeom or Dem3DofGeom geometry are meaningfully accepted; run-time typecheck can make this functor unnecessarily slow in general. Such design is problematic in itself, though – from http://www.mail-archive.com/yade-dev@lists.launchpad.net/msg02603.html:

> You have to suppose some exact type of InteractionGeometry in the Ip2 functor, but you don't know anything about it (Ip2 only guarantees you get certain InteractionPhysics types, via the dispatch mechanism).
>
> That means, unless you use Ig2 functor producing the desired type, the code will break (crash or whatever). The right behavior would be either to accept any type (what we have now, at least in principle), or really enforce InteractionGeometry type of the interation passed to that particular Ip2 functor.

Etc.

class Ip2_FrictMat_FrictMat_MindlinPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable***)**

Calculate some physical parameters needed to obtain the normal and shear stiffnesses according to the Hertz-Mindlin's formulation (as implemented in PFC).

class Ip2_MomentMat_MomentMat_MomentPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable***)**

Create MomentPhys from 2 instances of MomentMat.

1. If boolean userInputStiffness=true & useAlphaBeta=false, users can input Knormal, Kshear and Krotate directly. Then, kn,ks and kr will be equal to these values, rather than calculated E and v.
2. If boolean userInputStiffness=true & useAlphaBeta=true, users input Knormal, Alpha and Beta. Then ks and kr are calculated from alpha & beta respectively.
3. If both are false, it calculates kn and ks are calculated from E and v, whilst kr = 0.

Alpha(*=0***)**

Ratio of Ks/Kn

Beta(*=0***)**

Ratio to calculate Kr

Knormal(*=0***)**

Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

Krotate(*=0***)**

Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

Kshear(*=0***)**

Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

useAlphaBeta(*=false***)**

for users to choose whether to input stiffness directly or use ratios to calculate Ks/Kn

userInputStiffness(*=false***)**

for users to choose whether to input stiffness directly or use ratios to calculate Ks/Kn

class Ip2_RpmMat_RpmMat_RpmPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable***)**

Convert 2 RpmMat instances to RpmPhys with corresponding parameters.

initDistance(*=0***)**

Initial distance between spheres at the first step.

class Ip2_ViscElMat_ViscElMat_ViscElPhys(*inherits InteractionPhysicsFunctor → Functor → Serializable***)**

Convert 2 instances of ViscElMat to ViscElPhys using the rule of consecutive connection.

C.7.2. InteractionPhysicsDispatcher

class InteractionPhysicsDispatcher(*inherits Dispatcher → Engine → Serializable***)**

Dispatcher for creating/updating Interaction::interactionPhysics objects.

___init___() → None

object __init__(tuple args, dict kwds)

__init__((object)arg2) → **object :** Construct with list of associated functors.

dispFunctor(*(Material)arg2*, *(Material)arg3***)** → InteractionPhysicsFunctor

Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]**)** → dict

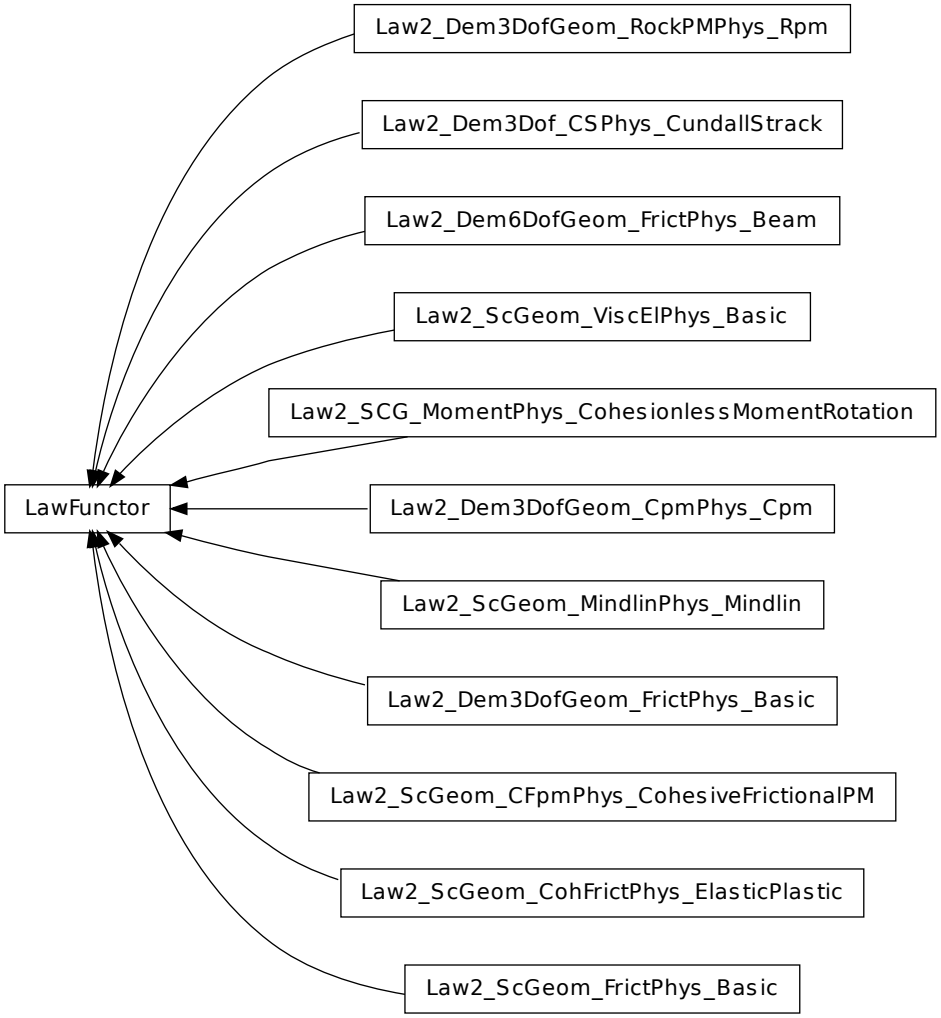
Return dictionary with contents of the dispatch matrix.

functors

Functors objects associated with this dispatcher.

C.8. Constitutive laws

C.8.1. LawFunctor



class LawFunctor(*inherits Functor → Serializable***)**

Functor for applying constitutive laws on interactions.

class Law2_Dem3DofGeom_CpmPhys_Cpm(*inherits LawFunctor → Functor → Serializable***)**

Constitutive law for the *cpm-model*.

epsSoft(*=-3e-3, approximates confinement -20MPa precisely, -100MPa a little over, -200 and -400 are OK (secant)***)**

Strain at which softening in compression starts (non-negative to deactivate)

funcG(*(float)kappaD*, *(float)epsCrackOnset*, *(float)epsFracture*[, *(bool)neverDamage=False*]**)** → float

Damage evolution law, evaluating the ω parameter. κ_D is historically maximum strain, *ep-*

sCrackOnset (ε_0) = CpmMat.epsCrackOnset, *epsFracture* = CpmMat.epsFracture; if *neverDamage* is **True**, the value returned will always be 0 (no damage).

omegaThreshold(=*1.*, >=*1. to deactivate, i.e. never delete any contacts*)
damage after which the contact disappears (<1), since omega reaches 1 only for strain →+∞

relKnSoft(=*.3*)
Relative rigidity of the softening branch in compression (0=perfect elastic-plastic, <0 softening, >0 hardening)

yieldEllipseShift(=*NaN*)
horizontal scaling of the ellipse (shifts on the +x axis as interactions with +y are given)

yieldLogSpeed(=*.1*)
scaling in the logarithmic yield surface (should be <1 for realistic results; >=0 for meaningful results)

yieldSigmaTMagnitude(*(float)sigmaN*, *(float)omega*, *(float)undamagedCohesion*, *(float)tanFrictionAngle*) → float
Return radius of yield surface for given material and state parameters; uses attributes of the current instance (*yieldSurfType* etc), change them before calling if you need that.

yieldSurfType(=*2*)
yield function: 0: mohr-coulomb (original); 1: parabolic; 2: logarithmic, 3: log+lin_tension, 4: elliptic, 5: elliptic+log

class Law2_Dem3DofGeom_FrictPhys_Basic(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for linear compression, no tension, and linear plasticity surface.
This class serves also as tutorial and is documented in detail at https://yade-dem.org/index.php/ConstitutiveLawHowto.

class Law2_Dem3DofGeom_RockPMPhys_Rpm(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the Rpm model

class Law2_Dem3Dof_CSPhys_CundallStrack(*inherits LawFunctor → Functor → Serializable*)
Basic constitutive law published originally by Cundall&Strack; it has normal and shear stiffnesses (Kn, Kn) and dry Coulomb friction. Operates on associated Dem3DofGeom and CSPhys instances.

class Law2_Dem6DofGeom_FrictPhys_Beam(*inherits LawFunctor → Functor → Serializable*)
Class for demonstrating beam-like behavior of contact (normal, shear, bend and twist) [broken][experimental]

class Law2_SCG_MomentPhys_CohesionlessMomentRotation(*inherits LawFunctor → Functor → Serializable*)
Contact law based on Plassiard et al. (2009) : A spherical discrete element model: calibration procedure and incremental response. The functionality has been verified with results in the paper. The contribution of stiffnesses are scaled according to the radius of the particle, as implemented in that paper.
See also associated classes MomentMat, Ip2_MomentMat_MomentMat_MomentPhys, MomentPhys.
Note: This constitutive law can be used with triaxial test, but the following significant changes in code have to be made: Ip2_MomentMat_MomentMat_MomentPhys and Law2_SCG_MomentPhys_CohesionlessMomentRotation have to be added. Since it uses ScGeom, it uses boxes rather than facets. Spheres and boxes have to be changed to MomentMat rather than FrictMat.

preventGranularRatcheting(=*false*)
??

class Law2_ScGeom_CFpmPhys_CohesiveFrictionalPM(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the CFpm model.

preventGranularRatcheting(=*false*)
If true rotations are computed such as granular ratcheting is prevented. See article [2], pg. 3-10 – and a lot more papers from the same authors).

class Law2_ScGeom_CohFrictPhys_ElasticPlastic(*inherits LawFunctor → Functor → Serializable*)
Law for linear traction-compression-bending-twisting, with cohesion+friction and Mohr-Coulomb plasticity surface. Can be elastic-fragile or perfectly elastic-plastic. Creep at contact can be enabled.

Note: This law uses ScGeom.

always_use_moment_law(=*false*)
: If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

creep_viscosity(=*1*)
: creep viscosity [Pa.s/m]. probably should be moved to Ip2_2xCohFrictMat_CohFrictPhys...

detectBrokenBodies(=*false*)

erosionActivated(=*false*)

momentRotationLaw(=*false*)
: use bending/twisting moment at contacts. See CohesiveFrictionalContactLaw::always_use_moment_law for details.

neverErase(=*false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

shear_creep(=*false*)
: activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

twist_creep(=*false*)
: activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

class **Law2_ScGeom_FrictPhys_Basic(***inherits LawFunctor → Functor → Serializable***)**
: Law for linear compression, without cohesion and Mohr-Coulomb plasticity surface.

Note: This law uses ScGeom; there is also functionally equivalent Law2_Dem3DofGeom_FrictPhys_Basic, which uses Dem3DofGeom (sphere-box interactions are not implemented for the latest).

elasticEnergy() → float
: Compute and return the total elastic energy in all "FrictPhys" contacts

initPlasticDissipation(*(float)arg2***)** → None
: Initialize cummulated plastic dissipation to a value (0 by default).

neverErase(=*false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

plasticDissipation() → float
: Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if Law2_ScGeom_FrictPhys_Basic::traceEnergy is true.

traceEnergy(=*false*)
: Define the total energy dissipated in plastic slips at all contacts.

useShear(=*false*)
: Use ScGeom::updateShear rather than ScGeom::rotateAndGetShear for shear force computation.

class **Law2_ScGeom_MindlinPhys_Mindlin(***inherits LawFunctor → Functor → Serializable***)**
: Constitutive law for the Mindlin's formulation.

preventGranularRatcheting(=*false*)
: bool to avoid granular ratcheting

class **Law2_ScGeom_ViscElPhys_Basic(***inherits LawFunctor → Functor → Serializable***)**
: Linear viscoelastic model operating on ScGeom and ViscElPhys.

C.8.2. LawDispatcher

class **LawDispatcher(***inherits Dispatcher → Engine → Serializable***)**
: Dispatcher for applying constitutive laws on interactions.

__init__() → None
: object __init__(tuple args, dict kwds)

__init__((object)arg2) → **object :** Construct with list of associated functors.

dispFunctor(*(InteractionGeometry)arg2*, *(InteractionPhysics)arg3***)** → LawFunctor
: Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

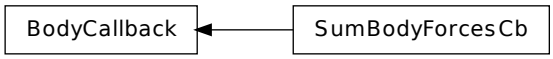
dispMatrix([*(bool)names=True*]**)** → dict
: Return dictionary with contents of the dispatch matrix.

functors

Functors objects associated with this dispatcher.

C.9. Callbacks

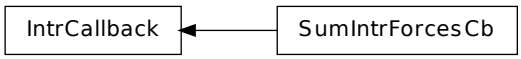
C.9.1. BodyCallback



class BodyCallback(*inherits Serializable***)**
Abstract callback object which will be called for every Body after being processed by NewtonIntegrator. See IntrCallback for details.

class SumBodyForcesCb(*inherits BodyCallback → Serializable***)**
Callback summing magnitudes of resultant forces over dynamic bodies.

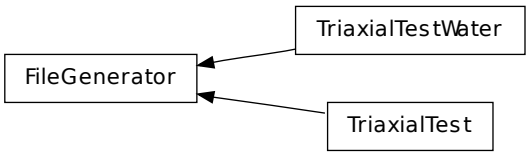
C.9.2. IntrCallback



class IntrCallback(*inherits Serializable***)**
Abstract callback object which will be called for every (real) Interaction after the interaction has been processed by InteractionDispatchers.
At the beginning of the interaction loop, **stepInit** is called, initializing the object; it returns either **NULL** (to deactivate the callback during this time step) or pointer to function, which will then be passed (1) pointer to the callback object itself and (2) pointer to Interaction.
Note: (NOT YET DONE) This functionality is accessible from python by passing 4th argument to InteractionDispatchers constructor, or by appending the callback object to InteractionDispatchers::callbacks.

class SumIntrForcesCb(*inherits IntrCallback → Serializable***)**
Callback summing magnitudes of forces over all interactions. InteractionPhysics of interactions must derive from NormShearPhys (responsability fo the user).

C.10. Preprocessors



class FileGenerator(*inherits Serializable***)**
Base class for scene generators, preprocessors.
generate(*(str)out***)** → None
Generate scene, save to given file
load() → None
Generate scene, save to temporary file and load immediately
outputFileName(*="./scene.xml"***)**
Filename to write resulting simulation to

class TriaxialTest(*inherits FileGenerator → Serializable***)**
Prepare a scene for triaxial tests. See full documentation at http://yade-dem.org/wiki/TriaxialTest.

