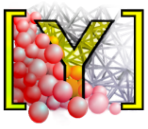


Yade Reference Documentation

Václav Šmilauer, Emanuele Catalano, Bruno Chareyre, Sergei Dorofeenko, Jerome Duriez, Anton Gladky, Janek Kozicki, Chiara Modenese, Luc Scholtès, Luc Sibille, Jan Stránský, Klaus Thoeni

February 17, 2011
(1st edition - from release bzr2718)

Editor

Václav Šmilauer
CVUT Prague - lab. 3SR Grenoble University

Authors

Václav Šmilauer
CVUT Prague - lab. 3SR Grenoble University

Emanuele Catalano
Grenoble INP, UJF, CNRS, lab. 3SR

Bruno Chareyre
Grenoble INP, UJF, CNRS, lab. 3SR

Sergei Dorofeenko
IPCP RAS, Chernogolovka

Jerome Duriez
Grenoble INP, UJF, CNRS, lab. 3SR

Anton Gladky
TU Bergakademie Freiberg

Janek Kozicki
Gdansk University of Technology - lab. 3SR Grenoble University

Chiara Modenese
University of Oxford

Luc Scholtès
Grenoble INP, UJF, CNRS, lab. 3SR

Luc Sibille
University of Nantes, lab. GeM

Jan Stránský
CVUT Prague

Klaus Thoeni
University of Newcastle (Australia)

Citing this document
Please use the following reference, as explained at http://yade-dem/doc/citing.html:

V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Reference Documentation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (http://yade-dem.org/doc/)

Abstract

This chapter describes all high level classes and functions, including contact laws, boundary controllers, pre- and post-processing tools. **Keywords**: Contact laws, boundary conditions, preprocessing, postprocessing.

Contents

Chapter 1

Class reference (yade.wrapper module)

1.1 Bodies

1.1.1 Body

class `yade.wrapper.Body`(*inherits Serializable*)

A particle, basic element of simulation; interacts with other bodies.

`aspherical`(=*false*)
: Whether this body has different inertia along principal axes; NewtonIntegrator makes use of this flag to call rotation integration routine for aspherical bodies, which is more expensive.

`bound`(=*uninitalized*)
: Bound, approximating volume for the purposes of collision detection.

`bounded`(=*true*)
: Whether this body should have Body.bound created. Note that bodies without a bound do not participate in collision detection. (In c++, use `Body::isBounded`/`Body::setBounded`)

`clumpId`
: Id of clump this body makes part of; invalid number if not part of clump; see Body::isStandalone, Body::isClump, Body::isClumpMember properties.

  Not meant to be modified directly from Python, use O.bodies.appendClumped instead.

`dynamic`(=*true*)
: Whether this body will be moved by forces. (In c++, use `Body::isDynamic`/`Body::setDynamic`)

`flags`(=*FLAG_BOUNDED*)
: Bits of various body-related flags. *Do not access directly.* In c++, use isDynamic/setDynamic, isBounded/setBounded, isAspherical/setAspherical. In python, use Body.dynamic, Body.bounded, Body.aspherical.

`groupMask`(=*1*)
: Bitmask for determining interactions.

`id`(=*Body::ID_NONE*)
: Unique id of this body.

`intrs`() → list
: Return all interactions in which this body participates.

isClump
True if this body is clump itself, false otherwise.

isClumpMember
True if this body is clump member, false otherwise.

isStandalone
True if this body is neither clump, nor clump member; false otherwise.

mask
Shorthand for Body::groupMask

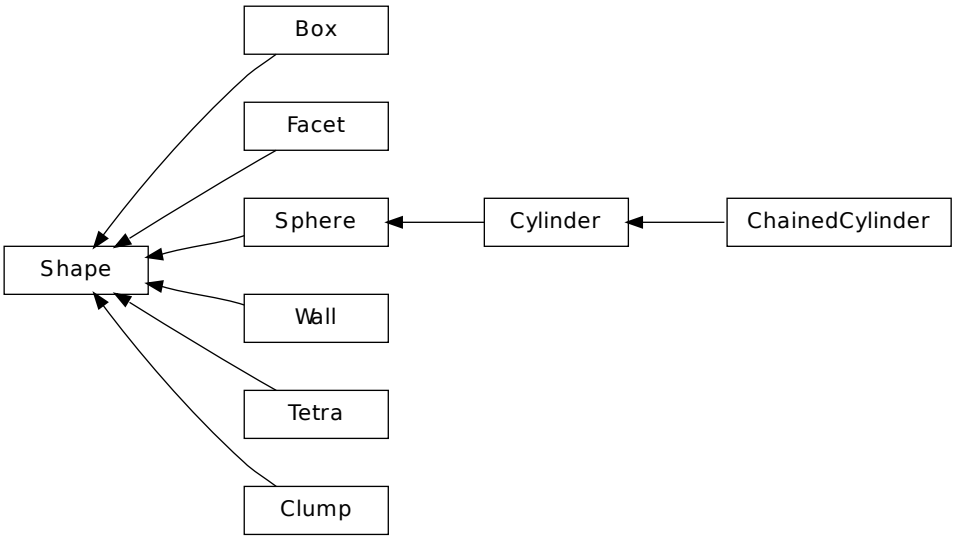
mat
Shorthand for Body::material

material(=*uninitalized*)
Material instance associated with this body.

shape(=*uninitalized*)
Geometrical Shape.

state(=*new State*)
Physical state.

1.1.2 Shape



class yade.wrapper.**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 yade.wrapper.**Box**(*inherits Shape → Serializable*)
Box (cuboid) particle geometry. (Avoid using in new code, prefer Facet instead.

extents(=*uninitalized*)
Half-size of the cuboid

class yade.wrapper.**ChainedCylinder**(*inherits Cylinder → Sphere → Shape → Serializable*)
Geometry of a deformable chained cylinder, using geometry Cylinder.

chainedOrientation(=*Quaternionr::Identity()*)
Deviation of node1 orientation from node-to-node vector

initLength(=*0*)
tensile-free length, used as reference for tensile strain

class yade.wrapper.**Clump**(*inherits Shape → Serializable*)
Rigid aggregate of bodies

members
Return clump members as {'id1':(relPos,relOri),...}

class yade.wrapper.**Cylinder**(*inherits Sphere → Shape → Serializable*)
Geometry of a cylinder, as Minkowski sum of line and sphere.

length(=*NaN*)
Length [m]

segment(=*Vector3r::Zero()*)
Length vector

class yade.wrapper.**Facet**(*inherits Shape → Serializable*)
Facet (triangular particle) geometry.

vertices(=*vector<Vector3r>(3, Vector3r(NaN, NaN, NaN))*)
Vertex positions in local coordinates.

class yade.wrapper.**Sphere**(*inherits Shape → Serializable*)
Geometry of spherical particle.

radius(=*NaN*)
Radius [m]

class yade.wrapper.**Tetra**(*inherits Shape → Serializable*)
Tetrahedron geometry.

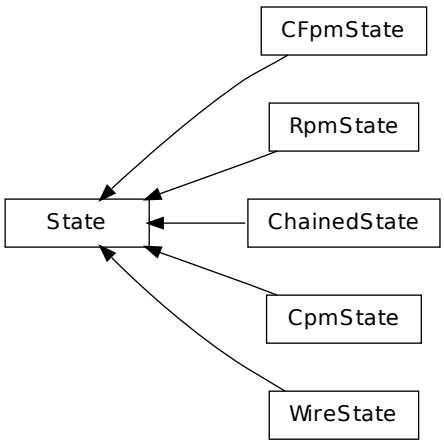
v(=*std::vector<Vector3r>(4)*)
Tetrahedron vertices in global coordinate system.

class yade.wrapper.**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

1.1.3 State



class `yade.wrapper.State`(*inherits Serializable*)

State of a body (spatial configuration, internal variables).

`angMom`(=*Vector3r::Zero()*)
: Current angular momentum

`angVel`(=*Vector3r::Zero()*)
: Current angular velocity

`blockedDOFs`
: Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

`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.

`displ`() → Vector3
: Displacement from reference position (pos - refPos

`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

`rot`() → Vector3
: Rotation from reference orientation (as rotation vector)

se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

vel(=*Vector3r::Zero()*)
Current linear velocity.

class yade.wrapper.**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 broken cohesive links. [-]

class yade.wrapper.**ChainedState**(*inherits State → Serializable*)
State of a chained bodies, containing information on connectivity in order to track contacts jumping over contiguous elements. Chains are 1D lists from which id of chained bodies are retrieved via :yref:rank<ChainedState::rank>‘ and :yref:chainNumber<ChainedState::chainNumber>‘.

addToChain(*(int)bodyId*) → None
Add body to current active chain

bId(=*-1*)
id of the body containing - for postLoad operations only

chainNumber(=*0*)
chain id

rank(=*0*)
rank in the chain

class yade.wrapper.**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.

class yade.wrapper.**RpmState**(*inherits State → Serializable*)
State information about Rpm body.

specimenMass(=*0*)
Indicates the mass of the whole stone, which owns the particle.

specimenMaxDiam(=*0*)
Indicates the maximal diametr of the specimen.

specimenNumber(=*0*)
The variable is used for particle size distribution analyze. Indicates, to which part of specimen belongs para of particles.

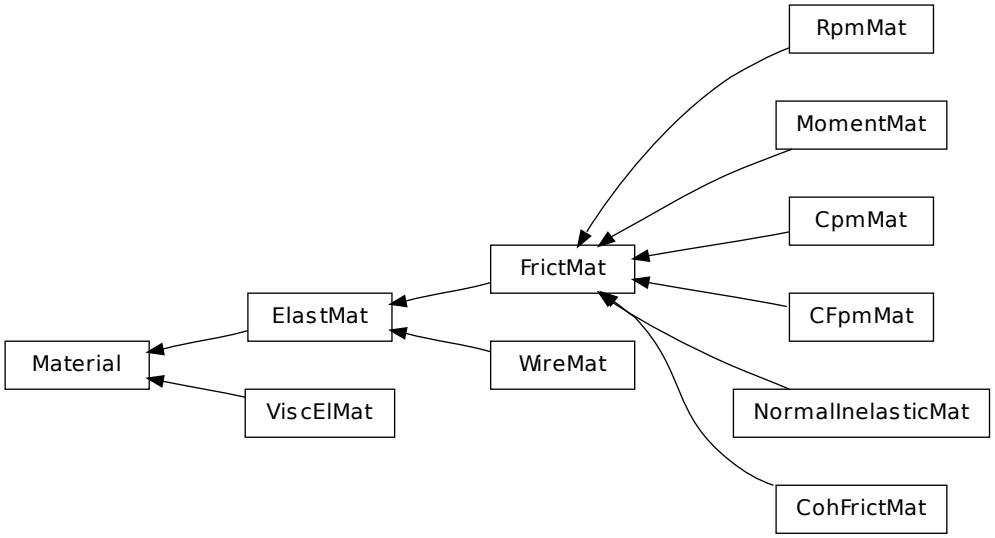
specimenVol(=*0*)
Indicates the mass of the whole stone, which owns the particle.

class yade.wrapper.**WireState**(*inherits State → Serializable*)
Wire state information of each body.

None of that is used for computation (at least not now), only for post-processing.

numBrokenLinks(=*0*)
Number of broken links (e.g. number of wires connected to the body which are broken). [-]

1.1.4 Material



class yade.wrapper.**Material**(*inherits Serializable*)
Material properties of a body.

density(=*1000*)
Density of the material [kg/m^3]

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(=*-1, not shared*)
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 was necessary since before boost::serialization was used, shared pointers were not tracked properly; it 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 `yade.wrapper.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 `yade.wrapper.CohFrictMat`(*inherits FrictMat → ElastMat → Material → Serializable*)

`alphaKr`(*=2.0*)
Dimensionless coefficient used for the rolling stiffness.

`alphaKtw`(*=2.0*)
Dimensionless coefficient used for the twist stiffness.

`etaRoll`(*=-1.*)
Dimensionless coefficient used to calculate the plastic rolling moment (if negative, plasticity will not be applied).

`isCohesive`(*=true*)

`momentRotationLaw`(*=false*)
Use bending/twisting moment at contact. The contact will have moments only if both bodies have this flag true. See CohFrictPhys::cohesionDisablesFriction for details.

`normalCohesion`(*=0*)

`shearCohesion`(*=0*)

class `yade.wrapper.CpmMat`(*inherits FrictMat → ElastMat → Material → Serializable*)
Concrete material, for use with other Cpm classes.

Note: Density is initialized to 4800 kgm ^{3}automatically, 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 `yade.wrapper.ElastMat`(*inherits Material → Serializable*)
Purely elastic material. The material parameters may have different meanings depending on the IPhysFunctor used : true Young and Poisson in Ip2_FrictMat_FrictMat_MindlinPhys, or contact stiffnesses in Ip2_FrictMat_FrictMat_FrictPhys.

`poisson`(*=.25*)
Poisson's ratio [-]

`young`(*=1e9*)
Young's modulus [Pa]

class `yade.wrapper.FrictMat`(*inherits ElastMat → Material → Serializable*)
Elastic material with contact friction. See also ElastMat.

`frictionAngle`(*=.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

class `yade.wrapper.MomentMat`(*inherits FrictMat → ElastMat → Material → Serializable*)
Material for constitutive law of (Plassiard & al., 2009); see Law2_SCG_MomentPhys_CohesionlessMomentRotation for details.

Users can input eta (constant for plastic moment) to Spheres and Boxes. For more complicated cases, users can modify TriaxialStressController to use different eta values during isotropic compaction.

`eta`(*=0*)
(has to be stored in this class and not by ContactLaw, because users may want to change its values before/after isotropic compaction.)

class `yade.wrapper.NormalInelasticMat`(*inherits FrictMat → ElastMat → Material → Serializable*)
Material class for particles whose contact obey to a normal inelasticity (governed by this *coeff_dech*).

`coeff_dech`(*=1.0*)
=kn(unload) / kn(load)

class `yade.wrapper.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 `yade.wrapper.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 analytical 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

class yade.wrapper.**WireMat**(*inherits ElastMat → Material → Serializable*)
Material for use with the Wire classes

as(=*0*)
Cross-section area of a single wire used for the computation of the limit normal contact forces. [m^2]

diameter(=*0.0027*)
(Diameter of the single wire in [m] (the diameter is used to compute the cross-section area of the wire).

isDoubleTwist(=*false*)
Type of the mesh. If true two particles of the same material which body ids differ by one will be considered as double-twisted interaction.

lambdaEps(=*0.4*)
Parameter between 0 and 1 to reduce the failure strain of the double-twisted wire (as used by [Bertrand2008]). [-]

lambdak(=*0.21*)
Parameter between 0 and 1 to compute the elastic stiffness of the double-twisted wire (as used by [Bertrand2008]): $k^D = 2(\lambda_k k_h + (1-\lambda_k)k^S)$. [-]

strainStressValues(=*uninitalized*)
Piecewise linear definition of the stress-strain curve by set of points (strain[-]>0,stress[Pa]>0) for one single wire. Tension only is considered and the point (0,0) is not needed!

1.1.5 Bound

class yade.wrapper.**Bound**(*inherits Serializable*)
Object bounding part of space taken by associated body; might be larger, used to optimalize collision detection

color(=*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`(=*Vector3r(NaN, NaN, NaN)*)
Lower corner of box containing this bound (and the Body as well)

`min`(=*Vector3r(NaN, NaN, NaN)*)
Lower corner of box containing this bound (and the Body as well)

class `yade.wrapper.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.)

1.2 Interactions

1.2.1 Interaction

class `yade.wrapper.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 thepriod information if the geometric functor again makes it real. Good to know after few days of debugging that :-)

`geom`(=*uninitalized*)
Geometry part of the interaction.

`id1`(=*0*)
Id of the first body in this interaction.

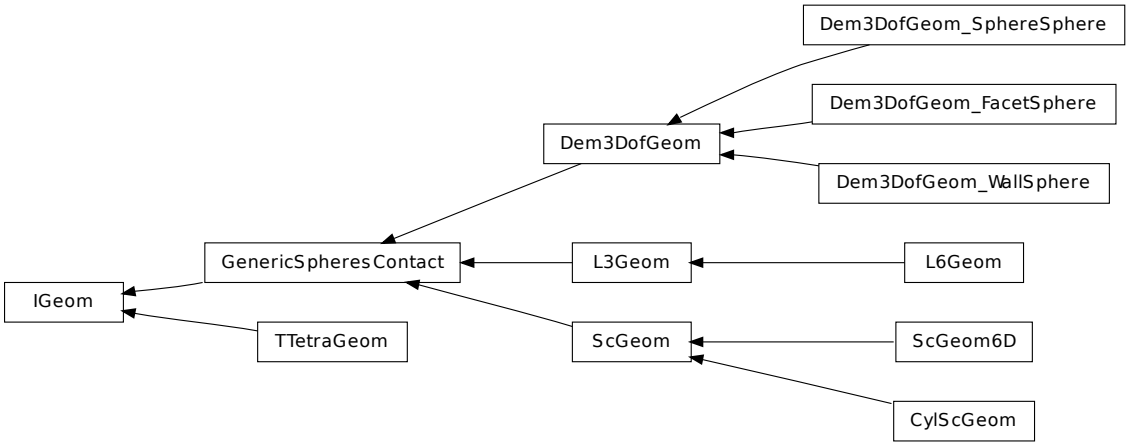
`id2`(=*0*)
Id of the first body in this interaction.

`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 geom and phys) created. (Should be touched only by IPhysDispatcher and InteractionLoop, therefore they are made friends of Interaction

`phys`(=*uninitalized*)
Physical (material) part of the interaction.

1.2.2 IGeom



class `yade.wrapper.IGeom`(*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 `yade.wrapper.CylScGeom`(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)

Geometry of a cylinder-sphere contact.

`end`(=*Vector3r::Zero()*)

position of 2nd node *(auto-updated)*

`id3`(=*0*)

id of next chained cylinder *(auto-updated)*

`isDuplicate`(=*0*)

this flag is turned true (1) automaticaly if the contact is shared between two chained cylinders. A duplicated interaction will be skipped once by the constitutive law, so that only one contact at a time is effective. If isDuplicate=2, it means one of the two duplicates has no longer geometric interaction, and should be erased by the constitutive laws.

`onNode`(=*false*)

contact on node?

`relPos`(=*0*)

position of the contact on the cylinder (0: node-, 1:node+) *(auto-updated)*

`start`(=*Vector3r::Zero()*)

position of 1st node *(auto-updated)*

`trueInt`(=*-1*)

Defines the body id of the cylinder where the contact is real, when CylScGeom::isDuplicate>0.

class `yade.wrapper.Dem3DofGeom`(*inherits GenericSpheresContact → IGeom → 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).

`displacementN`() → float

`displacementT`() → Vector3

`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)*

`slipToDisplacementTMax`(*(float)arg2*) → float

`slipToStrainTMax`(*(float)arg2*) → float

`strainN`() → float

`strainT`() → Vector3

class `yade.wrapper.Dem3DofGeom_FacetSphere`(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → 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 `yade.wrapper.Dem3DofGeom_SphereSphere`(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → 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 `yade.wrapper.Dem3DofGeom_WallSphere`(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → 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 `yade.wrapper.GenericSpheresContact`(*inherits IGeom → Serializable*)
Class uniting ScGeom and Dem3DofGeom, for the purposes of GlobalStiffnessTimeStepper. (It might be removed inthe future). Do not use this class directly.

`contactPoint`(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

`normal`(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

`refR1`(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

`refR2`(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

class `yade.wrapper.L3Geom`(*inherits GenericSpheresContact → IGeom → Serializable*)
Geometry of contact given in local coordinates with 3 degress of freedom: normal and two in shear plane. [experimental]

`F`(=*Vector3r::Zero()*)
Applied force in local coordinates [debugging only, will be removed]

`trsf`(=*Matrix3r::Identity()*)
Transformation (rotation) from global to local coordinates. (the translation part is in GenericSpheresContact.contactPoint)

`u`(=*Vector3r::Zero()*)
Displacement components, in local coordinates. *(auto-updated)*

`u0`
Zero displacement value; u0 should be always subtracted from the *geometrical* displacement u computed by appropriate IGeomFunctor, resulting in u. This value can be changed for instance

1. by IGeomFunctor, e.g. to take in account large shear displacement value unrepresentable by underlying geomeric algorithm based on quaternions)
2. by LawFunctor, to account for normal equilibrium position different from zero geometric overlap (set once, just after the interaction is created)
3. by LawFunctor to account for plastic slip.

Note: Never set an absolute value of *u0*, only increment, since both IGeomFunctor and LawFunctor use it. If you need to keep track of plastic deformation, store it in IPhys isntead (this might be changed: have *u0* for LawFunctor exclusively, and a separate value stored (when that is needed) inside classes deriving from L3Geom.

class `yade.wrapper.L6Geom`(*inherits L3Geom → GenericSpheresContact → IGeom → Serializable*)
Geometric of contact in local coordinates with 6 degrees of freedom. [experimental]

`phi`(=*Vector3r::Zero()*)
Rotation components, in local coordinates. *(auto-updated)*

`phi0`(=*Vector3r::Zero()*)
Zero rotation, should be always subtracted from *phi* to get the value. See L3Geom.u0.

class `yade.wrapper.ScGeom`(*inherits GenericSpheresContact → IGeom → Serializable*)
Class representing geometry of a contact point between two bodies with a non-spherical bodies (Facet, Plane, Box, ChainedCylinder), or between non-spherical bodies. The contact has 3 DOFs (normal and 2×shear) and uses incremental algorithm for updating shear.

We use symbols $\mathbf{x}$, $\boldsymbol{\nu}$, $\boldsymbol{\omega}$ 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

$$\boldsymbol{\nu}_{12} = (\boldsymbol{\nu}_2 + \boldsymbol{\omega}_2 \times (-r_2\mathbf{n})) - (\boldsymbol{\nu}_1 + \boldsymbol{\omega}_1 \times (r_1\mathbf{n}))$$

and its shear component

$$\Delta\boldsymbol{\nu}^s_{12} = \boldsymbol{\nu}_{12} - (\mathbf{n} \cdot \boldsymbol{\nu}_{12})\mathbf{n}.$$

Tangential displacement increment over last step then reads

$$\mathbf{x}^s_{12} = \Delta t \boldsymbol{\nu}^s_{12}.$$

`incidentVel`(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction.

`penetrationDepth`(*=NaN*)
Penetration distance of spheres (positive if overlapping)

`relAngVel`(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

`shearInc`(*=Vector3r::Zero()*)
Shear displacement increment in the last step

class `yade.wrapper.ScGeom6D`(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)
Class representing geometry of two bodies in contact. The contact has 6 DOFs (normal, 2×shear, twist, 2xbending) and uses ScGeom incremental algorithm for updating shear.

`bending`(*=Vector3r::Zero()*)
Bending at contact as a vector defining axis of rotation and angle (angle=norm).

`initialOrientation1`(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 1 one at initialisation time *(auto-updated)*

`initialOrientation2`(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

`twist`(*=0*)
Elastic twist angle of the contact.

`twistCreep`(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

class `yade.wrapper.TTetraGeom`(*inherits IGeom → 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*)
??

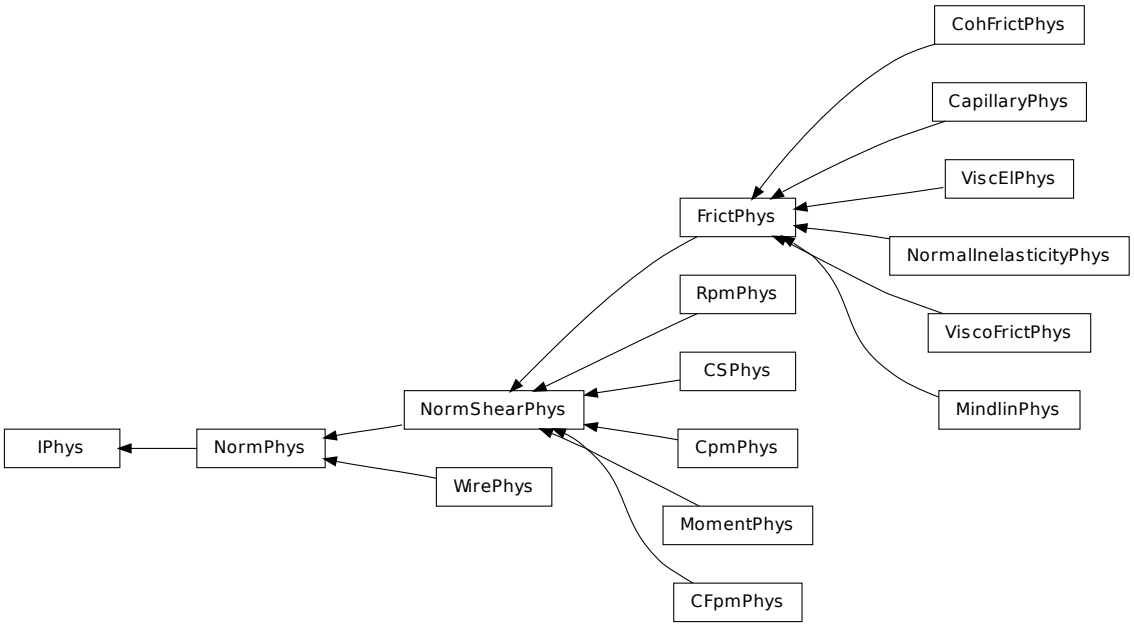
`maxPenetrationDepthA`(*=NaN*)
??

`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]

1.2.3 IPhys



class yade.wrapper.**IPhys**(*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 yade.wrapper.**CFpmPhys**(*inherits NormShearPhys → NormPhys → IPhys → 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 `yade.wrapper.CSPhys`(*inherits NormShearPhys → NormPhys → IPhys → 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 `yade.wrapper.CapillaryPhys`(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Physics (of interaction) for Law2_ScGeom_CapillaryPhys_Capillarity.

`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 `yade.wrapper.CohFrictPhys`(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → 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?

creep_viscosity(=*-1*)
creep viscosity [Pa.s/m].

creepedShear(=*Vector3r(0, 0, 0)*)
Creeped force (parallel)

fragile(=*true*)
do cohesion disapear when contact strength is exceeded?

kr(=*0*)
rotational stiffness [N.m/rad]

ktw(=*0*)
twist stiffness [N.m/rad]

maxRollPl(=*0.0*)
Coefficient to determine the maximum plastic rolling moment.

maxTwistMoment(=*Vector3r::Zero()*)
Maximum elastic value for the twisting moment (if zero, plasticity will not be applied). In CohFrictMat a parameter should be added to decide what value should be attributed to this threshold value.

momentRotationLaw(=*false*)
use bending/twisting moment at contacts. See CohFrictPhys::cohesionDisablesFriction for details.

moment_bending(=*Vector3r(0, 0, 0)*)
Bending moment

moment_twist(=*Vector3r(0, 0, 0)*)
Twist moment

normalAdhesion(=*0*)
tensile strength

shearAdhesion(=*0*)
cohesive part of the shear strength (a frictional term might be added depending on Law2_ScGeom6D_CohFrictPhys_CohesionMoment::always_use_moment_law)

unp(=*0*)
plastic normal displacement, only used for tensile behaviour and if CohFrictPhys::fragile‘=false. :ydefault:‘0

unpMax(=*0*)
maximum value of plastic normal displacement, after that the interaction breaks even if CohFrictPhys::fragile‘=false. The default value (0) means no maximum. :ydefault:‘0

class yade.wrapper.**CpmPhys**(*inherits NormShearPhys → NormPhys → IPhys → 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 [m^2]

`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(=Vector3r::Zero())`
Total shear strain (either computed from increments with ScGeom or simple copied with Dem3DofGeom) *(auto-updated)*

`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 yade.wrapper.**FrictPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
The simple linear elastip-plastic interaction with friction angle, like in the traditional [Cundall-Strack1979]

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

class yade.wrapper.**MindlinPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of an interaction of the Hertz-Mindlin type.