Key(=*"*"**)**
A code that is added to output filenames.
StabilityCriterion(=*0.01*)
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.
WallStressRecordFile(=*"./WallStresses"+Key*)
autoCompressionActivation(=*true*)
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?
autoStopSimulation(=*false*)
freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)
autoUnload(=*true*)
auto adjust the isotropic stress state from TriaxialTest::sigmaIsoCompaction to TriaxialTest::sigmaLateralConfinement if they have different values. See docs for TriaxialCompressionEngine::autoUnload
biaxial2dTest(=*false*)
FIXME : what is that?
boxFrictionDeg(=*0.0*)
Friction angle [°] of boundaries contacts.
boxKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for boxes.
boxWalls(=*true*)
Use boxes for boundaries (recommended).
boxYoungModulus(=*15000000.0*)
Stiffness of boxes.
compactionFrictionDeg(=*sphereFrictionDeg*)
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overriden by TriaxialTest::sphereFrictionDeg before triaxial testing.
dampingForce(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)
dampingMomentum(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)
defaultDt(=*-1*)
Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.
density(=*2600*)
density of spheres
facetWalls(=*false*)
Use facets for boundaries (not tested)
finalMaxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (secondary precise adjustment)
fixedBoxDims(=*"*"**)**
string that contains some subset (max. 2) of {'x','y','z'} ; containes axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.
fixedPoroCompaction(=*false*)
flag to choose an isotropic compaction until a fixed porosity choosing a same translation speed for the six walls
fixedPorosity(=*1*)
FIXME : what is that?
importFilename(=*"*"**)**
File with positions and sizes of spheres.
internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.
lowerCorner(=*Vector3r(0, 0, 0)*)
Lower corner of the box.
maxMultiplier(=*1.01*)

max multiplier of diameters during internal compaction (initial fast increase)

maxWallVelocity(=*10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

noFiles(=*false*)
Do not create any files during run (.xml, .spheres, wall stress records)

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and TriaxialTest::numberOfGrains

radiusStdDev(=*0.3*)
Normalized standard deviation of generated sizes.

recordIntervalIter(=*20*)
interval between file outputs

sigmaIsoCompaction(=*50000*)
Confining stress during isotropic compaction.

sigmaLateralConfinement(=*50000*)
Lateral stress during triaxial loading. An isotropic unloading is performed if the value is not equal to TriaxialTest::SigmaIsoCompaction.

sphereFrictionDeg(=*18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(=*15000000.0*)
Stiffness of spheres.

strainRate(=*0.1*)
Strain rate in triaxial loading.

thickness(=*0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepUpdateInterval(=*50*)
interval for GlobalStiffnessTimeStepper

upperCorner(=*Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(=*1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(=*10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(=*false*)
Use walls for boundaries (not tested)

class **TriaxialTestWater(***inherits FileGenerator → Serializable***)**

This preprocessor is a variant of TriaxialTest, including the model of capillary forces developped as part of the PhD of Luc Scholtès. See the documentation of Law2_ScGeom_CapillaryPhys_Capillarity or the main page https://yade-dem.org/wiki/CapillaryTriaxialTest, for more details. Results obtained with this preprocessor were reported for instance in 'Scholtes et al. Micromechanics of granular materials with capillary effects. International Journal of Engineering Science 2009,(47)1, 64-75.'

CapillaryPressure(=*0*)
Define succion in the packing [Pa]. This is the value used in the capillary model.

Key(=*""*)
A code that is added to output filenames.

Rdispersion(=*0.3*)
Normalized standard deviation of generated sizes.

StabilityCriterion(=*0.01*)
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

WallStressRecordFile(=*"./WallStressesWater"+Key*)

autoCompressionActivation(=*true*)
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?
autoStopSimulation(=*false*)
freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)
autoUnload(=*true*)
auto adjust the isotropic stress state from TriaxialTest::sigmaIsoCompaction to TriaxialTest::sigmaLateralConfinement if they have different values. See docs for TriaxialCompressionEngine::autoUnload
biaxial2dTest(=*false*)
FIXME : what is that?
binaryFusion(=*true*)
Defines how overlapping bridges affect the capillary forces (see TriaxialTestWater::fusionDetection). If binary=true, the force is null as soon as there is an overlap detected, if not, the force is divided by the number of overlaps.
boxFrictionDeg(=*0.0*)
Friction angle [°] of boundaries contacts.
boxKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for boxes.
boxWalls(=*true*)
Use boxes for boundaries (recommended).
boxYoungModulus(=*15000000.0*)
Stiffness of boxes.
capillaryStressRecordFile(=*"./capStresses"+Key*)
compactionFrictionDeg(=*sphereFrictionDeg*)
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overriden by TriaxialTest::sphereFrictionDeg before triaxial testing.
contactStressRecordFile(=*"./contStresses"+Key*)
dampingForce(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)
dampingMomentum(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)
defaultDt(=*0.0001*)
Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.
density(=*2600*)
density of spheres
facetWalls(=*false*)
Use facets for boundaries (not tested)
finalMaxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (secondary precise adjustment)
fixedBoxDims(=*"*)
string that contains some subset (max. 2) of {'x','y','z'} ; containes axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.
fixedPoroCompaction(=*false*)
flag to choose an isotropic compaction until a fixed porosity choosing a same translation speed for the six walls
fixedPorosity(=*1*)
FIXME : what is that?
fusionDetection(=*false*)
test overlaps between liquid bridges on modify forces if overlaps exist
importFilename(=*"*)
File with positions and sizes of spheres.
internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.
lowerCorner(=*Vector3r(0, 0, 0)*)

Lower corner of the box.

maxMultiplier(=*1.01*)
max multiplier of diameters during internal compaction (initial fast increase)

maxWallVelocity(=*10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

noFiles(=*false*)
Do not create any files during run (.xml, .spheres, wall stress records)

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and TriaxialTest::numberOfGrains

recordIntervalIter(=*20*)
interval between file outputs

sigmaIsoCompaction(=*50000*)
Confining stress during isotropic compaction.

sigmaLateralConfinement(=*50000*)
Lateral stress during triaxial loading. An isotropic unloading is performed if the value is not equal to TriaxialTestWater::SigmaIsoCompaction.

sphereFrictionDeg(=*18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(=*15000000.0*)
Stiffness of spheres.

strainRate(=*1*)
Strain rate in triaxial loading.

thickness(=*0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepOutputInterval(=*50*)
interval for outputing general informations on the simulation (stress,unbalanced force,...)

timeStepUpdateInterval(=*50*)
interval for GlobalStiffnessTimeStepper

upperCorner(=*Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(=*1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(=*10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(=*false*)
Use walls for boundaries (not tested)

water(=*true*)
activate capillary model

C.11. Rendering

C.11.1. OpenGLRenderingEngine

class OpenGLRenderingEngine(*inherits Serializable***)**
Class responsible for rendering scene on OpenGL devices.

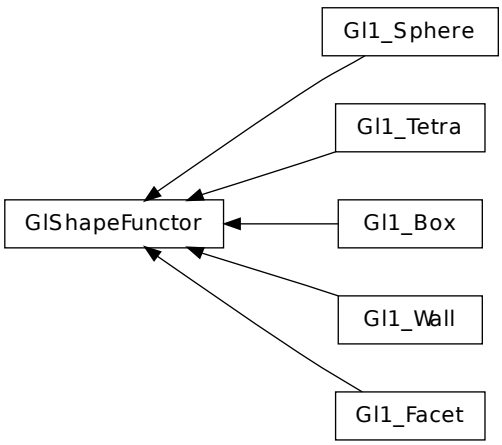
bgColor(=*Vector3r(.2, .2, .2)*)
Color of the backgroud canvas (RGB)

bound(=*false*)
Render body Bound

clipPlaneActive(=*vector<int>(numClipPlanes, 0)*)
Activate/deactivate respective clipping planes

clipPlaneSe3(=*vector<Se3r>(numClipPlanes, Se3r(Vector3r::Zero(), Quaternionr::Identity()))*)
Position and orientation of clipping planes
dispScale(=*Vector3r::Ones()*, *disable scaling*)
Artificially enlarge (scale) dispalcements from bodies' reference positions by this relative amount, so that they become better visible (independently in 3 dimensions). Disbled if (1,1,1).
dof(=*false*)
Show which degrees of freedom are blocked for each body
id(=*false*)
Show body id's
intrAllWire(=*false*)
Draw wire for all interactions, blue for potential and green for real ones (mostly for debugging)
intrGeom(=*false*)
Render Interaction::interactionGeometry objects.
intrPhys(=*false*)
Render Interaction::interactionPhysics objects
intrWire(=*false*)
If rendering interactions, use only wires to represent them.
lightPos(=*Vector3r(75, 130, 0)*)
Position of OpenGL light source in the scene.
mask(=*~0*, *draw everything*)
Bitmask for showing only bodies where ((mask & Body::mask)!=0)
rotScale(=*1.*, *disable scaling*)
Artificially enlarge (scale) rotations of bodies relative to their reference orientation, so the they are better visible.
selectBodyLimit(=*1000*)
Limit number of bodies to allow picking body with mouse (performance reasons)
setRefSe3() → None
Make current positions and orientation reference for scaleDisplacements and scaleRotations.
shape(=*true*)
Render body Shape
wire(=*false*)
Render all bodies with wire only (faster)

C.11.2. GlShapeFunctor



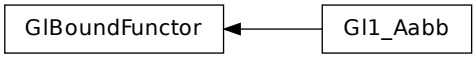
class GlShapeFunctor(*inherits Functor → Serializable***)**
Abstract functor for rendering Body::shape objects.
class Gl1_Box(*inherits GlShapeFunctor → Functor → Serializable***)**
Renders Box object
class Gl1_Facet(*inherits GlShapeFunctor → Functor → Serializable***)**
Renders Facet object

class Gl1_Sphere(*inherits GlShapeFunctor → Functor → Serializable***)**
Renders Sphere object
class Gl1_Tetra(*inherits GlShapeFunctor → Functor → Serializable***)**
Renders Tetra object
class Gl1_Wall(*inherits GlShapeFunctor → Functor → Serializable***)**
Renders Wall object

C.11.3. GlStateFunctor

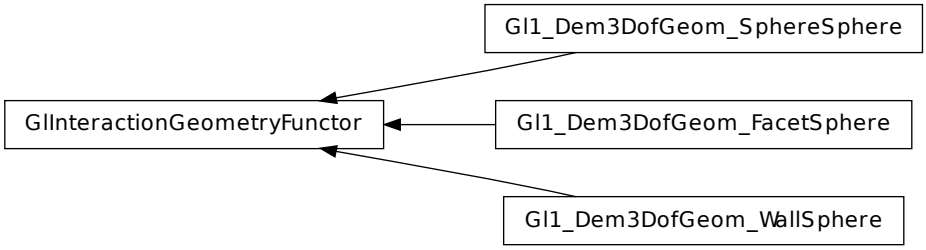
class GlStateFunctor(*inherits Functor → Serializable***)**
Abstract functor for rendering Body::state objects.

C.11.4. GlBoundFunctor



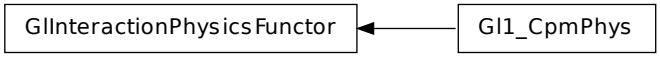
class GlBoundFunctor(*inherits Functor → Serializable***)**
Abstract functor for rendering Body::bound objects.
class Gl1_Aabb(*inherits GlBoundFunctor → Functor → Serializable***)**
Render Axis-aligned bounding box (Aabb).

C.11.5. GlInteractionGeometryFunctor



class GlInteractionGeometryFunctor(*inherits Functor → Serializable***)**
Abstract functor for rendering Interaction::interactionGeometry objects.
class Gl1_Dem3DofGeom_FacetSphere(*inherits GlInteractionGeometryFunctor → Functor → Serializable***)**
Render interaction of facet and sphere (represented by Dem3DofGeom_FacetSphere)
class Gl1_Dem3DofGeom_SphereSphere(*inherits GlInteractionGeometryFunctor → Functor → Serializable***)**
Render interaction of 2 spheres (represented by Dem3DofGeom_SphereSphere)
class Gl1_Dem3DofGeom_WallSphere(*inherits GlInteractionGeometryFunctor → Functor → Serializable***)**
Render interaction of wall and sphere (represented by Dem3DofGeom_WallSphere)

C.11.6. GlInteractionPhysicsFunctor



class GlInteractionPhysicsFunctor(*inherits Functor → Serializable***)**
Abstract functor for rendering Interaction::interactionPhysics objects.

class Gl1_CpmPhys(*inherits GlInteractionPhysicsFunctor → Functor → Serializable***)**
Render CpmPhys objects of interactions.

C.12. Simulation data

C.12.1. Omega

class Omega

bodies
Bodies in the current simulation (container supporting index access by id and iteration)

cell
Periodic cell of the current scene (None if the scene is aperiodic).

childClassesNonrecursive(*(str)arg2***)** → list
Return list of all classes deriving from given class, as registered in the class factory

disableGdb() → None
Revert SEGV and ABRT handlers to system defaults.

dt
Current timestep (Δt) value.
- assigning zero enables dynamic Δt control via a TimeStepper (raises an exception if there is no TimeStepper among O.engines)
- assigning negative value enables dynamic Δt (as in the previous case) and sets positive timestep **O.dt=**|Δ**t**| (will be used until the timestepper is run and updates it)
- assigning positive value sets Δt to that value and disables dynamic Δt (via TimeStepper, if there is one).

dynDt can be used to query whether dynamic Δt is in use.

dynDt
Whether a TimeStepper is used for dynamic Δt control. See dt on how to enable/disable TimeStepper.

engines
List of engines in the simulation (Scene::engines).

exitNoBacktrace([*(int)status=0*]**)** → None
Disable SEGV handler and exit, optionally with given status number.

forceSyncCount
Counter for number of syncs in ForceContainer, for profiling purposes.

forces
ForceContainer (forces, torques, displacements) in the current simulation.

initializers
List of initializers (Scene::initializers).

interactions
Interactions in the current simulation (container supporting index acces by either (id1,id2) or interactionNumber and iteration)

isChildClassOf(*(str)arg2*, *(str)arg3***)** → bool
Tells whether the first class derives from the second one (both given as strings).

iter
Get current step number

labeledEngine(*(str)arg2***)** → object
Return instance of engine/functor with the given label. This function shouldn't be called by the user directly; every ehange in O.engines will assign respective global python variables according to labels.
For example:: O.engines=[InsertionSortCollider(label='collider')] collider.nBins=5 ## collider has become a variable after assignment to O.engines automatically)

load(*(str)arg2***)** → None
Load simulation from file.

load2(*(str)arg2***)** → None
[EXPERIMENTAL] load using boost::serialization (handles compression, XML/binary)

loadTmp([*(str)mark=''*]**)** → None

Load simulation previously stored in memory by saveTmp. *mark* optionally distinguishes multiple saved simulations

materials
Shared materials; they can be accessed by id or by label

miscParams
MiscParams in the simulation (Scene::mistParams), usually used to save serializables that don't fit anywhere else, like GL functors

numThreads
Get maximum number of threads openMP can use.

pause() → None
Stop simulation execution. (May be called from within the loop, and it will stop after the current step).

periodic
Get/set whether the scene is periodic or not (True/False).

plugins() → list
Return list of all plugins registered in the class factory.

realtime
Return clock (human world) time the simulation has been running.

reload() → None
Reload current simulation

reset() → None
Reset simulations completely (including another scene!).

resetThisScene() → None
Reset current scene.

resetTime() → None
Reset simulation time: step number, virtual and real time. (Doesn't touch anything else, including timings).

run([*(int)nSteps=-1*[, *(bool)wait=False*]]**)** → None
Run the simulation. *nSteps* how many steps to run, then stop (if positive); *wait* will cause not returning to python until simulation will have stopped.

runEngine(*(Engine)arg2***)** → None
Run given engine exactly once; simulation time, step number etc. will not be incremented (use only if you know what you do).

save(*(str)arg2***)** → None
Save current simulation to file (should be .xml or .xml.bz2)

save2(*(str)arg2***)** → None
[EXPERIMENTAL] save using boost::serialization (handles compression, XML/binary)

saveTmp([*(str)mark=''*]**)** → None
Save simulation to memory (disappears at shutdown), can be loaded later with loadTmp. *mark* optionally distinguishes different memory-saved simulations.

step() → None
Advance the simulation by one step. Returns after the step will have finished.

stopAtIter
Get/set number of iteration after which the simulation will stop.

switchScene() → None
Switch to alternative simulation (while keeping the old one). Calling the function again switches back to the first one. Note that most variables from the first simulation will still refer to the first simulation even after the switch (e.g. b=O.bodies[4]; O.switchScene(); [b still refers to the body in the first simulation here])

tags
Tags (string=string dictionary) of the current simulation (container supporting string-index access/assignment)

time
Return virtual (model world) time of the simulation.

timingEnabled
Globally enable/disable timing services (see documentation of yade.timing).

tmpFilename() → str
Return unique name of file in temporary directory which will be deleted when yade exits.

tmpToFile(*(str)fileName*[, *(str)mark=''*]) → None
Save XML of saveTmp'd simulation into *fileName*.
tmpToString([*(str)mark=''*]) → str
Return XML of saveTmp'd simulation as string.
wait() → None
Don't return until the simulation will have been paused. (Returns immediately if not running).

C.12.2. BodyContainer

class BodyContainer

__init__(*(BodyContainer)arg2***)** → None
append(*(Body)arg2***)** → int
Append one Body instance, return its id.
append((BodyContainer)arg1, (object)arg2) → **object :** Append list of Body instance, return list of ids
appendClumped(*(object)arg2***)** → tuple
Append given list of bodies as a clump (rigid aggregate); return list of ids.
clear() → None
Remove all bodies (interactions not checked)
erase(*(int)arg2***)** → bool
Erase body with the given id; all interaction will be deleted by InteractionDispatchers in the next step.
replace(*(object)arg2***)** → object

C.12.3. InteractionContainer

class InteractionContainer
Access to interactions of simulation, by using
1.id's of both Bodies of the interactions, e.g. **O.interactions[23,65]**
2.iteration over the whole container:

```
for i in O.interactions: print i.id1,i.id2
```

Note: Iteration silently skips interactions that are not real.
__init__(*(InteractionContainer)arg2***)** → None
clear() → None
Remove all interactions
countReal() → int
Return number of interactions that are "real", i.e. they have phys and geom.
eraseNonReal() → None
Erase all interactions that are not real .
nth(*(int)arg2***)** → Interaction
Return n-th interaction from the container (usable for picking random interaction).
serializeSorted
withBody(*(int)arg2***)** → list
Return list of real interactions of given body.
withBodyAll(*(int)arg2***)** → list
Return list of all (real as well as non-real) interactions of given body.

C.12.4. ForceContainer

class ForceContainer

__init__(*(ForceContainer)arg2***)** → None
addF(*(int)id*, *(Vector3)f***)** → None
Apply force on body (accumulates).
addMove(*(int)id*, *(Vector3)m***)** → None
Apply displacement on body (accumulates).

addRot(*(int)id*, *(Vector3)r***)** → None
Apply rotation on body (accumulates).
addT(*(int)id*, *(Vector3)t***)** → None
Apply torque on body (accumulates).
f(*(int)id***)** → Vector3
Force applied on body.
m(*(int)id***)** → Vector3
Deprecated alias for t (torque).
move(*(int)id***)** → Vector3
Displacement applied on body.
rot(*(int)id***)** → Vector3
Rotation applied on body.
syncCount
Number of synchronizations of ForceContainer (cummulative); if significantly higher than number of steps, there might be unnecessary syncs hurting performance.
t(*(int)id***)** → Vector3
Torque applied on body.

C.13. Other classes

class Engine(*inherits Serializable***)**
Basic execution unit of simulation, called from the simulation loop (O.engines)
execCount
Cummulative count this engine was run (only used if O.timingEnabled==True).
execTime
Cummulative time this Engine took to run (only used if O.timingEnabled==True).
label(*=uninitalized***)**
Textual label for this object; must be valid python identifier, you can refer to it directly from python.
timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

class TimingDeltas

data
Get timing data as list of tuples (label, execTime[nsec], execCount) (one tuple per checkpoint)
reset() → None
Reset timing information

class ParallelEngine(*inherits Engine* → *Serializable***)**
Engine for running other Engine in parallel.
__init__() → None
object __init__(tuple args, dict kwds)
__init__((list)arg2) → **object :** Construct from (possibly nested) list of slaves.
slaves
List of lists of Engines; each top-level group will be run in parallel with other groups, while Engines inside each group will be run sequentially, in given order.

class Functor(*inherits Serializable***)**
Function-like object that is called by Dispatcher, if types of arguments match those the Functor declares to accept.
bases
Ordered list of types (as strings) this functor accepts.
label(*=uninitalized***)**
Textual label for this object; must be valid python identifier, you can refer to it directly fron python (must be a valid python identifier).
timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

class Serializable

dict() → dict
Return dictionary of attributes.

has_key(*(str)arg2***)** → bool
Predicate telling whether given attribute exists.

keys() → list
Return list of attribute names

name
Name of the class

postProcessAttributes([*(bool)deserializing=True*]**)** → None
Call Serializable::postProcessAttributes c++ method.

updateAttrs(*(dict)arg2***)** → None
Update object attributes from given dictionary

updateExistingAttrs(*(dict)arg2***)** → list
Update object attributes from given dictionary, skipping those that the instance doesn't have. Return list of attributes that did *not* exist and were not updated.

class Cell(*inherits Serializable***)**
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

Hsize(*=Matrix3r::Zero()***)**
The current period size (one column per box edge) *(auto-updated)*

refSize
Reference size of the cell.

size
Current size of the cell, i.e. lengths of 3 cell lateral vectors after applying current trsf. Update automatically at every step.

trsf
Transformation matrix of the cell.

velGrad(*=Matrix3r::Zero()***)**
Velocity gradient of the transformation; used in NewtonIntegrator.

class Dispatcher(*inherits Engine* → *Serializable***)**
Engine dispatching control to its associated functors, based on types of argument it receives.

functorArguments(*=uninitalized***)**
Instances of functors

functorNames(*=uninitalized***)**
Names of functor classes

D. Yade modules

D.1. yade.eudoxos module

Miscillaneous functions that are not believed to be generally usable, therefore kept in my "private" module here.

They comprise notably oofem export and various CPM-related functions.

class yade.eudoxos.IntrSmooth3d

Return spatially weigted gaussian average of arbitrary quantity defined on interactions.

At construction time, all real interactions are put inside spatial grid, permitting fast search for points in neighbourhood defined by distance.

Parameters for the distribution are standard deviation σ and relative cutoff distance *relThreshold* (3 by default) which will discard points farther than *relThreshold* $\times\sigma$.

Given central point p_0, points are weighted by gaussian function

$$\rho(p_0,p)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-\|p_0-p\|^2}{2\sigma^2}\right)$$

To get the averaged value, simply call the instance, passing central point and callable object which received interaction object and returns the desired quantity:

```
>>> is3d=IntrSmooth3d(0.003)
>>> is3d(Vector3r(0,0,0),lambda i: i.phys.omega)
```

bounds()

count()

yade.eudoxos.displacementsInteractionsExport(*fName***)**

yade.eudoxos.eliminateJumps(*eps*, *sigma*, *numSteep=10*, *gapWidth=5*, *movWd=40***)**

yade.eudoxos.estimatePoissonYoung(*principalAxis*, *stress=0*, *plot=False*, *cutoff=0.0***)**

Estimate Poisson's ration given the "principal" axis of straining. For every base direction, homogenized strain is computed (slope in linear regression on discrete function particle coordinate $\rightarrow\rightarrow$ particle displacement in the same direction as returned by utils.coordsAndDisplacements) and, (if axis '0' is the strained axis) the poisson's ratio is given as $-\frac{1}{2}(\epsilon 1+\epsilon 2)/\epsilon_0$.

Young's modulus is computed as σ/ϵ_0; if stress σ is not given (default 0), the result is 0.

cutoff, if $>0.$, will take only smaller part (centered) or the specimen into account

yade.eudoxos.estimateStress(*strain*, *cutoff=0.0***)**

Use summed stored energy in contacts to compute macroscopic stress over the same volume, provided known strain.

yade.eudoxos.oofemDirectExport(*fileBase*, *title=None*, *negIds=[]*, *posIds=[]***)**

yade.eudoxos.oofemPrescribedDisplacementsExport(*fileName***)**

yade.eudoxos.oofemTextExport(*fName***)**

Export simulation data in text format

The format is line-oriented as follows: # 3 lines of material parameters: 1. E G # elastic 2. epsCrackOnset relDuctility xiShear transStrainCoeff #tension; epsFr=epsCrackOnset*relDuctility 3. cohesionT tanPhi # shear 4. [number of spheres] [number of links] 5. id x y z r -1/0/1[on negative/no/positive boundary] # spheres ... n. id1 id2 contact_point_x cp_y cp_z A # interactions

yade.eudoxos.particleConfinement() $\rightarrow$ None

yade.eudoxos.testNumpy() $\rightarrow$ dict

yade.eudoxos.velocityTowardsAxis(*(Vector3)axisPoint*, *(Vector3)axisDirection*, *(float)timeToAxis*[, *(float)subtractDist*[, *(float)perturbation*]]) $\rightarrow$ None

class yade._eudoxos.InteractionLocator

Locate all (real) interactions in space by their contact point. When constructed, all real interactions are spatially indexed (uses vtkPointLocator internally). Use intrsWithinDistance to use those data. **Note:** Data might become inconsistent with real simulation state if simulation is being run between creation of this object and spatial queries.

bounds

Return coordinates of lower and uppoer corner of axis-aligned abounding box of all interactions

count

Number of interactions held

intrsAroundPt(*(Vector3)point*, *(float)maxDist***)** → list

Return list of real interactions that are not further than *maxDist* from *point*.

macroAroundPt(*(Vector3)point*, *(float)maxDist***)** → tuple

Return tuple of averaged stress tensor (as Matrix3), average omega and average kappa values.

class yade._eudoxos.SpiralInteractionLocator2d

Locate all real interactions in 2d plane (reduced by spiral projection from 3d, using **Shop::spiralProject**, which is the same as utils.spiralProject) using their contact points.

Note: Do not run simulation while using this object.

___init___(*(float)dH_dTheta*[, *(int)axis=0*[, *(float)theta0=0*]])** → None

Parameters

dH_dTheta: float Spiral inclination, i.e. height increase per 1 radian turn; axis: int Axis of rotation (0=x,1=y,2=z) theta: float Spiral angle at zero height (theta intercept)

intrsAroundPt(*(Vector2)pt2d*, *(float)radius***)** → list

Return list of interaction objects that are not further from *pt2d* than *radius* in the projection plane

macroStressAroundPt(*(Vector2)pt2d*, *(float)radius***)** → Matrix3

Compute macroscopic stress around given point, rotating the interaction to the projection plane first. The formula used is

$$\sigma_{ij} = \frac{1}{V}\sum_{IJ} d^{IJ}A^{IJ}\left[\sigma^{N,IJ}n_i^{IJ}n_j^{IJ} + \frac{1}{2}\left(\sigma_i^{T,IJ}n_j^{IJ} + \sigma_j^{T,IJ}n_i^{IJ}\right)\right]$$

where the sum is taken over volume V containing interactions IJ between spheres I and J;

- i, j indices denote Cartesian components of vectors and tensors,
- d^{IJ} is current distance between spheres I and J,
- A^{IJ} is area of contact IJ,
- n is interaction normal (unit vector pointing from center of I to the center of J)
- $\sigma^{N,IJ}$ is normal stress (as scalar) in contact IJ,
- $\sigma^{T,IJ}$ is shear stress in contact IJ in global coordinates.

σ^T and n are transformed by angle ϑ as given by utils.spiralProject.

yade._eudoxos.particleConfinement() → None

yade._eudoxos.testNumpy() → dict

yade._eudoxos.velocityTowardsAxis(*(Vector3)axisPoint*, *(Vector3)axisDirection*, *(float)timeToAxis*[, *(float)subtractDist*[, *(float)perturbation*]])** → None

D.2. yade.linterpolation module

Module for rudimentary support of manipulation with piecewise-linear functions (which are usually interpolations of higher-order functions, whence the module name). Interpolation is always given as two lists of the same length, where the x-list must be increasing.

Periodicity is supported by supposing that the interpolation can wrap from the last x-value to the first x-value (which should be 0 for meaningful results).

Non-periodic interpolation can be converted to periodic one by padding the interpolation with constant head and tail using the sanitizeInterpolation function.

There is a c++ template function for interpolating on such sequences in pkg/common/Engine/PartialEngine/LinearInterpolate.hpp (stateful, therefore fast for sequential reads).

TODO: Interpolating from within python is not (yet) supported.

yade.linterpolation.integral(*x*, *y***)**
Return integral of piecewise-linear function given by points x0,x1,... and y0,y1,...
yade.linterpolation.revIntegrateLinear(*I*, *x0*, *y0*, *x1*, *y1***)**
Helper function, returns value of integral variable x for linear function f passing through (x0,y0),(x1,y1) such that 1. x⊠[x0,x1] 2. ⊠_x0^x f dx=I and raise exception if such number doesn't exist or the solution is not unique (possible?)
yade.linterpolation.sanitizeInterpolation(*x*, *y*, *x0*, *x1***)**
Extends piecewise-linear function in such way that it spans at least the x0...x1 interval, by adding constant padding at the beginning (using y0) and/or at the end (using y1) or not at all.
yade.linterpolation.xFractionalFromIntegral(*integral*, *x*, *y***)**
Return x within range x0...xn such that ⊠_x0^x f dx==integral. Raises error if the integral value is not reached within the x-range.
yade.linterpolation.xFromIntegral(*integralValue*, *x*, *y***)**
Return x such that ⊠_x0^x f dx==integral. x wraps around at xn. For meaningful results, therefore, x0 should == 0

D.3. yade.log module

Acess and manipulation of log4cxx loggers.
yade.log.loadConfig(*(str)fileName***)** → None
Load configuration from file (log4cxx::PropertyConfigurator::configure)
yade.log.setLevel(*(str)logger*, *(int)level***)** → None
Set minimum severity *level* (constants **TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, **FATAL**) for given logger. Leading 'yade.' will be appended automatically to the logger name; if logger is '', the root logger 'yade' will be operated on.