Fs(=*Vector2r::Zero()*)
Shear force in local axes (computed incrementally)

adhesionForce(=*0.0*)
Force of adhesion as predicted by DMT

alpha(=*0.0*)
Constant coefficient to define contact viscous damping for non-linear elastic force-displacement relationship.

betan(=*0.0*)
Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

betas(=*0.0*)
Fraction of the viscous damping coefficient (shear direction) equal to $\frac{c_s}{C_{s,crit}}$.

isAdhesive(=*false*)
bool to identify if the contact is adhesive, that is to say if the contact force is attractive

isSliding(=*false*)
check if the contact is sliding (useful to calculate the ratio of sliding contacts)

kno(=*0.0*)
Constant value in the formulation of the normal stiffness

kr(=*0.0*)
Rotational stiffness

kso(=*0.0*)
Constant value in the formulation of the tangential stiffness

ktw(=*0.0*)
Rotational stiffness

maxBendPl(=*0.0*)
Coefficient to determine the maximum plastic moment to apply at the contact

momentBend(=*Vector3r::Zero()*)
Artificial bending moment to provide rolling resistance in order to account for some degree of interlocking between particles

momentTwist(=*Vector3r::Zero()*)
Artificial twisting moment (no plastic condition can be applied at the moment)

normalViscous(=*Vector3r::Zero()*)
Normal viscous component

prevU(=*Vector3r::Zero()*)
Previous local displacement; only used with Law2_L3Geom_FrictPhys_HertzMindlin.

radius(=*NaN*)
Contact radius (only computed with Law2_ScGeom_MindlinPhys_Mindlin::calcEnergy)

shearElastic(=*Vector3r::Zero()*)
Total elastic shear force

shearViscous(=*Vector3r::Zero()*)
Shear viscous component

usElastic(=*Vector3r::Zero()*)
Total elastic shear displacement (only elastic part)

usTotal(=*Vector3r::Zero()*)
Total elastic shear displacement (elastic+plastic part)

class yade.wrapper.**MomentPhys**(*inherits NormShearPhys → NormPhys → IPhys → 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 yade.wrapper.**NormPhys**(*inherits IPhys → 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 yade.wrapper.**NormShearPhys**(*inherits NormPhys → IPhys → 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 `yade.wrapper.NormalInelasticityPhys`(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

Physics (of interaction) for using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity : with inelastic unloadings

`forMaxMoment`(=*1.0*)
: parameter stored for each interaction, and allowing to compute the maximum value of the exchanged torque : TorqueMax= forMaxMoment * NormalForce

`knLower`(=*0.0*)
: the stifness corresponding to a virgin load for example

`kr`(=*0.0*)
: the rolling stiffness of the interaction

`moment_bending`(=*Vector3r(0, 0, 0)*)
: Bending moment. Defined here, being initialized as it should be, to be used in Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

`moment_twist`(=*Vector3r(0, 0, 0)*)
: Twist moment. Defined here, being initialized as it should be, to be used in Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

`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 `yade.wrapper.RpmPhys`(*inherits NormShearPhys → NormPhys → IPhys → 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 [m^2]

`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 `yade.wrapper.ViscElPhys`(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

IPhys created from ViscElMat, for use with Law2_ScGeom_ViscElPhys_Basic.

`cn`(=*NaN*)
: Normal viscous constant

cs(=*NaN*)
Shear viscous constant

class yade.wrapper.**ViscoFrictPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Temporary version of FrictPhys for compatibility with e.g. Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

creepedShear(=*Vector3r(0, 0, 0)*)
Creeped force (parallel)

class yade.wrapper.**WirePhys**(*inherits NormPhys → IPhys → Serializable*)
Representation of a single interaction of the WirePM type, storage for relevant parameters

displForceValues(=*uninitalized*)
Defines the values for force-displacement curve.

initD(=*0*)
Equilibrium distance for particles. Computed as the initial inter-particular distance when particle are linked.

isDoubleTwist(=*false*)
If true the properties of the interaction will be defined as a double-twisted wire.

isLinked(=*false*)
If true particles are linked and will interact. Interactions are linked automatically by the definition of the corresponding interaction radius. The value is false if the wire breaks (no more interaction).

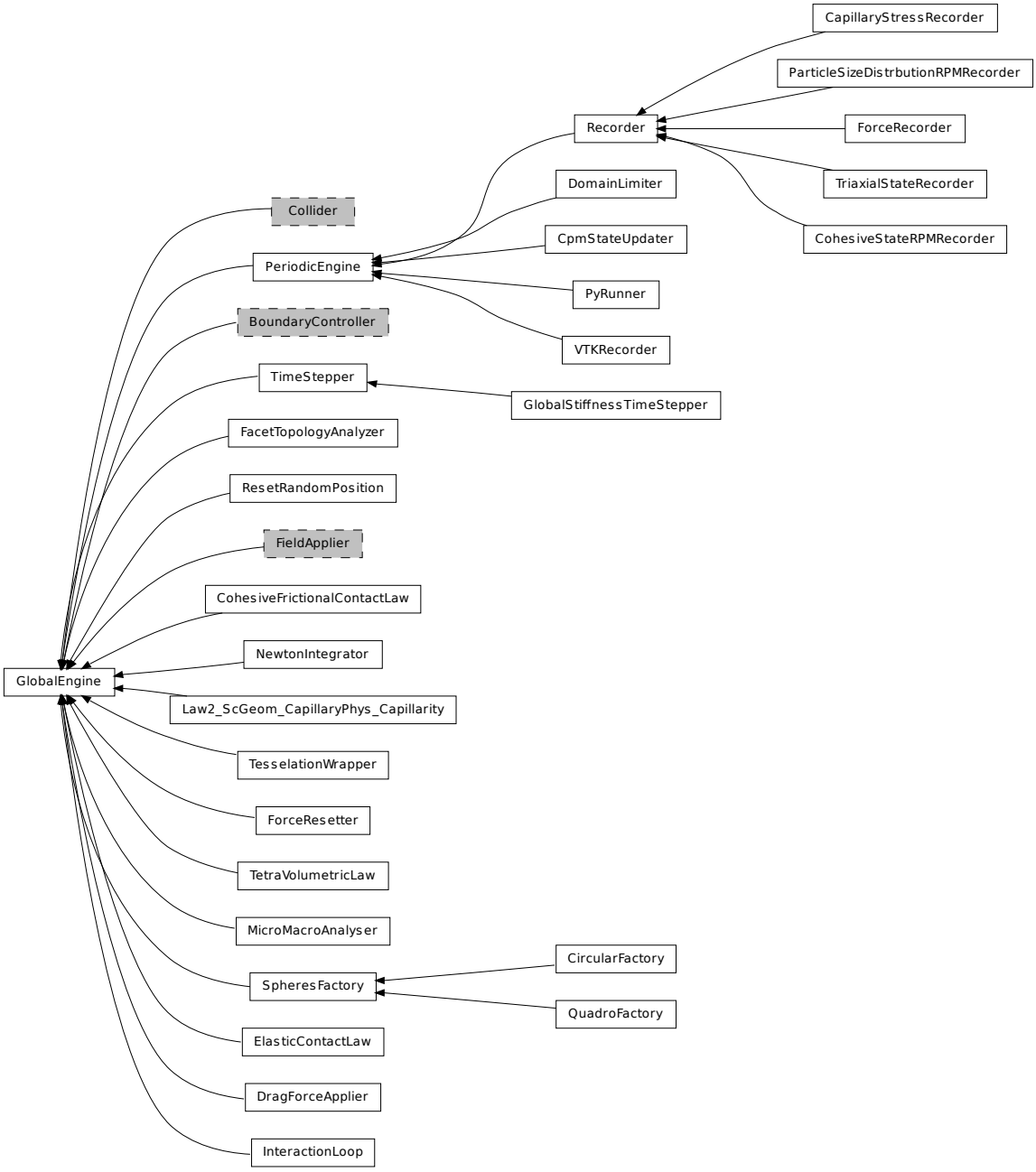
plastD
Plastic part of the inter-particular distance of the previous step.

Note: Only elastic displacements are reversible (the elastic stiffness is used for unloading) and compressive forces are inadmissible. The compressive stiffness is assumed to be equal to zero (see [Bertrand2005]).

stiffnessValues(=*uninitalized*)
Defines the values for the different stiffness (first value corresponds to elastic stiffness kn).

1.3 Global engines

1.3.1 GlobalEngine



class `yade.wrapper.GlobalEngine`(*inherits Engine → Serializable*)
Engine that will generally affect the whole simulation (contrary to PartialEngine).

class `yade.wrapper.CapillaryStressRecorder`(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Records information from capillary meniscii on samples submitted to triaxial compressions. -> New formalism needs to be tested!!!

class `yade.wrapper.CircularFactory`(*inherits SpheresFactory → GlobalEngine → Engine → Serializable*)
Circular geometry of the SpheresFactory region. It can be disk (given by radius and center), or cylinder (given by radius, length and center).

center(=*Vector3r(NaN, NaN, NaN)*)
Center of the region

length(=*0*)
Length of the cylindrical region (0 by default)

radius(=*NaN*)
Radius of the region

class yade.wrapper.**CohesiveFrictionalContactLaw**(*inherits GlobalEngine → Engine → Serializable*)
[DEPRECATED] Loop over interactions applying Law2_ScGeom6D_CohFrictPhys_CohesionMoment on all interactions.

Note: Use InteractionLoop and Law2_ScGeom6D_CohFrictPhys_CohesionMoment 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.

creepStiffness(=*10*)
...

creep_viscosity(=*false*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys...

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.

shear_creep2(=*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 yade.wrapper.**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 yade.wrapper.**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 yade.wrapper.**DomainLimiter**(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)
Delete particles that are out of axis-aligned box given by *lo* and *hi*.

hi(=*Vector3r(0, 0, 0)*)
Upper corner of the domain.

`lo`(=*Vector3r(0, 0, 0)*)
Lower corner of the domain.

`nDeleted`(=*0*)
Cummulative number of particles deleted.

class `yade.wrapper.DragForceApplier`(*inherits GlobalEngine → Engine → Serializable*)
Apply drag force on particles, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -\frac{v}{|v|}\frac{1}{2}\rho|v|^2 C_d A$$

where ρ is the medium density (density), v is particle's velocity, A is particle projected area (disc), C_d is the drag coefficient (0.47 for Sphere),

Note: Drag force is only applied to spherical particles.

Warning: Not tested.

`density`(=*0*)
Density of the medium.

class `yade.wrapper.ElasticContactLaw`(*inherits GlobalEngine → Engine → Serializable*)
[DEPRECATED] Loop over interactions applying Law2_ScGeom_FrictPhys_CundallStrack on all interactions.

Note: Use InteractionLoop and Law2_ScGeom_FrictPhys_CundallStrack 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)

class `yade.wrapper.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 `yade.wrapper.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.

`ids`(=*uninitalized*)
Lists of bodies whose state will be measured

class `yade.wrapper.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. In addition, reset those energies that should be reset, if energy tracing is enabled.

class yade.wrapper.**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 yade.wrapper.**InteractionLoop**(*inherits GlobalEngine → Engine → Serializable*)

Unified dispatcher for handling interaction loop at every step, for parallel performance reasons.

Special constructor

Constructs from 3 lists of Ig2, Ip2, Law functors respectively; they will be passed to interal dispatchers, which you might retrieve.

callbacks(*=uninitalized*)
Callbacks which will be called for every Interaction, if activated.

geomDispatcher(*=new IGeomDispatcher*)
IGeomDispatcher object that is used for dispatch.

lawDispatcher(*=new LawDispatcher*)
LawDispatcher object used for dispatch.

physDispatcher(*=new IPhysDispatcher*)
IPhysDispatcher object used for dispatch.

class yade.wrapper.**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 [Scholtes2009d] (lot of documentation)
2. in english [Scholtes2009b] (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 yade.wrapper.**MicroMacroAnalyser**(*inherits GlobalEngine → Engine → Serializable*)

Compute fabric tensor, local porosity, local deformation, and other micromechanicaly defined quantities based on triangulation/tesselation of the packing.

compDeformation(*=false*)
Is the engine just saving states or also computing and outputing deformations for each increment?

`compIncrt(=false)`
Should increments of force and displacements be defined on [n,n+1]? If not, states will be saved with only positions and forces (no displacements).

`incrtNumber(=1)`

`interval(=100)`
Number of timesteps between analyzed states.

`outputFile(="MicroMacroAnalysis")`
Base name for increment analysis output file.

`stateFileName(="state")`
Base name of state files.

`stateNumber(=0)`
A number incremented and appended at the end of output files to reflect increment number.

class `yade.wrapper.NewtonIntegrator(`*inherits GlobalEngine → Engine → Serializable*`)`
Engine integrating newtonian motion equations.

`damping(=0.2)`
damping coefficient for Cundall's non viscous damping (see [Chareyre2005]) [-]

`exactAsphericalRot(=true)`
Enable more exact body rotation integrator for aspherical bodies *only*, using formulation from [Allen1989], pg. 89.

`kinSplit(=false)`
Whether to separately track translational and rotational kinetic energy.

`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)*

`warnNoForceReset(=true)`
Warn when forces were not resetted in this step by ForceResetter; this mostly points to ForceResetter being forgotten incidentally and should be disabled only with a good reason.

class `yade.wrapper.ParticleSizeDistrbutionRPMRecorder(`*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*`)`
Store number of PSD in RPM model to file.

`numberCohesiveContacts(=0)`
Number of cohesive contacts found at last run. [-]

class `yade.wrapper.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 PyRunner, which derives from PeriodicEngine; likely to be encountered in python scripts):

```
PyRunner(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 yade.wrapper.**PyRunner**(*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 yade.wrapper.**QuadroFactory**(*inherits SpheresFactory → GlobalEngine → Engine → Serializable*)
Quadro geometry of the SpheresFactory region, given by extents and center

center(=*Vector3r(NaN, NaN, NaN)*)
Center of the region

extents(=*Vector3r(NaN, NaN, NaN)*)
Extents of the region

class yade.wrapper.**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.

addIterNum(=*false*)
Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

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 `yade.wrapper.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 attempts 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 `yade.wrapper.SpheresFactory`(*inherits GlobalEngine → Engine → Serializable*)
Engine for spitting spheres based on mass flow rate, particle size distribution etc. Initial velocity of particles is given by *vMin*, *vMax*, the *massFlowRate* determines how many particles to generate at each step. When *goalMass* is attained or positive *maxParticles* is reached, the engine does not produce particles anymore. Geometry of the region should be defined in a derived engine by overriden SpheresFactory::pickRandomPosition().

A sample script for this engine is in scripts/spheresFactory.py.

`goalMass`(=*0*)
Total mass that should be attained at the end of the current step. *(auto-updated)*

`massFlowRate`(=*NaN*)
Mass flow rate [kg/s]

`materialId`(=*-1*)
Shared material id to use for newly created spheres (can be negative to count from the end)

`maxAttempt`(=*5000*)
Maximum number of attempts to position a new sphere randomly.

`maxParticles`(=*100*)
The number of particles at which to stop generating new ones (regardless of massFlowRate

`normal`(=*Vector3r(NaN, NaN, NaN)*)
Spitting direction (and orientation of the region's geometry).

`numParticles`(=*0*)
Cummulative number of particles produces so far *(auto-updated)*

rMax(=*NaN*)
Maximum radius of generated spheres (uniform distribution)

rMin(=*NaN*)
Minimum radius of generated spheres (uniform distribution)

silent(=*false*)
If true no complain about excessing maxAttempt but disable the factory (by set massFlowRate=0).

totalMass(=*0*)
Mass of spheres that was produced so far. *(auto-updated)*

vAngle(=*NaN*)
Maximum angle by which the initial sphere velocity deviates from the nozzle normal.

vMax(=*NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

vMin(=*NaN*)
Minimum velocity norm of generated spheres (uniform distribution)

class yade.wrapper.**TesselationWrapper**(*inherits GlobalEngine → Engine → Serializable*)
Handle the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities : currently volume and porosity of each Voronoï sphere. Example script :

tt=TriaxialTest()

tt.generate('test.xml')

O.load('test.xml')

O.run(100) //for unknown reasons, this procedure crashes at iteration 0

TW=TesselationWrapper()

TW.triangulate() //compute regular Delaunay triangulation, don't construct tesselation TW.computeVolumes() //will silently tesselate the packing

TW.volume(10) //get volume associated to sphere of id 10

Note: This engine needs yade built with 'cgal' feature.

computeVolumes() → None
Compute volumes of all Voronoi's cells.

getVolPoroDef(*[(bool)deformation=False]*) → dict
Return a table with per-sphere computed quantities. Include deformations on the increment defined by states 0 and 1 if deformation=True (make sure to define states 0 and 1 consistently).

n_spheres(=*0*)
(auto-computed)

setState(*[(bool)state=0]*) → None
Make the current state the initial (0) or final (1) configuration for the definition of displacement increments, use only state=0 if you just want to get only volmumes and porosity.

triangulate(*[(bool)reset=True]*) → None
triangulate spheres of the packing

volume(*[(int)id=0]*) → float
Returns the volume of Voronoi's cell of a sphere.

class yade.wrapper.**TetraVolumetricLaw**(*inherits GlobalEngine → Engine → Serializable*)
Calculate physical response of 2 tetrahedra in interaction, based on penetration configuration given by TTetraGeom.

class yade.wrapper.**TimeStepper**(*inherits GlobalEngine → Engine → Serializable*)
Engine defining time-step (fundamental class)

`active`(=*true*)
is the engine active?

`timeStepUpdateInterval`(=*1*)
dt update interval

class `yade.wrapper.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 `yade.wrapper.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.

`ascii`(=*false*)
Store data as readable text in the XML file (sets vtkXMLWriter data mode to `vtkXMLWriter::Ascii`, while the default is `Appended`

`compress`(=*false*)
Compress output XML files [experimental].

`fileName`(=*"“*)
Base file name; it will be appended with {spheres,intrs,facets}-243100.vtu (unless *multiblock* is `True`) 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). `all` (the default value) enables all base and generic recorders.

Base recorders

Base recorders save the geometry (unstructured grids) on which other data is defined. They are implicitly activated by many of the other recorders. Each of them creates a new file (or a block, if multiblock is set).

`spheres` Saves positions and radii (`radii`) of spherical particles.

`facets` Save facets positions (vertices).

`intr` Store interactions as lines between nodes at respective particles positions. Additionally stores magnitude of normal (`forceN`) and shear (`absForceT`) forces on interactions (the geom).

Generic recorders

Generic recorders do not depend on specific model being used and save commonly useful data.

`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**.

stress Saves stresses of spheres and of facets as Vector3 and length; only active if **spheres** or **facets** are activated.

Specific recorders

The following should only be activated in appropriate cases, otherwise crashes can occur due to violation of type presuppositions.

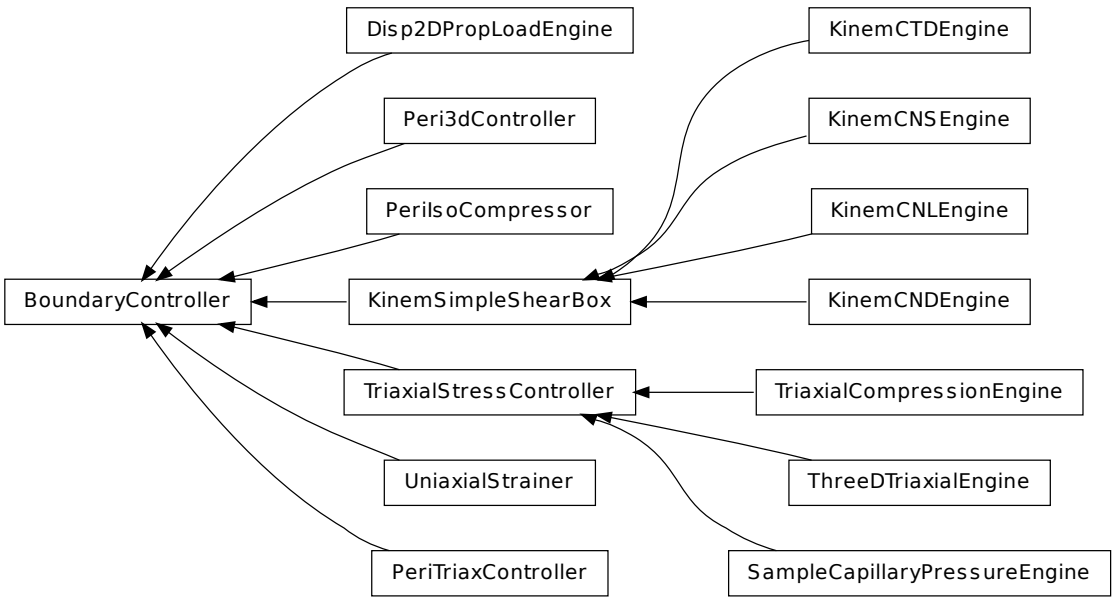
cpm Saves data pertaining to the concrete model: **cpmDamage** (normalized residual strength averaged on particle), **cpmSigma** (stress on particle, normal components), **cpmTau** (shear components of stress on particle), **cpmSigmaM** (mean stress around particle); **intr** is activated automatically by **cpm**

rpm Saves data pertaining to the rock particle model: **rpmSpecNum** shows different pieces of separated stones, only ids. **rpmSpecMass** shows masses of separated stones.

skipFacetIntr(=*true*)
Skip interactions with facets, when saving interactions

skipNondynamic(=*false*)
Skip non-dynamic spheres (but not facets).

1.3.2 BoundaryController



class yade.wrapper.**BoundaryController**(*inherits GlobalEngine → Engine → Serializable*)
Base for engines controlling boundary conditions of simulations. Not to be used directly.

class yade.wrapper.**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 `yade.wrapper.KinemCNDEngine`(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)

To apply a Constant Normal Displacement (CND) shear for a parallelogram box

This engine, designed for simulations implying a simple shear box (SimpleShear Preprocessor or scripts/simpleShear.py), 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.

`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]

`shearSpeed`(=*0.0*)
: the speed at which the shear is performed : speed of the upper plate [m/s]

class `yade.wrapper.KinemCNLEngine`(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)

To apply a constant normal stress shear (i.e. Constant Normal Load : CNL) for a parallelogram box (simple shear box : SimpleShear Preprocessor or scripts/simpleShear.py)

This engine 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 chosen 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.

`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]

`shearSpeed`(*=0.0*)
the speed at wich the shearing is performed : speed of the upper plate [m/s]

class `yade.wrapper.KinemCNSEngine`(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)

To apply a Constant Normal Stifness (CNS) shear for a parallelogram box (simple shear)

This engine, useable in simulations implying one deformable parallelepipedic box, 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

`KnC`(*=10.0e6*)
the normal rigidity chosen by the user [MPa/mm] - the conversion in Pa/m will be made

`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]

shearSpeed(*=0.0*)
the speed at wich the shearing is performed : speed of the upper plate [m/s]

class yade.wrapper.**KinemCTDEngine**(*inherits KinemSimpleShearBox → 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 (thus, the CTD = Constant Tangential Displacement). The lateral boxes move also to keep always contact. All that until this box is submitted to a given stress (=*targetSigma*). Moreover saves are executed at each value of stresses stored in the vector *sigma_save*, and at *targetSigma*

compSpeed(*=0.0*)
(vertical) speed of the upper box : >0 for real compression, <0 for unloading [m/s]

sigma_save(*=uninitalized*)
vector with the values of sigma at which a save of the simulation should be performed [kPa]

targetSigma(*=0.0*)
the value of sigma at which the compression should stop [kPa]

class yade.wrapper.**KinemSimpleShearBox**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
This class is supposed to be a mother class for all Engines performing loadings on the simple shear box of SimpleShear. It is not intended to be used by itself, but its declaration and implentation will thus contain all what is useful for all these Engines. The script simpleShear.py illustrates the use of the various corresponding Engines.

Key(*=""*)
string to add at the names of the saved files

LOG(*=false*)
boolean controling the output of messages on the screen

alpha(*=Mathr::PI/2.0*)
the angle from the lower box to the left box (trigo wise). Measured by this Engine, not to be changed by the user.

f0(*=0.0*)
the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of KinemCNSEngine or KinemCNLEngine will be done according to this initial value [N]. Not to be changed by the user.]

firstRun(*=true*)
boolean set to false as soon as the engine has done its job one time : useful to know if initial height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Not to be changed by the user.

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 control σ (CNL or CNS cases) [m/s]

temoin_save(=*uninitalized*)
vector (same length as 'gamma_save' for ex), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0). Not to be changed by the user.

wallDamping(=*0.2*)
the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

y0(=*0.0*)
the height of the upper plate at the very first time step : the engine finds its value [m]. Not to be changed by the user.

class yade.wrapper.**Peri3dController**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
Class for controlling independently all 6 components of "engineering" stress and strain of periodic :yref:"Cell'. goal are the goal values, while stressMask determines which components prescribe stress and which prescribe strain.

If the strain is prescribed, appropeiate strain rate is directly applied. If the stress is prescribed, the strain predictor is used: from stress values in two previous steps the value of strain rate is prescribed so as the value of stress in the next step is as close as possible to the ideal one. Current algorithm is extremly simple and probably will be changed in future, but is roboust enough and mostly works fine.

Stress error (difference between actual and ideal stress) is evaluated in current and previous steps ($d\sigma_i, d\sigma_{i-1}$. Linear extrapolation is used to estimate error in the next step

$$d\sigma_{i+1} = 2d\sigma_i - d\sigma_{i-1}$$

According to this error, the strain rate is modified by mod parameter

$$d\sigma_{i+1} \begin{cases} > 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i - \max(\mathrm{abs}(\dot{\boldsymbol{\varepsilon}}_i)) \cdot \mathrm{mod} \\ < 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i + \max(\mathrm{abs}(\dot{\boldsymbol{\varepsilon}}_i)) \cdot \mathrm{mod} \end{cases}$$

According to this fact, the prescribed stress will (almost) never have exact prescribed value, but the difference would be very small (and decreasing for increasing nSteps. This approach works good if one of the dominant strain rates is prescribed. If all stresses are prescribed or if all goal strains is prescribed as zero, a good estimation is needed for the first step, therefore the compliance matrix is estimated (from user defined estimations of macroscopic material parameters youngEstimation and poissonEstimation) and respective strain rates is computed form prescribed stress rates and compliance matrix (the estimation of compliance matrix could be computed autamatically avoiding user inputs of this kind).

The simulation on rotated periodic cell is also supported. Firstly, the polar decomposition is performed on cell's transformation matrix trsf $\mathcal{T} = \mathbf{UP}$, where $\mathbf{U}$ is orthogonal (unitary) matrix representing rotation and $\mathbf{P}$ is a positive semi-definite Hermitian matrix representing strain. A logarithm of $\mathbf{P}$ should be used to obtain realistic values at higher strain values (not implemented yet). A prescribed strain increment in global coordinates $dt \cdot \dot{\boldsymbol{\varepsilon}}$ is properly rotated to cell's local coordinates and added to $\mathbf{P}$

$$\mathbf{P}_{i+1} = \mathbf{P} + \mathbf{U}^T dt \cdot \dot{\boldsymbol{\varepsilon}} \mathbf{U}$$

The new value of trsf is computed at $\mathbf{T}_{i+1} = \mathbf{U}\mathbf{P}_{i+1}$. From current and next trsf the cell's velocity gradient velGrad is computed (according to its definition) as

$$\mathbf{V} = (\mathbf{T}_{i+1}\mathbf{T}^{-1} - \mathbf{I})/dt$$

Current implementation allow user to define independent loading "path" for each prescribed component. i.e. define the prescribed value as a function of time (or progress or steps). See Paths.

Examples scripts/test/peri3dController_example1 and scripts/test/peri3dController_triaxialCompression explain usage and inputs of Peri3dController, scripts/test/peri3dController_shear is an example of using shear components and also simulation on rotatd cell.

doneHook(=*uninitalized*)
Python command (as string) to run when nSteps is achieved. If empty, the engine will be set dead.

goal(=*Vector6r::Zero()*)
Goal state; only the upper triangular matrix is considered; each component is either prescribed stress or strain, depending on stressMask.

maxStrain(=*1e6*)
Maximal asolute value of strain allowed in the simulation. If reached, the simulation is considered as finished

maxStrainRate(=*1e3*)
Maximal absolute value of strain rate (both normal and shear components of strain)

mod(=*.1*)
Predictor modificator, by trail-and-error analysis the value 0.1 was found as the best.

nSteps(=*1000*)
Number of steps of the simulation.

poissonEstimation(=*.25*)
Estimation of macroscopic Poisson's ratio, used used for the first simulation step

progress(=*0.*)
Actual progress of the simulation with Controller.

strain(=*Vector6r::Zero()*)
Current strain (deformation) vector (ε_x,ε_y,ε_z,γ_{yz},γ_{zx},γ_{xy}) *(auto-updated)*.

strainRate(=*Vector6r::Zero()*)
Current strain rate vector.

stress(=*Vector6r::Zero()*)
Current stress vector (σ_x,σ_y,σ_z,τ_{yz},τ_{zx},τ_{xy})|yupdate|.

stressIdeal(=*Vector6r::Zero()*)
Ideal stress vector at current time step.

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).

stressRate(=*Vector6r::Zero()*)
Current stress rate vector (that is prescribed, the actual one slightly differ).

xxPath
"Time function" (piecewise linear) for xx direction. Sequence of couples of numbers. First number is time, second number desired value of respective quantity (stress or strain). The last couple is considered as final state (equal to (nSteps, goal)), other values are relative to this state.

Example: nSteps=1000, goal[0]=300, xxPath=((2,3),(4,1),(5,2))

at step 400 (=5*1000/2) the value is 450 (=3*300/2),

at step 800 (=4*1000/5) the value is 150 (=1*300/2),

at step 1000 (=5*1000/5=nSteps) the value is 300 (=2*300/2=goal[0]).

See example scripts/test/peri3dController_example1 for illusration.

xyPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for xy direction, see xxPath

youngEstimation(=*1e20*)
Estimation of macroscopic Young's modulus, used for the first simulation step

yyPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for yy direction, see xxPath

yzPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for yz direction, see xxPath

zxPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for zx direction, see xxPath

zzPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for zz direction, see xxPath

class yade.wrapper.**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 yade.wrapper.**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.

`externalWork`(=*0*)
Work input from boundary controller.

`globUpdate`(=*5*)
How often to recompute average stress, stiffness and unbalaced force.

`goal`
Desired stress or strain values (depending on stressMask), strains defined as `strain(i)=log(Fii)`.

> **Warning:** Strains are relative to the O.cell.refSize (reference cell size), not the current one (e.g. at the moment when the new strain value is set).

`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); if not set and dynCell is used, it will be computed as sum of masses of all particles.

`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 some constitutive laws (practicaly all laws based on ScGeom), normalForce and shearForce on interactions are in the reverse sense and this flag must be true (mandatory). 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 interactions if !dynCell)

class `yade.wrapper.SampleCapillaryPressureEngine`(*inherits* *TriaxialStressController* → *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)
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 `yade.wrapper.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 yade.wrapper.**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: Most of the algorithms used have been developed initialy for simulations reported in [Chareyre2002a] and [Chareyre2005]. They have been ported to Yade in a second step and used in e.g. [Kozicki2008],[Scholtes2009b]_,[Jerier2010b].

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*)
There are 5 possible states in which TriaxialCompressionEngine can be. See above wrapper.TriaxialCompressionEngine

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

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 yade.wrapper.**TriaxialStressController**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
An engine maintaining constant stresses on some boundaries of a parallepipedic packing. See also TriaxialCompressionEngine

Note: The algorithms used have been developed initialy for simulations reported in [Chareyre2002a] and [Chareyre2005]. They have been ported to Yade in a second step and used in e.g. [Kozicki2008],[Scholtes2009b]_,[Jerier2010b].

boxVolume
Total packing volume.

computeStressStrainInterval(=*10*)

depth(=*0*)
size of the box (2-axis) *(auto-updated)*

depth0(=*0*)
Reference size for strain definition. See TriaxialStressController::depth

externalWork(=*0*)
Energy provided by boundaries.|yupdate|

finalMaxMultiplier(=*1.00001*)
max multiplier of diameters during internal compaction (secondary precise adjustment - TriaxialStressController::maxMultiplier is used in the initial stage)

height(=*0*)
size of the box (1-axis) *(auto-updated)*

height0(=*0*)
Reference size for strain definition. See TriaxialStressController::height

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 (applies at each iteration and overrides user-set values of s1,2,3)

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. *(auto-updated)*

porosity
Porosity of the packing.

previousMultiplier(=*1*)
(auto-updated)

previousStress(=*0*)
(auto-updated)

radiusControlInterval(=*10*)

sigma1(=*0*)
prescribed stress on axis 1 (see TriaxialStressController::isAxisymetric)

sigma2(=*0*)
prescribed stress on axis 2 (see TriaxialStressController::isAxisymetric)

sigma3(=*0*)
prescribed stress on axis 3 (see TriaxialStressController::isAxisymetric)

`sigma_iso`(*=0*)
prescribed confining stress (see TriaxialStressController::isAxisymetric)

`spheresVolume`
Total volume pf spheres.

`stiffnessUpdateInterval`(*=10*)
target strain rate (./s)

`strain`
Current strain (logarithmic).

`stress`(*(int)id*) → Vector3
Return the mean stress vector acting on boundary 'id', with 'id' between 0 and 5.

`thickness`(*=-1*)
thickness of boxes (needed by some functions)

`volumetricStrain`(*=0*)
Volumetric strain (see TriaxialStressController::strain).|yupdate|

`wallDamping`(*=0.25*)
wallDamping coefficient - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

`wall_back_activated`(*=true*)
if true, the engine is keeping stress constant on this boundary.

`wall_back_id`(*=0*)
id of boundary ; coordinate 2-

`wall_bottom_activated`(*=true*)
if true, the engine is keeping stress constant on this boundary.

`wall_bottom_id`(*=0*)
id of boundary ; coordinate 1-

`wall_front_activated`(*=true*)
if true, the engine is keeping stress constant on this boundary.

`wall_front_id`(*=0*)
id of boundary ; coordinate 2+

`wall_left_activated`(*=true*)
if true, the engine is keeping stress constant on this boundary.

`wall_left_id`(*=0*)
id of boundary ; coordinate 0-

`wall_right_activated`(*=true*)
if true, the engine is keeping stress constant on this boundary.

`wall_right_id`(*=0*)
id of boundary ; coordinate 0+

`wall_top_activated`(*=true*)
if true, the engine is keeping stress constant on this boundary.

`wall_top_id`(*=0*)
id of boundary ; coordinate 1+

`width`(*=0*)
size of the box (0-axis) *(auto-updated)*

`width0`(*=0*)
Reference size for strain definition. See TriaxialStressController::width

class `yade.wrapper.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; must be given explicitly [m^2]

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?

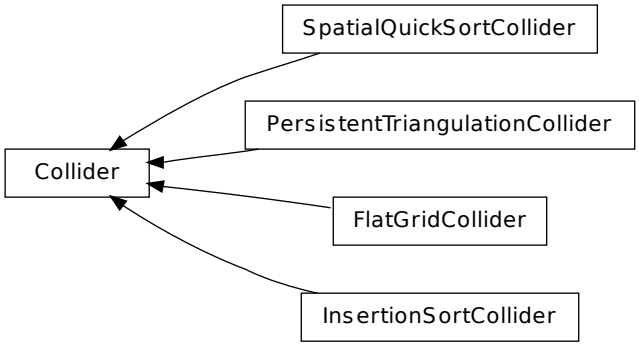
stopStrain(=*NaN*)
Strain at which we will pause simulation; inactive (nan) by default; must be reached from below (in absolute value)

strain(=*0*)
Current strain value, elongation/originalLength *(auto-updated)* [-]

strainRate(=*NaN*)
Rate of strain, starting at 0, linearly raising to strainRate. [-]

stressUpdateInterval(=*10*)
How often to recompute stress on supports.

1.3.3 Collider



class `yade.wrapper.Collider`(*inherits GlobalEngine → Engine → Serializable*)

Abstract class for finding spatial collisions between bodies.

Special constructor

Derived colliders (unless they override `pyHandleCustomCtorArgs`) can be given list of BoundFunctors which is used to initialize the internal boundDispatcher instance.

`boundDispatcher`(=*new BoundDispatcher*)

BoundDispatcher object that is used for creating bounds on collider's request as necessary.

class `yade.wrapper.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 `yade.wrapper.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 `verletDist` (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 `verletDist`; instead, they will be put to bins (in the statistical sense) based on magnitude of their velocity; `verletDist` 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(=2)`
: 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)

`dumpBounds()` → tuple
: Return representation of the internal sort data. The format is `([...],[...],[...])` for 3 axes, where each `...` is a list of entries (bounds). The entry is a tuple with the fllowing items:

  - coordinate (float)
  - body id (int), but negated for negative bounds
  - period numer (int), if the collider is in the periodic regime.

`fastestBodyMaxDist(=-1)`
: Maximum displacement of the fastest body since last run; if >= verletDist, 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(=5)`
: Number of velocity bins for striding. If <=0, bin-less strigin is used (this is however DEPRECATED).

`numReinit(=0)`
: Cummulative number of bound array re-initialization.

`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 verletDist, only on the computed stride. [DEPRECATED, is used only when bins are not used].

`verletDist`(*=-.05*, *Automatically initialized*)
: Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if zero. Negative value will trigger automatic computation, so that the real value will be **|verletDist|** × minimum spherical particle radius; if there are no spherical particles, it will be disabled.

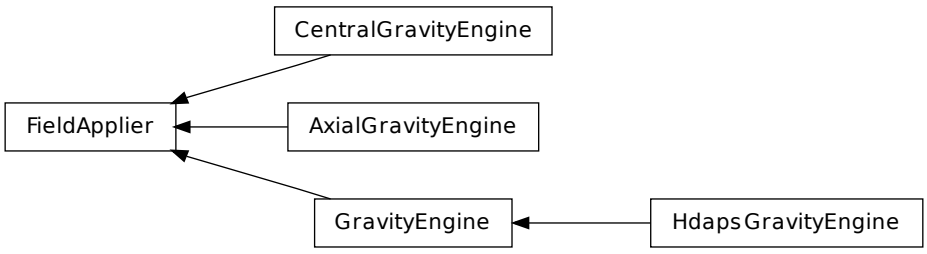
class `yade.wrapper.PersistentTriangulationCollider`(*inherits Collider → GlobalEngine → Engine → Serializable*)
Collision detection engine based on regular triangulation. Handles spheres and flat boundaries (considered as infinite-sized bounding spheres).

`haveDistantTransient`(*=false*)
: Keep distant interactions? If True, don't delete interactions once bodies don't overlap anymore; constitutive laws will be responsible for requesting deletion. If False, delete as soon as there is no object penetration.

class `yade.wrapper.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.

1.3.4 FieldApplier



class `yade.wrapper.FieldApplier`(*inherits GlobalEngine → Engine → Serializable*)
Base for engines applying force files on particles. Not to be used directly.

class `yade.wrapper.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 `yade.wrapper.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 `yade.wrapper.GravityEngine`(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)
Engine applying constant acceleration to all bodies.

`gravity`(=*Vector3r::Zero()*)
: Acceleration [kgms ^{2}]

class `yade.wrapper.HdapsGravityEngine`(*inherits GravityEngine → FieldApplier → GlobalEngine → Engine → Serializable*)
Read accelerometer in Thinkpad laptops (HDAPS and accordingly set gravity within the simulation. This code draws from hdaps-gl . See scripts/test/hdaps.py for an example.

`accel`(=*Vector2i::Zero()*)
: reading from the sysfs file

`calibrate`(=*Vector2i::Zero()*)
: Zero position; if NaN, will be read from the *hdapsDir* / calibrate.

`calibrated`(=*false*)
: Whether *calibrate* was already updated. Do not set to `True` by hand unless you also give a meaningful value for *calibrate*.

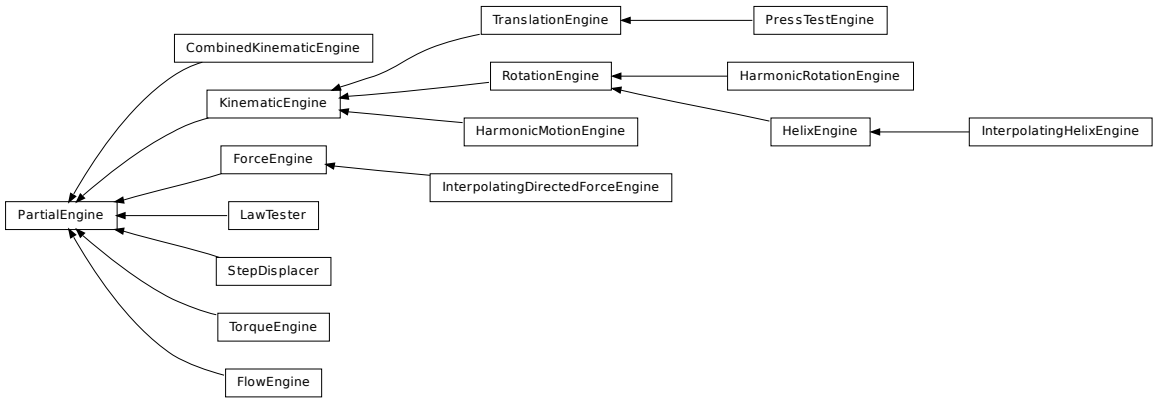
`hdapsDir`(=*"/sys/devices/platform/hdaps"*)
: Hdaps directory; contains `position` (with accelerometer readings) and `calibration` (zero acceleration).

`msecUpdate`(=*50*)
: How often to update the reading.

`updateThreshold`(=*4*)
: Minimum difference of reading from the file before updating gravity, to avoid jitter.

`zeroGravity`(=*Vector3r(0, 0, -1)*)
: Gravity if the accelerometer is in flat (zero) position.

1.4 Partial engines



class `yade.wrapper.PartialEngine`(*inherits Engine → Serializable*)
Engine affecting only particular bodies in the simulation, defined by *ids*.

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

class `yade.wrapper.CombinedKinematicEngine`(*inherits PartialEngine → Engine → Serializable*)
Engine for applying combined displacements on pre-defined bodies. Constructed using `+` operator on regular KinematicEngines. The `ids` operated on are those of the first engine in the combination (assigned automatically).

`comb`(=*uninitalized*)
Kinematic engines that will be combined by this one, run in the order given.

class `yade.wrapper.FlowEngine`(*inherits PartialEngine → Engine → Serializable*)
An engine to solve flow problem in saturated granular media

`BACK_Boundary_MaxMin`(=*1*)
If true bounding sphere is added as function fo max/min sphere coord, if false as function of yade wall position

`BOTTOM_Boundary_MaxMin`(=*1*)
If true bounding sphere is added as function fo max/min sphere coord, if false as function of yade wall position

`CachedForces`(=*true*)
Des/Activate the cached forces calculation

`Debug`(=*false*)
Activate debug messages

`EpsVolPercent_RTRG`(=*0.01*)
Percentuage of cumulate eps_vol at which retriangulation of pore space is performed

`FRONT_Boundary_MaxMin`(=*1*)
If true bounding sphere is added as function fo max/min sphere coord, if false as function of yade wall position

`Flow_imposed_BACK_Boundary`(=*true*)
if false involve pressure imposed condition

`Flow_imposed_BOTTOM_Boundary`(=*true*)
if false involve pressure imposed condition

`Flow_imposed_FRONT_Boundary`(=*true*)
if false involve pressure imposed condition

`Flow_imposed_LEFT_Boundary(=true)`
if false involve pressure imposed condition

`Flow_imposed_RIGHT_Boundary(=true)`
if false involve pressure imposed condition

`Flow_imposed_TOP_Boundary(=true)`
if false involve pressure imposed condition

`K(=0)`
Permeability of the sample

`LEFT_Boundary_MaxMin(=1)`
If true bounding sphere is added as function fo max/min sphere coord, if false as function of yade wall position

`MaxPressure(=0)`
Maximal value of water pressure within the sample

`P_zero(=0)`
Initial internal pressure for oedometer test

`PermuteInterval(=100000)`
Pore space re-triangulation period

`Pressure_BACK_Boundary(=0)`
Pressure imposed on back boundary

`Pressure_BOTTOM_Boundary(=0)`
Pressure imposed on bottom boundary

`Pressure_FRONT_Boundary(=0)`
Pressure imposed on front boundary

`Pressure_LEFT_Boundary(=0)`
Pressure imposed on left boundary

`Pressure_RIGHT_Boundary(=0)`
Pressure imposed on right boundary

`Pressure_TOP_Boundary(=0)`
Pressure imposed on top boundary

`RIGHT_Boundary_MaxMin(=1)`
If true bounding sphere is added as function fo max/min sphere coord, if false as function of yade wall position

`Relax(=1.9)`
Gauss-Seidel relaxation

`Sinus_Amplitude(=0)`
Pressure value (amplitude) when sinusoidal pressure is applied

`Sinus_Average(=0)`
Pressure value (average) when sinusoidal pressure is applied

`TOP_Boundary_MaxMin(=1)`
If true bounding sphere is added as function fo max/min sphere coord, if false as function of yade wall position

`Tolerance(=1e-06)`
Gauss-Seidel Tolerance

`Update_Triangulation(=0)`
If true the medium is retriangulated

`WaveAction(=false)`
Allow sinusoidal pressure condition to simulate ocean waves

`blocked_grains(=false)`
Grains will/won't be moved by forces

bottom_seabed_pressure(*=0*)
Fluid pressure measured at the bottom of the seabed on the symmetry axe

clearImposedPressure() → None
Clear the list of points with pressure imposed.

compute_K(*=false*)
Activates permeability measure within a granular sample

consolidation(*=false*)
Enable/Disable storing consolidation files

currentStrain(*=0*)
Current value of axial strain

currentStress(*=0*)
Current value of axial stress

eps_vol_max(*=0*)
Maximal absolute volumetric strain computed at each iteration

first(*=true*)
Controls the initialization/update phases

getFlux(*(int)cond*) → float
Get influx in cell associated to an imposed P (indexed using 'cond').

id_sphere(*=0*)
Average velocity will be computed for all cells incident to that sphere

imposePressure(*(Vector3)pos*, *(float)p*) → None
Impose pressure in cell of location 'pos'.

intervals(*=30*)
Number of layers for pressure measurements

isActivated(*=true*)
Activates Flow Engine

liquefaction(*=false*)
Compute bottom_seabed_pressure if true, see below

loadFactor(*=1.1*)
Load multiplicator for oedometer test

meanK_correction(*=true*)
Local permeabilities' correction through meanK threshold

meanK_opt(*=false*)
Local permeabilities' correction through an optimized threshold

permeability_factor(*=1.0*)
a permability multiplicator

porosity(*=0*)
Porosity computed at each retriangulation

save_mgpost(*=false*)
Enable/disable mgpost file creation

save_mplot(*=false*)
Enable/disable mplot files creation

save_vtk(*=false*)
Enable/disable vtk files creation for visualization

slice_pressures(*=false*)
Enable/Disable slice pressure measurement

slip_boundary(*=true*)
Controls friction condition on lateral walls

`useSolver`(=*0*)
Solver to use

`velocity_profile`(=*false*)
Enable/Disable slice velocity measurement

class `yade.wrapper.ForceEngine`(*inherits PartialEngine → Engine → Serializable*)
Apply contact force on some particles at each step.

`force`(=*Vector3r::Zero()*)
Force to apply.

class `yade.wrapper.HarmonicMotionEngine`(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)
This engine implements the harmonic oscillation of bodies. http://en.wikipedia.org/wiki/Simple_harmonic_motion#Dynamics_of_simple_harmonic_motion

`A`(=*Vector3r::Zero()*)
Amplitude [m]

`f`(=*Vector3r::Zero()*)
Frequency [hertz]

`fi`(=*Vector3r(Mathr::PI/2.0, Mathr::PI/2.0, Mathr::PI/2.0)*)
Initial phase [radians]. By default, the body oscillates around initial position.

class `yade.wrapper.HarmonicRotationEngine`(*inherits RotationEngine → KinematicEngine → PartialEngine → Engine → Serializable*)
This engine implements the harmonic-rotation oscillation of bodies. http://en.wikipedia.org/wiki/Simple_harmonic_motion#Dynamics_of_simple_harmonic_motion ; please, set dynamic=False for bodies, droven by this engine, otherwise amplitude will be 2x more, than awaited.

`A`(=*0*)
Amplitude [rad]

`f`(=*0*)
Frequency [hertz]

`fi`(=*Mathr::PI/2.0*)
Initial phase [radians]. By default, the body oscillates around initial position.

class `yade.wrapper.HelixEngine`(*inherits RotationEngine → KinematicEngine → PartialEngine → Engine → Serializable*)
Engine applying both rotation and translation, along the same axis, whence the name HelixEngine

`angleTurned`(=*0*)
How much have we turned so far. *(auto-updated)* [rad]

`linearVelocity`(=*0*)
Linear velocity [m/s]

class `yade.wrapper.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 `yade.wrapper.InterpolatingHelixEngine`(*inherits HelixEngine → RotationEngine → KinematicEngine → 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 `yade.wrapper.KinematicEngine`(*inherits PartialEngine → Engine → Serializable*)
Abstract engine for applying prescribed displacement.

Note: Derived classes should override the `apply` with given list of `ids` (not `action` with PartialEngine.ids), so that they work when combined together; velocity and angular velocity of all subscribed bodies is reset before the `apply` method is called, it should therefore only increment those quantities.

class `yade.wrapper.LawTester`(*inherits PartialEngine → Engine → Serializable*)
Prescribe and apply deformations of an interaction in terms of local mutual displacements and rotations. The loading path is specified either using path (as sequence of 6-vectors containing generalized displacements u_x, u_y, u_z, φ_x, φ_y, φ_z) or disPath (u_x, u_y, u_z) and rotPath (φ_x, φ_y, φ_z). Time function with time values (step numbers) corresponding to points on loading path is given by pathSteps. Loading values are linearly interpolated between given loading path points, and starting zero-value (the initial configuration) is assumed for both path and pathSteps. hooks can specify python code to run when respective point on the path is reached; when the path is finished, doneHook will be run.

LawTester should be placed between InteractionLoop and NewtonIntegrator in the simulation loop, since it controls motion via setting linear/angular velocities on particles; those velocities are integrated by NewtonIntegrator to yield an actual position change, which in turn causes IGeom to be updated (and contact law applied) when InteractionLoop is executed. Constitutive law generating forces on particles will not affect prescribed particle motion, since both particles have all DoFs blocked when first used with LawTester.

LawTester uses, as much as possible, IGeom to provide useful data (such as local coordinate system), but is able to compute those independently if absent in the respective IGeom:

IGeom	#DoFs	LawTester support level
L3Geom	3	full
L6Geom	6	full
ScGeom	3	emulate local coordinate system
ScGeom6D	6	emulate local coordinate system
Dem3DofGeom	3	*not supported*

Depending on IGeom, 3 (u_x, u_y, u_z) or 6 (u_x, u_y, u_z, φ_x, φ_y, φ_z) degrees of freedom (DoFs) are controlled with LawTester, by prescribing linear and angular velocities of both particles in contact. All DoFs controlled with LawTester are orthogonal (fully decoupled) and are controlled independently.

When 3 DoFs are controlled, rotWeight controls whether local shear is applied by moving particle on arc around the other one, or by rotating without changing position; although such rotation induces mutual rotation on the interaction, it is ignored with IGeom with only 3 DoFs. When 6 DoFs are controlled, only arc-displacement is applied for shear, since otherwise mutual rotation would occur.

idWeight distributes prescribed motion between both particles (resulting local deformation is the same if `id1` is moved towards `id2` or `id2` towards `id1`). This is true only for u_x, u_y, u_z, φ_x however ; bending rotations φ_y, φ_z are nevertheless always distributed regardless of `idWeight` to both spheres in inverse proportion to their radii, so that there is no shear induced.

LawTester knows current contact deformation from 2 sources: from its own internal data (which are used for prescribing the displacement at every step), which can be accessed in uTest, and from IGeom itself (depending on which data it provides), which is stored in uGeom. These two values should be identical (disregarding numerical percision), and it is a way to test whether IGeom and related functors compute what they are supposed to compute.

LawTester-operated interactions can be rendered with GlExtra_LawTester renderer.

See scripts/test/law-test.py for an example.

`disPath`(=*uninitalized*)
: Loading path, where each Vector3 contains desired normal displacement and two components of the shear displacement (in local coordinate system, which is being tracked automatically. If shorter than rotPath, the last value is repeated.

`displIsRel`(=*true*)
: Whether displacement values in *disPath* are normalized by reference contact length (r1+r2 for 2 spheres).

`doneHook`(=*uninitalized*)
: Python command (as string) to run when end of the path is achieved. If empty, the engine will be set dead.

`hooks`(=*uninitalized*)
: Python commands to be run when the corresponding point in path is reached, before doing other things in that particular step. See also doneHook.

`idWeight`(=*1*)
: Float, usually ⟨0,1⟩, determining on how are displacements distributed between particles (0 for id1, 1 for id2); intermediate values will apply respective part to each of them. This parameter is ignored with 6-DoFs IGeom.

`pathSteps`(=*vector<int>(1, 1), (constant step)*)
: Step number for corresponding values in path; if shorter than path, distance between last 2 values is used for the rest.

`refLength`(=*0*)
: Reference contact length, for rendering only.

`renderLength`(=*0*)
: Characteristic length for the purposes of rendering, set equal to the smaller radius.

`rotPath`(=*uninitalized*)
: Rotational components of the loading path, where each item contains torsion and two bending rotations in local coordinates. If shorter than path, the last value is repeated.

`rotWeight`(=*1*)
: Float ⟨0,1⟩ determining whether shear displacement is applied as rotation or displacement on arc (0 is displacement-only, 1 is rotation-only). Not effective when mutual rotation is specified.

`step`(=*1*)
Step number in which this engine is active; determines position in path, using pathSteps.

`trsf`(=*uninitalized*)
Transformation matrix for the local coordinate system. *(auto-updated)*

`uGeom`(=*Vector6r::Zero()*)
Current generalized displacements (3 displacements, 3 rotations), as stored in the interation itself. They should corredpond to uTest, otherwise a bug is indicated.

`uTest`(=*Vector6r::Zero()*)
Current generalized displacements (3 displacements, 3 rotations), as they should be according to this LawTester. Should correspond to uGeom.

`uuPrev`(=*Vector6r::Zero()*)
Generalized displacement values reached in the previous step, for knowing which increment to apply in the current step.

class `yade.wrapper.PressTestEngine`(*inherits TranslationEngine → KinematicEngine → 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 `yade.wrapper.RotationEngine`(*inherits KinematicEngine → 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 `yade.wrapper.StepDisplacer`(*inherits PartialEngine → Engine → Serializable*)
Apply generalized displacement (displacement or rotation) stepwise on subscribed bodies. Could be used for purposes of contact law tests (by moving one sphere compared to an other), but in this case, see rather LawTester

`mov`(=*Vector3r::Zero()*)
Linear displacement step to be applied per iteration, by addition to State.pos.

`rot`(=*Quaternionr::Identity()*)
Rotation step to be applied per iteration (via rotation composition with State.ori).

`setVelocities`(=*false*)
If false, positions and orientations are directly updated, without changing the speeds of concerned bodies. If true, only velocity and angularVelocity are modified. In this second case integrator is supposed to be used, so that, thanks to this Engine, the bodies will have the prescribed jump over one iteration (dt).

class `yade.wrapper.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.

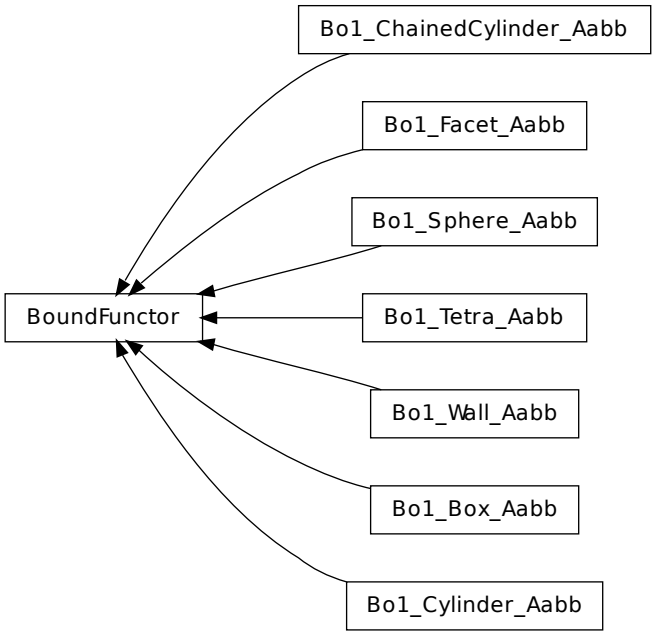
class yade.wrapper.**TranslationEngine**(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)
This engine is the base class for different engines, which require any kind of motion.

translationAxis(=*uninitalized*)
Direction [Vector3]

velocity(=*uninitalized*)
Velocity [m/s]

1.5 Bounding volume creation

1.5.1 BoundFunctor



class yade.wrapper.**BoundFunctor**(*inherits Functor → Serializable*)
Functor for creating/updating Body::bound.

class yade.wrapper.**Bo1_Box_Aabb**(*inherits BoundFunctor → Functor → Serializable*)
Create/update an Aabb of a Box.

class yade.wrapper.**Bo1_ChainedCylinder_Aabb**(*inherits BoundFunctor → Functor → Serializable*)
Functor creating Aabb from ChainedCylinder.

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 IGeomFunctor which will not simply discard such interactions: Ig2_Cylinder_Cylinder_Dem3DofGeom::distFactor / Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class yade.wrapper.**Bo1_Cylinder_Aabb**(*inherits BoundFunctor → Functor → Serializable*)
Functor creating Aabb from Cylinder.

`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 IGeomFunctor which will not simply discard such interactions: Ig2_Cylinder_Cylinder_-Dem3DofGeom::distFactor / Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class `yade.wrapper.Bo1_Facet_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Creates/updates an Aabb of a Facet.

class `yade.wrapper.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 IGeomFunctor 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 `yade.wrapper.Bo1_Tetra_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Create/update Aabb of a Tetra

class `yade.wrapper.Bo1_Wall_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Creates/updates an Aabb of a Wall

1.5.2 BoundDispatcher

class `yade.wrapper.BoundDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`activated`(=*true*)
Whether the engine is activated (only should be changed by the collider)

`dispFunctor`(*(Shape)arg2*) → BoundFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

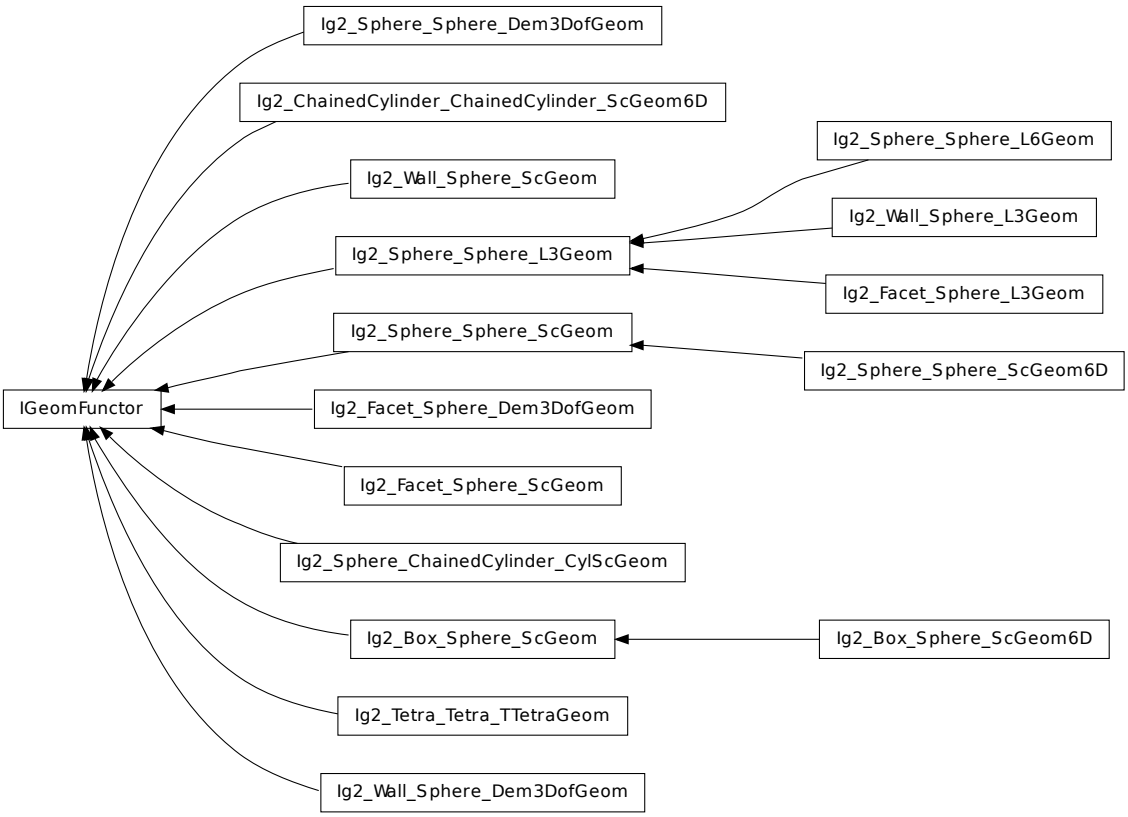
`dispMatrix`([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

`functors`
Functors 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).

1.6 Interaction Geometry creation

1.6.1 IGeomFunctor



class `yade.wrapper.IGeomFunctor`(*inherits Functor → Serializable*)
Functor for creating/updating Interaction::geom objects.

class `yade.wrapper.Ig2_Box_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)
Create an interaction geometry ScGeom from Box and Sphere, representing the box with a projected virtual sphere of same radius.

class `yade.wrapper.Ig2_Box_Sphere_ScGeom6D`(*inherits Ig2_Box_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)
Create an interaction geometry ScGeom6D from Box and Sphere, representing the box with a projected virtual sphere of same radius.

class `yade.wrapper.Ig2_ChainedCylinder_ChainedCylinder_ScGeom6D`(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a ScGeom instance representing connexion between chained cylinders.

`interactionDetectionFactor`(*=1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

class `yade.wrapper.Ig2_Facet_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → 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 `yade.wrapper.Ig2_Facet_Sphere_L3Geom`(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)
Incrementally compute L3Geom for contact between Facet and Sphere. Uses attributes of Ig2_Sphere_Sphere_L3Geom.

class `yade.wrapper.Ig2_Facet_Sphere_ScGeom`(*inherits IGeomFunctor → 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 `yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of two Spheres.

`interactionDetectionFactor`(*=1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

class `yade.wrapper.Ig2_Sphere_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → 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 `yade.wrapper.Ig2_Sphere_Sphere_L3Geom`(*inherits IGeomFunctor → Functor → Serializable*)

Functor for computing incrementally configuration of 2 Spheres stored in L3Geom; the configuration is positioned in global space by local origin $\mathbf{c}$ (contact point) and rotation matrix $\mathbf{T}$ (orthonormal transformation matrix), and its degrees of freedom are local displacement $\mathbf{u}$ (in one normal and two shear directions); with Ig2_Sphere_Sphere_L6Geom and L6Geom, there is additionally $\boldsymbol{\varphi}$. The first row of $\mathbf{T}$, i.e. local x-axis, is the contact normal noted $\mathbf{n}$ for brevity. Additionally, quasi-constant values of $\mathbf{u}_0$ (and $\boldsymbol{\varphi}_0$) are stored as shifted origins of $\mathbf{u}$ (and $\boldsymbol{\varphi}$); therefore, current value of displacement is always $\mathbf{u}^\circ - \mathbf{u}_0$.

Suppose two spheres with radii r_i, positions $\mathbf{x}_i$, velocities $\boldsymbol{v}_i$, angular velocities $\boldsymbol{\omega}_i$.

When there is not yet contact, it will be created if $u_N = |\mathbf{x}_2^\circ - \mathbf{x}_1^\circ| - |f_d|(r_1 + r2) < 0$, where f_d is distFactor (sometimes also called "interaction radius"). If $f_d > 0$, then $\mathbf{u}_{0x}$ will be initalized to u_N, otherwise to 0. In another words, contact will be created if spheres enlarged by $|f_d|$ touch, and the "equilibrium distance" (where $\mathbf{u}_x - \mathbf{u} - 0x$ is zero) will be set to the current distance if f_d is positive, and to the geometrically-touching distance if negative.

Local axes (rows of $\mathbf{T}$) are initially defined as follows:

- local x-axis is $\mathbf{n} = \mathbf{x}_l = \widehat{\mathbf{x}_2 - \mathbf{x}_1}$;
- local y-axis positioned arbitrarily, but in a deterministic manner: aligned with the xz plane (if $\mathbf{n}_y < \mathbf{n}_z$) or xy plane (otherwise);
- local z-axis $\mathbf{z}_l = \mathbf{x}_l \times \mathbf{y}_l$.

If there has already been contact between the two spheres, it is updated to keep track of rigid motion of the contact (one that does not change mutual configuration of spheres) and mutual configuration changes. Rigid motion transforms local coordinate system and can be decomposed in rigid translation (affecting $\mathbf{c}$), and rigid rotation (affecting $\mathbf{T}$), which can be split in rotation $\mathbf{o}_r$ perpendicular to the normal and rotation $\mathbf{o}_t$ ("twist") parallel with the normal:

$$\mathbf{o}_r^{\ominus} = \mathbf{n}^- \times \mathbf{n}^\circ.$$

Since velocities are known at previous midstep $(t - \Delta t/2)$, we consider mid-step normal

$$\mathbf{n}^{\ominus} = \frac{\mathbf{n}^- + \mathbf{n}^\circ}{2}.$$

For the sake of numerical stability, $\mathbf{n}^{\ominus}$ is re-normalized after being computed, unless prohibited by approxMask. If approxMask has the appropriate bit set, the mid-normal is not compute, and we simply use $\mathbf{n}^{\ominus} \approx \mathbf{n}^{-}$.

Rigid rotation parallel with the normal is

$$\mathbf{o}_t^{\ominus} = \mathbf{n}^{\ominus} \left(\mathbf{n}^{\ominus} \cdot \frac{\boldsymbol{\omega}_1^{\ominus} + \boldsymbol{\omega}_2^{\ominus}}{2} \right) \Delta t.$$

Branch vectors $\mathbf{b}_1$, $\mathbf{b}_2$ (connecting $\mathbf{x}_1^{\circ}$, $\mathbf{x}_2^{\circ}$ with $\mathbf{c}^{\circ}$ are computed depending on noRatch (see here).

$$\mathbf{b}_1 = \begin{cases} r_1 \mathbf{n}^{\circ} & \text{with } \texttt{noRatch} \\ \mathbf{c}^{\circ} - \mathbf{x}_1^{\circ} & \text{otherwise} \end{cases}$$

$$\mathbf{b}_2 = \begin{cases} -r_2 \mathbf{n}^{\circ} & \text{with } \texttt{noRatch} \\ \mathbf{c}^{\circ} - \mathbf{x}_2^{\circ} & \text{otherwise} \end{cases}$$

Relative velocity at $\mathbf{c}^{\circ}$ can be computed as

$$\boldsymbol{v}_r^{\ominus} = (\tilde{\boldsymbol{v}}_2^{\ominus} + \boldsymbol{\omega}_2 \times \mathbf{b}_2) - (\boldsymbol{v}_1 + \boldsymbol{\omega}_1 \times \mathbf{b}_1)$$

where $\tilde{\boldsymbol{v}}_2$ is $\boldsymbol{v}_2$ without mean-field velocity gradient in periodic boundary conditions (see Cell.homoDeform). In the numerial implementation, the normal part of incident velocity is removed (since it is computed directly) with $\boldsymbol{v}_{r2}^{\ominus} = \boldsymbol{v}_r^{\ominus} - (\mathbf{n}^{\ominus} \cdot \boldsymbol{v}_r^{\ominus})\mathbf{n}^{\ominus}$.

Any vector $\mathbf{a}$ expressed in global coordinates transforms during one timestep as

$$\mathbf{a}^{\circ} = \mathbf{a}^{-} + \boldsymbol{v}_r^{\ominus} \Delta t - \mathbf{a}^{-} \times \mathbf{o}_r^{\ominus} - \mathbf{a}^{-} \times \mathbf{t}_r^{\ominus}$$

where the increments have the meaning of relative shear, rigid rotation normal to $\mathbf{n}$ and rigid rotation parallel with $\mathbf{n}$. Local coordinate system orientation, rotation matrix $\mathbf{T}$, is updated by rows, i.e.

$$\mathbf{T}^{\circ} = \begin{pmatrix} \mathbf{n}_x^{\circ} & \mathbf{n}_y^{\circ} & \mathbf{n}_z^{\circ} \\ \mathbf{T}_{1,\bullet}^{-} - \mathbf{T}_{1,\bullet}^{-} \times \mathbf{o}_r^{\ominus} - \mathbf{T}_{1,\bullet}^{-} \times \mathbf{o}_t^{\ominus} \\ \mathbf{T}_{2,\bullet}^{-} - \mathbf{T}_{2,\bullet}^{-} \times \mathbf{o}_r^{\ominus} - \mathbf{T}_{,\bullet}^{-} \times \mathbf{o}_t^{\ominus} \end{pmatrix}$$

This matrix is re-normalized (unless prevented by approxMask) and mid-step transformation is computed using quaternion spherical interpolation as

$$\mathbf{T}^{\ominus} = \mathrm{Slerp}\left(\mathbf{T}^{-}; \mathbf{T}^{\circ}; t = 1/2\right).$$

Depending on approxMask, this computation can be avoided by approximating $\mathbf{T}^{\ominus} = \mathbf{T}^{-}$.

Finally, current displacement is evaluated as

$$\mathbf{u}^{\circ} = u^{-} + \mathbf{T}^{\ominus} \boldsymbol{v}_r^{\ominus} \Delta t.$$

For the normal component, non-incremental evaluation is preferred, giving

$$\mathbf{u}_x^{\circ} = |\mathbf{x}_2^{\circ} - \mathbf{x}_1^{\circ}| - (r_1 + r_2)$$

If this functor is called for L6Geom, local rotation is updated as

$$\boldsymbol{\varphi}^{\circ} = \boldsymbol{\varphi}^{-} + \mathbf{T}^{\ominus}\Delta t(\boldsymbol{\omega}_2 - \boldsymbol{\omega}_1)$$

`approxMask`
Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1	use previous transformation to transform velocities (which are known at mid-steps), instead of mid-step transformation computed as quaternion slerp at t=0.5.
2	do not take average (mid-step) normal when computing relative shear displacement, use previous value instead
4	do not re-normalize average (mid-step) normal, if used....

By default, the mask is zero, wherefore none of these approximations is used.

`distFactor(=1)`
Create interaction if spheres are not futher than |distFactor|*(r1+r2). If negative, zero normal deformation will be set to be the initial value (otherwise, the geometrical distance is the 'zero' one).

`noRatch(=true)`
See Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting.

`trsfRenorm(=100)`
How often to renormalize trsf; if non-positive, never renormalized (simulation might be unstable)

class `yade.wrapper.Ig2_Sphere_Sphere_L6Geom`(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)
Incrementally compute L6Geom for contact of 2 spheres.

class `yade.wrapper.Ig2_Sphere_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a ScGeom instance representing the geometry of a contact point between two :yref:'Spheres<Sphere>'s.

`avoidGranularRatcheting`
Define relative velocity so that ratcheting is avoided. It applies for sphere-sphere contacts. It eventualy also apply for sphere-emulating interactions (i.e. convertible into the ScGeom type), if the virtual sphere's motion is defined correctly (see e.g. Ig2_Sphere_ChainedCylinder_CylScGeom.

Short explanation of what we want to avoid :

Numerical ratcheting is best understood considering a small elastic cycle at a contact between two grains : assuming b1 is fixed, impose this displacement to b2 :

1.translation *dx* in the normal direction

2.rotation *a*

3.translation -*dx* (back to the initial position)

4.rotation -*a* (back to the initial orientation)

If the branch vector used to define the relative shear in rotation×branch is not constant (typically if it is defined from the vector center→contactPoint), then the shear displacement at the end of this cycle is not zero: rotations *a* and -*a* are multiplied by branches of different lengths.

It results in a finite contact force at the end of the cycle even though the positions and orientations are unchanged, in total contradiction with the elastic nature of the problem. It could also be seen as an *inconsistent energy creation or loss*. Given that DEM simulations tend to generate oscillations around equilibrium (damped mass-spring), it can have a significant

impact on the evolution of the packings, resulting for instance in slow creep in iterations under constant load.

The solution adopted here to avoid ratcheting is as proposed by McNamara and co-workers. They analyzed the ratcheting problem in detail - even though they comment on the basis of a cycle that differs from the one shown above. One will find interesting discussions in e.g. DOI 10.1103/PhysRevE.77.031304, even though solution it suggests is not fully applied here (equations of motion are not incorporating alpha, in contradiction with what is suggested by McNamara et al.).

`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.

class `yade.wrapper.Ig2_Sphere_Sphere_ScGeom6D`(*inherits Ig2_Sphere_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)
Create/update a ScGeom6D instance representing the geometry of a contact point between two :yref:‘Spheres<Sphere>‘s, including relative rotations.

`creep`(=*false*)
Substract rotational creep from relative rotation. The rotational creep ScGeom6D::twistCreep is a quaternion and has to be updated inside a constitutive law, see for instance Law2_ScGeom6D_CohFrictPhys_CohesionMoment.

`updateRotations`(=*true*)