D.4. yade.pack module

Creating packings and filling volumes defined by boundary representation or constructive solid geometry.
For examples, see

- scripts/test/gts-horse.py
- scripts/test/gts-operators.py
- scripts/test/gts-random-pack-obb.py
- scripts/test/gts-random-pack.py
- scripts/test/pack-cloud.py
- scripts/test/pack-predicates.py
- examples/regular-sphere-pack/regular-sphere-pack.py

yade.pack.cloudBestFitOBB(*(tuple)arg1***)** → tuple
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) of best-fit oriented bounding-box for given tuple of points (uses brute-force velome minimization, do not use for very large clouds).
yade.pack.filterSpherePack(*predicate*, *spherePack*, ***kw***)**
Using given SpherePack instance, return spheres the satisfy predicate. The packing will be recentered to match the predicate and warning is given if the predicate is larger than the packing.
yade.pack.gtsSurface2Facets(*surf*, ***kw***)**
Construct facets from given GTS surface. **kw is passed to utils.facet.
yade.pack.gtsSurfaceBestFitOBB(*surf***)**
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) describing best-fit oriented bounding box (OBB) for the given surface. See cloudBestFitOBB for details.
class **yade.pack.inGtsSurface_py(***inherits Predicate***)**
This class was re-implemented in c++, but should stay here to serve as reference for implementing Predicates in pure python code. C++ allows us to play dirty tricks in GTS which are not accessible through pygts itself; the performance penalty of pygts comes from fact that if constructs and destructs bb tree for the surface at every invocation of gts.Point().is_inside(). That is cached in the c++ code, provided that the surface is not manipulated with during lifetime of the object (user's responsibility).
—
Predicate for GTS surfaces. Constructed using an already existing surfaces, which must be closed.

import gts surf=gts.read(open('horse.gts')) inGtsSurface(surf)

Note: Padding is optionally supported by testing 6 points along the axes in the pad distance. This must be enabled in the ctor by saying doSlowPad=True. If it is not enabled and pad is not zero, warning is issued.

aabb()

class yade.pack.inSpace(*inherits Predicate***)**

Predicate returning True for any points, with infinite bounding box.

aabb()

center()

dim()

yade.pack.randomDensePack(*predicate*, *radius*, *material=-1*, *dim=None*, *cropLayers=0*, *rRelFuzz=0.0*, *spheresInCell=0*, *memoizeDb=None*, *useOBB=True*, *memoDbg=False*, *color=None***)**

Generator of random dense packing with given geometry properties, using TriaxialTest (aperiodic) or PeriIsoCompressor (periodic). The priodicity depens on whether the spheresInCell parameter is given.

O.switchScene() magic is used to have clean simulation for TriaxialTest without deleting the original simulation. This function therefore should never run in parallel with some code accessing your simulation.

Parameters

- **predicate** – solid-defining predicate for which we generate packing
- **spheresInCell** – if given, the packing will be periodic, with given number of spheres in the periodic cell.
- **radius** – mean radius of spheres
- **rRelFuzz** – relative fuzz of the radius – e.g. radius=10, rRelFuzz=.2, then spheres will have radii $10 \pm (10*.2)$). 0 by default, meaning all spheres will have exactly the same radius.
- **cropLayers** – (aperiodic only) how many layers of spheres will be added to the computed dimension of the box so that there no (or not so much, at least) boundary effects at the boundaries of the predicate.
- **dim** – dimension of the packing, to override dimensions of the predicate (if it is infinite, for instance)
- **memoizeDb** – name of sqlite database (existent or nonexistent) to find an already generated packing or to store the packing that will be generated, if not found (the technique of caching results of expensive computations is known as memoization). Fuzzy matching is used to select suitable candidate – packing will be scaled, rRelFuzz and dimensions compared. Packing that are too small are dictarded. From the remaining candidate, the one with the least number spheres will be loaded and returned.
- **useOBB** – effective only if a inGtsSurface predicate is given. If true (default), oriented bounding box will be computed first; it can reduce substantially number of spheres for the triaxial compression (like $10\times$ depending on how much asymmetric the body is), see scripts/test/gts-triax-pack-obb.py.
- **memoDbg** – show packigns that are considered and reasons why they are rejected/accepted

Returns SpherePack object with spheres, filtered by the predicate.

yade.pack.randomPeriPack(*radius*, *rRelFuzz*, *initSize*, *memoizeDb=None***)**

Generate periodic dense packing. EXPERIMENTAL, you at your own risk.

A cell of initSize is stuffed with as many spheres as possible (ignore the warning from SpherePack::makeCloud about not being able to add any more spheres), then we run periodic compression with PeriIsoCompressor, just like with randomDensePack.

Parameters

- **radius** – mean sphere radius
- **rRelFuzz** – relative fuzz of sphere radius (equal distribution); see the same param for randomDensePack.
- **initSize** – initial size of the periodic cell.

Returns SpherePack object, which also contains periodicity information.

Todo memoization in db; what criteria??

yade.pack.regularHexa(*predicate*, *radius*, *gap*, ***kw***)**
Return set of spheres in regular hexagonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

yade.pack.regularOrtho(*predicate*, *radius*, *gap*, ***kw***)**
Return set of spheres in regular orthogonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

yade.pack.revolutionSurfaceMeridians(*sects*, *angles*, *origin=Vector3(0, 0, 0)*, *orientation=Quaternion((1, 0, 0), 0)***)**
Revolution surface given sequences of 2d points and sequence of corresponding angles, returning sequences of 3d points representing meridian sections of the revolution surface. The 2d sections are turned around z-axis, but they can be transformed using the origin and orientation arguments to give arbitrary orientation.

yade.pack.sweptPolylines2gtsSurface(*pts*, *threshold=0*, *capStart=False*, *capEnd=False***)**
Create swept suface (as GTS triangulation) given same-length sequences of points (as 3-tuples). If threshold is given (>0), then

- degenerate faces (with edges shorter than threshold) will not be created
- gts.Surface().cleanup(threshold) will be called before returning, which merges vertices mutually closer than threshold. In case your pts are closed (last point concident with the first one) this will the surface strip of triangles. If you additionally have capStart==True and capEnd==True, the surface will be closed.

Note: capStart and capEnd make the most naive polygon triangulation (diagonals) and will perhaps fail for non-convex sections.

Warning: the algorithm connects points sequentially; if two polylines are mutually rotated or have inverse sense, the algorithm will not detect it and connect them regardless in their given order.

Creation, manipulation, IO for generic sphere packings.

class yade._packSpheres.SpherePack
Set of spheres represented as centers and radii. This class is returned by pack.randomDensePack, pack.randomPeriPack and others. The object supports iteration over spheres, as in

```
>>> sp=SpherePack()
>>> for center,radius in sp: print center,radius
```

```
>>> for sphere in sp: print sphere[0],sphere[1]   ## same, but without unpacking the tuple automatically
```

```
>>> for i in range(0,len(sp)): print sp[i][0], sp[i][1]   ## same, but accessing spheres by index
```

___init___([*(list)list*]**)** → None
Empty constructor, optionally taking list [((cx,cy,cz),r), ...] for initial data.

aabb() → tuple
Get axis-aligned bounding box coordinates, as 2 3-tuples.

add(*(Vector3)arg2*, *(float)arg3***)** → None
Add single sphere to packing, given center as 3-tuple and radius

cellFill(*(Vector3)arg2***)** → None
Repeat the packing (if periodic) so that the results has dim() >= given size. The packing retains periodicity, but changes cellSize. Raises exception for non-periodic packing.

cellRepeat(*(Vector3i)arg2***)** → None
Repeat the packing given number of times in each dimension. Periodicity is retained, cellSize changes. Raises exception for non-periodic packing.

cellSize
Size of periodic cell; is Vector3(0,0,0) if not periodic. (Change this property only if you know what you're doing).

center() → Vector3
Return coordinates of the bounding box center.

dim() → Vector3
Return dimensions of the packing in terms of aabb(), as a 3-tuple.

fromList(*(list)arg2***)** → None
Make packing from given list, same format as for constructor. Discards current data.

fromSimulation() → None
Make packing corresponding to the current simulation. Discards current data.
load(*(str)fileName***)** → None
Load packing from external text file (current data will be discarded).
makeCloud(*(Vector3)minCorner*, *(Vector3)maxCorner*, *(float)rMean*, *(float)rRelFuzz*[, *(int)num=-1*[, *(bool)periodic=False*[, *(float)porosity=-1*]]]) → int
Create random packing enclosed in box given by minCorner and maxCorner, containing num spheres. Returns number of created spheres, which can be < num if the packing is too tight. If porosity>0, recompute meanRadius (porosity>0.65 recommended) and try generating this porosity with num spheres.
relDensity() → float
Relative packing density, measured as sum of spheres' volumes / aabb volume. (Sphere overlaps are ignored.)
rotate(*(Vector3)axis*, *(float)angle***)** → None
Rotate all spheres around packing center (in terms of aabb()), given axis and angle of the rotation.
save(*(str)fileName***)** → None
Save packing to external text file (will be overwritten).
scale(*(float)arg2***)** → None
Scale the packing around its center (in terms of aabb()) by given factor (may be negative).
toList() → list
Return packing data as python list.
toList_pointsAsTuples() → list
Return packing data as python list, but using only pure-python data types (3-tuples instead of Vector3) (for pickling with cPickle)
translate(*(Vector3)arg2***)** → None
Translate all spheres by given vector.

class **yade._packSpheres.SpherePackIterator**

__init__(*(SpherePackIterator)arg2***)** → None
next() → tuple
Spatial predicates for volumes (defined analytically or by triangulation).

class **yade._packPredicates.Predicate**

aabb() → tuple
aabb((Predicate)arg1) → None
center() → Vector3
dim() → Vector3

class **yade._packPredicates.PredicateBoolean(***inherits Predicate***)**
Boolean operation on 2 predicates (abstract class)
A
B
__init__()
Raises an exception This class cannot be instantiated from Python

class **yade._packPredicates.PredicateDifference(***inherits PredicateBoolean → Predicate***)**
Difference (conjunction with negative predicate) of 2 predicates. A point has to be inside the first and outside the second predicate. Can be constructed using the - operator on predicates: **pred1 - pred2**.
__init__(*(object)arg2*, *(object)arg3***)** → None

class **yade._packPredicates.PredicateIntersection(***inherits PredicateBoolean → Predicate***)**
Intersection (conjunction) of 2 predicates. A point has to be inside both predicates. Can be constructed using the **&** operator on predicates: **pred1 & pred2**.
__init__(*(object)arg2*, *(object)arg3***)** → None

class **yade._packPredicates.PredicateSymmetricDifference(***inherits PredicateBoolean → Predicate***)**
SymmetricDifference (exclusive disjunction) of 2 predicates. A point has to be in exactly one predicate of the two. Can be constructed using the ^ operator on predicates: **pred1 ^ pred2**.
__init__(*(object)arg2*, *(object)arg3***)** → None

class yade._packPredicates.PredicateUnion(*inherits PredicateBoolean → Predicate***)**
Union (non-exclusive disjunction) of 2 predicates. A point has to be inside any of the two predicates to be inside. Can be constructed using the | operator on predicates: **pred1** | **pred2**.

___init___(*(object)arg2*, *(object)arg3***)** → None

class yade._packPredicates.inAlignedBox(*inherits Predicate***)**
Axis-aligned box predicate

___init___(*(Vector3)minAABB*, *(Vector3)maxAABB***)** → None
Ctor taking minumum and maximum points of the box (as 3-tuples).

class yade._packPredicates.inCylinder(*inherits Predicate***)**
Cylinder predicate

___init___(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius***)** → None
Ctor taking centers of the lateral walls (as 3-tuples) and radius.

class yade._packPredicates.inEllipsoid(*inherits Predicate***)**
Ellipsoid predicate

___init___(*(Vector3)centerPoint*, *(Vector3)abc***)** → None
Ctor taking center of the ellipsoid (3-tuple) and its 3 radii (3-tuple).

class yade._packPredicates.inGtsSurface(*inherits Predicate***)**
GTS surface predicate

___init___(*(object)surface*[, *(bool)noPad*]**)** → None
Ctor taking a gts.Surface() instance, which must not be modified during instance lifetime. The optional noPad can disable padding (if set to True), which speeds up calls several times. Note: padding checks inclusion of 6 points along +- cardinal directions in the pad distance from given point, which is not exact.

surf
The associated gts.Surface object.

class yade._packPredicates.inHyperboloid(*inherits Predicate***)**
Hyperboloid predicate

___init___(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*, *(float)skirt***)** → None
Ctor taking centers of the lateral walls (as 3-tuples), radius at bases and skirt (middle radius).

class yade._packPredicates.inSphere(*inherits Predicate***)**
Sphere predicate.

___init___(*(Vector3)center*, *(float)radius***)** → None
Ctor taking center (as a 3-tuple) and radius

class yade._packPredicates.notInNotch(*inherits Predicate***)**
Outside of infinite, rectangle-shaped notch predicate

___init___(*(Vector3)centerPoint*, *(Vector3)edge*, *(Vector3)normal*, *(float)aperture***)** → None
Ctor taking point in the symmetry plane, vector pointing along the edge, plane normal and aperture size. The side inside the notch is edge×normal. Normal is made perpendicular to the edge. All vectors are normalized at construction time.

Computation of oriented bounding box for cloud of points.

yade._packObb.cloudBestFitOBB(*(tuple)arg1***)** → tuple
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) of best-fit oriented bounding-box for given tuple of points (uses brute-force velome minimization, do not use for very large clouds).

class yade._packSpherePadder.SpherePadder
Geometrical algorithm for filling tetrahedral mesh with spheres; the algorithm was designed by Jean-François Jerier and is described in [24].