Precompute relative rotations. Turning this false can speed up simulations when rotations are not needed in constitutive laws (e.g. when spheres are compressed without cohesion and moment in early stage of a triaxial test), but is not foolproof. Change this value only if you know what you are doing.

class `yade.wrapper.Ig2_Tetra_Tetra_TTetraGeom`(*inherits IGeomFunctor → Functor → Serializable*)
Create/update geometry of collision between 2 tetrahedra (TTetraGeom instance)

class `yade.wrapper.Ig2_Wall_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → Functor → Serializable*)
Create/update contact of Wall and Sphere (Dem3DofGeom_WallSphere instance)

class `yade.wrapper.Ig2_Wall_Sphere_L3Geom`(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)
Incrementally compute L3Geom for contact between Wall and Sphere. Uses attributes of Ig2_Sphere_Sphere_L3Geom.

class `yade.wrapper.Ig2_Wall_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a ScGeom instance representing intersection of Wall and Sphere.

`noRatch`(=*true*)
Avoid granular ratcheting

1.6.2 IGeomDispatcher

class `yade.wrapper.IGeomDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

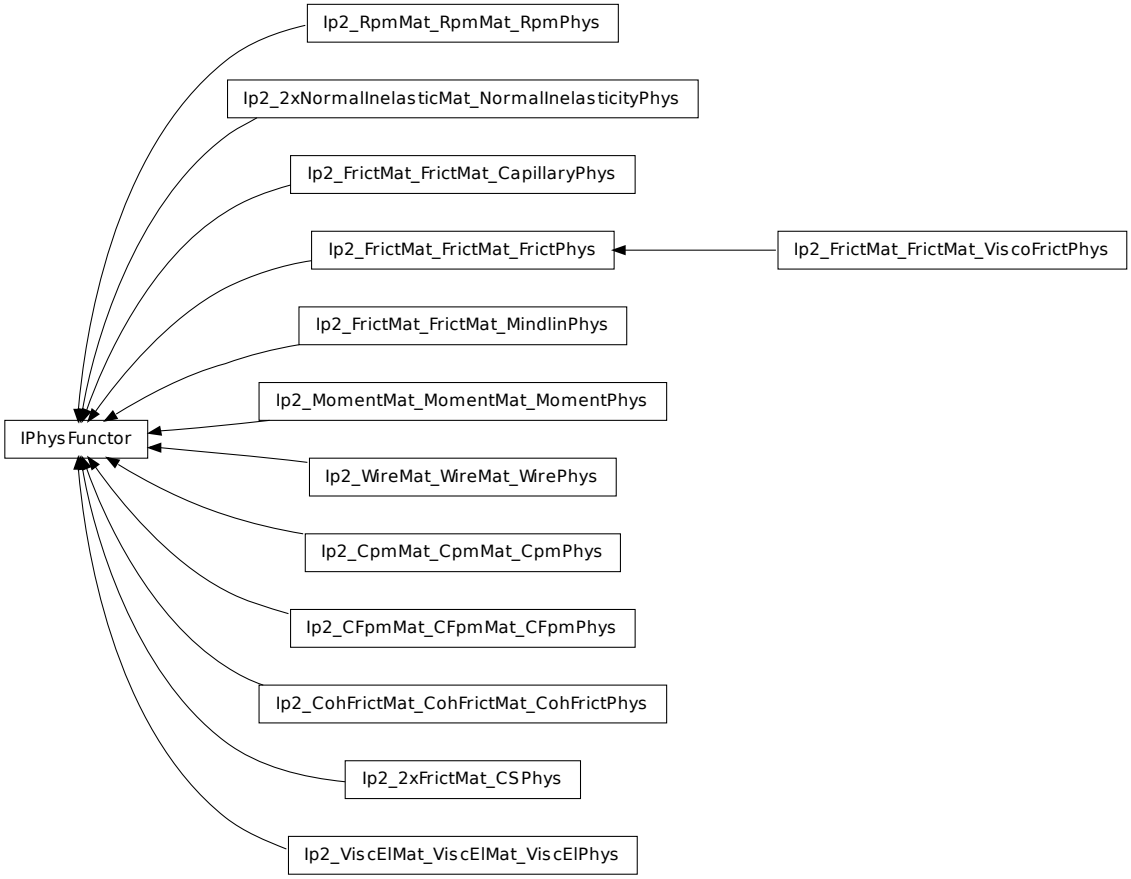
`dispFunctor`(*(Shape)arg2*, *(Shape)arg3*) → IGeomFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

1.7 Interaction Physics creation

1.7.1 IPhysFunctor



class yade.wrapper.**IPhysFunctor**(*inherits Functor → Serializable*)
Functor for creating/updating Interaction::phys objects.

class yade.wrapper.**Ip2_2xFrictMat_CSPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Functor creating CSPhys from two FrictMat. See Law2_Dem3Dof_CSPhys_CundallStrack for details.

class yade.wrapper.**Ip2_2xNormalInelasticMat_NormalInelasticityPhys**(*inherits IPhysFunctor → Functor → Serializable*)
The RelationShips for using Law2_ScGeom6D_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

class yade.wrapper.**Ip2_CFpmMat_CFpmMat_CFpmPhys**(*inherits IPhysFunctor → 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 yade.wrapper.**Ip2_CohFrictMat_CohFrictMat_CohFrictPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Generates cohesive-frictional interactions with moments. Used in the contact law Law2_ScGeom6D_CohFrictPhys_CohesionMoment.

setCohesionNow(=*false*)
If true, assign cohesion to all existing contacts in current time-step. The flag is turned false automatically, so that assignment is done in the current timestep only.

setCohesionOnNewContacts(=*false*)
If true, assign cohesion at all new contacts. If false, only existing contacts can be cohesive (also see Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow), and new contacts are only frictional.

class yade.wrapper.**Ip2_CpmMat_CpmMat_CpmPhys**(*inherits IPhysFunctor → 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 yade.wrapper.**Ip2_FrictMat_FrictMat_CapillaryPhys**(*inherits IPhysFunctor → 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 `yade.wrapper.Ip2_FrictMat_FrictMat_FrictPhys`(*inherits IPhysFunctor → 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 ElastMat::poisson, as ks=poisson*kn, and the resulting shear stiffness of the interaction will be also an harmonic average.

frictAngle(*=uninitalized*)
Instance of MatchMaker determining how to compute interaction's friction angle. If `None`, minimum value is used.

class `yade.wrapper.Ip2_FrictMat_FrictMat_MindlinPhys`(*inherits IPhysFunctor → 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).

Viscous parameters can be specified either using coefficients of restitution (e_n, e_s) or viscous damping coefficient (β_n, β_s). The following rules apply: #. If the β_n (β_s) coefficient is given, it is assigned to MindlinPhys.betan (MindlinPhys.betas) directly. #. If e_n is given, MindlinPhys.betan is computed using $\beta_n = -(\log e_n)/\sqrt{\pi^2 + (\log e_n)^2}$. The same applies to e_s, MindlinPhys.betas. #. It is an error (exception) to specify both e_n and β_n (e_s and β_s). #. If neither e_n nor β_n is given, zero value for MindlinPhys.betan is used; there will be no viscous effects. #.If neither e_s nor β_s is given, the value of MindlinPhys.betan is used for MindlinPhys.betas as well.

The e_n, β_n, e_s, β_s are MatchMaker objects; they can be constructed from float values to always return constant value.

See scripts/test/shots.py for an example of specifying e_n based on combination of parameters.

betan(*=uninitalized*)
Normal viscous damping coefficient β_n.

betas(*=uninitalized*)
Shear viscous damping coefficient β_s.

en(*=uninitalized*)
Normal coefficient of restitution e_n.

es(*=uninitalized*)
Shear coefficient of restitution e_s.

eta(*=0.0*)
Coefficient to determine the plastic bending moment

gamma(*=0.0*)
Surface energy parameter [J/m^2] per each unit contact surface, to derive DMT formulation from HM

krot(*=0.0*)
Rotational stiffness for moment contact law

ktwist(*=0.0*)
Torsional stiffness for moment contact law

class `yade.wrapper.Ip2_FrictMat_FrictMat_ViscoFrictPhys`(*inherits Ip2_FrictMat_FrictMat_FrictPhys → IPhysFunctor → 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 ElastMat::poisson, as ks=poisson*kn, and the resulting shear stiffness of the interaction will be also an harmonic average.

class `yade.wrapper.Ip2_MomentMat_MomentMat_MomentPhys`(*inherits IPhysFunctor → 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 `yade.wrapper.Ip2_RpmMat_RpmMat_RpmPhys`(*inherits IPhysFunctor → Functor → Serializable*)

Convert 2 RpmMat instances to RpmPhys with corresponding parameters.

`initDistance`(*=0*)
: Initial distance between spheres at the first step.

class `yade.wrapper.Ip2_ViscElMat_ViscElMat_ViscElPhys`(*inherits IPhysFunctor → Functor → Serializable*)

Convert 2 instances of ViscElMat to ViscElPhys using the rule of consecutive connection.

class `yade.wrapper.Ip2_WireMat_WireMat_WirePhys`(*inherits IPhysFunctor → Functor → Serializable*)

Converts 2 WireMat instances to WirePhys with corresponding parameters.

`linkThresholdIteration`(*=1*)
: Iteration to create the link.

1.7.2 IPhysDispatcher

class `yade.wrapper.IPhysDispatcher`(*inherits Dispatcher → Engine → Serializable*)

Dispatcher calling functors based on received argument type(s).

dispFunctor(*(Material)arg2*, *(Material)arg3*) → IPhysFunctor
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 associated with this dispatcher.

1.8 Constitutive laws

1.8.1 LawFunctor

class yade.wrapper.**LawFunctor**(*inherits Functor → Serializable*)
Functor for applying constitutive laws on interactions.

class yade.wrapper.**Law2_CylScGeom_FrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)
Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from [CundallStrack1979] (see also [Pfc3dManual30]). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The

shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

Note: This law uses ScGeom; there is also functionally equivalent Law2_Dem3DofGeom_-FrictPhys_CundallStrack, which uses Dem3DofGeom (sphere-box interactions are not implemented for the latest).

Note: This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. [Scholtes2009b] and other papers from the same authors). It is generalised by Law2_ScGeom6D_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

`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)

class `yade.wrapper.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)epsCrackOnset*, *(float)epsFracture*[, *(bool)neverDamage=False*]) → float
: Damage evolution law, evaluating the ω parameter. κ_D is historically maximum strain, *epsCrackOnset* (ε_0) = CpmPhys.epsCrackOnset, *epsFracture* = CpmPhys.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 `yade.wrapper.Law2_Dem3DofGeom_FrictPhys_CundallStrack`(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for linear compression, no tension, and linear plasticity surface.

No longer maintained and linking to known bugs; :consider using yref:*Law2_ScGeom_FrictPhys_-CundallStrack*.

class `yade.wrapper.Law2_Dem3DofGeom_RockPMPhys_Rpm`(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the Rpm model

class `yade.wrapper.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 `yade.wrapper.Law2_L3Geom_FrictPhys_ElPerfPl`(*inherits LawFunctor → Functor → Serializable*)

Basic law for testing L3Geom; it bears no cohesion (unless *noBreak* is `True`), and plastic slip obeys the Mohr-Coulomb criterion (unless *noSlip* is `True`).

`noBreak`(=*false*)
Do not break contacts when particles separate.

`noSlip`(=*false*)
No plastic slipping.

class `yade.wrapper.Law2_L6Geom_FrictPhys_Linear`(*inherits Law2_L3Geom_FrictPhys_ElPerfPl → LawFunctor → Functor → Serializable*)

Basic law for testing L6Geom – linear in both normal and shear sense, without slip or breakage.

`charLen`(=*1*)
Characteristic length with the meaning of the stiffness ratios bending/shear and torsion/normal.

class `yade.wrapper.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 `yade.wrapper.Law2_ScGeom6D_CohFrictPhys_CohesionMoment`(*inherits LawFunctor → Functor → Serializable*)

Law for linear traction-compression-bending-twisting, with cohesion+friction and Mohr-Coulomb plasticity surface. This law adds adhesion and moments to Law2_ScGeom_FrictPhys_CundallStrack.

The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n * u_n, a_n)$, with a_n the normal adhesion. The shear force is $F_s = k_s * u_s$, the plasticity condition defines the maximum value of the shear force, by default $F_s^{max} = F_n * \tan(\varphi) + a_s$, with φ the friction angle and a_n the shear adhesion. If CohFrictPhys::cohesionDisableFriction is True, friction is ignored as long as adhesion is active, and the maximum shear force is only $F_s^{max} = a_s$.

If the maximum tensile or maximum shear force is reached and CohFrictPhys::fragile =True (default), the cohesive link is broken, and a_n, a_s are set back to zero. If a tensile force is present, the contact is lost, else the shear strength is $F_s^{max} = F_n * \tan(\varphi)$. If CohFrictPhys::fragile =False (in course of implementation), the behaviour is perfectly plastic, and the shear strength is kept constant.

If Law2_ScGeom6D_CohFrictPhys_CohesionMoment::momentRotationLaw =True, bending and twisting moments are computed using a linear law with moduli respectively k_t and k_r (the two values are the same currently), so that the moments are : $M_b = k_b * \Theta_b$ and $M_t = k_t * \Theta_t$, with $\Theta_{b,t}$ the relative rotations between interacting bodies. There is no maximum value of moments in the current implementation, though they could be added in the future.

Creep at contact is implemented in this law, as defined in [Hassan2010]. If activated, there is a viscous behaviour of the shear and twisting components, and the evolution of the elastic parts of shear displacement and relative twist is given by $du_{s,e}/dt = -F_s/\nu_s$ and $d\Theta_{t,e}/dt = -M_t/\nu_t$.

Note: Periodicity is not handled yet in this law.

`always_use_moment_law(=false)`
: If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

`creepStiffness(=1)`
: ...

`creep_viscosity(=1)`
: creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys...

`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.

`shear_creep2(=false)`
: activate SLS (http://en.wikipedia.org/wiki/Standard_Linear_Solid_model) creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

`twist_creep(=false)`
: activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

`useIncrementalForm(=false)`
: use the incremental formulation to compute bending and twisting moments. Creep on the twisting moment is not included in such a case.

class `yade.wrapper.Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity`(*inherits LawFunctor → Functor → Serializable*)

Contact law used to simulate granulate filler in rock joints [Duriez2009a], [Duriez2010]. It includes possibility of cohesion, moment transfer and inelastic compression behaviour (to reproduce the normal inelasticity observed for rock joints, for the latter).

The moment transfer relation corresponds to the adaptation of the work of Plassiard & Belheine (see in [DeghmReport2006] for example), which was realized by J. Kozicki, and is now coded in ScGeom6D.

As others LawFunctor, it uses pre-computed data of the interactions (rigidities, friction angles -with their tan()-, orientations of the interactions); this work is done here in Ip2_2xNormalInelasticMat_NormalInelasticityPhys.

To use this you should also use NormalInelasticMat as material type of the bodies.

The effects of this law are illustrated in scripts/normalInelasticityTest.py

`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 `yade.wrapper.Law2_ScGeom_CFpmPhys_CohesiveFrictionalPM`(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the CFpm model.

preventGranularRatcheting(=*true*)
If true rotations are computed such as granular ratcheting is prevented. See article [Alonso2004], pg. 3-10 – and a lot more papers from the same authors).

class yade.wrapper.**Law2_ScGeom_FrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)
Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from [CundallStrack1979] (see also [Pfc3dManual30]). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. [Scholtes2009b] and other papers from the same authors). It is generalised by Law2_ScGeom6D_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

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_CundallStrack::traceEnergy is true.

sphericalBodies(=*true*)
If true, compute branch vectors from radii (faster), else use contactPoint-position. Turning this flag true is safe for sphere-sphere contacts and a few other specific cases. It will give wrong values of torques on facets or boxes.

traceEnergy(=*false*)
Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

class yade.wrapper.**Law2_ScGeom_MindlinPhys_HertzWithLinearShear**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the Hertz formulation (using MindlinPhys.kno) and linear beahvior in shear (using MindlinPhys.kso for stiffness and FrictPhys.tangensOfFrictionAngle).

Note: No viscosity or damping. If you need those, look at Law2_ScGeom_MindlinPhys_Mindlin, which also includes non-linear Mindlin shear.

nonLin(=*0*)
Shear force nonlinearity (the value determines how many features of the non-linearity are taken in account). 1: ks as in HM 2: shearElastic increment computed as in HM 3. granular ratcheting disabled.

class yade.wrapper.**Law2_ScGeom_MindlinPhys_Mindlin**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the Hertz-Mindlin formulation. It includes non linear elasticity in the normal direction as predicted by Hertz for two non-conforming elastic contact bodies. In the shear direction, instead, it reseambles the simplified case without slip discussed in Mindlin's paper, where a linear relationship between shear force and tangential displacement is provided. Finally, the Mohr-Coulomb criterion is employed to established the maximum friction force which can be developed at the contact. Moreover, it is also possible to include the effect of linear viscous damping through the definition of the parameters β_n and β_s.

calcEnergy(=*false*)
bool to calculate energy terms (shear potential energy, dissipation of energy due to friction

and dissipation of energy due to normal and tangential damping)

contactsAdhesive() → float
Compute total number of adhesive contacts.

frictionDissipation(=*uninitalized*)
Energy dissipation due to sliding

includeAdhesion(=*false*)
bool to include the adhesion force following the DMT formulation. If true, also the normal elastic energy takes into account the adhesion effect.

includeMoment(=*false*)
bool to consider rolling resistance (if Ip2_FrictMat_FrictMat_MindlinPhys::eta is 0.0, no plastic condition is applied.)

normDampDissip(=*uninitalized*)
Energy dissipated by normal damping

normElastEnergy() → float
Compute normal elastic potential energy. It handles the DMT formulation if Law2_ScGeom_-MindlinPhys_Mindlin::includeAdhesion is set to true.

preventGranularRatcheting(=*true*)
bool to avoid granular ratcheting

ratioSlidingContacts() → float
Return the ratio between the number of contacts sliding to the total number at a given time.

shearDampDissip(=*uninitalized*)
Energy dissipated by tangential damping

shearEnergy(=*uninitalized*)
Shear elastic potential energy

class yade.wrapper.**Law2_ScGeom_MindlinPhys_MindlinDeresiewitz**(*inherits LawFunctor → Functor → Serializable*)
Hertz-Mindlin contact law with partial slip solution, as described in [Thornton1991].

class yade.wrapper.**Law2_ScGeom_ViscElPhys_Basic**(*inherits LawFunctor → Functor → Serializable*)
Linear viscoelastic model operating on ScGeom and ViscElPhys.

class yade.wrapper.**Law2_ScGeom_ViscoFrictPhys_CundallStrack**(*inherits Law2_ScGeom_-FrictPhys_CundallStrack → LawFunctor → Functor → Serializable*)
Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from [CundallStrack1979] (see also [Pfc3dManual30]). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. [Scholtes2009b] and other papers from the same authors). It is generalised by Law2_ScGeom6D_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

creepStiffness(=*1*)

shearCreep(=*false*)

viscosity(=*1*)

class yade.wrapper.**Law2_ScGeom_WirePhys_WirePM**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the wire model.

linkThresholdIteration(=*1*)
Iteration to create the link.

1.8.2 LawDispatcher

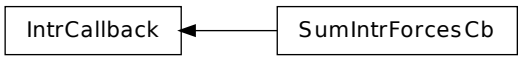
class `yade.wrapper.LawDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(IGeom)arg2*, *(IPhys)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 associated with this dispatcher.

1.9 Callbacks



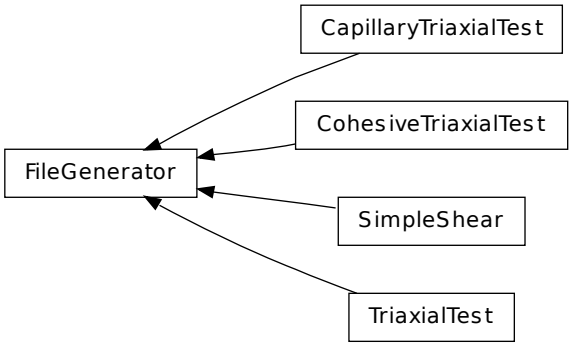
class `yade.wrapper.IntrCallback`(*inherits Serializable*)
Abstract callback object which will be called for every (real) Interaction after the interaction has been processed by InteractionLoop.

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 InteractionLoop constructor, or by appending the callback object to InteractionLoop::callbacks.

class `yade.wrapper.SumIntrForcesCb`(*inherits IntrCallback → Serializable*)
Callback summing magnitudes of forces over all interactions. IPhys of interactions must derive from NormShearPhys (responsability fo the user).

1.10 Preprocessors



class `yade.wrapper.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

class `yade.wrapper.CapillaryTriaxialTest`(*inherits FileGenerator → Serializable*)
This preprocessor is a variant of TriaxialTest, including the model of capillary forces developed 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 CapillaryTriaxialTest::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 overridden 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'} ; contains 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 CapillaryTriaxialTest::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 information 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

class yade.wrapper.**CohesiveTriaxialTest**(*inherits FileGenerator → Serializable*)
This preprocessor is a variant of TriaxialTest using the cohesive-frictional contact law with moments. It sets up a scene for cohesive triaxial tests. See full documentation at http://yade-dem.org/wiki/TriaxialTest.

Cohesion is initially 0 by default. The suggested usage is to define cohesion values in a second step, after isotropic compaction : define shear and normal cohesions in Ip2_CohFrictMat_CohFrictMat_CohFrictPhys, then turn Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow true to assign them at each contact at next iteration.

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(=*"./CohesiveWallStresses"+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 overridden 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(=*0.001*)
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'} ; contains 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)

normalCohesion(=*0*)
Material parameter used to define contact strength in tension.

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusDeviation(=*0.3*)
Normalized standard deviation of generated sizes.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and TriaxialTest::numberOfGrains

recordIntervalIter(=*20*)
interval between file outputs

setCohesionOnNewContacts(=*false*)
create cohesionless (False) or cohesive (True) interactions for new contacts.

shearCohesion(=*0*)
Material parameter used to define shear strength of contacts.

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 yade.wrapper.**SimpleShear**(*inherits FileGenerator → Serializable*)
Preprocessor for creating a numerical model of a simple shear box.

- Boxes (6) constitute the different sides of the box itself
- Spheres are contained in the box. The sample could be generated via the same method used in TriaxialTest Preprocesor (=> see GenerateCloud) or by reading a text file containing positions and radii of a sample (=> see ImportCloud). This last one is the one by default used by this PreProcessor as it is written here => you need to have such a file.

Thanks to the Engines (in pkg/common/Engine/PartialEngine) KinemCNDEngine, KinemCNSEngine and KinemCNLEngine, respectively constant normal displacement, constant normal rigidity and constant normal stress are possible to execute over such samples.

NB about micro-parameters : their values correspond to those used in [Duriez2009a].

boxPoissonRatio(=*0.04*)
value of ElastMat::poisson for the spheres [-]

boxYoungModulus(=*4.0e9*)
value of ElastMat::young for the boxes [Pa]

density(=*2600*)
density of the spheres [kg/m^3]

filename(=*"../porosite0_44.txt"*)
file with the list of spheres centers and radii

gravApplied(=*false*)
depending on this, GravityEngine is added or not to the scene to take into account the weight of particles

gravity(=*Vector3r(0, -9.81, 0)*)
vector corresponding to used gravity [m/s^2]

height(=*0.02*)
initial height (along y-axis) of the shear box [m]

length(=*0.1*)
initial length (along x-axis) of the shear box [m]

sphereFrictionDeg(=*37*)
value of ElastMat::poisson for the spheres [°] (the necessary conversion in rad is done automatically)

spherePoissonRatio(=*0.04*)
value of ElastMat::poisson for the spheres [-]

sphereYoungModulus(=*4.0e9*)
value of ElastMat::young for the spheres [Pa]

thickness(=*0.001*)
thickness of the boxes constituting the shear box [m]

timeStepUpdateInterval(=*50*)
value of TimeStepper::timeStepUpdateInterval for the TimeStepper used here

width(=*0.04*)
initial width (along z-axis) of the shear box [m]

class yade.wrapper.**TriaxialTest**(*inherits FileGenerator → Serializable*)
Create a scene for triaxal test.

Introduction Yade includes tools to simulate triaxial tests on particles assemblies. This preprocessor (and variants like e.g. CapillaryTriaxialTest) illustrate how to use them. It generates a scene which will - by default - go through the following steps :

- generate random loose packings in a parallelepiped.
- compress the packing isotropicaly, either squeezing the packing between moving rigid boxes or expanding the particles while boxes are fixed (depending on flag internalCompaction). The confining pressure in this stage is defined via sigmaIsoCompaction.
- when the packing is dense and stable, simulate a loading path and get the mechanical response as a result.

The default loading path corresponds to a constant lateral stress (sigmaLateralConfinement) in 2 directions and constant strain rate on the third direction. This default loading path is

performed when the flag autoCompressionActivation it `True`, otherwise the simulation stops after isotropic compression.

Different loading paths might be performed. In order to define them, the user can modify the flags found in engine TriaxialStressController at any point in the simulation (in c++). If `TriaxialStressController.wall_X_activated` is `true` boundary X is moved automatically to maintain the defined stress level *sigmaN* (see axis conventions below). If `false` the boundary is not controlled by the engine at all. In that case the user is free to prescribe fixed position, constant velocity, or more complex conditions.

Note: *Axis conventions.* Boundaries perpendicular to the x axis are called "left" and "right", y corresponds to "top" and "bottom", and axis z to "front" and "back". In the default loading path, strain rate is assigned along y, and constant stresses are assigned on x and z.

Essential engines

1. The TrixaialCompressionEngine is used for controlling the state of the sample and simulating loading paths. TriaxialCompressionEngine inherits from TriaxialStressController, which computes stress- and strain-like quantities in the packing and maintain a constant level of stress at each boundary. TriaxialCompressionEngine has few more members in order to impose constant strain rate and control the transition between isotropic compression and triaxial test. Transitions are defined by changing some flags of the TriaxialStressController, switching from/to imposed strain rate to/from imposed stress.
2. The class TriaxialStateRecorder is used to write to a file the history of stresses and strains.
3. TriaxialTest is using GlobalStiffnessTimeStepper to compute an appropriate Δt for the numerical scheme.

Note: `TriaxialStressController::ComputeUnbalancedForce` returns a value that can be useful for evaluating the stability of the packing. It is defined as (mean force on particles)/(mean contact force), so that it tends to 0 in a stable packing. This parameter is checked by TriaxialCompressionEngine to switch from one stage of the simulation to the next one (e.g. stop isotropic confinment and start axial loading)

Frequently Asked Questions

1. **How is generated the packing? How to change particles sizes distribution? Why do I have a m**

   The initial positioning of spheres is done by generating random (x,y,z) in a box and checking if a sphere of radius R (R also randomly generated with respect to a uniform distribution between mean*(1-std_dev) and mean*(1+std_dev) can be inserted at this location without overlaping with others.

   If the sphere overlaps, new (x,y,z)'s are generated until a free position for the new sphere is found. This explains the message you have: after 3000 trial-and-error, the sphere couldn't be placed, and the algorithm stops.

   You get the message above if you try to generate an initialy dense packing, which is not possible with this algorithm. It can only generate clouds. You should keep the default value of porosity (n~0.7), or even increase if it is still to low in some cases. The dense state will be obtained in the second step (compaction, see below).

2. **How is the compaction done, what are the parameters maxWallVelocity and finalMaxMultiplie**

   Compaction is done

   (a) by moving rigid boxes or

   (b) by increasing the sizes of the particles (decided using the option internalCompaction size increase).

   Both algorithm needs numerical parameters to prevent instabilities. For instance, with the method (1) maxWallVelocity is the maximum wall velocity, with method (2) finalMaxMultiplier is the max value of the multiplier applied on sizes at each iteration (always something like 1.00001).

3. **During the simulation of triaxial compression test, the wall in one direction moves with an incr**
   The control of stress on a boundary is based on the total stiffness K of all contacts between the packing and this boundary. In short, at each step, displacement=stress_error/K. This algorithm is implemented in TriaxialStressController, and the control itself is in `TriaxialStressController::ControlExternalStress`. The control can be turned off independently for each boundary, using the flags `wall_XXX_activated`, with *XXX* {*top*, *bottom*, *left*, *right*, *back*, *front*}. The imposed sress is a unique value (sigma_iso) for all directions if TriaxialStressController.isAxisymetric, or 3 independent values sigma1, sigma2, sigma3.

4. **Which value of friction angle do you use during the compaction phase of the Triaxial Test?**
   The friction during the compaction (whether you are using the expansion method or the compression one for the specimen generation) can be anything between 0 and the final value used during the Triaxial phase. Note that higher friction than the final one would result in volumetric collapse at the beginning of the test. The purpose of using a different value of friction during this phase is related to the fact that the final porosity you get at the end of the sample generation essentially depends on it as well as on the assumed Particle Size Distribution. Changing the initial value of friction will get to a different value of the final porosity.

5. **Which is the aim of the `bool isRadiusControlIteration`?** This internal variable (updated automatically) is true each N timesteps (with N=radiusControlInterval). For other timesteps, there is no expansion. Cycling without expanding is just a way to speed up the simulation, based on the idea that 1% increase each 10 iterations needs less operations than 0.1% at each iteration, but will give similar results.

6. **How comes the unbalanced force reaches a low value only after many timesteps in the compact**
   The value of unbalanced force (dimensionless) is expected to reach low value (i.e. identifying a static-equilibrium condition for the specimen) only at the end of the compaction phase. The code is not aiming at simulating a quasistatic isotropic compaction process, it is only giving a stable packing at the end of it.

`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.

`boxYoungModulus`(=*15000000.0*)
: Stiffness of boxes.

compactionFrictionDeg(=*sphereFrictionDeg*)
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden 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'} ; contains axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

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)

1.11 Rendering

1.11.1 OpenGLRenderer

class yade.wrapper.**OpenGLRenderer**(*inherits Serializable*)
Class responsible for rendering scene on OpenGL devices.

bgColor(=*Vector3r(.2, .2, .2)*)
Color of the background canvas (RGB)

bound(=*false*)
Render body Bound

clipPlaneActive(=*vector<bool>(numClipPlanes, false)*)
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

extraDrawers(=*uninitalized*)
Additional rendering components (GlExtraDrawer).

ghosts(=*true*)
Render objects crossing periodic cell edges by cloning them in multiple places (periodic simulations only).

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::geom objects.

intrPhys(=*false*)
Render Interaction::phys objects

intrWire(=*false*)
If rendering interactions, use only wires to represent them.

light1(=*true*)
Turn light 1 on.

light2(=*true*)
Turn light 2 on.

light2Color(=*Vector3r(0.5, 0.5, 0.1)*)
Per-color intensity of secondary light (RGB).

light2Pos(=*Vector3r(-130, 75, 30)*)
Position of secondary OpenGL light source in the scene.

lightColor(=*Vector3r(0.6, 0.6, 0.6)*)
Per-color intensity of primary light (RGB).

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)

render() → None
Render the scene in the current OpenGL context.

rotScale(=*1., disable scaling*)
Artificially enlarge (scale) rotations of bodies relative to their reference orientation, so the they are better visible.

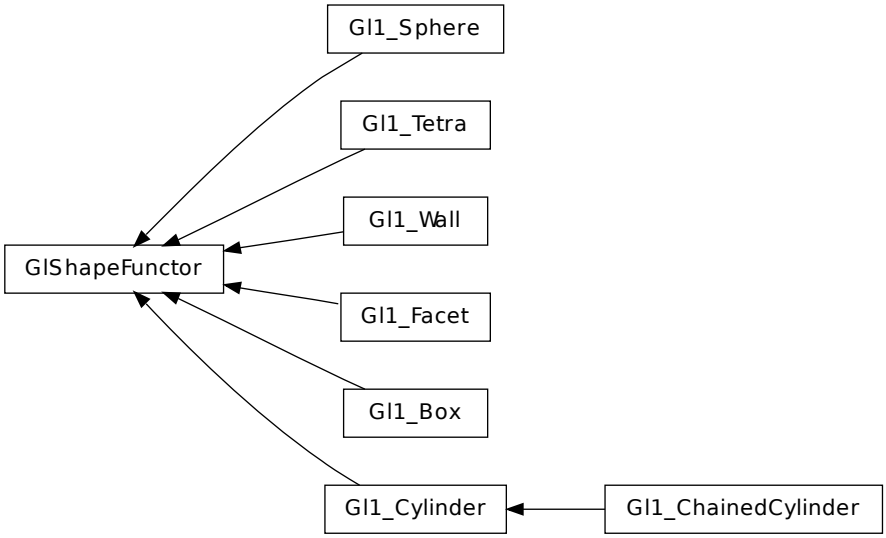
selId(=*Body::ID_NONE*)
Id of particle that was selected by the user.

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)

1.11.2 GlShapeFunctor



class `yade.wrapper.GlShapeFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering Shape objects.

class `yade.wrapper.Gl1_Box`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Box object

class `yade.wrapper.Gl1_ChainedCylinder`(*inherits Gl1_Cylinder → GlShapeFunctor → Functor → Serializable*)
Renders ChainedCylinder object including a shift for compensating flexion.

class `yade.wrapper.Gl1_Cylinder`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Cylinder object

`wire`(=*false* [**static**])
Only show wireframe (controlled by `glutSlices` and `glutStacks`.

`glutNormalize`(=*true* [**static**])
Fix normals for non-wire rendering

`glutSlices`(=*8* [**static**])
Number of sphere slices.

`glutStacks`(=*4* [**static**])
Number of sphere stacks.

class `yade.wrapper.Gl1_Facet`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Facet object

`normals`(=*false* [**static**])
In wire mode, render normals of facets and edges; facet's colors are disregarded in that case.

class `yade.wrapper.Gl1_Sphere`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Sphere object

`quality`(=*1.0* [**static**])
Change discretization level of spheres. quality>1 for better image quality, at the price of more cpu/gpu usage, 0<quality<1 for faster rendering. If mono-color sphres are displayed (Gl1_Sphere::stripes=False), quality mutiplies :yref:'Gl1_Sphere::glutSlices and Gl1_Sphere::glutStacks. If striped spheres are displayed (:yref:'Gl1_Sphere::stripes=True), only integer increments are meaningfull : quality=1 and quality=1.9 will give the same result, quality=2 will give finer result.

`wire`(=*false* [**static**])
Only show wireframe (controlled by `glutSlices` and `glutStacks`.

`stripes`(=*false* [**static**])
In non-wire rendering, show stripes clearly showing particle rotation.

`localSpecView`(=*true* [**static**])
Compute specular light in local eye coordinate system.

`glutSlices`(=*12* [**static**])
Base number of sphere slices, multiplied by Gl1_Sphere::quality before use); not used with `stripes` (see glut{Solid,Wire}Sphere reference)

`glutStacks`(=*6* [**static**])
Base number of sphere stacks, multiplied by Gl1_Sphere::quality before use; not used with `stripes` (see glut{Solid,Wire}Sphere reference)

class `yade.wrapper.Gl1_Tetra`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Tetra object

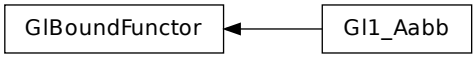
class `yade.wrapper.Gl1_Wall`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Wall object

`div`(=*20* [**static**])
Number of divisions of the wall inside visible scene part.

1.11.3 GlStateFunctor

class `yade.wrapper.GlStateFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering State objects.

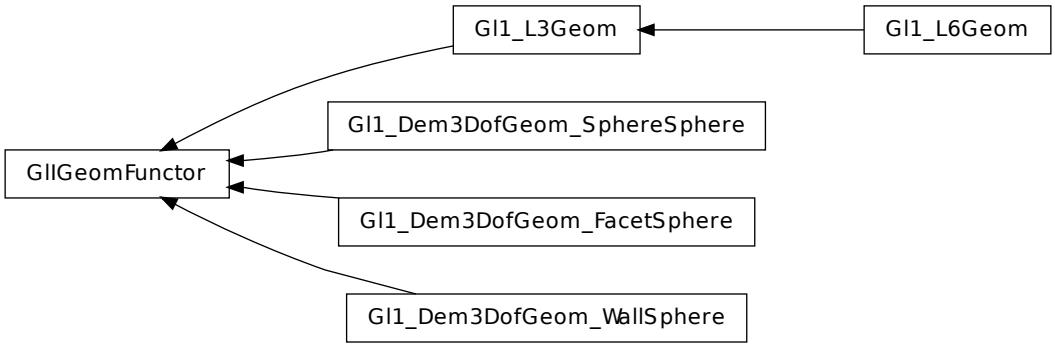
1.11.4 GlBoundFunctor



class `yade.wrapper.GlBoundFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering Bound objects.

class `yade.wrapper.Gl1_Aabb`(*inherits GlBoundFunctor → Functor → Serializable*)
Render Axis-aligned bounding box (Aabb).

1.11.5 GlIGeomFunctor



class yade.wrapper.**GlIGeomFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering IGeom objects.

class yade.wrapper.**Gl1_Dem3DofGeom_FacetSphere**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of facet and sphere (represented by Dem3DofGeom_FacetSphere)

normal(=*false* [**static**])
Render interaction normal

rolledPoints(=*false* [**static**])
Render points rolled on the sphere & facet (original contact point)

unrolledPoints(=*false* [**static**])
Render original contact points unrolled to the contact plane

shear(=*false* [**static**])
Render shear line in the contact plane

shearLabel(=*false* [**static**])
Render shear magnitude as number

class yade.wrapper.**Gl1_Dem3DofGeom_SphereSphere**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of 2 spheres (represented by Dem3DofGeom_SphereSphere)

normal(=*false* [**static**])
Render interaction normal

rolledPoints(=*false* [**static**])
Render points rolled on the spheres (tracks the original contact point)

unrolledPoints(=*false* [**static**])
Render original contact points unrolled to the contact plane

shear(=*false* [**static**])
Render shear line in the contact plane

shearLabel(=*false* [**static**])
Render shear magnitude as number

class yade.wrapper.**Gl1_Dem3DofGeom_WallSphere**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of wall and sphere (represented by Dem3DofGeom_WallSphere)

normal(=*false* [**static**])
Render interaction normal

rolledPoints(=*false* [**static**])
Render points rolled on the spheres (tracks the original contact point)

unrolledPoints(=*false* [**static**])
Render original contact points unrolled to the contact plane

shear(=*false* [**static**])
Render shear line in the contact plane

shearLabel(=*false* [**static**])
Render shear magnitude as number

class yade.wrapper.**Gl1_L3Geom**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render L3Geom geometry.

axesLabels(=*false* [**static**])
Whether to display labels for local axes (x,y,z)

axesScale(=*1.* [**static**])
Scale local axes, their reference length being half of the minimum radius.

axesWd(=*1.* [**static**])
Width of axes lines, in pixels; not drawn if non-positive

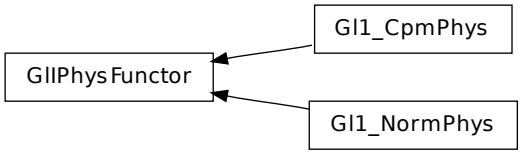
uPhiWd(=*2.* [**static**])
Width of lines for drawing displacements (and rotations for L6Geom); not drawn if non-positive.

uScale(=*1.* [**static**])
Scale local displacements (u - u0); 1 means the true scale, 0 disables drawing local displacements; negative values are permissible.

class yade.wrapper.**Gl1_L6Geom**(*inherits Gl1_L3Geom → GlIGeomFunctor → Functor → Serializable*)
Render L6Geom geometry.

phiScale(=*1.* [**static**])
Scale local rotations (phi - phi0). The default scale is to draw π rotation with length equal to minimum radius.

1.11.6 GlIPhysFunctor



class yade.wrapper.**GlIPhysFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering IPhys objects.

class yade.wrapper.**Gl1_CpmPhys**(*inherits GlIPhysFunctor → Functor → Serializable*)
Render CpmPhys objects of interactions.

contactLine(=*true* [**static**])
Show contact line

dmgLabel(=*true* [**static**])
Numerically show contact damage parameter

dmgPlane(=*false* [**static**])
[what is this?]

epsT(=*false* [**static**])
Show shear strain

epsTAxes(=*false* [**static**])
Show axes of shear plane

normal(=*false* [**static**])
Show contact normal

colorStrainRatio(=*-1* [**static**])
If positive, set the interaction (wire) color based on ε_N normalized by $\varepsilon_0 \times$ *colorStrainRatio* (ε_0=:yref:*CpmPhys.epsCrackOnset*). Otherwise, color based on the residual strength.

epsNLabel(=*false* [**static**])
Numerically show normal strain

class yade.wrapper.**Gl1_NormPhys**(*inherits GlIPhysFunctor → Functor → Serializable*)
Renders NormPhys objects as cylinders of which diameter and color depends on NormPhys:normForce magnitude.

maxFn(=*0* [**static**])
Value of NormPhys.normalForce corresponding to maxDiameter. This value will be increased (but *not decreased*) automatically.

signFilter(=*0* [**static**])
If non-zero, only display contacts with negative (-1) or positive (+1) normal forces; if zero, all contacts will be displayed.

refRadius(=*std::numeric_limits<Real>::infinity()* [**static**])
Reference (minimum) particle radius; used only if maxRadius is negative. This value will be decreased (but *not increased*) automatically. *(auto-updated)*

maxRadius(=*-1* [**static**])
Cylinder radius corresponding to the maximum normal force. If negative, auto-updated refRadius will be used instead.

slices(=*6* [**static**])
Number of sphere slices; (see glutCylinder reference)

stacks(=*1* [**static**])
Number of sphere stacks; (see glutCylinder reference)

maxWeakFn(=*NaN* [**static**])
Value that divides contacts by their normal force into the "weak fabric" and "strong fabric". This value is set as side-effect by utils.fabricTensor.

weakFilter(=*0* [**static**])
If non-zero, only display contacts belonging to the "weak" (-1) or "strong" (+1) fabric.

weakScale(=*1.* [**static**])
If maxWeakFn is set, scale radius of the weak fabric by this amount (usually smaller than 1). If zero, 1 pixel line is displayed. Colors are not affected by this value.

1.12 Simulation data

1.12.1 Omega

class yade.wrapper.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 negative value enables dynamic Δt (by looking for a TimeStepper in O.engine) and sets positive timestep `O.dt=|Δt|` (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.

dynDtAvailable
Whether a TimeStepper is amongst O.engines, activated or not.

energy
EnergyTracker of the current simulation. (meaningful only with O.trackEnergy)

engines
List of engines in the simulation (Scene::engines).

exitNoBacktrace([*(int)status=0*]) → None
Disable SEGV handler and exit, optionally with given status number.

filename
Filename under which the current simulation was saved (None if never saved).

forceSyncCount
Counter for number of syncs in ForceContainer, for profiling purposes.

forces
ForceContainer (forces, torques, displacements) in the current simulation.

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)file*[, *(bool)quiet=False*]) → None
Load simulation from file.

loadTmp([*(str)mark=''*[, *(bool)quiet=False*]]) → None
Load simulation previously stored in memory by saveTmp. *mark* optionally distinguishes multiple saved simulations

lsTmp() → list
Return list of all memory-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([*(bool)quiet=False*]) → 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).

running
Whether background thread is currently running a simulation.

save(*(str)file*[, *(bool)quiet=False*]) → None
Save current simulation to file (should be .xml or .xml.bz2)

saveTmp([*(str)mark=''*[, *(bool)quiet=False*]]) → 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.

subStep
Get the current subStep number (only meaningful if O.subStepping==True); -1 when outside the loop, otherwise either 0 (O.subStepping==False) or number of engine to be run (O.subStepping==True)

subStepping
Get/set whether subStepping is active.

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 the timing module).

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.

`trackEnergy`
When energy tracking is enabled or disabled in this simulation.

`wait`() → None
Don't return until the simulation will have been paused. (Returns immediately if not running).

1.12.2 BodyContainer

class `yade.wrapper.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)

`clump`(*(object)arg2*) → int
Clump given bodies together (creating a rigid aggregate); returns clump id.

`erase`(*(int)arg2*) → bool
Erase body with the given id; all interaction will be deleted by InteractionLoop in the next step.

`replace`(*(object)arg2*) → object

1.12.3 InteractionContainer

class `yade.wrapper.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.

`erase`(*(int)arg2*, *(int)arg3*) → None
Erase one interaction, given by id1, id2 (internally, `requestErase` is called – the interaction might still exist as potential, if the Collider decides so).

`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.

1.12.4 ForceContainer

class `yade.wrapper.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.

1.12.5 MaterialContainer

class `yade.wrapper.MaterialContainer`
Container for Materials. A material can be accessed using

1. numerical index in range(0,len(cont)), like cont[2];
2. textual label that was given to the material, like cont['steel']. This etails traversing all materials and should not be used frequently.

`__init__`(*(MaterialContainer)arg2*) → None

`append`(*(Material)arg2*) → int

Add new shared Material; changes its id and return it.

append((MaterialContainer)arg1, (object)arg2) → object : Append list of Material instances, return list of ids.

`index`(*(str)arg2*) → int
Return id of material, given its label.

1.12.6 Scene

class `yade.wrapper.Scene`(*inherits Serializable*)
Object comprising the whole simulation.

`compressionNegative`
Whether the convention is that compression has negative sign (set by Ig2Functor.

`dt`(*=1e-8*)
Current timestep for integration.

`flags`(*=0*)
Various flags of the scene; 1 (Scene::LOCAL_COORDS): use local coordinate system rather than global one for per-interaction quantities (set automatically from the functor).

`isPeriodic`(*=false*)
Whether periodic boundary conditions are active.

`iter`(*=0*)
Current iteration (computational step) number

`localCoords`
Whether local coordianate system is used on interactions (set by Ig2Functor.

`selectedBody`(*=-1*)
Id of body that is selected by the user

`stopAtIter`(*=0*)
Iteration after which to stop the simulation.

`subStep`(*=-1*)
Number of sub-step; not to be changed directly. -1 means to run loop prologue (cell integration), 0…n-1 runs respective engines (n is number of engines), n runs epilogue (increment step number and time.

`subStepping`(*=false*)
Whether we currently advance by one engine in every step (rather than by single run through all engines).

`tags`(*=uninitalized*)
Arbitrary key=value associations (tags like mp3 tags: author, date, version, description etc.)

`time`(*=0*)
Simulation time (virtual time) [s]

`trackEnergy`(*=false*)
Whether energies are being traced.

1.12.7 Cell

class `yade.wrapper.Cell`(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`hSize`
Base cell vectors (columns of the matrix), updated at every step from velGrad (trsf accumulates applied velGrad transformations). Setting *hSize* during a simulation is not supported by most contact laws, it is only meant to be used at iteration 0 before any interactions have been created.

`hSize0`
Value of untransformed hSize, with respect to current trsf (computed as trsf<Cell.trsf>‘ [1] × :yref:‘hSize.

`homoDeform`(*=3*)
Deform (velGrad) the cell homothetically, by adjusting positions or velocities of particles. The values have the following meaning: 0: no homothetic deformation, 1: set absolute particle positions directly (when `velGrad` is non-zero), but without changing their velocity, 2: adjust

particle velocity (only when `velGrad` changed) with Δv_i=Δ v x_i. 3: as 2, but include a 2nd order term in addition – the derivative of 1 (convective term in the velocity update).

`prevVelGrad`(=*Matrix3r::Zero()*)
Velocity gradient in the previous step.

`refHSize`(=*Matrix3r::Identity()*)
Reference cell configuration, only used with OpenGLRenderer.dispScale. Updated automatically when hSize or trsf is assigned directly; also modified by utils.setRefSe3 (called e.g. by the **:gui:'Reference'** button in the UI).

`refSize`
Reference size of the cell (lengths of initial cell vectors, i.e. column norms of hSize).

Note: Modifying this value is deprecated, use setBox instead.

`setBox`(*(Vector3)arg2*) → None

Set Cell shape to be rectangular, with dimensions along axes specified by given argument. Shorthand for assigning diagonal matrix with respective entries to hSize.

setBox((Cell)arg1, (float)arg2, (float)arg3, (float)arg4) → None : Set Cell shape to be rectangular, with dimensions along `x`, `y`, `z` specified by arguments. Shorthand for assigning diagonal matrix with the respective entries to hSize.

`shearPt`(*(Vector3)arg2*) → Vector3
Apply shear (cell skew+rot) on the point

`shearTrsf`
Current skew+rot transformation (no resize)

`size`
Current size of the cell, i.e. lengths of the 3 cell lateral vectors contained in Cell.hSize columns. Updated automatically at every step.

`trsf`
Current transformation matrix of the cell, obtained from time integration of Cell.velGrad.

`unshearPt`(*(Vector3)arg2*) → Vector3
Apply inverse shear on the point (removes skew+rot of the cell)

`unshearTrsf`
Inverse of the current skew+rot transformation (no resize)

`velGrad`(=*Matrix3r::Zero()*)
Velocity gradient of the transformation; used in NewtonIntegrator. Values of velGrad accumulate in trsf at every step.

`volume`
Current volume of the cell.

`wrap`(*(Vector3)arg2*) → Vector3
Transform an arbitrary point into a point in the reference cell

`wrapPt`(*(Vector3)arg2*) → Vector3
Wrap point inside the reference cell, assuming the cell has no skew+rot.

1.13 Other classes

class `yade.wrapper.Engine`(*inherits Serializable*)
Basic execution unit of simulation, called from the simulation loop (O.engines)

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`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 `yade.wrapper.Cell`(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`hSize`
Base cell vectors (columns of the matrix), updated at every step from velGrad (trsf accumulates applied velGrad transformations). Setting *hSize* during a simulation is not supported by most contact laws, it is only meant to be used at iteration 0 before any interactions have been created.

`hSize0`
Value of untransformed hSize, with respect to current trsf (computed as trsf<Cell.trsf>‘ [1] × :yref:‘hSize.

`homoDeform`(=*3*)
Deform (velGrad) the cell homothetically, by adjusting positions or velocities of particles. The values have the following meaning: 0: no homothetic deformation, 1: set absolute particle positions directly (when `velGrad` is non-zero), but without changing their velocity, 2: adjust particle velocity (only when `velGrad` changed) with Δv_i=Δ v x_i. 3: as 2, but include a 2nd order term in addition – the derivative of 1 (convective term in the velocity update).

`prevVelGrad`(=*Matrix3r::Zero()*)
Velocity gradient in the previous step.

`refHSize`(=*Matrix3r::Identity()*)
Reference cell configuration, only used with OpenGLRenderer.dispScale. Updated automatically when hSize or trsf is assigned directly; also modified by utils.setRefSe3 (called e.g. by the **:gui:‘Reference‘** button in the UI).

`refSize`
Reference size of the cell (lengths of initial cell vectors, i.e. column norms of hSize).

Note: Modifying this value is deprecated, use setBox instead.

`setBox`(*(Vector3)arg2*) → None

Set Cell shape to be rectangular, with dimensions along axes specified by given argument. Shorthand for assigning diagonal matrix with respective entries to hSize.

setBox((Cell)arg1, (float)arg2, (float)arg3, (float)arg4) → None : Set Cell shape to be rectangular, with dimensions along `x`, `y`, `z` specified by arguments. Shorthand for assigning diagonal matrix with the respective entries to hSize.

`shearPt`(*(Vector3)arg2*) → Vector3
Apply shear (cell skew+rot) on the point

`shearTrsf`
Current skew+rot transformation (no resize)

`size`
Current size of the cell, i.e. lengths of the 3 cell lateral vectors contained in Cell.hSize columns. Updated automatically at every step.

`trsf`
Current transformation matrix of the cell, obtained from time integration of Cell.velGrad.

`unshearPt`(*(Vector3)arg2*) → Vector3
Apply inverse shear on the point (removes skew+rot of the cell)

`unshearTrsf`
Inverse of the current skew+rot transformation (no resize)

`velGrad`(=*Matrix3r::Zero()*)
Velocity gradient of the transformation; used in NewtonIntegrator. Values of velGrad accumulate in trsf at every step.

`volume`
Current volume of the cell.

`wrap`(*(Vector3)arg2*) → Vector3
Transform an arbitrary point into a point in the reference cell

`wrapPt`(*(Vector3)arg2*) → Vector3
Wrap point inside the reference cell, assuming the cell has no skew+rot.

class `yade.wrapper.TimingDeltas`

`data`
Get timing data as list of tuples (label, execTime[nsec], execCount) (one tuple per checkpoint)

`reset`() → None
Reset timing information

class `yade.wrapper.GlExtraDrawer`(*inherits Serializable*)
Performing arbitrary OpenGL drawing commands; called from OpenGLRenderer (see OpenGLRenderer.extraDrawers) once regular rendering routines will have finished.

This class itself does not render anything, derived classes should override the *render* method.

`dead`(=*false*)
Deactivate the object (on error/exception).

class `yade.wrapper.GlIGeomDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(IGeom)arg2*) → GlIGeomFunctor
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 associated with this dispatcher.

class `yade.wrapper.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 `yade.wrapper.GlShapeDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(Shape)arg2*) → GlShapeFunctor
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 associated with this dispatcher.

class `yade.wrapper.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 `yade.wrapper.Serializable`

`dict`() → dict
Return dictionary of attributes.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlExtra_LawTester`(*inherits GlExtraDrawer → Serializable*)
Find an instance of LawTester and show visually its data.

`tester`(=*uninitalized*)
Associated LawTester object.

class `yade.wrapper.GlStateDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(State)arg2*) → GlStateFunctor
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 associated with this dispatcher.

class `yade.wrapper.MatchMaker`(*inherits Serializable*)
Class matching pair of ids to return pre-defined (for a pair of ids defined in matches) or derived value (computed using algo) of a scalar parameter. It can be called (`id1`, `id2`, `val1=NaN`, `val2=NaN`) in both python and c++.

Note: There is a *converter* from python number defined for this class, which creates a new MatchMaker returning the value of that number; instead of giving the object instance therefore, you can only pass the number value and it will be converted automatically.

`algo`
Alogorithm used to compute value when no match for ids is found. Possible values are

- 'avg' (arithmetic average)
- 'min' (minimum value)
- 'max' (maximum value)
- 'harmAvg' (harmonic average)

The following algo algorithms do *not* require meaningful input values in order to work:

- 'val' (return value specified by val)
- 'zero' (always return 0.)

`computeFallback`(*(float)val1*, *(float)val2*) → float
Compute algo value for *val1* and *val2*, using algorithm specified by algo.

`matches`(=*uninitalized*)
Array of `(id1,id2,value)` items; queries matching `id1` + `id2` or `id2` + `id1` will return `value`

`val`(=*NaN*)
Constant value returned if there is no match and algo is `val`

class `yade.wrapper.GlBoundDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(Bound)arg2*) → GlBoundFunctor
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 associated with this dispatcher.

class `yade.wrapper.GlIPhysDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(IPhys)arg2*) → GlIPhysFunctor
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 associated with this dispatcher.

class `yade.wrapper.GlExtra_OctreeCubes`(*inherits GlExtraDrawer → Serializable*)
Render boxed read from file

`boxesFile`(=*uninitalized*)
File to read boxes from; ascii files with `x0 y0 z0 x1 y1 z1 c` records, where `c` is an integer specifying fill (0 for wire, 1 for filled).

`fillRangeDraw`(=*Vector2i(-2, 2)*)
Range of fill indices that will be rendered.

`fillRangeFill`(=*Vector2i(2, 2)*)
Range of fill indices that will be filled.

`levelRangeDraw`(=*Vector2i(-2, 2)*)
Range of levels that will be rendered.

`noFillZero`(=*true*)
Do not fill 0-fill boxed (those that are further subdivided)

class `yade.wrapper.Dispatcher`(*inherits Engine → Serializable*)
Engine dispatching control to its associated functors, based on types of argument it receives. This abstract base class provides no functionality in itself.

class `yade.wrapper.EnergyTracker`(*inherits Serializable*)
Storage for tracing energies. Only to be used if O.traceEnergy is True.

`clear`() → None
Clear all stored values.

`energies`(=*uninitalized*)
Energy values, in linear array

`items()` → list
Return contents as list of (name,value) tuples.

`keys()` → list
Return defined energies.

`total()` → float
Return sum of all energies.

Chapter 2

Yade modules

2.1 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:

```
>>> O.reset()
>>> from yade import utils
>>> O.bodies.append([utils.sphere((0,0,0),1),utils.sphere((0,0,1.9),1)])
[0, 1]
>>> O.engines=[InteractionLoop([Ig2_Sphere_Sphere_Dem3DofGeom(),],[Ip2_FrictMat_FrictMat_Frict
>>> utils.createInteraction(0,1)
<Interaction instance at 0x...>
```

>> is3d=IntrSmooth3d(0.003) >> is3d((0,0,0),lambda i: i.phys.normalForce) Vector3(0,0,0)

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 → → 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 -½(ε1+ε2)/ε .

Young's modulus is computed as σ/ε ; 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:

```
E G                                                 # elastic material parameters
epsCrackOnset relDuctility xiShear transStrainCoeff # tensile parameters; epsFr=epsCrackOnset*relDuctili
cohesionT tanPhi                                    # shear parameters
number_of_spheres number_of_links
id x y z r boundary                                 # spheres; boundary: -1 negative, 0 none, 1 positive
…
id1 id2 cp_x cp_y cp_z A                            # interactions; cp = contact point; A = cross-secti
```

yade.eudoxos.**particleConfinement**() → None

yade.eudoxos.**velocityTowardsAxis**(*(Vector3)axisPoint*, *(Vector3)axisDirection*, *(float)timeToAxis[]*, *(float)subtractDist[]*, *(float)perturbation[]*) → None

class yade._eudoxos.**HelixInteractionLocator2d**
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)periodStart=nan[]*, *(float)theta0=0[]*, *(float)thetaMin=nan[]*, *(float)thetaMax=nan[]*) → 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)
- **thetaMin** (*float*) – only interactions with ϑ *thetaMin* will be considered (NaN to deactivate)
- **thetaMax** (*float*) – only interactions with ϑ *thetaMax* will be considered (NaN to deactivate)

See utils.spiralProject.

hi
Return upper corner of the rectangle containing all interactions.

intrsAroundPt(*(Vector2)pt2d*, *(float)radius*) → list
Return list of interaction objects that are not further from *pt2d* than *radius* in the projection plane

lo
Return lower corner of the rectangle containing all interactions.

macroAroundPt(*(Vector2)pt2d*, *(float)radius*) → tuple
Compute macroscopic stress around given point; the interaction ($\mathfrak{n}$ and σ^{T} are rotated to the projection plane by ϑ (as given by utils.spiralProject) first, but no skew is applied). 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 (ϑ-rotated) 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 and ϑ-rotated.

Additionally, computes average of CpmPhys.omega ($\bar{\omega}$) and CpmPhys.kappaD ($\bar{\kappa}_D$). N is the number of interactions in the volume given.

Returns tuple of (N, $\boldsymbol{\sigma}$, $\bar{\omega}$, $\bar{\kappa}_D$).

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 instance methods 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*[, *(float)forceVolume=-1*]) → tuple

Return tuple of averaged stress tensor (as Matrix3), average omega and average kappa values. *forceVolume* can be used (if positive) rather than the sphere (with *maxDist* radius) volume for the computation. (This is useful if *point* and *maxDist* encompass empty space that you want to avoid.)

yade._eudoxos.**particleConfinement**() → None

yade._eudoxos.**velocityTowardsAxis**(*(Vector3)axisPoint*, *(Vector3)axisDirection*, *(float)timeToAxis*[, *(float)subtractDist*[, *(float)perturbation*]]) → None

2.2 yade.export module

Export geometry to various formats.

class yade.export.**VTKWriter**

USAGE: create object vtk_writer = VTKWriter('base_file_name'), add to engines PyRunner with command='vtk_writer.snapshot()'

snapshot()

yade.export.**text**(*filename*, *consider=<function <lambda> at 0x56f8c08>*)

Save sphere coordinates into a text file; the format of the line is: x y z r. Non-spherical bodies are silently skipped. Example added to examples/regular-sphere-pack/regular-sphere-pack.py

Parameters

***filename*: string** the name of the file, where sphere coordinates will be exported.

***consider*:** anonymous function(optional)

Returns number of spheres which were written.

`yade.export.textExt(`*filename*, *format='x_y_z_r'*, *consider=<function <lambda> at 0x56f8a28>*, *comment=''*`)`

Save sphere coordinates and other parameters into a text file in specific format.
Non-spherical bodies are silently skipped. Users can add here their own specific format, giving meaningful names. The first file row will contain the format name. Be sure to add the same format specification in ymport.textExt.

parameters

***filename*: string** the name of the file, where sphere coordinates will be exported.

***format*:** the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_matId*

***comment*:** the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '

#' for next lines.

***consider*:** anonymous function(optional)

Returns number of spheres which were written.

2.3 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

2.4 yade.log module

Access 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.

2.5 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.**SpherePack_toSimulation**(*self*, *rot=Matrix3(1, 0, 0, 0, 1, 0, 0, 0, 1)*, ***kw*)
Append spheres directly to the simulation. In addition calling O.bodies.append, this method also appropriately sets periodic cell information of the simulation.