___init___(*(str)fileName*[, *(str)meshType=''*]**)** → None
Initialize using tetrahedral mesh stored in *fileName*. Type of file is determined by extension: .gmsh implies *meshType*='GMSH', .inp implies *meshType*='INP'. If the extension is different, specify *meshType* explicitly. Possible values are 'GMSH' and 'INP'.

asSpherePack() → SpherePack

densify() → None

insert_sphere(*(float)arg2*, *(float)arg3*, *(float)arg4*, *(float)arg5***)** → None

maxNumberOfSpheres

maxOverlapRate

maxSolidFractioninProbe

meanSolidFraction

numberOfSpheres
pad_5() → None
place_virtual_spheres() → None
radiusRange
radiusRatio
save_mgpost(*(str)arg2***)** → None
setRadiusRatio(*(float)arg2*, *(float)arg3***)** → None
Like radiusRatio, but taking 2nd parameter.
virtualRadiusFactor

D.5. yade.plot module

Module containing utility functions for plotting inside yade. See scripts/simple-scene-plot.py or examples/concrete/uniax.py for example of usage.

yade.plot.addData(**d_in*, ***kw***)**
Add data from arguments name1=value1,name2=value2 to yade.plot.data. (the old {'name1':value1,'name2':value2} is deprecated, but still supported)
New data will be left-padded with nan's, unspecified data will be nan. This way, equal length of all data is assured so that they can be plotted one against any other.
Nan's don't appear in graphs.

yade.plot.data
Global dictionary containing all data values, common for all plots, in the form {'name':[value,...],...}. Data should be added using plot.addData function. All [value,...] columns have the same length, they are padded with NaN if unspecified.

yade.plot.labels
Dictionary converting names in data to human-readable names (TeX names, for instance); if a variable is not specified, it is left untranslated.

yade.plot.plot(*noShow=False***)**
Do the actual plot, which is either shown on screen (and nothing is returned: if *noShow* is False) or returned as object (if *noShow* is True).
You can use

```
>>> from yade import plot
>>> plot.plot(noShow=True).saveFig('someFile.pdf')
```

to save the figure to file automatically.

yade.plot.plots
dictionary x-name -> (yspec,...), where yspec is either y-name or (y-name,'line-specification')

yade.plot.reset()
Reset all plot-related variables (data, plots, labels)

yade.plot.resetData()
Reset all plot data; keep plots and labels intact.

yade.plot.reverseData()
Reverse yade.plot.data order.
Useful for tension-compression test, where the initial (zero) state is loaded and, to make data continuous, last part must *end* in the zero state.

yade.plot.saveGnuplot(*baseName*, *term='wxt'*, *extension=None*, *timestamp=False*, *comment=None*, *title=None*, *varData=False***)**
Save data added with plot.addData into (compressed) file and create .gnuplot file that attempts to mimick plots specified with plot.plots.

Parameters

baseName: used for creating baseName.gnuplot (command file for gnuplot), associated baseName.data (data) and output files (if applicable) in the form baseName.[plot number].extension

term: specify the gnuplot terminal; defaults to x11, in which case gnuplot will draw persistent windows to screen and terminate; other useful terminals are 'png', 'cairopdf' and so on

extension: defaults to terminal name; fine for png for example; if you use 'cairopdf', you should also say extension='pdf' however

timestamp: append numeric time to the basename

varData: whether file to plot will be declared as variable or be in-place in the plot expression

comment: a user comment (may be multiline) that will be embedded in the control file

Returns name fo the gnuplot file created.

yade.plot.splitData()

Make all plots discontinuous at this point (adds nan's to all data fields)

D.6. yade.post2d module

Module for 2d postprocessing, containing classes to project points from 3d to 2d in various ways, providing basic but flexible framework for extracting arbitrary scalar values from bodies and plotting the results. There are 2 basic components: flatteners and extractors.

D.6.1. Flatteners

Instance of classes that convert 3d (model) coordinates to 2d (plot) coordinates. Their interface is defined by the Flatten class (__call__, planar, normal).

D.6.2. Extractors

Callable objects returning scalar or vector value, given a body object. If a 3d vector is returned, Flattener.planar is called, which should return only in-plane components of the vector.

D.6.3. Example

This example can be found in examples/concrete/uniax-post.py

```
from yade import post2d
import pylab # the matlab-like interface of matplotlib

O.load('/tmp/uniax-tension.xml.bz2')

# flattener that project to the xz plane
flattener=post2d.AxisFlatten(useRef=False,axis=1)
# return scalar given a Body instance
extractDmg=lambda b: b.state.normDmg
# will call flattener.planar implicitly
# the same as: extractVelocity=lambda b: flattener.planar(b,b.state.vel)
extractVelocity=lambda b: b.state.vel

# create new figure
pylab.figure()
# plot raw damage
post2d.plot(post2d.data(extractDmg,flattener))

# plot smooth damage into new figure
pylab.figure(); ax,map=post2d.plot(post2d.data(extractDmg,flattener,stDev=2e-3))
# show color scale
pylab.colorbar(map,orientation='horizontal')

# raw velocity (vector field) plot
pylab.figure(); post2d.plot(post2d.data(extractVelocity,flattener))

# smooth velocity plot; data are sampled at regular grid
pylab.figure(); ax,map=post2d.plot(post2d.data(extractVelocity,flattener,stDev=1e-3))
# save last (current) figure to file
pylab.gcf().savefig('/tmp/foo.png')
```

```
# show the figures
pylab.show()
```

class yade.post2d.AxisFlatten(*inherits Flatten***)**

__init__()
:parameters: 'useRef': bool use reference positions rather than actual positions. 'axis': {0,1,2} axis normal to the plane; the return value will be simply position with this component dropped.

normal()

planar()

class yade.post2d.CylinderFlatten(*inherits Flatten***)**
Class for converting 3d point to 2d based on projection from circle. The y-axis in the projection corresponds to the rotation axis; the x-axis is distance form the axis.

__init__()
:param useRef: (bool) use reference positions rather than actual positions :param axis: axis of the cylinder, ☒{0,1,2}

normal()

planar()

class yade.post2d.Flatten
Abstract class for converting 3d point into 2d. Used by post2d.data2d.

normal()
Given position and vector value, return lenght of the vector normal to the flat plane.

planar()
Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

class yade.post2d.SpiralFlatten(*inherits Flatten***)**
Class converting 3d point to 2d based on projection from spiral. The y-axis in the projection corresponds to the rotation axis

__init__()
:parameters: 'useRef': bool use reference positions rather than actual positions 'thetaRange': (thetaMin,thetaMax) tuple bodies outside this range will be discarded 'dH_dTheta':float inclination of the spiral (per radian) 'axis': {0,1,2} axis of rotation of the spiral 'periodStart': float height of the spiral for zero angle

normal()

planar()

yade.post2d.data(*extractor*, *flattener*, *onlyDynamic=True*, *stDev=None*, *relThreshold=3.0*, *div=(50, 50)*, *margin=(0, 0)***)**
Filter all bodies (spheres only), project them to 2d and extract required scalar value; return either discrete array of positions and values, or smoothed data, depending on whether the stDev value is specified.

Parameters

***extractor*: callable** receives Body instance, should return scalar, a 2-tuple (vector fields) or None (to skip that body)

***flattener*: callable** receives Body instance and returns its 2d coordinates or None (to skip that body)

***onlyDynamic*: bool** skip all non-dynamic bodies

stDev*: float or *None standard deviation for averaging, enables smoothing; None (default) means raw mode.

***relThreshold*: float** threshold for the gaussian weight function relative to stDev (smooth mode only)

***div*: (int,int)** number of cells for the gaussian grid (smooth mode only)

***margin*: (float,float)** margin around bounding box for data (smooth mode only)

Returns dictionary

Returned dictionary always containing keys 'type' (one of 'rawScalar','rawVector','smoothScalar','smoothVector', depending on value of smooth and on return value from extractor), 'x', 'y', 'bbox'.
Raw data further contains 'radii'.
Scalar fields contain 'val' (value from *extractor*), vector fields have 'valX' and 'valY' (2 components returned by the *extractor*).

yade.post2d.plot(*data*, *axes=None*, *alpha=0.5*, *clabel=True*, ***kw***)**

Given output from post2d.data, plot the scalar as discrete or smooth plot.

For raw discrete data, plot filled circles with radii of particles, colored by the scalar value.

For smooth discrete data, plot image with optional contours and contour labels.

For vector data (raw or smooth), plot quiver (vector field), with arrows colored by the magnitude.

Parameters

***axes*: matplotlib.axes instance** axes where the figure will be plotted; if None, will be created from scratch.

***data*:** value returned by post2d.data

***clable*: bool** show contour labels (smooth mode only)

Returns tuple of (axes,mappable); mappable can be used in further calls to pylab.colorbar.

D.7. yade.qt module

Access/manipulation of the qt3-based yade gui.

yade.qt.Controller() → None

Start simulation controller

yade.qt.Generator() → None

Start simulation generator (preprocessor interface)

yade.qt.Renderer() → OpenGLRenderingEngine

Return wrapped OpenGLRenderingEngine; the renderer is constructed if it doesn't exist yet.

yade.qt.View() → GLViewer

Create new 3d view.

yade.qt.activate() → None

Attempt to activate the Qt GUI.

yade.qt.center() → None

Center all views.

yade.qt.close() → None

Close all open qt windows.

yade.qt.isActive() → bool

Whether the Qt GUI is being used.

yade.qt.makePlayerVideo(**args*, ***kw***)**

yade.qt.makeSimulationVideo(*output*, *realPeriod=1*, *virtPeriod=0*, *iterPeriod=0*, *viewNo=0*, *fps=24*, *msecSleep=0***)**

Create video by running simulation. SnapshotEngine is added (and removed once done), temporary files are deleted. The video is theora-encoded in the ogg container. Periodicity is controlled in the same way as for PeriodicEngine (SnapshotEngine is a PeriodicEngine and realPeriod, virtPeriod and iterPeriod are passed to the new SnapshotEngine).

viewNo is 0-based GL view number. 0 is the primary view and will be created if it doesn't exist. It is an error if viewNo>0 and the view doesn't exist.

The simulation will run until it stops by itself. Either set Omega().stopAtIter or have an engine that will call Omega().pause() at some point.

See makePlayerVideo for more documentation.

yade.qt.views() → list

Return list of all open qt.GLViewer objects

yade._qt.Controller() → None

Start simulation controller

class yade._qt.GLViewer

__init__() → None

__init__((int)arg2) → None

axes

Show arrows for axes.

center([*(bool)median=True*]**)** → None

Center view. View is centered either so that all bodies fit inside (*median*=False), or so that 75% of bodies fit inside (*median*=True).

eyePosition

Camera position.

fitAABB(*(Vector3)mn*, *(Vector3)mx***)** → None
Adjust scene bounds so that Axis-aligned bounding box given by its lower and upper corners *mn*, *mx* fits in.

fitSphere(*(Vector3)center*, *(float)radius***)** → None
Adjust scene bounds so that sphere given by *center* and *radius* fits in.

fps
Show frames per second indicator.

grid
Display square grid in zero planes, as 3-tuple of bools for yz, xz, xy planes.

loadState(*(int)slot***)** → None
Load display parameters from slot saved previously into, identified by its number.

lookAt
Point at which camera is directed.

ortho
Whether orthographic projection is used; if false, use perspective projection.

saveState(*(int)slot***)** → None
Save display parameters into numbered memory slot. Saves state for both GLViewer and associated OpenGLRenderingEngine.

scale
Scale of the view (?)

sceneRadius
Visible scene radius.

screenSize
Size of the viewer's window, in scree pixels

showEntireScene() → None

timeDisp
Time displayed on in the vindow; is a string composed of characters *r*, *v*, *i* standing respectively for real time, virtual time, iteration number.

upVector
Vector that will be shown oriented up on the screen.

viewDir
Camera orientation (as vector).

yade._qt.Generator() → None
Start simulation generator (preprocessor interface)

yade._qt.Renderer() → OpenGLRenderingEngine
Return wrapped OpenGLRenderingEngine; the renderer is constructed if it doesn't exist yet.

yade._qt.View() → GLViewer
Create new 3d view.

yade._qt.activate() → None
Attempt to activate the Qt GUI.

yade._qt.center() → None
Center all views.

yade._qt.close() → None
Close all open qt windows.

yade._qt.isActive() → bool
Whether the Qt GUI is being used.

yade._qt.views() → list
Return list of all open qt.GLViewer objects

D.8. yade.timing module

Functions for accessing timing information stored in engines and functors.

See *Timing* section of the programmer's manual, wiki page for some examples.

yade.timing.reset()
Zero all timing data.

yade.timing.stats()
Print summary table of timing information from engines and functors. Absolute times as well as

percentages are given. Sample output:

```
Name                                    Count                  Time        Rel. time
-------------------------------------------------------------------------------------
ForceResetter                             400                  9449µs            0.01%
BoundingVolumeMetaEngine                  400               1171770µs            1.15%
PersistentSAPCollider                     400               9433093µs            9.24%
InteractionGeometryMetaEngine             400              15177607µs           14.87%
InteractionPhysicsMetaEngine              400               9518738µs            9.33%
ConstitutiveLawDispatcher                 400              64810867µs           63.49%
  ef2_Spheres_Brefcom_BrefcomLaw
         setup                                 4926145        7649131µs           15.25%
         geom                                  4926145       23216292µs           46.28%
         material                              4926145        8595686µs           17.14%
         rest                                  4926145       10700007µs           21.33%
         TOTAL                                               50161117µs          100.00%
"damper"                                  400               1866816µs            1.83%
"strainer"                                400                 21589µs            0.02%
"plotDataCollector"                       160                 64284µs            0.06%
"damageChecker"                             9                  3272µs            0.00%
TOTAL                                                      102077490µs          100.00%
```

D.9. yade.utils module

Heap of functions that don't (yet) fit anywhere else.

Devs: please DO NOT ADD more functions here, it is getting too crowded!

yade.utils.NormalRestitution2DampingRate(*en*)

Compute the normal damping rate as a function of the normal coefficient of restitution e_n. For $e_n \in \langle 0, 1 \rangle$ damping rate equals

$$-\frac{\log e_n}{\sqrt{e_n^2 + \pi^2}}$$

yade.utils.PWaveTimeStep() → float

Get timestep accoring to the velocity of P-Wave propagation; computed from sphere radii, rigidities and masses.

yade.utils.SpherePWaveTimeStep(*radius*, *density*, *young*)

Compute P-wave critical timestep for a single (presumably representative) sphere, using formula for P-Wave propagation speed $\Delta t_c = \frac{r}{\sqrt{E/\rho}}$. If you want to compute minimum critical timestep for all spheres in the simulation, use utils.PWaveTimeStep instead.

```
>>> SpherePWaveTimeStep(1e-3,2400,30e9)
2.8284271247461903e-07
```

class **yade.utils.TableParamReader**

Class for reading simulation parameters from text file.

Each parameter is represented by one column, each parameter set by one line. Colums are separated by blanks (no quoting).

First non-empty line contains column titles (without quotes). You may use special column named 'description' to describe this parameter set; if such colum is absent, description will be built by concatenating column names and corresponding values (**param1=34,param2=12.22,param4=foo**)

- from columns ending in **!** (the **!** is not included in the column name)
- from all columns, if no columns end in **!**.

Empty lines within the file are ignored (although counted); **#** starts comment till the end of line. Number of blank-separated columns must be the same for all non-empty lines.

A special value **=** can be used instead of parameter value; value from the previous non-empty line will be used instead (works recursively).

This class is used by utils.readParamsFromTable.

__init__()

Setup the reader class, read data into memory.

paramDict()

Return dictionary containing data from file given to constructor. Keys are line numbers (which might be non-contiguous and refer to real line numbers that one can see in text editors), values are dictionaries mapping parameter names to their values given in the file. The special value '=' has already been interpreted, **!** (bangs) (if any) were already removed from column titles, **description** column has already been added (if absent).

yade.utils.aabbDim(*cutoff=0.0*, *centers=False***)**

Return dimensions of the axis-aligned bounding box, optionally with relative part *cutoff* cut away.

yade.utils.aabbExtrema([*(float)cutoff=0.0*[, *(bool)centers=False*]]**)** → tuple

Return coordinates of box enclosing all bodies

Parameters

- **centers** (*bool*) – do not take sphere radii in account, only their centroids
- **cutoff** (*float*☒⟨*0...1*⟩) – relative dimension by which the box will be cut away at its boundaries.

Returns (lower corner, upper corner) as (Vector3,Vector3)

yade.utils.aabbExtrema2d(*pts***)**

Return 2d bounding box for a sequence of 2-tuples.

yade.utils.aabbWalls(*extrema=None*, *thickness=None*, *oversizeFactor=1.5*, ***kw***)**

Return 6 boxes that will wrap existing packing as walls from all sides; extrema are extremal points of the Aabb of the packing (will be calculated if not specified) thickness is wall thickness (will be 1/10 of the X-dimension if not specified) Walls will be enlarged in their plane by oversizeFactor. returns list of 6 wall Bodies enclosing the packing, in the order minX,maxX,minY,maxY,minZ,maxZ.

yade.utils.approxSectionArea(*(float)arg1*, *(int)arg2***)** → float

Compute area of convex hull when when taking (swept) spheres crossing the plane at coord, perpendicular to axis.

yade.utils.avgNumInteractions(*cutoff=0.0***)**

Return average numer of interactions per particle, also known as *coordination number*.

Parameters

- **cutoff** – cut some relative part of the sample's bounding box away.

yade.utils.bodyNumInteractionsHistogram([*(tuple)aabb*]**)** → tuple

yade.utils.box(*center*, *extents*, *orientation=*[, *1*, *0*, *0*, *0*], *dynamic=True*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1***)**

Create box (cuboid) with given parameters.

Parameters

***extents*: Vector3** half-sizes along x,y,z axes

See utils.sphere's documentation for meaning of other parameters.

yade.utils.coordsAndDisplacements(*(int)axis*[, *(tuple)Aabb=()*]**)** → tuple

Return tuple of 2 same-length lists for coordinates and displacements (coordinate minus reference coordinate) along given axis (1st arg); if the Aabb=((x_min,y_min,z_min),(x_max,y_max,z_max)) box is given, only bodies within this box will be considered.

yade.utils.createInteraction(*(int)id1*, *(int)id2***)** → Interaction

Create interaction between given bodies by hand.

Current engines are searched for InteractionGeometryDispatcher and InteractionPhysicsDispatcher (might be both hidden in InteractionDispatchers). Geometry is created using **force** parameter of the geometry dispatcher, wherefore the interaction will exist even if bodies do not spatially overlap and the functor would return **false** under normal circumstances.

This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

yade.utils.defaultMaterial()

Return default material, when creating bodies with utils.sphere and friends, material is unspecified and there is no shared material defined yet. By default, this function returns:
FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5,label='defaultMat')

yade.utils.downCast(*obj*, *newClassName***)**

Cast given object to class deriving from the same yade root class and copy all parameters from given object. Obj should be up in the inheritance tree, otherwise some attributes may not be defined in the new class.

yade.utils.elasticEnergy(*(tuple)arg1***)** → float

yade.utils.encodeVideoFromFrames(*frameSpec*, *out*, *renameNotOverwrite=True*, *fps=24***)**

Create .ogg video from external image files.

Parameters

***frameSpec*: wildcard | sequence of filenames** If string, wildcard in format understood by GStreamer's multifilesrc plugin (e.g. '/tmp/frame-%04d.png'). If list or tuple, filenames to be encoded in given order.

> **Warning:** GStreamer is picky about the wildcard; if you pass a wrong one, if will not complain, but silently stall.

***out*: filename** file to save video into

***renameNotOverwrite*: bool** if True, existing same-named video file will have ~[number] appended; will be overwritten otherwise.

***fps*:** Frames per second.

yade.utils.facet(*vertices*, *dynamic=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1***)**

Create facet with given parameters.

Parameters

***vertices*: [Vector3,Vector3,Vector3]** coordinates of vertices in the global coordinate system.

***wire*: bool** if True, facets are shown as skeleton; otherwise facets are filled

***noBound*:** do not assign Body().bound

***color*: Vector3 or None** random color will be assigned if None

***material*: int | string | Material instance | callable returning Material instance**

- if int, O.materials[material] will be used; as a special case, if material==-1 and there is no shared materials defined, utils.defaultMaterial() will be assigned to O.materials[0]
- if string, it is label of an existing material that will be used
- if Material instance, this instance will be used
- if callable, it will be called without arguments; returned Material value will be used (Material factory object, if you like)

***mask*: integer** Body.mask for the body

See utils.sphere's documentation for meaning of other parameters.

yade.utils.facetBox(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw***)**

Create arbitrarily-aligned box composed of facets, with given center, extents and orientation. If any of the box dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the box.

Parameters

***center*: Vector3** center of the created box

***extents*: (eX,eY,eZ)** lengths of the box sides

***orientation*: quaternion** orientation of the box

***wallMask*: bitmask** determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls;

*****kw*: (unused keyword arguments)** passed to utils.facet

Returns list of facets forming the box.

yade.utils.facetCylinder(*center*, *radius*, *height*, *orientation=*[*1, 0, 0, 0*]*, segmentsNumber=10*, *wallMask=7*, *closed=1*, ***kw***)**

Create arbitrarily-aligned cylinder composed of facets, with given center, radius, height and orientation. Return List of facets forming the cylinder;

Parameters

***center*: Vector3** center of the created cylinder

***radius*: float** cylinder radius

***height*: float** cylinder height

***orientation*: Quaternion** orientation of the cylinder

***segmentsNumber*: int** number of edges on the cylinder surface (>=5)

***wallMask*: bitmask** determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls;

*****kw*: (unused keyword arguments)** passed to utils.facet;

yade.utils.flipCell([*(Matrix3)flip=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]**)** → Matrix3

Flip periodic cell so that angles between R^3 axes and transformed axes are as small as possible. This function relies on the fact that periodic cell defines by repetition or its corners regular grid of points in R^3; however, all cells generating identical grid are equivalent and can be flipped one over another. This necessiatates adjustment of Interaction.cellDist for interactions that cross boundary and didn't before (or vice versa), and re-initialization of collider. The *flip* argument can be used to specify desired flip: integers, each column for one axis; if zero matrix, best fit (minimizing the angles) is computed automatically.

In c++, this function is accessible as **Shop::flipCell**.

This function is currently broken and should not be used.

yade.utils.forcesOnCoordPlane(*(float)arg1*, *(int)arg2***)** → Vector3

yade.utils.forcesOnPlane(*(Vector3)planePt*, *(Vector3)normal***)** → Vector3

Find all interactions deriving from NormShearPhys that cross given plane and sum forces (both normal and shear) on them.

Parameters

- **planePt** (*Vector3*) – a point on the plane
- **normal** (*Vector3*) – plane normal (will be normalized).

yade.utils.fractionalBox(*fraction=1.0*, *minMax=None***)**

retrurn (min,max) that is the original minMax box (or aabb of the whole simulation if not specified) linearly scaled around its center to the fraction factor

yade.utils.getSpheresVolume() → float

Compute the total volume of spheres in the simulation (might crash for now if dynamic bodies are not spheres)

yade.utils.getViscoelasticFromSpheresInteraction(*(float)m*, *(float)tc*, *(float)en*, *(float)es***)** → dict

Get viscoelastic interaction parameters from analytical solution of a pair spheres collision problem.

Parameters

'm' : float sphere mass 'tc' : float collision time 'en' : float normal restitution coefficient 'es' : float tangential restitution coefficient.

Returns

dict with keys:

kn : float normal elastic coefficient computed as:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$

cn : float normal viscous coefficient computed as:

$$c_n = -\frac{2m}{t_c}\ln e_n$$

kt : float tangential elastic coefficient computed as:

$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$

ct : float tangential viscous coefficient computed as:

$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t.$$

For details see [48].

yade.utils.highlightNone() → None

Reset highlight on all bodies.

yade.utils.inscribedCircleCenter(*(Vector3)v1*, *(Vector3)v2*, *(Vector3)v3***)** → Vector3

Return center of inscribed circle for triangle given by its vertices *v1*, *v2*, *v3*.

yade.utils.interactionAnglesHistogram(*(int)axis*[, *(int)mask*[, *(int)bins*[, *(tuple)aabb*]]]) → tuple

yade.utils.kineticEnergy() → float

Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i v_i^2 + \omega(I\omega^T)\right).$$

No transformation of inertia tensor (in local frame) I is done, although it is multiplied by angular velocity ω (in global frame); the value will not be accurate for aspherical particles.

yade.utils.loadVars(*mark=None***)**

Load variables from saveVars, which are saved inside the simulation. If mark==None, all save variables are loaded. Otherwise only those with the mark passed.

yade.utils.negPosExtremeIds(*(int)axis*[, *(float)distFactor*]) → tuple

Return list of ids for spheres (only) that are on extremal ends of the specimen along given axis; distFactor multiplies their radius so that sphere that do not touch the boundary coordinate can also be returned.

yade.utils.perpendicularArea(*axis***)**

Return area perpendicular to given axis (0=x,1=y,2=z) generated by bodies for which the function consider returns True (defaults to returning True always) and which is of the type Sphere.

yade.utils.plotDirections(*aabb=()*, *mask=0*, *bins=20*, *numHist=True***)**

Plot 3 histograms for distribution of interaction directions, in yz,xz and xy planes and (optional but default) histogram of number of interactions per body.

yade.utils.plotNumInteractionsHistogram(*cutoff=0.0***)**

Plot histogram with number of interactions per body, optionally cutting away *cutoff* relative axis-aligned box from specimen margin.

yade.utils.pointInsidePolygon(*(tuple)arg1*, *(object)arg2***)** → bool

yade.utils.ptInAABB(*(tuple)arg1*, *(tuple)arg2*, *(tuple)arg3***)** → bool

Return True/False whether the point (3-tuple) p is within box given by its min (3-tuple) and max (3-tuple) corners

yade.utils.randomColor()

Return random Vector3 with each component in interval 0...1 (uniform distribution)

yade.utils.randomizeColors(*onlyDynamic=False***)**

Assign random colors to Shape::color.

If onlyDynamic is true, only dynamic bodies will have the color changed.

yade.utils.readParamsFromTable(*tableFileLine=None*, *noTableOk=False*, *unknownOk=False*, ***kw***)**

Read parameters from a file and assign them to __builtin__ variables.

The format of the file is as follows (commens starting with # and empty lines allowed):

```
# commented lines allowed anywhere
name1 name2 … # first non-blank line are column headings
                        # empty line is OK, with or without comment
val1  val2  … # 1st parameter set
val2  val2  … # 2nd
…
```

Assigned tags:

- *description* column is assigned to Omega().tags['description']; this column is synthesized if absent (see utils.TableParamReader)
- Omega().tags['params']="name1=val1,name2=val2,..."
- Omega().tags['defaultParams']="unassignedName1=defaultValue1,..."

All parameters (default as well as settable) are saved using saveVars('table').

Parameters

***tableFile*:** text file (with one value per blank-separated columns)

***tableLine*:** number of line where to get the values from.

***noTableOk*: bool** do not raise exception if the file cannot be open; use default values

***unknownOk*: bool** do not raise exception if unknown column name is found in the file; assign it as well

Returns number of assigned parameters.

yade.utils.replaceCollider(*colliderEngine***)**

Replaces collider (Collider) engine with the engine supplied. Raises error if no collider is in engines.

yade.utils.saveVars(*mark=''*, *loadNow=False*, ***kw***)**

Save passed variables into the simulation so that it can be recovered when the simulation is loaded again.

For example, variables a=5, b=66 and c=7.5e-4 are defined. To save those, use:

```
>>> from yade import utils
>>> utils.saveVars('mark',a=1,b=2,c=3,loadNow=True)
>>> a,b,c
(1, 2, 3)
```

those variables will be save in the .xml file, when the simulation itself is saved. To recover those variables once the .xml is loaded again, use

```
>>> utils.loadVars('mark')
```

and they will be defined in the __builtin__ namespace (i.e. available from anywhere in the python code).

If *loadParam*==True, variables will be loaded immediately after saving. That effectively makes ***kw* available in builtin namespace.

yade.utils.scalarOnColorScale(*(float)arg1*, *(float)arg2*, *(float)arg3***)** → Vector3

yade.utils.setRefSe3() → None

Set reference positions and orientations of all bodies equal to their current positions and orientations.

yade.utils.sphere(*center*, *radius*, *dynamic=True*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1***)**

Create sphere with given parameters; mass and inertia computed automatically.

Last assigned material is used by default (*material*=-1), and utils.defaultMaterial() will be used if no material is defined at all.

Parameters

***center*: Vector3** center

***radius*: float** radius

***color*: Vector3 or None** random color will be assigned if None

***material*: int | string | Material instance | callable returning Material instance**

- if int, O.materials[material] will be used; as a special case, if material==-1 and there is no shared materials defined, utils.defaultMaterial() will be assigned to O.materials[0]
- if string, it is label of an existing material that will be used
- if Material instance, this instance will be used
- if callable, it will be called without arguments; returned Material value will be used (Material factory object, if you like)

***mask*: integer** Body.mask for the body

Returns A Body instance with desired characteristics.

Creating default shared material if none exists neither is given:

```
>>> O.reset()
>>> from yade import utils
>>> len(O.materials)
0
>>> s0=utils.sphere([2,0,0],1)
>>> len(O.materials)
1
```

Instance of material can be given:

```
>>> s1=utils.sphere([0,0,0],1,wire=False,color=(0,1,0),material=ElastMat(young=30e9,density=2e3))
>>> s1.shape.wire
False
>>> s1.shape.color
Vector3(0,1,0)
>>> s1.mat.density
2000.0
```

Material can be given by label:

```
>>> O.materials.append(FrictMat(young=10e9,poisson=.11,label='myMaterial'))
1
>>> s2=utils.sphere([0,0,2],1,material='myMaterial')
>>> s2.mat.label
'myMaterial'
>>> s2.mat.poisson
0.11
```

Finally, material can be a callable object (taking no arguments), which returns a Material instance. Use this if you don't call this function directly (for instance, through yade.pack.randomDensePack), passing only 1 *material* parameter, but you don't want material to be shared.
For instance, randomized material properties can be created like this:

```
>>> import random
>>> def matFactory(): return ElastMat(young=1e10*random.random(),density=1e3+1e3*random.random())
...
>>> s3=utils.sphere([0,2,0],1,material=matFactory)
>>> s4=utils.sphere([1,2,0],1,material=matFactory)
```

yade.utils.spiralProject(*(Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]) → tuple

yade.utils.sumFacetNormalForces(*(object)ids*[, *(int)axis=-1*]) → float

Sum force magnitudes on given bodies (must have shape of the Facet type), considering only part of forces perpendicular to each facet's face; if *axis* has positive value, then the specified axis (0=x, 1=y, 2=z) will be used instead of facet's normals.

yade.utils.sumForces(*(tuple)ids*, *(Vector3)direction*) → float

Return summary force on bodies with given *ids*, projected on the *direction* vector.

yade.utils.sumTorques(*(tuple)ids*, *(Vector3)axis*, *(Vector3)axisPt*) → float

Sum forces and torques on bodies given in *ids* with respect to axis specified by a point *axisPt* and its direction *axis*.

yade.utils.totalForceInVolume() → tuple

Return summed forces on all interactions and average isotropic stiffness, as tuple (Vector3,float)

yade.utils.typedEngine(*name*)

Return first engine from current O.engines, identified by its type (as string). For example:

```
>>> from yade import utils
>>> O.engines=[InsertionSortCollider(),NewtonIntegrator(),GravityEngine()]
>>> utils.typedEngine("NewtonIntegrator") == O.engines[1]
True
```

yade.utils.unbalancedForce([*(bool)useMaxForce=False*]) → float

Compute the ratio of mean (or maximum, if *useMaxForce*) summary force on bodies and maximum force magnitude on interactions. For perfectly static equilibrium, summary force on all bodies is zero (since forces from interactions cancel out and induce no acceleration of particles); this ratio will tend to zero as simulation stabilizes, though zero is never reached because of finite precision computation. Sufficiently small value can be e.g. 1e-2 or smaller, depending on how much equilibrium it should be.

yade.utils.uniaxialTestFeatures(*filename=None*, *areaSections=10*, *axis=-1*, ***kw*)

Get some data about the current packing useful for uniaxial test:

1. Find the dimensions that is the longest (uniaxial loading axis)
2. Find the minimum cross-section area of the specimen by examining several (areaSections) sections perpendicular to axis, computing area of the convex hull for each one. This will work also for non-prismatic specimen.
3. Find the bodies that are on the negative/positive boundary, to which the straining condition should be applied.