```
>>> from yade import pack; from math import *
>>> sp=pack.SpherePack()
```

Create random periodic packing with 20 spheres:

```
>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20)
20
```

Virgin simulation is aperiodic:

```
>>> O.reset()
>>> O.periodic
False
```

Add generated packing to the simulation, rotated by 45° along +z

```
>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Periodic properties are transferred to the simulation correctly, including rotation:

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
```

```
>>> O.cell.hSize
Matrix3(3.53553,-3.53553,0, 3.53553,3.53553,0, 0,0,5)
```

The current state (even if rotated) is taken as mechanically undeformed, i.e. with identity transformation:

```
>>> O.cell.trsf
Matrix3(1,0,0, 0,1,0, 0,0,1)
```

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set Cell.trsf as well.
- ****kw** – passed to utils.sphere

Returns list of body ids added (like O.bodies.append)

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*, *returnSpherePack=None*, ***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 periodicity 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 ± (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× 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*, *initSize*, *rRelFuzz=0.0*, *memoizeDb=None*`)`
Generate periodic dense packing.

A cell of initSize is stuffed with as many spheres as possible, 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.

`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 automaticall

>>> for i in range(0,len(sp)): print sp[i][0], sp[i][1]   ## same, but accessing spheres by index
```

Special constructors

Construct from list of `[(c1,r1),(c2,r2),…]`. To convert two same-length lists of `centers` and `radii`, construct with `zip(centers,radii)`.

__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

appliedPsdScaling
A factor between 0 and 1, uniformly applied on all sizes of of the PSD.

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.

fromList((SpherePack)arg1, (object)centers, (object)radii) → **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.

`getClumps()` → tuple
Return lists of sphere ids sorted by clumps they belong to. The return value is (standalones,[clump1,clump2,…]), where each item is list of id's of spheres.

`hasClumps()` → bool
Whether this object contains clumps.

`load(`*(str)fileName*`)` → None
Load packing from external text file (current data will be discarded).