Parameters

***filename*:** if given, spheres will be loaded from this file (ASCII format); if not, current simulation will be used.

***areaSection*:** number of section that will be used to estimate cross-section

***axis*:** if given, force strained axis, rather than computing it from predominant length

Returns dictionary with keys 'negIds', 'posIds', 'axis', 'area'.

> **Warning:** The function utils.approxSectionArea uses convex hull algorithm to find the area, but the implementation is reported to be *buggy* (bot works in some cases). Always check this number, or fix the convex hull algorithm (it is documented in the source, see py/_utils.cpp).

yade.utils.vmData()
Return memory usage data from Linux's /proc/[pid]/status, line VmData.

yade.utils.wall(*position*, *axis*, *sense=0*, *color=None*, *material=-1*, *mask=1***)**
Return ready-made wall body.

Parameters

***position*: float or Vector3** center of the wall. If float, it is the position along given axis, the other 2 components being zero

***axis*: {0,1,2}** orientation of the wall normal (0,1,2) for x,y,z (sc. planes yz, xz, xy)

***sense*: {-1,0,1}** sense in which to interact (0: both, -1: negative, +1: positive; see Wall reference documentation)

***mask*: bitmask (as int)** Body.mask

See utils.sphere's documentation for meaning of other parameters.

yade.utils.wireAll() → None
Set Shape::wire on all bodies to True, rendering them with wireframe only.

yade.utils.wireNoSpheres() → None
Set Shape::wire to True on non-spherical bodies (Facets, Walls).

yade.utils.wireNone() → None
Set Shape::wire on all bodies to False, rendering them as solids.

yade.utils.xMirror(*half***)**
Mirror a sequence of 2d points around the x axis (changing sign on the y coord). The sequence should start up and then it will wrap from y downwards (or vice versa). If the last point's x coord is zero, it will not be duplicated.

yade._utils.PWaveTimeStep() → float
Get timestep accoring to the velocity of P-Wave propagation; computed from sphere radii, rigidities and masses.

yade._utils.aabbExtrema([*(float)cutoff=0.0*[, *(bool)centers=False*]]**)** → tuple
Return coordinates of box enclosing all bodies

Parameters

- **centers** (*bool*) – do not take sphere radii in account, only their centroids
- **cutoff** (*float*☒⟨*0...1*⟩) – relative dimension by which the box will be cut away at its boundaries.

Returns (lower corner, upper corner) as (Vector3,Vector3)

yade._utils.approxSectionArea(*(float)arg1*, *(int)arg2***)** → float
Compute area of convex hull when when taking (swept) spheres crossing the plane at coord, perpendicular to axis.

yade._utils.bodyNumInteractionsHistogram([*(tuple)aabb*]**)** → tuple

yade._utils.coordsAndDisplacements(*(int)axis*[, *(tuple)Aabb=()*]**)** → tuple
Return tuple of 2 same-length lists for coordinates and displacements (coordinate minus reference coordinate) along given axis (1st arg); if the Aabb=((x_min,y_min,z_min),(x_max,y_max,z_max)) box is given, only bodies within this box will be considered.

yade._utils.createInteraction(*(int)id1*, *(int)id2***)** → Interaction
Create interaction between given bodies by hand.
Current engines are searched for InteractionGeometryDispatcher and InteractionPhysicsDispatcher (might be both hidden in InteractionDispatchers). Geometry is created using **force** parameter of the geometry dispatcher, wherefore the interaction will exist even if bodies do not spatially overlap and the functor would return **false** under normal circumstances.
This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

yade._utils.elasticEnergy(*(tuple)arg1***)** → float

yade._utils.flipCell([*(Matrix3)flip=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]**)** → Matrix3
Flip periodic cell so that angles between R^3 axes and transformed axes are as small as possible. This function relies on the fact that periodic cell defines by repetition or its corners regular grid of

points in R^3; however, all cells generating identical grid are equivalent and can be flipped one over another. This necessiatates adjustment of Interaction.cellDist for interactions that cross boundary and didn't before (or vice versa), and re-initialization of collider. The *flip* argument can be used to specify desired flip: integers, each column for one axis; if zero matrix, best fit (minimizing the angles) is computed automatically.

In c++, this function is accessible as **Shop::flipCell**.

This function is currently broken and should not be used.

yade._utils.forcesOnCoordPlane(*(float)arg1*, *(int)arg2***)** → Vector3

yade._utils.forcesOnPlane(*(Vector3)planePt*, *(Vector3)normal***)** → Vector3

Find all interactions deriving from NormShearPhys that cross given plane and sum forces (both normal and shear) on them.

Parameters

- **planePt** (*Vector3*) – a point on the plane
- **normal** (*Vector3*) – plane normal (will be normalized).

yade._utils.getSpheresVolume() → float

Compute the total volume of spheres in the simulation (might crash for now if dynamic bodies are not spheres)

yade._utils.getViscoelasticFromSpheresInteraction(*(float)m*, *(float)tc*, *(float)en*, *(float)es***)** → dict

Get viscoelastic interaction parameters from analytical solution of a pair spheres collision problem.

Parameters

'm' : float sphere mass 'tc' : float collision time 'en' : float normal restitution coefficient 'es' : float tangential restitution coefficient.

Returns

dict with keys:

kn : float normal elastic coefficient computed as:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$

cn : float normal viscous coefficient computed as:

$$c_n = -\frac{2m}{t_c}\ln e_n$$

kt : float tangential elastic coefficient computed as:

$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$

ct : float tangential viscous coefficient computed as:

$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t.$$

For details see [48].

yade._utils.highlightNone() → None

Reset highlight on all bodies.

yade._utils.inscribedCircleCenter(*(Vector3)v1*, *(Vector3)v2*, *(Vector3)v3***)** → Vector3

Return center of inscribed circle for triangle given by its vertices *v1*, *v2*, *v3*.

yade._utils.interactionAnglesHistogram(*(int)axis*[, *(int)mask*[, *(int)bins*[, *(tuple)aabb*]]]**)** → tuple

yade._utils.kineticEnergy() → float

Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i v_i^2 + \omega(I\omega^T)\right).$$

No transformation of inertia tensor (in local frame) I is done, although it is multiplied by angular velocity ω (in global frame); the value will not be accurate for aspherical particles.

yade._utils.negPosExtremeIds(*(int)axis*[, *(float)distFactor*]**)** → tuple
Return list of ids for spheres (only) that are on extremal ends of the specimen along given axis; distFactor multiplies their radius so that sphere that do not touch the boundary coordinate can also be returned.

yade._utils.pointInsidePolygon(*(tuple)arg1*, *(object)arg2***)** → bool

yade._utils.ptInAABB(*(tuple)arg1*, *(tuple)arg2*, *(tuple)arg3***)** → bool
Return True/False whether the point (3-tuple) p is within box given by its min (3-tuple) and max (3-tuple) corners

yade._utils.scalarOnColorScale(*(float)arg1*, *(float)arg2*, *(float)arg3***)** → Vector3

yade._utils.setRefSe3() → None
Set reference positions and orientations of all bodies equal to their current positions and orientations.

yade._utils.spiralProject(*(Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]**)** → tuple

yade._utils.sumFacetNormalForces(*(object)ids*[, *(int)axis=-1*]**)** → float
Sum force magnitudes on given bodies (must have shape of the Facet type), considering only part of forces perpendicular to each facet's face; if *axis* has positive value, then the specified axis (0=x, 1=y, 2=z) will be used instead of facet's normals.

yade._utils.sumForces(*(tuple)ids*, *(Vector3)direction***)** → float
Return summary force on bodies with given *ids*, projected on the *direction* vector.

yade._utils.sumTorques(*(tuple)ids*, *(Vector3)axis*, *(Vector3)axisPt***)** → float
Sum forces and torques on bodies given in *ids* with respect to axis specified by a point *axisPt* and its direction *axis*.

yade._utils.totalForceInVolume() → tuple
Return summed forces on all interactions and average isotropic stiffness, as tuple (Vector3,float)

yade._utils.unbalancedForce([*(bool)useMaxForce=False*]**)** → float
Compute the ratio of mean (or maximum, if *useMaxForce*) summary force on bodies and maximum force magnitude on interactions. For perfectly static equilibrium, summary force on all bodies is zero (since forces from interactions cancel out and induce no acceleration of particles); this ratio will tend to zero as simulation stabilizes, though zero is never reached because of finite precision computation. Sufficiently small value can be e.g. 1e-2 or smaller, depending on how much equilibrium it should be.

yade._utils.wireAll() → None
Set Shape::wire on all bodies to True, rendering them with wireframe only.

yade._utils.wireNoSpheres() → None
Set Shape::wire to True on non-spherical bodies (Facets, Walls).

yade._utils.wireNone() → None
Set Shape::wire on all bodies to False, rendering them as solids.

D.10. yade.ymport module

Import geometry from various formats ('import' is python keyword, hence the name 'ymport').

yade.ymport.gengeo(*mntable*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw***)**
Imports geometry from LSMGenGeo library and creates spheres.

Parameters

***mntable*: mntable** object, which creates by LSMGenGeo library, see example

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.sphere

LSMGenGeo library allows to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/regular-sphere-pack/regular-sphere-pack.py, usage of LSMGenGeo library in scripts/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-

module.html

- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

yade.ymport.gengeoFile(*fileName='file.geo', shift=[0.0, 0.0, 0.0], scale=1.0, **kw***)**

Imports geometry from LSMGenGeo .geo file and creates spheres.

Parameters

***filename*: string** file which has 4 colums [x, y, z, radius].

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.sphere

Returns list of spheres.

LSMGenGeo library allows to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/regular-sphere-pack/regular-sphere-pack.py, usage of LSMGenGeo library in scripts/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

yade.ymport.gmsh(*meshfile='file.mesh', shift=[0.0, 0.0, 0.0], scale=1.0, orientation=Quaternion((1, 0, 0), 0), **kw***)**

Imports geometry from mesh file and creates facets.

Parameters

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

***orientation*: quaternion** orientation of the imported mesh

*****kw*: (unused keyword arguments)** is passed to utils.facet

Returns list of facets forming the specimen.

mesh files can be easily created with GMSH. Example added to examples/regular-sphere-pack/regular-sphere-pack.py

Additional examples of mesh-files can be downloaded from http://www-roc.inria.fr/gamma/download/download.php

yade.ymport.gts(*meshfile, shift=(0, 0, 0), scale=1.0, **kw***)**

Read given meshfile in gts format.

Parameters

***meshfile*: string** name of the input file.

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.facet

Returns list of facets.

yade.ymport.stl(*file, dynamic=False, wire=True, color=None, highlight=False, noBound=False, material=-1***)**

Import geometry from stl file, return list of created facets.

yade.ymport.text(*fileName, shift=[0.0, 0.0, 0.0], scale=1.0, **kw***)**

Load sphere coordinates from file, create spheres, insert them to the simulation.

Parameters

***filename*: string** file which has 4 colums [x, y, z, radius].

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.sphere

Returns list of spheres.

Lines starting with # are skipped

yade.ymport.textExt(*fileName, format='x_y_z_r', shift=[0.0, 0.0, 0.0], scale=1.0, **kw***)**

Load sphere coordinates from file in specific format, create spheres, insert them to the simulation.

Parameters *filename*: string *format*:

the name of output format. Supported *'x_y_z_r'(default), 'x_y_z_r_matId*

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

****kw*: (unused keyword arguments)** is passed to utils.sphere

Returns list of spheres.

Lines starting with # are skipped

Bibliography

[1] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Clarendon Press, New York, NY, USA, 1989. ISBN 0-19-855645-4.

[2] F. Alonso-Marroquín, R. García-Rojo, and H. J. Herrmann. Micro-mechanical investigation of the granular ratcheting. In T. Triantafyllidis, editor, *Cyclic Behaviour of Soils and Liquefaction Phenomena*, pages 3–10. Taylor & Francis, april 2004. ISBN 9058096203. URL http://www.comphys.ethz.ch/hans/p/334.pdf.

[3] Peter F. Ash and Ethan D. Bolker. Generalized dirichlet tessellations. *Geometriae Dedicata*, 20(2): 209–243, April 1986. ISSN 0046-5755 (Print) 1572-9168 (Online). doi: 10.1007/BF00164401. URL http://www.springerlink.com/content/j334537p07370405/.

[4] N. Bićanić. Discrete Element Methods. In E. Stein, R. de Borst, and T.J.R. Hughes, editors, *Encyclopedia of Computational Mechanics: Fundametals*, pages 311–337. Wiley and Sons, 2004.

[5] F. Camborde, C. Mariotti, and F. V. Donzé. Numerical study of rock and concrete behaviour by discrete element modelling. *Computers and Geotechnics*, 27(4):225–247, 2000. URL http://dx.doi.org/10.1016/S0266-352X(00)00013-6.

[6] Ferhun C. Caner and Zdenek P. Bažant. Microplane model m4 for concrete. ii: Algorithm and calibration. *Journal of Engineering Mechanics*, 126(9):954–961, 2000. doi: 10.1061/(ASCE)0733-9399(2000)126:9(954). URL http://www.civil.northwestern.edu/people/bazant/PDFs/Papers/394.pdf.

[7] Bruno Chareyre and Pascal Villard. Dynamic spar elements and discrete element methods in two dimensions for the modeling of soil-inclusion problems. *Journal of Engineering Mechanics*, 131 (7):689–698, 2005. doi: 10.1061/(ASCE)0733-9399(2005)131:7(689). URL http://link.aip.org/link/?QEM/131/689/1.

[8] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*, 29(1):47–65, 1979.

[9] Gianluca Cusatis, Zdenek P. Bažant, and Luigi Cedolin. Confinement-shear lattice model for concrete damage in tension and compression: I. theory. *Journal of Engineering Mechanics*, 129 (12):1439–1448, 2003. URL http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=JENMDT000129000012001439000001&idtype=cvips&gifs=yes.

[10] G. A. D'Addetta, F. Kun, E. Ramm, and H. J. Herrmann. From solids to granulates - Discrete element simulations of fracture and fragmentation processes in geomaterials. In P. A. Vermeer, S. Diebels, W. Ehlers, H. J. Herrmann, S. Luding, & E. Ramm, editor, *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*, volume 568 of *Lecture Notes in Physics, Berlin Springer Verlag*, pages 231–+, 2001. URL http://www.comphys.ethz.ch/hans/p/267.pdf.

[11] F. V. Donzé. Spherical discrete element code, 1997. URL http://geo.hmg.inpg.fr/frederic/articles/sdec_v2.00.pdf.

[12] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL http://people.redhat.com/drepper/cpumemory.pdf.

[13] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–323, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: http://doi.acm.org/10.1145/236337.236369.

[14] ESyS-Particle. Esys-particle, February 2010. URL https://launchpad.net/esys-particle/.

[15] Euler angles. Euler angles, February 2010. URL http://en.wikipedia.org/wiki/Euler_angles.

[16] Peter Grassl and Milan Jirásek. Damage-plastic model for concrete failure. *International Journal of Solids and Structures*, 43(22-23):7166–7196, 2006. ISSN 0020-7683. doi: DOI:10.1016/j.ijsolstr.2006.06.032. URL http://www.sciencedirect.com/science/article/B6VJS-4K8NWXK-1/2/e3a41b693b156ce13a70e44e973d505f.

[17] D. V. Griffiths and G. G. W. Mustoe. Modelling of elastic continua using a grillage of structural elements based on discrete element concepts. *International Journal for Numerical Methods in Engineering*, 50(7):1759–1775, 2001. doi: 10.1002/nme.99. URL http://80.www3.interscience.wiley.com.dialog.cvut.cz/journal/76509666/abstract.

[18] Sir William Rowan Hamilton. Lectures on quaternions, 1853. URL http://books.google.com/books?id=TCwPAAAAIAAJ.

[19] Sébastien Hentz. *Modélisation d'une Structure en Béton Armé Soumise à un Choc par la méthode des Éléments Discrets*. PhD thesis, Université Grenoble 1 – Joseph Fourier, October 2003.

[20] Sébastien Hentz, Laurent Daudeville, and Frédéric V. Donzé. Identification and validation of a discrete element model for concrete. *Journal of Engineering Mechanics*, 130(6):709–719, June 2004.

[21] Alexander Hrennikoff. Solution of problems of elasticity by the frame-work method. *ASME Journal of Applied Mechanics*, (8):A619–A715, 1941.

[22] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 15(3):179–210, 1996. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/231731.231732.

[23] ICG. *PFC3D (Particle Flow Code in 3D) Theory and Background Manual, version 3.0*. Itasca Consulting Group, 2003.

[24] Jean-François Jerier, Didier Imbault, Fréderic-Victor Donzé, and Pierre Doremus. A geometric algorithm based on tetrahedral meshes to generate a dense polydisperse sphere packing. *Granular Matter*, 11(1):43–52, 2009. doi: 10.1007/s10035-008-0116-0. URL http://www.springerlink.com/content/w0x307g110421035.

[25] Scott M. Johnson, John R. Williams, and Benjamin K. Cook. Quaternion-based rigid body rotation integration algorithms for use in particle methods. *International Journal for Numerical Methods in Engineering*, 74(8):1303–1313, 2008. doi: 10.1002/nme.2210. URL http://80.www3.interscience.wiley.com.dialog.cvut.cz/journal/116835638/abstract.

[26] Derek Jung and Kamal K. Gupta. Octree-based hierarchical distance maps for collision detection. *Journal of Robotic Systems*, 14(11):789–806, 1997. doi: http://dx.doi.org/10.1002/(SICI)1097-4563(199711)14:11<789::AID-ROB3>3.0.CO;2-Q. URL http://www3.interscience.wiley.com/journal/51764/abstract?CRETRY=1&SRETRY=0.

[27] Tadahiko Kawai. New element models in discrete structural analysis. *Journal of the Society of Naval Architects of Japan*, (141):174–180, 19770600. ISSN 05148499. URL http://ci.nii.ac.jp/naid/110003878089/en/.

[28] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4:21–36, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.6555&rep=rep1&type=pdf.

[29] J. Kozicki and F. V. Donzé. A new open-source software developed for numerical simulations using discrete modeling methods. *Computer Methods in Applied Mechanics and Engineering*, 197:4429–4443, 2008. doi: doi:10.1016/j.cma.2008.05.023. URL http://linkinghub.elsevier.com/retrieve/pii/S0045782508002119.

[30] J. Kozicki and F.V. Donzé. Yade-open dem: an open-source software using a discrete element method to simulate granular material. *Engineering Computations: Int J for Computer-Aided Engineering*, 26(7):786–805, 2009. doi: 10.1108/02644400910985170. URL http://www.ingentaconnect.com/content/mcb/182/2009/00000026/00000007/art00003.

[31] E. Kuhl, G. A. D'Addetta, M. Leukart, and E. Ramm. Microplane modelling and particle modelling of cohesive-frictional materials. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*, volume 568 of *Lecture Notes in Physics*, pages 31–46. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-41525-1. doi: 10.1007/3-540-44424-6_3. URL http://www.springerlink.com/content/e50544266r506615.

[32] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer, 3rd edition, February 2009. ISBN 3540739157. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540739157.

[33] J. P. B. Leite, V. Slowik, and H. Mihashi. Computer simulation of fracture processes of concrete using mesolevel models of lattice structures. *Cement and Concrete Research*, 34(6):1025–1033, 2004. doi: DOI:10.1016/j.cemconres.2003.11.011. URL http://www.sciencedirect.com/science/article/B6TWG-4B8X294-2/2/51f72ac6eb39cfeaf744c5980dd2fc2f.

[34] Ching-Lung Liao, Ta-Peng Chang, Dong-Hwa Young, and Ching S. Chang. Stress-strain relationship for granular materials based on the hypothesis of best fit. *International Journal of Solids and Structures*, 34(31–32):4087–4100, 1997. ISSN 0020-7683. doi: DOI:10.1016/S0020-7683(97)00015-2. URL http://www.sciencedirect.com/science/article/B6VJS-3XDHMP5-10/2/9311e21c602280eb84adca51eb4dc744.

[35] G. Lilliu and J. G. M. van Mier. 3d lattice type fracture model for concrete. *Engineering Fracture Mechanics*, 70(7–8):927–941, 2003. ISSN 0013-7944. doi: DOI:10.1016/S0013-7944(02)00158-3. URL http://www.sciencedirect.com/science/article/B6V2R-47DM661-2/2/b3ec6fb13217ef0e8f7a854f6aa166de.

[36] Stefan Luding. Introduction to discrete element methods. In Félix Darve and Jean-Pierre Ollivier, editors, *European Journal of Environmental and Civil Engineering*, pages 785–826. Lavoisier, 2008. ISBN 978-2-7462-2258-8.

[37] A. Munjiza. *The Combined Finite-Discrete Element Method*. John Wiley & Sons, Ltd, 2004.

[38] A. Munjiza and K. R. F. Andrews. NBS contact detection algorithm for bodies of similar size. *International Journal for Numerical Methods in Engineering*, 43(1):131–149, 1998. doi: 10.1002/(SICI)1097-0207(19980915)43:1<131::AID-NME447>3.0.CO;2-S. URL http://www3.interscience.wiley.com/journal/10005234/abstract.

[39] A. Munjiza, D. R. J. Owen, and N. Bićanić. A combined finite-discrete element method in transient dynamics of fracuring solids. *Engineering Computations*, 12:145–174, 1995.

[40] A. Munjiza, E. Rougier, and N. W. M. John. MR linear contact detection algorithm. *International Journal for Numerical Methods in Engineering*, 66(1):46–71, 2006. doi: 10.1002/nme.1538. URL http://dx.doi.org/10.1002/nme.1538.

[41] Kouhei Nagai, Yasuhiko Sato, Tamon Ueda, and Yoshio Kakuta. Numerical simulation of fracture process of concrete model by rigid body spring method. コンクリート工学年次論文集, 24(2):163–168, 2002. URL http://211.10.28.144/data_pdf/24/024-01-2028.pdf.

[42] Natale Neto and Luca Bellucci. A new algorithm for rigid body molecular dynamics. *Chemical Physics*, 328(1–3):259–268, 2006. ISSN 0301-0104. doi: DOI:10.1016/j.chemphys.2006.07.009. URL http://www.sciencedirect.com/science/article/B6TFM-4KCXF8Y-5/2/d1394e8a82938b31093afa52c3d47863.

[43] Erfan G. Nezami, Youssef M.A. Hashash, Dawei Zhao, and Jamshid Ghaboussi. A fast contact detection algorithm for 3-d discrete element method. *Computers and Geotechnics*, 31(7):575–587, 2004. ISSN 0266-352X. doi: 10.1016/j.compgeo.2004.08.002. URL http://www.sciencedirect.com/science/article/B6V2C-4DMW3PT-1/2/d109e9e249daf37d294e6a10d24f8d31.

[44] Igor P. Omelyan. A new leapfrog integrator of rotational motion. the revised angular-momentum approach. *Molecular Simulation*, 22(3), 1999. doi: 10.1080/08927029908022097. URL http://arxiv.org/pdf/physics/9901025.

[45] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Open multi-methods for c++. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and compo-*

nent engineering, pages 123–134, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8. doi: http://doi.acm.org/10.1145/1289971.1289993.

[46] A. V. Potapov, C. S. Campbell, and M. A. Hopkins. A two-dimensional dynamic simulation of solid fracture part i: description of the model. *International Journal of Modern Physics*, 6(3):371–397, 1995.

[47] Alexander V. Potapov, Charles S. Campbell, and Mark A. Hopkins. A two-dimensional dynamic simulation of solid fracture part ii: examples. *International Journal of Modern Physics*, 6(3), 1995. doi: 10.1142/S0129183195000289.

[48] L. Pournin, Th. M. Liebling, and A. Mocellin. Molecular-dynamics force models for better control of energy dissipation in numerical simulations of dense granular media. *Phys. Rev. E*, 65(1):011302, Dec 2001. doi: 10.1103/PhysRevE.65.011302.

[49] Mathew Price, Vasile Murariu, and Garry Morrison. Sphere clump generation and trajectory comparison for real particles. In *Proceedings of Discrete Element Modelling 2007*, 2007. URL http://www.cogency.co.za/images/info/dem2007_sphereclump.pdf.

[50] Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 147–154. Canadian Human-Computer Communications Society, 1995. URL http://citeseer.ist.psu.edu/provot96deformation.html.

[51] Quaternion. Quaternion, February 2010. URL http://en.wikipedia.org/wiki/Quaternion.

[52] Quaternion-rotations. Quaternions and spatial rotation, February 2010. URL http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation.

[53] Frédéric Ragueneau and Fabrice Gatuingt. Inelastic behavior modelling of concrete in low and high strain rate dynamics. *Computers & Structures*, 81(12):1287–1299, 2003. ISSN 0045-7949. doi: DOI: 10.1016/S0045-7949(03)00043-9. Advanced Computational Models and Techniques in Dynamics.

[54] Jessice Rousseau, Emmanuel Frangin, Phiippe Marin, and Laurent Daudeville. Multidomain finite and discrete elements method for impact analysis of a concrete structure. *Engineering structures*, 43(1–2):2735–2743, 2009. URL http://geo.hmg.inpg.fr/%7Edaudevil/publis/engstruct2.pdf.

[55] E. Schlangen and E. J. Garboczi. New method for simulating fracture using an elastically uniform random geometry lattice. *International Journal of Engineering Science*, 34(10):1131–1144, 1996. URL http://www.fire.nist.gov/bfrlpubs/build96/PDF/b96022.pdf.

[56] L. Scholtès, B. Chareyre, F. Nicot, and F. Darve. Micromechanics of granular materials with capillary effects. *International Journal of Engineering Science*, 47(1):64–75, 2009. ISSN 0020-7225. doi: 10.1016/j.ijengsci.2008.07.002. URL http://www.sciencedirect.com/science/article/B6V32-4TDJGDJ-1/2/9878efa5e573197aff6ee14d5abb58a2.

[57] Luc Scholtès. *Modélisation micromécanique des milieux granulaires partiellement saturés*. PhD thesis, Institut National Polytechnique de Grenoble, 2009. URL http://tel.archives-ouvertes.fr/tel-00363961/en/.

[58] Gen-Hua Shi. Discontinuous deformation analysis: a new numerical model for the statics and dynamics of deformable block structures. *Engineering computations*, 9:157–168, 1992.

[59] W. J. Shiu, F. V. Donzé, and L. Daudeville. Compaction process in concrete during missile impact: a dem analysis. *Computers and Concrete*, 5(4):329–342, 2008. URL http://geo.hmg.inpg.fr/%7Edaudevil/publis/Computers&Concrete2.pdf.

[60] Danil Shopyrin. Multimethods in c++: Finding a complete solution, February 2010. URL http://www.codeproject.com/KB/recipes/mmcppfcs.aspx.

[61] Slerp. Slerp, February 2010. URL http://en.wikipedia.org/wiki/Slerp.

[62] Julian Smith. Cmm (c++ with multimethods), February 2010. URL http://www.op59.net/cmm/cmm-0.28/readme.html.

[63] Jan Stránský, Milan Jirásek, and Václav Šmilauer. Macroscopic elastic properties of particle models. In *Proceedings of the Interaction Conference on Modelling and Simulation 2010, Prague*. preprint, June 2010.

[64] Guido van Rossum. Five-minute multimethods in python, February 2010. URL http://www.artima.com/weblogs/viewpost.jsp?thread=101605.

[65] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159(1):98, Jul 1967. doi: 10.1103/PhysRev.159.98.

[66] Adri Vervuurt. *Interface Fracture in Concrete (proefschrift)*. Technische Universiteit Delft, 1997.

[67] Václav Šmilauer. The splendors and miseries of yade design. *Annual Report of Discrete Element Group for Hazard Mitigation*, 2006.

[68] Yucang Wang. A new algorithm to model the dynamics of 3-d bonded rigid bodies with rotations. *Acta Geotechnica*, 4(2):117–127, July 2009. ISSN 1861-1125 (Print) 1861-1133 (Online). doi: 10.1007/s11440-008-0072-1. URL http://www.springerlink.com/content/l2306412v1004871/.

[69] Yucang Wang and Peter Mora. Macroscopic elastic properties of regular lattices. *Journal of the Mechanics and Physics of Solids*, 56(12):3459–3474, 2008. ISSN 0022-5096. doi: DOI:10.1016/j.jmps.2008.08.011. URL http://www.sciencedirect.com/science/article/B6TXB-4TF2J92-1/2/c988bc1c23c664e4562f2cd04c19e00e.

[70] J. G. Williams. The analysis of dynamic fracture using lumped mass-spring models. *International Journal of Fracture*, 33(1):47–59, January 1987. ISSN 0376-9429 (Print) 1573-2673 (Online). doi: 10.1007/BF00034898. URL http://www.springerlink.com/content/h31089636u20h601/.

[71] Yade. Yade history, February 2010. URL http://www.yade-dem.org/index.php/Yade_history.