`makeCloud(`[*(Vector3)minCorner=Vector3(0, 0, 0)*[, *(Vector3)maxCorner=Vector3(0, 0, 0)*[, *(float)rMean=-1*[, *(float)rRelFuzz=0*[, *(int)num=-1*[, *(bool)periodic=False*[, *(float)porosity=0.5*[, *(object)psdSizes=*[][, *(object)psdCumm=*[][, *(bool)distributeMass=False*[, *(int)seed=0*[, *(Matrix3)hSize=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]]]]]]]]]]]]) → int
Create random loose packing enclosed in a parallelepiped. Sphere radius distribution can be specified using one of the following ways:

1. *rMean*, *rRelFuzz* and *num* gives uniform radius distribution in *rMean* (1 ± *rRelFuzz*). Less than *num* spheres can be generated if it is too high.
2. *rRelFuzz*, *num* and (optional) *porosity*, which estimates mean radius so that *porosity* is attained at the end. *rMean* must be less than 0 (default). *porosity* is only an initial guess for the generation algorithm, which will retry with higher porosity until the prescibed *num* is obtained.
3. *psdSizes* and *psdCumm*, two arrays specifying points of the particle size distribution function. As many spheres as possible are generated.
4. *psdSizes*, *psdCumm*, *num*, and (optional) *porosity*, like above but if *num* is not obtained, *psdSizes* will be scaled down uniformly, until *num* is obtained (see appliedPsdScaling).

By default (with `distributeMass==False`), the distribution is applied to particle radii. The usual sense of "particle size distribution" is the distribution of *mass fraction* (rather than particle count); this can be achieved with `distributeMass=True`.

If *num* is defined, then sizes generation is deterministic, giving the best fit of target distribution. It enables spheres placement in descending size order, thus giving lower porosity than the random generation.

Parameters

- **minCorner** (*Vector3*) – lower corner of an axis-aligned box
- **maxCorner** (*Vector3*) – upper corner of an axis-aligned box
- **hSize** (*Matrix3*) – base vectors of a generalized box (arbitrary parallelepiped, typically Cell::hSize), superseeds minCorner and maxCorner if defined. For periodic boundaries only.
- **rMean** (*float*) – mean radius or spheres
- **rRelFuzz** (*float*) – dispersion of radius relative to rMean

- **num** (*int*) – number of spheres to be generated. If negavite (default), generate as many as possible with stochastic sizes, ending after a fixed number of tries to place the sphere in space, else generate exactly *num* spheres with deterministic size distribution.
- **periodic** (*bool*) – whether the packing to be generated should be periodic
- **porosity** (*float*) – initial guess for the iterative generation procedure (if *num*>1). The algorithm will be retrying until the number of generated spheres is *num*. The first iteration tries with the provided porosity, but next iterations increase it if necessary (hence an initialy high porosity can speed-up the algorithm). If *psdSizes* is not defined, *rRelFuzz* (z) and *num* (N) are used so that the porosity given (ρ) is approximately achieved at the end of generation, $r_m = \sqrt[3]{\frac{V(1-\rho)}{\frac{4}{3}\pi(1+z^2)N}}$. The default is ρ=0.5. The optimal value depends on *rRelFuzz* or *psdSizes*.
- **psdSizes** – sieve sizes (particle diameters) when particle size distribution (PSD) is specified
- **psdCumm** – cummulative fractions of particle sizes given by *psdSizes*; must be the same length as *psdSizes* and should be non-decreasing
- **distributeMass** (*bool*) – if `True`, given distribution will be used to distribute sphere's mass rather than radius of them.
- **seed** – number used to initialize the random number generator.

Returns number of created spheres, which can be lower than *num* depending on the method used.

`makeClumpCloud`(*(Vector3)minCorner*, *(Vector3)maxCorner*, *(object)clumps*[, *(bool)periodic=False*[, *(int)num=-1*]]) → int

Create random loose packing of clumps within box given by *minCorner* and *maxCorner*. Clumps are selected with equal probability. At most *num* clumps will be positioned if *num* is positive; otherwise, as many clumps as possible will be put in space, until maximum number of attemps to place a new clump randomly is attained.

`particleSD`(*(Vector3)minCorner*, *(Vector3)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=*[][, *(object)passing=*[][, *(bool)passingIsNotPercentageButCount=False*[, *(int)seed=0*]]]]) → int

Create random packing enclosed in box given by minCorner and maxCorner, containing numSph spheres. Returns number of created spheres, which can be < num if the packing is too tight. The computation is done according to the given psd.

`particleSD2`(*(object)radii*, *(object)passing*, *(int)numSph*[, *(bool)periodic=False*[, *(float)cloudPorosity=0.80000000000000004*[, *(int)seed=0*]]]) → int

Create random packing following the given particle size distribution (radii and volume/mass passing for each fraction) and total number of particles *numSph*. The cloud size (periodic or aperiodic) is computed from the PSD and is always cubic.

`psd`([*(int)bins=50*[, *(bool)mass=True*]]) → tuple

Return particle size distribution of the packing. :param int bins: number of bins between minimum and maximum diameter :param mass: Compute relative mass rather than relative particle count for each bin. Corresponds to distributeMass parameter for makeCloud. :returns: tuple of `(cumm,edges)`, where `cumm` are cummulative fractions for respective diameters and `edges` are those diameter values. Dimension of both arrays is equal to `bins+1`.

`psdScaleExponent`

[Deprecated] Defined for compatibility, no effect.

`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.

toSimulation()
Append spheres directly to the simulation. In addition calling O.bodies.append, this method also appropriately sets periodic cell information of the simulation.

>>> from yade import pack; from math import * >>> sp=pack.SpherePack()

Create random periodic packing with 20 spheres:

>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20) 20

Virgin simulation is aperiodic:

>>> O.reset() >>> O.periodic False

Add generated packing to the simulation, rotated by 45° along +z

>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1)) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Periodic properties are transferred to the simulation correctly, including rotation:

>>> O.periodic True >>> O.cell.refSize Vector3(5,5,5) >>> O.cell.hSize Matrix3(3.53553,-3.53553,0, 3.53553,3.53553,0, 0,0,5)

The current state (even if rotated) is taken as mechanically undeformed, i.e. with identity transformation:

>>> O.cell.trsf Matrix3(1,0,0, 0,1,0, 0,0,1)

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set Cell.trsf as well.
- ****kw** – passed to utils.sphere

Returns list of body ids added (like O.bodies.append)

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.inParallelepiped`(*inherits Predicate*)
Parallelepiped predicate

`__init__`(*(Vector3)o*, *(Vector3)a*, *(Vector3)b*, *(Vector3)c*) → None
Ctor taking four points: `o` (for origin) and then `a`, `b`, `c` which define endpoints of 3 respective edges from `o`.

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).

2.6 yade.plot module

Module containing utility functions for plotting inside yade. See examples/simple-scene/simple-scene-plot.py or examples/concrete/uniax.py for example of usage.

`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.plots`
dictionary x-name -> (yspec,...), where yspec is either y-name or (y-name,'line-specification'). If `(yspec,...)` is `None`, then the plot has meaning of image, which will be taken from respective field of plot.imgData.

`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.live`
Enable/disable live plot updating. Disabled by default for now, since it has a few rough edges.

`yade.plot.liveInterval`
Interval for the live plot updating, in seconds.

`yade.plot.autozoom`
Enable/disable automatic plot rezooming after data update.

`yade.plot.plot`(*noShow=False*, *subPlots=True*)
Do the actual plot, which is either shown on screen (and nothing is returned: if *noShow* is `False`) or, if *noShow* is `True`, returned as matplotlib's Figure object or list of them.

You can use

```
>>> from yade import plot
>>> plot.resetData()
>>> plot.plots={'foo':('bar',)}
>>> plot.plot(noShow=True).savefig('someFile.pdf')
>>> import os
>>> os.path.exists('someFile.pdf')
True
```

to save the figure to file automatically.

Note: For backwards compatibility reasons, *noShow* option will return list of figures for multiple figures but a single figure (rather than list with 1 element) if there is only 1 figure.

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.**splitData**()
Make all plots discontinuous at this point (adds nan's to all data fields)

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.**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 padded with nan's, unspecified data will be nan (nan's don't appear in graphs). This way, equal length of all data is assured so that they can be plotted one against any other.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.resetData()
>>> plot.addData(a=1)
>>> plot.addData(b=2)
>>> plot.addData(a=3,b=4)
>>> pprint(plot.data)
{'a': [1, nan, 3], 'b': [nan, 2, 4]}
```

Some sequence types can be given to addData; they will be saved in synthesized columns for individual components.

```
>>> plot.resetData()
>>> plot.addData(c=Vector3(5,6,7),d=Matrix3(8,9,10, 11,12,13, 14,15,16))
>>> pprint(plot.data)
{'c_x': [5.0],
 'c_y': [6.0],
 'c_z': [7.0],
 'd_xx': [8.0],
 'd_xy': [9.0],
 'd_xz': [10.0],
 'd_yy': [12.0],
 'd_yz': [11.0],
 'd_zx': [14.0],
 'd_zy': [15.0],
 'd_zz': [16.0]}
```

yade.plot.**addAutoData**()
Add data by evaluating contents of plot.plots. Expressions rasing exceptions will be handled gracefully, but warning is printed for each.

```
>>> from yade import plot
>>> from pprint import pprint
>>> O.reset()
>>> plot.resetData()
>>> plot.plots={'O.iter':('O.time',None,'numParticles=len(O.bodies)')}
>>> plot.addAutoData()
>>> pprint(plot.data)
{'O.iter': [0], 'O.time': [0.0], 'numParticles': [0]}
```

Note that each item in plot.plots can be

- an expression to be evaluated (using the `eval` builtin);
- `name=expression` string, where `name` will appear as label in plots, and expression will be evaluated each time;
- a dictionary-like object – current keys are labels of plots and current values are added to plot.data. The contents of the dictionary can change over time, in which case new lines will be created as necessary.

A simple simulation with plot can be written in the following way; note how the energy plot is specified.

```
>>> from yade import plot, utils
>>> plot.plots={'i=O.iter':(O.energy,None,'total energy=O.energy.total()')}
>>> # we create a simple simulation with one ball falling down
>>> plot.resetData()
>>> O.bodies.append(utils.sphere((0,0,0),1))
0
>>> O.dt=utils.PWaveTimeStep()
>>> O.engines=[
...    ForceResetter(),
...    GravityEngine(gravity=(0,0,-10)),
...    NewtonIntegrator(damping=.4,kinSplit=True),
...    # get data required by plots at every step
...    PyRunner(command='yade.plot.addAutoData()',iterPeriod=1,initRun=True)
... ]
>>> O.trackEnergy=True
>>> O.run(2,True)
>>> pprint(plot.data)
{'gravWork': [0.0, -25.13274...],
 'i': [0, 1],
 'kinRot': [0.0, 0.0],
 'kinTrans': [0.0, 7.5398...],
 'nonviscDamp': [0.0, 10.0530...],
 'total energy': [0.0, -7.5398...]}
```

`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.bz2` (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** – extension for `baseName` defaults to terminal name; fine for png for example; if you use `cairopdf`, you should also say `extension='pdf'` however
- **timestamp** (*bool*) – append numeric time to the basename
- **varData** (*bool*) – 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 of the gnuplot file created.

`yade.plot.saveDataTxt`(*fileName*, *vars=None*)

Save plot data into a (optionally compressed) text file. The first line contains a comment (starting

with #) giving variable name for each of the columns. This format is suitable for being loaded for further processing (outside yade) with `numpy.genfromtxt` function, which recognizes those variable names (creating numpy array with named entries) and handles decompression transparently.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.reset()
>>> plot.addData(a=1,b=11,c=21,d=31)  # add some data here
>>> plot.addData(a=2,b=12,c=22,d=32)
>>> pprint(plot.data)
{'a': [1, 2], 'b': [11, 12], 'c': [21, 22], 'd': [31, 32]}
>>> plot.saveDataTxt('/tmp/dataFile.txt.bz2',vars=('a','b','c'))
>>> import numpy
>>> d=numpy.genfromtxt('/tmp/dataFile.txt.bz2',dtype=None,names=True)
>>> d['a']
array([1, 2])
>>> d['b']
array([11, 12])
```

Parameters

- **fileName** – file to save data to; if it ends with `.bz2` / `.gz`, the file will be compressed using bzip2 / gzip.
- **vars** – Sequence (tuple/list/set) of variable names to be saved. If `None` (default), all variables in plot.plot are saved.

`yade.plot.savePlotSequence`(*fileBase*, *stride=1*, *imgRatio=(5, 7)*, *title=None*, *titleFrames=20*, *lastFrames=30*)

Save sequence of plots, each plot corresponding to one line in history. It is especially meant to be used for utils.makeVideo.

Parameters

- **stride** – only consider every stride-th line of history (default creates one frame per each line)
- **title** – Create title frame, where lines of title are separated with newlines (`\n`) and optional subtitle is separated from title by double newline.
- **titleFrames** (*int*) – Create this number of frames with title (by repeating its filename), determines how long the title will stand in the movie.
- **lastFrames** (*int*) – Repeat the last frame this number of times, so that the movie does not end abruptly.

Returns List of filenames with consecutive frames.

2.7 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/interactions and plotting the results. There are 2 basic components: flatteners and extractors.

The algorithms operate on bodies (default) or interactions, depending on the `intr` parameter of post2d.data.

2.7.1 Flatteners

Instance of classes that convert 3d (model) coordinates to 2d (plot) coordinates. Their interface is defined by the post2d.Flatten class (`__call__`, `planar`, `normal`).

2.7.2 Extractors

Callable objects returning scalar or vector value, given a body/interaction object. If a 3d vector is returned, Flattener.planar is called, which should return only in-plane components of the vector.

2.7.3 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__()
:param bool useRef: use reference positions rather than actual positions (only meaningful when operating on Bodies) :param {0,1,2} axis: 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 onto plane 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.HelixFlatten`(*inherits Flatten*)
Class converting 3d point to 2d based on projection from helix. The y-axis in the projection corresponds to the rotation axis

`__init__()`
:param bool useRef: use reference positions rather than actual positions :param (ϑmin,ϑmax) thetaRange: bodies outside this range will be discarded :param float dH_dTheta: inclination of the spiral (per radian) :param {0,1,2} axis: axis of rotation of the spiral :param float periodStart: height of the spiral for zero angle

`normal()`

`planar()`

`yade.post2d.data`(*extractor*, *flattener*, *intr=False*, *onlyDynamic=True*, *stDev=None*, *relThreshold=3.0*, *perArea=0*, *div=(50, 50)*, *margin=(0, 0)*, *radius=1*)
Filter all bodies/interactions, 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.

The `intr` parameter determines whether we operate on bodies or interactions; the extractor provided should expect to receive body/interaction.

Parameters

- **extractor** (*callable*) – receives Body (or Interaction, if `intr` is `True`) instance, should return scalar, a 2-tuple (vector fields) or None (to skip that body/interaction)
- **flattener** (*callable*) – post2d.Flatten instance, receiving body/interaction, returns its 2d coordinates or `None` (to skip that body/interaction)
- **intr** (*bool*) – operate on interactions rather than bodies
- **onlyDynamic** (*bool*) – skip all non-dynamic bodies
- **stDev** (*float/None*) – standard deviation for averaging, enables smoothing; `None` (default) means raw mode, where discrete points are returned
- **relThreshold** (*float*) – threshold for the gaussian weight function relative to stDev (smooth mode only)
- **perArea** (*int*) – if 1, compute weightedSum/weightedArea rather than weighted average (weightedSum/sumWeights); the first is useful to compute average stress; if 2, compute averages on subdivision elements, not using weight function
- **div** (*(int,int)*) – number of cells for the gaussian grid (smooth mode only)
- **margin** (*(float,float)*) – x,y margins around bounding box for data (smooth mode only)
- **radius** (*float/callable*) – Fallback value for radius (for raw plotting) for non-spherical bodies or interactions; if a callable, receives body/interaction and returns radius

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*, *cbar=False*, *aspect='equal'*, ***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.axesinstance where the figure will be plotted; if None, will be created from scratch.
- **data** – value returned by post2d.data
- **clabel** (*bool*) – show contour labels (smooth mode only), or annotate cells with numbers inside (with perArea==2)
- **cbar** (*bool*) – show colorbar (equivalent to calling pylab.colorbar(mappable) on the returned mappable)

Returns tuple of `(axes,mappable)`; mappable can be used in further calls to pylab.colorbar.

2.8 yade.qt module

Common initialization core for yade.

This file is executed when anything is imported from yade for the first time. It loads yade plugins and injects c++ class constructors to the ___builtins___ (that might change in the future, though) namespace, making them available everywhere.

`yade.qt.Renderer`() → OpenGLRenderer
Return the active OpenGLRenderer object.

`yade.qt.View`() → GLViewer
Create a new 3d view.

`yade.qt.bin`()
bin(QTextStream) -> QTextStream

`yade.qt.center`() → None
Center all views.

`yade.qt.hex`()
hex(QTextStream) -> QTextStream

`yade.qt.oct`()
oct(QTextStream) -> QTextStream

`yade.qt.views`() → list
Return list of all open qt.GLViewer objects

class `yade.qt._GLViewer.GLViewer`

`__init__`()
Raises an exception This class cannot be instantiated from Python

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).

close() → None

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 OpenGLRenderer.

scale
Scale of the view (?)

sceneRadius
Visible scene radius.

screenSize
Size of the viewer's window, in scree pixels

selection

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._GLViewer.**Renderer**() → OpenGLRenderer
Return the active OpenGLRenderer object.

class yade.qt._GLViewer.**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. snap00004.png).

counter(=*0*)
Number that will be appended to fileBase when the next snapshot is saved (incremented at every save). *(auto-updated)*

deadTimeout(=*3*)
Timeout for 3d operations (opening new view, saving snapshot); after timing out, throw exception (or only report error if *ignoreErrors*) and make myself dead. [s]

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*)
Only report errors instead of throwing exceptions, in case of timeouts.

msecSleep(=*0*)
number of msec to sleep after snapshot (to prevent 3d hw problems) [ms]

plot(=*uninitalized*)
Name of field in plot.imgData to which taken snapshots will be appended automatically.

snapshots(=*uninitalized*)
Files that have been created so far

yade.qt._GLViewer.**View**() → GLViewer
Create a new 3d view.

yade.qt._GLViewer.**center**() → None
Center all views.

yade.qt._GLViewer.**views**() → list
Return list of all open qt.GLViewer objects

2.9 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%
```

2.10 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*, *skipFree=False*)
Return average numer of interactions per particle, also known as *coordination number* Z. This number is defined as

$$Z = 2C/N$$

where C is number of contacts and N is number of particles.

With *skipFree*, particles not contributing to stable state of the packing are skipped, following equation (8) given in [Thornton2000]:

$$Z_m = \frac{2C - N_1}{N - N_0 - N_1}$$

Parameters

- **cutoff** – cut some relative part of the sample's bounding box away.
- **skipFree** – see above.

yade.utils.**bodyNumInteractionsHistogram**([*(tuple)aabb*]) → tuple

yade.utils.**bodyStressTensors**([*(bool)revertSign=False*]) → tuple
Compute and return a table with per-particle stress tensors. Each tensor represents the average stress in one particle, obtained from the contour integral of applied load as detailed below. This definition is considering each sphere as a continuum. It can be considered exact in the context of spheres at static equilibrium, interacting at contact points with negligible volume changes of the solid phase (this last assumption is not restricting possible deformations and volume changes at the packing scale).

Proof:

First, we remark the identity: $\sigma_{ij} = \delta_{ij}\sigma_{ij} = x_{i,j}\sigma_{ij} = (x_i\sigma_{ij})_{,j} - x_i\sigma_{ij,j}$.

At equilibrium, the divergence of stress is null: $\sigma_{ij,j} = 0$. Consequently, after divergence theorem: $\frac{1}{V}\int_V \sigma_{ij}dV = \frac{1}{V}\int_V (x_i\sigma_{ij})_{,j}dV = \frac{1}{V}\int_{\partial V} x_i.\sigma_{ij}.\mathbf{n_j}.dS = \frac{1}{V}\sum_k x_i^k.f_j^k$.

The last equality is implicitely based on the representation of external loads as Dirac distributions whose zeros are the so-called *contact points*: 0-sized surfaces on which the *contact forces* are applied, located at x_i in the deformed configuration.

A weighted average of per-body stresses will give the average stress inside the solid phase. There is a simple relation between the stress inside the solid phase and the stress in an equivalent continuum in the absence of fluid pressure. For porosity n, the relation reads: $\sigma_{ij}^{equ.} = (1-n)\sigma_{ij}^{solid}$.

Parameters

- **revertSign** (*bool*) – invert the sign of returned tensors components.

`yade.utils.box`(*center*, *extents*, *orientation=*[, *1*, *0*, *0*, *0*], *dynamic=None*, *fixed=False*, *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.chainedCylinder`(*begin=Vector3(0, 0, 0)*, *end=Vector3(1, 0, 0)*, *radius=0.20000000000000001*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*)
Create and connect a chainedCylinder with given parameters. The shape generated by repeted calls of this function is the Minkowski sum of polyline and sphere.

Parameters

- **radius** (*Real*) – radius of sphere in the Minkowski sum.
- **begin** (*Vector3*) – first point positioning the line in the Minkowski sum
- **last** (*Vector3*) – last point positioning the line in the Minkowski sum

In order to build a correct chain, last point of element of rank N must correspond to first point of element of rank N+1 in the same chain (with some tolerance, since bounding boxes will be used to create connections.

Returns Body object with the ChainedCylinder shape.

`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 IGeomDispatcher and IPhysDispatcher (might be both hidden in InteractionLoop). 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.elasticEnergy`(*(tuple)arg1*) → float

`yade.utils.fabricTensor`([*(bool)splitTensor=False*[, *(bool)revertSign=False*[, *(float)thresholdForce=nan*]]]) → tuple
Compute the fabric tensor of the periodic cell. The original paper can be found in [Satake1982].

Parameters

- **splitTensor** (*bool*) – split the fabric tensor into two parts related to the strong and weak contact forces respectively.

- **revertSign** (*bool*) – it must be set to true if the contact law's convention takes compressive forces as positive.
- **thresholdForce** (*Real*) – if the fabric tensor is split into two parts, a threshold value can be specified otherwise the mean contact force is considered by default. It is worth to note that this value has a sign and the user needs to set it according to the convention adopted for the contact law. To note that this value could be set to zero if one wanted to make distinction between compressive and tensile forces.

`yade.utils.facet`(*vertices*, *dynamic=None*, *fixed=True*, *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** (*bool*) – set Body.bounded
- **color** (*Vector3-or-None*) – color of the facet; random color will be assigned if `None`.

See utils.sphere's documentation for meaning of other parameters.

`yade.utils.flipCell`(`[`*(Matrix3)flip=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*`]`) → Matrix3
Flip periodic cell so that angles between $\mathbb{R}^3$ 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 $\mathbb{R}^3$; 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)tc*, *(float)en*, *(float)es*) → dict
Compute viscoelastic interaction parameters from analytical solution of a pair spheres collision problem:

:nowrap:

begin{align*}k_n&=frac{m}{t_c^2}left(pi^2+(ln e_n)^2right)\ c_n&=-frac{2m}{t_c}ln e_n \k_t&=frac27frac{m}{t_c^2}left(pi^2+(ln e_t)^2right) \ c_t=-frac27frac{m}{t_c}ln e_t end{align*}

where k_n, c_n are normal elastic and viscous coefficients and k_t, c_t shear elastic and viscous coefficients. For details see [Pournin2001].

Parameters

- **m** (*float*) – sphere mass m
- **tc** (*float*) – collision time t_c
- **en** (*float*) – normal restitution coefficient e_n
- **es** (*float*) – tangential restitution coefficient e_s

Returns dictionary with keys `kn` (the value of k_n), `cn` (c_n), `kt` (k_t), `ct` (c_t).

`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(`[*(bool)findMaxId=False*]`)` → object
Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i v_i^2 + \boldsymbol{\omega}(\mathbf{I}\boldsymbol{\omega}^T)\right).$$

For aspherical bodies, the inertia tensor $\mathbf{I}$ is transformed to global frame, before multiplied by $\boldsymbol{\omega}$, therefore the value should be accurate.

`yade.utils.loadVars(`*mark=None*`)`
Load variables from utils.saveVars, which are saved inside the simulation. If `mark==None`, all save variables are loaded. Otherwise only those with the mark passed.

`yade.utils.makeVideo(`*frameSpec*, *out*, *renameNotOverwrite=True*, *fps=24*, *kbps=6000*, *bps=None*`)`
Create a video from external image files using mencoder. Two-pass encoding using the default mencoder codec (mpeg4) is performed, running multi-threaded with number of threads equal to number of OpenMP threads allocated for Yade.

Parameters

- **frameSpec** – wildcard | sequence of filenames. If list or tuple, filenames to be encoded in given order; otherwise wildcard understood by mencoder's mf:// URI option (shell wildcards such as `/tmp/snap-*.png` or and printf-style pattern like `/tmp/snap-%05d.png`)
- **out** (*str*) – file to save video into
- **renameNotOverwrite** (*bool*) – if True, existing same-named video file will have *-number* appended; will be overwritten otherwise.
- **fps** (*int*) – Frames per second (`-mf fps=…`)
- **kbps** (*int*) – Bitrate (`-lavcopts vbitrate=…`) in kb/s

`yade.utils.maxOverlapRatio()` → float
Return maximum overlap ration in interactions (with ScGeom) of two spheres. The ratio is computed as $\frac{u_N}{2(r_1 r_2)/r_1+r_2}$, where u_N is the current overlap distance and r_1, r_2 are radii of the two spheres in contact.

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.**normalShearStressTensors**([*(bool)compressionPositive=False*]) → tuple
Compute overall stress tensor of the periodic cell decomposed in 2 parts, one contributed by normal forces, the other by shear forces. The formulation can be found in [Thornton2000], eq. (3):

$$\sigma_{ij} = \frac{2}{V}\sum \mathsf{RN}\mathbf{n}_i\mathbf{n}_j + \frac{2}{V}\sum \mathsf{RT}\mathbf{n}_i\mathbf{t}_j$$

where V is the cell volume, R is "contact radius" (in our implementation, current distance between particle centroids), $\mathbf{n}$ is the normal vector, $\mathbf{t}$ is a vector perpendicular to $\mathbf{n}$, N and T are norms of normal and shear forces.

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*, *noShow=False*)
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.

Returns If *noShow* is `False`, displays the figure and returns nothing. If *noShow*, the figure object is returned without being displayed (works the same way as plot.plot).

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.**porosity**([*(float)volume=-1*]) → float
Compute packing porosity $\frac{V-V_s}{V}$ where V is overall volume and V_s is volume of spheres.

Parameters

- **volume** (*float*) – overall volume which must be specified for aperiodic simulations. For periodic simulations, current volume of the Cell is used.

yade.utils.**ptInAABB**(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*) → bool
Return True/False whether the point p is within box given by its min and max 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=True*, *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 (the `description` column is synthesized if absent,see utils.TableParamReader);

O.tags['description']=... # assigns the description column; might be synthesized O.tags['params']="name1=val1,name2=val2,..." # all explicitly assigned parameters O.tags['defaultParams']="unassignedName1=defaultValue1,..." # parameters that were left at their defaults O.tags['d.id']=O.tags['id']+'.'+O.tags['description'] O.tags['id.d']=O.tags['description']+'.'+O.tags['id']

All parameters (default as well as settable) are saved using utils.saveVars('table').

Parameters

- **tableFile** – text file (with one value per blank-separated columns)
- **tableLine** (*int*) – number of line where to get the values from
- **noTableOk** (*bool*) – if False, raise exception if the file cannot be open; use default values otherwise
- **unknownOk** (*bool*) – do not raise exception if unknown column name is found in the file, and 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.**runningInBatch**()
Tell whether we are running inside the batch or separately.

yade.utils.**saveVars**(*mark=''*, *loadNow=True*, ***kw*)
Save passed variables into the simulation so that it can be recovered when the simulation is loaded again.

For example, variables *a*, *b* and *c* are defined. To save them, use:

```
>>> from yade import utils
>>> utils.saveVars('mark',a=1,b=2,c=3)
>>> from yade.params.mark import *
>>> 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 yade.params.*mark* module. The *loadNow* parameter calls utils.loadVars after saving automatically.

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=None*, *fixed=False*, *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
- **Vector3-or-None** – body's color, as normalized RGB; random color will be assigned if ''None'.
- **material** –

specify Body.material; different types are accepted:

- 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]
- string: label of an existing material that will be used
- Material instance: this instance will be used
- callable: will be called without arguments; returned Material value will be used (Material factory object, if you like)

- **mask** (*int*) – 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.stressTensorOfPeriodicCell([*(bool)smallStrains=False*]) → Matrix3

Compute overall (macroscopic) stress of periodic cell using equation published in [Kuhl2001]:

$$\sigma = \frac{1}{V} \sum_{c} l^c [\mathbf{N}^c f_N^c + \mathbf{T}^{cT} \cdot \mathbf{f}_T^c],$$

where V is volume of the cell, l^c length of interaction c, f_N^c normal force and $\mathbf{f}_T^c$ shear force. Sumed are values over all interactions c. $\mathbf{N}^c$ and $\mathbf{T}^{cT}$ are projection tensors (see the original publication for more details):

$$\mathbf{N} = \mathbf{n} \otimes \mathbf{n} \rightarrow N_{ij} = n_i n_j$$

$$\mathbf{T}^T = \mathbf{I}_{sym} \cdot \mathbf{n} - \mathbf{n} \otimes \mathbf{n} \otimes \mathbf{n} \rightarrow T^T_{ijk} = \frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})n_l - n_i n_j n_k$$

$$\mathbf{T}^T \cdot \mathbf{f}_T \equiv T^T_{ijk} f_k = (\delta_{ik} n_j/2 + \delta_{jk} n_i/2 - n_i n_j n_k) f_k = n_j f_i/2 + n_i f_j/2 - n_i n_j n_k f_k,$$

where $\mathbf{n}$ is unit vector oriented along the interaction (normal) and δ is Kronecker's delta. As $\mathbf{n}$ and $\mathbf{f}_T$ are perpendicular (therfore $n_i f_i = 0$) we can write

$$\sigma_{ij} = \frac{1}{V} \sum l[n_i n_j f_N + n_j f_i^T/2 + n_i f_j^T/2]$$

Parameters

- **smallStrains** (*bool*) – if false (large strains), real values of volume and interaction lengths are computed. If true, only refLength of interactions and initial volume are computed (can save some time).

Returns macroscopic stress tensor as Matrix3

`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** (*float*) – number of section that will be used to estimate cross-section
- **axis** (*{0,1,2}*) – 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.waitIfBatch()`
Block the simulation if running inside a batch. Typically used at the end of script so that it does not finish prematurely in batch mode (the execution would be ended in such a case).

`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)

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(`$\left[$*(float)cutoff=0.0*$\left[$, *(bool)centers=False*$\right]\right]$`)` → 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.bodyStressTensors`([*(bool)revertSign=False*]) → tuple
Compute and return a table with per-particle stress tensors. Each tensor represents the average stress in one particle, obtained from the contour integral of applied load as detailed below. This definition is considering each sphere as a continuum. It can be considered exact in the context of spheres at static equilibrium, interacting at contact points with negligible volume changes of the solid phase (this last assumption is not restricting possible deformations and volume changes at the packing scale).

Proof:

First, we remark the identity: $\sigma_{ij} = \delta_{ij}\sigma_{ij} = x_{i,j}\sigma_{ij} = (x_i\sigma_{ij})_{,j} - x_i\sigma_{ij,j}$.

At equilibrium, the divergence of stress is null: $\sigma_{ij,j} = 0$. Consequently, after divergence theorem: $\frac{1}{V}\int_V \sigma_{ij}\,dV = \frac{1}{V}\int_V (x_i\sigma_{ij})_{,j}\,dV = \frac{1}{V}\int_{\partial V} x_i.\sigma_{ij}.n_j.dS = \frac{1}{V}\sum_k x_i^k.f_j^k$.

The last equality is implicitely based on the representation of external loads as Dirac distributions whose zeros are the so-called *contact points*: 0-sized surfaces on which the *contact forces* are applied, located at x_i in the deformed configuration.

A weighted average of per-body stresses will give the average stress inside the solid phase. There is a simple relation between the stress inside the solid phase and the stress in an equivalent continuum in the absence of fluid pressure. For porosity n, the relation reads: $\sigma_{ij}^{equ.} = (1-n)\sigma_{ij}^{solid}$.

Parameters

- **revertSign** (*bool*) – invert the sign of returned tensors components.

`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 IGeomDispatcher and IPhysDispatcher (might be both hidden in InteractionLoop). 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.fabricTensor`([*(bool)splitTensor=False*[, *(bool)revertSign=False*[, *(float)thresholdForce=nan*]]]) → tuple
Compute the fabric tensor of the periodic cell. The original paper can be found in [Satake1982].

Parameters

- **splitTensor** (*bool*) – split the fabric tensor into two parts related to the strong and weak contact forces respectively.

- **revertSign** (*bool*) – it must be set to true if the contact law's convention takes compressive forces as positive.
- **thresholdForce** (*Real*) – if the fabric tensor is split into two parts, a threshold value can be specified otherwise the mean contact force is considered by default. It is worth to note that this value has a sign and the user needs to set it according to the convention adopted for the contact law. To note that this value could be set to zero if one wanted to make distinction between compressive and tensile forces.

`yade._utils.flipCell(`[*(Matrix3)flip=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]`)` → Matrix3

Flip periodic cell so that angles between $\mathbb{R}^3$ 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 $\mathbb{R}^3$; 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)tc*, *(float)en*, *(float)es*`)` → dict

Compute viscoelastic interaction parameters from analytical solution of a pair spheres collision problem:

:nowrap:

begin{align*}k_n&=frac{m}{t_c^2}left(pi^2+(ln e_n)^2right)\ c_n&=-frac{2m}{t_c}ln e_-n \k_t&=frac27frac{m}{t_c^2}left(pi^2+(ln e_t)^2right) \ c_t=-frac27frac{m}{t_c}ln e_t end{align*}

where k_n, c_n are normal elastic and viscous coefficients and k_t, c_t shear elastic and viscous coefficients. For details see [Pournin2001].

Parameters

- **m** (*float*) – sphere mass m
- **tc** (*float*) – collision time t_c
- **en** (*float*) – normal restitution coefficient e_n
- **es** (*float*) – tangential restitution coefficient e_s

Returns dictionary with keys `kn` (the value of k_n), `cn` (c_n), `kt` (k_t), `ct` (c_t).

`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**([*(bool)findMaxId=False*]) → object
Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i v_i^2 + \boldsymbol{\omega}(\mathbf{I}\boldsymbol{\omega}^T)\right).$$

For aspherical bodies, the inertia tensor $\mathbf{I}$ is transformed to global frame, before multiplied by $\boldsymbol{\omega}$, therefore the value should be accurate.

yade._utils.**maxOverlapRatio**() → float
Return maximum overlap ration in interactions (with ScGeom) of two spheres. The ratio is computed as $\frac{u_N}{2(r_1 r_2)/r_1+r_2}$, where u_N is the current overlap distance and r_1, r_2 are radii of the two spheres in contact.

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.**normalShearStressTensors**([*(bool)compressionPositive=False*]) → tuple
Compute overall stress tensor of the periodic cell decomposed in 2 parts, one contributed by normal forces, the other by shear forces. The formulation can be found in [Thornton2000], eq. (3):

$$\boldsymbol{\sigma}_{ij} = \frac{2}{V}\sum RN\mathbf{n}_i\mathbf{n}_j + \frac{2}{V}\sum RT\mathbf{n}_i\mathbf{t}_j$$

where V is the cell volume, R is "contact radius" (in our implementation, current distance between particle centroids), $\mathbf{n}$ is the normal vector, $\mathbf{t}$ is a vector perpendicular to $\mathbf{n}$, N and T are norms of normal and shear forces.

yade._utils.**pointInsidePolygon**(*(tuple)arg1*, *(object)arg2*) → bool

yade._utils.**porosity**([*(float)volume=-1*]) → float
Compute packing porosity $\frac{V-V_s}{V}$ where V is overall volume and V_s is volume of spheres.

Parameters

- **volume** (*float*) – overall volume which must be specified for aperiodic simulations. For periodic simulations, current volume of the Cell is used.

yade._utils.**ptInAABB**(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*) → bool
Return True/False whether the point p is within box given by its min and max 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.**stressTensorOfPeriodicCell**([*(bool)smallStrains=False*]) → Matrix3
Compute overall (macroscopic) stress of periodic cell using equation published in [Kuhl2001]:

$$\boldsymbol{\sigma} = \frac{1}{V}\sum_c l^c[\mathbf{N}^c f_N^c + \mathbf{T}^{cT}\cdot\mathbf{f}_T^c],$$

where V is volume of the cell, l^c length of interaction c, f_N^c normal force and $\mathbf{f}_T^c$ shear force. Sumed are values over all interactions c. $\mathbf{N}^c$ and $\mathbf{T}^{cT}$ are projection tensors (see the original publication for more details):

$$\mathbf{N} = \mathbf{n} \otimes \mathbf{n} \rightarrow N_{ij} = n_i n_j$$

$$\mathbf{T}^T = \mathbf{I}_{sym} \cdot \mathbf{n} - \mathbf{n} \otimes \mathbf{n} \otimes \mathbf{n} \rightarrow T^T_{ijk} = \frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})n_l - n_i n_j n_k$$

$$\mathbf{T}^T \cdot \mathbf{f}_T \equiv T^T_{ijk} f_k = (\delta_{ik} n_j/2 + \delta_{jk} n_i/2 - n_i n_j n_k) f_k = n_j f_i/2 + n_i f_j/2 - n_i n_j n_k f_k,$$

where $\mathbf{n}$ is unit vector oriented along the interaction (normal) and δ is Kronecker's delta. As $\mathbf{n}$ and $\mathbf{f}_T$ are perpendicular (therfore $n_i f_i = 0$) we can write

$$\sigma_{ij} = \frac{1}{V} \sum l[n_i n_j f_N + n_j f_i^T/2 + n_i f_j^T/2]$$

Parameters

- **smallStrains** (*bool*) – if false (large strains), real values of volume and interaction lengths are computed. If true, only refLength of interactions and initial volume are computed (can save some time).

Returns macroscopic stress tensor as Matrix3

`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.

2.11 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=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, ***kw*)

Imports geometry from LSMGenGeo .geo file and creates spheres.

Parameters

***filename*: string** file which has 4 colums [x, y, z, radius].

***shift*: Vector3** Vector3(X,Y,Z) parameter moves the specimen.

***scale*: float** factor scales the given data.

***orientation*: quaternion** orientation of the imported geometry

*****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=Vector3(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=None*, *fixed=True*, *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=Vector3(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=Vector3(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

Chapter 3

External modules

3.1 miniEigen (math) module

Basic math functions for Yade: small matrix, vector and quaternion classes. This module internally wraps small parts of the Eigen library. Refer to its documentation for details. All classes in this module support pickling.

class `miniEigen.Matrix3`

3x3 float matrix.

Supported operations (`m` is a Matrix3, `f` if a float/int, `v` is a Vector3): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

`__init__()` → None

___init___((Matrix3)m) → None

___init___((float)m00, (float)m01, (float)m02, (float)m10, (float)m11, (float)m12, (float)m20, (float)m21, (float)m22) → object

`col(`*(int)arg2*`)` → Vector3

`determinant()` → float

`diagonal()` → Vector3

`inverse()` → Matrix3

`polarDecomposition()` → tuple

`row(`*(int)arg2*`)` → Vector3

`toVoigt(`[*(bool)strain=False*]`)` → Vector6

Convert 2nd order tensor to 6-vector (Voigt notation), symmetrizing the tensor; if *strain* is `True`, multiply non-diagonal compoennts by 2.

`trace()` → float

`transpose()` → Matrix3

class `miniEigen.Quaternion`

Quaternion representing rotation.

Supported operations (`q` is a Quaternion, `v` is a Vector3): `q*q` (rotation composition), `q*=q`, `q*v` (rotating `v` by `q`), `q==q`, `q!=q`.

`Rotate(`*(Vector3)v*`)` → Vector3

`__init__()` → None

___init___((Vector3)axis, (float)angle) → object

___init___((float)angle, (Vector3)axis) → object

___init___((float)w, (float)x, (float)y, (float)z) → None : Initialize from coefficients.

Note: The order of coefficients is *w*, *x*, *y*, *z*. The [] operator numbers them differently, 0…4 for *x y z w*!

___init___((Quaternion)other) → None

`conjugate()` → Quaternion

`inverse()` → Quaternion

`norm()` → float

`normalize()` → None

`setFromTwoVectors(`*(Vector3)u*, *(Vector3)v*`)` → Quaternion

`toAngleAxis()` → tuple

`toAxisAngle()` → tuple

`toRotationMatrix()` → Matrix3

class `miniEigen.Vector2`

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 2 floats.

`__init__()` → None

___init___((Vector2)other) → None

___init___((float)x, (float)y) → None

`dot(`*(Vector2)arg2*`)` → float

`norm()` → float

`normalize()` → None

`squaredNorm()` → float

class `miniEigen.Vector2i`

2-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector2i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 2 integers.

`__init__()` → None

___init___((Vector2i)other) → None

___init___((int)x, (int)y) → None

`dot(`*(Vector2i)arg2*`)` → float

`norm()` → int

`normalize()` → None

`squaredNorm()` → int

class `miniEigen.Vector3`

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, plus operations with `Matrix3` and `Quaternion`.

Implicit conversion from sequence (list,tuple, …) of 3 floats.

`__init__()` → None

___init___((Vector3)other) → None

___init___((float)x, (float)y, (float)z) → None

`cross(`*(Vector3)arg2*`)` → Vector3

`dot(`*(Vector3)arg2*`)` → float

`norm()` → float

`normalize()` → None

`normalized()` → Vector3

`squaredNorm()` → float

class `miniEigen.Vector3i`

3-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector3i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 3 integers.

`__init__()` → None

___init___((Vector3i)other) → None

___init___((int)x, (int)y, (int)z) → None

`cross(`*(Vector3i)arg2*`)` → Vector3i

`dot(`*(Vector3i)arg2*`)` → float

`norm()` → int

`squaredNorm()` → int

class `miniEigen.Vector6`

6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 6 floats.

`__init__()` → None

___init___((Vector6)other) → None

___init___((float)v0, (float)v1, (float)v2, (float)v3, (float)v4, (float)v5) → object

`head()` → Vector3

`norm()` → float

`normalize()` → None

`normalized()` → Vector6

`squaredNorm()` → float

`tail()` → Vector3

`toSymmTensor(`[*(bool)strain=False*]`)` → Matrix3

Convert Vector6 in the Voigt notation to the corresponding 2nd order symmetric tensor (as Matrix3); if *strain* is `True`, multiply non-diagonal components by .5

class `miniEigen.Vector6i`

6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 6 floats.

`__init__()` → None

___init___((Vector6i)other) → None

___init___((int)v0, (int)v1, (int)v2, (int)v3, (int)v4, (int)v5) → object

`head()` → Vector3i

`norm()` → int

`normalize()` → None

`normalized()` → Vector6i

`squaredNorm()` → int

`tail()` → Vector3i

3.2 gts (GNU Triangulated surface) module

A package for constructing and manipulating triangulated surfaces.

PyGTS is a python binding for the GNU Triangulated Surface (GTS) Library, which may be used to build, manipulate, and perform computations on triangulated surfaces.

The following geometric primitives are provided:

> Point - a point in 3D space Vertex - a Point in 3D space that may be used to define a Segment Segment - a line defined by two Vertex end-points Edge - a Segment that may be used to define the edge of a Triangle Triangle - a triangle defined by three Edges Face - a Triangle that may be used to define a face on a Surface Surface - a surface composed of Faces

A tetrahedron is assembled from these primitives as follows. First, create Vertices for each of the tetrahedron's points:

> import gts
>
> v1 = gts.Vertex(1,1,1) v2 = gts.Vertex(-1,-1,1) v3 = gts.Vertex(-1,1,-1) v4 = gts.Vertex(1,-1,-1)

Next, connect the four vertices to create six unique Edges:

> e1 = gts.Edge(v1,v2) e2 = gts.Edge(v2,v3) e3 = gts.Edge(v3,v1) e4 = gts.Edge(v1,v4) e5 = gts.Edge(v4,v2) e6 = gts.Edge(v4,v3)

The four triangular faces are composed using three edges each:

> f1 = gts.Face(e1,e2,e3) f2 = gts.Face(e1,e4,e5) f3 = gts.Face(e2,e5,e6) f4 = gts.Face(e3,e4,e6)

Finally, the surface is assembled from the faces:

> s = gts.Surface() for face in [f1,f2,f3,f4]:
>
> s.add(face)

Some care must be taken in the orientation of the faces. In the above example, the surface normals are pointing inward, and so the surface technically defines a void, rather than a solid. To create a tetrahedron with surface normals pointing outward, use the following instead:

> f1.revert() s = Surface() for face in [f1,f2,f3,f4]:
>
> **if not face.is_compatible(s):** face.revert()
>
> s.add(face)

Once the Surface is constructed, there are many different operations that can be performed. For example, the volume can be calculated using:

> s.volume()

The difference between two Surfaces s1 and s2 is given by:

> s3 = s2.difference(s1)

Etc.

It is also possible to read in GTS data files and plot surfaces to the screen. See the example programs packaged with PyGTS for more information.

class gts.Edge(*inherits Segment → Object → object*)

Bases: gts.Segment

Edge object

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

belongs_to_tetrahedron
Returns True if this Edge e belongs to a tetrahedron. Otherwise False.

Signature: e.belongs_to_tetrahedron()

contacts
Returns number of sets of connected triangles share this Edge e as a contact Edge.

Signature: e.contacts()

face_number
Returns number of faces using this Edge e on Surface s.

Signature: e.face_number(s)

is_boundary
Returns True if this Edge e is a boundary on Surface s. Otherwise False.

Signature: e.is_boundary(s)

is_ok
True if this Edge e is not degenerate or duplicate. False otherwise. Degeneracy implies e.v1.id == e.v2.id.

Signature: e.is_ok()

is_unattached
True if this Edge e is not part of any Triangle.

Signature: e.is_unattached()

class gts.Face(*inherits Triangle → Object → object*)

Bases: gts.Triangle

Face object

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

is_compatible
True if Face f is compatible with all neighbors in Surface s. False otherwise.

Signature: f.is_compatible(s).

is_ok
True if this Face f is non-degenerate and non-duplicate. False otherwise.

Signature: f.is_ok()

is_on
True if this Face f is on Surface s. False otherwise.

Signature: f.is_on(s).

is_unattached
True if this Face f is not part of any Surface.

Signature: f.is_unattached().

neighbor_number
Returns the number of neighbors of Face f belonging to Surface s.

Signature: f.neighbor_number(s).

neighbors
Returns a tuple of neighbors of this Face f belonging to Surface s.

Signature: f.neighbors(s).

class gts.Object(*inherits object*)
Bases: object

Base object

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

id
GTS object id

is_unattached
True if this Object o is not attached to another Object. Otherwise False.

Trace: o.is_unattached().

class gts.Point(*inherits Object → object*)
Bases: gts.Object

Point object

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

closest
Set the coordinates of Point p to the Point on Segment s or Triangle t closest to the Point p2

Signature: p.closest(s,p2) or p.closest(t,p2)

Returns the (modified) Point p.

coords
Returns a tuple of the x, y, and z coordinates for this Point p.

Signature: p.coords(x,y,z)

distance
Returns Euclidean distance between this Point p and other Point p2, Segment s, or Triangle t. Signature: p.distance(p2), p.distance(s) or p.distance(t)

distance2
Returns squared Euclidean distance between Point p and Point p2, Segment s, or Triangle t.

Signature: p.distance2(p2), p.distance2(s), or p.distance2(t)

is_in
Tests if this Point p is inside or outside Triangle t. The planar projection (x,y) of Point p is tested against the planar projection of Triangle t.

Signature: p.in_circle(p1,p2,p3) or p.in_circle(t)

Returns a +1 if p lies inside, -1 if p lies outside, and 0 if p lies on the triangle.

is_in_circle
Tests if this Point p is inside or outside circumcircle. The planar projection (x,y) of Point p is tested against the circumcircle defined by the planar projection of p1, p2 and p3, or alternatively the Triangle t

Signature: p.in_circle(p1,p2,p3) or p.in_circle(t)

Returns +1 if p lies inside, -1 if p lies outside, and 0 if p lies on the circle. The Points p1, p2, and p3 must be in counterclockwise order, or the sign of the result will be reversed.

is_in_rectangle
True if this Point p is in box with bottom-left and upper-right Points p1 and p2.

Signature: p.is_in_rectange(p1,p2)

is_inside
True if this Point p is inside or outside Surface s. False otherwise.

Signature: p.in_inside(s)

is_ok
True if this Point p is OK. False otherwise. This method is useful for unit testing and debugging.

Signature: p.is_ok().

orientation_3d
Determines if this Point p is above, below or on plane of 3 Points p1, p2 and p3.

Signature: p.orientation_3d(p1,p2,p3)

Below is defined so that p1, p2 and p3 appear in counterclockwise order when viewed from above the plane.

The return value is positive if p4 lies below the plane, negative if p4 lies above the plane, and zero if the four points are coplanar. The value is an approximation of six times the signed volume of the tetrahedron defined by the four points.

orientation_3d_sos
Determines if this Point p is above, below or on plane of 3 Points p1, p2 and p3.

Signature: p.orientation_3d_sos(p1,p2,p3)

Below is defined so that p1, p2 and p3 appear in counterclockwise order when viewed from above the plane.

The return value is +1 if p4 lies below the plane, and -1 if p4 lies above the plane. Simulation of Simplicity (SoS) is used to break ties when the orientation is degenerate (i.e. the point lies on the plane definedby p1, p2 and p3).

rotate
Rotates Point p around vector dx,dy,dz by angle a. The sense of the rotation is given by the right-hand-rule.

Signature: p.rotate(dx=0,dy=0,dz=0,a=0)

scale
Scales Point p by vector dx,dy,dz.

Signature: p.scale(dx=1,dy=1,dz=1)

set
Sets x, y, and z coordinates of this Point p.

Signature: p.set(x,y,z)

translate
Translates Point p by vector dx,dy,dz.

Signature: p.translate(dx=0,dy=0,dz=0)

x
x value

y
y value

z
z value

class gts.Segment(*inherits Object → object*)
Bases: gts.Object

Segment object

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

connects
Returns True if this Segment s1 connects Vertices v1 and v2. False otherwise.

Signature: s1.connects(v1,v2).

intersection
Returns the intersection of Segment s with Triangle t

This function is geometrically robust in the sense that it will return None if s and t do not intersect and will return a Vertex if they do. However, the point coordinates are subject to round-off errors. None will be returned if s is contained in the plane defined by t.

Signature: s.intersection(t) or s.intersection(t,boundary).

If boundary is True (default), the boundary of s is taken into account.

Returns a summit of t (if boundary is True), one of the endpoints of s, a new Vertex at the intersection of s with t, or None if s and t don't intersect.

intersects
Checks if this Segment s1 intersects with Segment s2. Returns 1 if they intersect, 0 if an endpoint of one Segment lies on the other Segment, -1 otherwise

Signature: s1.intersects(s2).

is_ok
True if this Segment s is not degenerate or duplicate. False otherwise. Degeneracy implies s.v1.id == s.v2.id.

Signature: s.is_ok().

midvertex
Returns a new Vertex at the mid-point of this Segment s.

Signature: s.midvertex().

touches
Returns True if this Segment s1 touches Segment s2 (i.e., they share a common Vertex). False otherwise.

Signature: s1.touches(s2).

v1
Vertex 1

v2
Vertex 2

class gts.Surface(*inherits Object → object*)
Bases: gts.Object

Surface object

Nedges
The number of unique edges

Nfaces
The number of unique faces

Nvertices
The number of unique vertices

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

add
Adds a Face f or Surface s2 to Surface s1.

Signature: s1.add(f) or s2.add(f)

area
Returns the area of Surface s. The area is taken as the sum of the signed areas of the Faces of s.

Signature: s.area()

boundary
Returns a tuple of boundary Edges of Surface s.

Signature: s.boundary()

center_of_area
Returns the coordinates of the center of area of Surface s.

Signature: s.center_of_area()

center_of_mass
Returns the coordinates of the center of mass of Surface s.

Signature: s.center_of_mass()

cleanup
Cleans up the Vertices, Edges, and Faces on a Surface s.

Signature: s.cleanup() or s.cleanup(threhold)

If threhold is given, then Vertices that are spaced less than the threshold are merged. Degenerate Edges and Faces are also removed.

coarsen
Reduces the number of vertices on Surface s.

Signature: s.coarsen(n) and s.coarsen(amin)

n is the smallest number of desired edges (but you may get fewer). amin is the smallest angle between Faces.

copy
Copys all Faces, Edges and Vertices of Surface s2 to Surface s1.

Signature: s1.copy(s2)

Returns s1.

difference
Returns the difference of this Surface s1 with Surface s2.

Signature: s1.difference(s2)

distance
Calculates the distance between the faces of this Surface s1 and the nearest Faces of other s2, and (if applicable) the distance between the boundary of this Surface s1 and the nearest boundary Edges of other s2.

One or two dictionaries are returned (where applicable), the first for the face range and the second for the boundary range. The fields in each dictionary describe statistical results for each population: {min,max,sum,sum2,mean,stddev,n}.

Signature: s1.distance(s2) or s1.distance(s2,delta)

The value delta is a spatial increment defined as the percentage of the diagonal of the bounding box of s2 (default 0.1).

edges
Returns tuple of Edges on Surface s that have Vertex in list. If a list is not given then all of the Edges are returned.

Signature: s.edges(list) or s.edges()

face_indices
Returns a tuple of 3-tuples containing Vertex indices for each Face in Surface s. The index for each Vertex in a face corresponds to where it is found in the Vertex tuple vs.

Signature: s.face_indices(vs)

faces
Returns tuple of Faces on Surface s that have Edge in list. If a list is not given then all of the Faces are returned.

Signature: s.faces(list) s.faces()

fan_oriented
Returns a tuple of outside Edges of the Faces fanning from Vertex v on this Surface s. The Edges are given in counter-clockwise order.

Signature: s.fan_oriented(v)

intersection
Returns the intersection of this Surface s1 with Surface s2.

Signature: s1.intersection(s2)

is_closed
True if Surface s is closed, False otherwise. Note that a closed Surface is also a manifold.

Signature: s.is_closed()

is_manifold
True if Surface s is a manifold, False otherwise.

Signature: s.is_manifold()

is_ok
True if this Surface s is OK. False otherwise.

Signature: s.is_ok()

is_orientable
True if Faces in Surface s have compatible orientation, False otherwise. Note that a closed surface is also a manifold. Note that an orientable surface is also a manifold.

Signature: s.is_orientable()

is_self_intersecting
Returns True if this Surface s is self-intersecting. False otherwise.

Signature: s.is_self_intersecting()

manifold_faces
Returns the 2 manifold Faces of Edge e on this Surface s if they exist, or None.

Signature: s.manifold_faces(e)

next
x.next() -> the next value, or raise StopIteration

parent
Returns Face on this Surface s that has Edge e, or None if the Edge is not on this Surface.

Signature: s.parent(e)

quality_stats
Returns quality statistics for this Surface f in a dict. The statistics include the {min, max, sum, sum2, mean, stddev, and n} for populations of face_quality, face_area, edge_length, and edge_angle. Each of these names are dictionary keys. See Triangle.quality() for an explanation of the face_quality.

Signature: s.quality_stats()

remove
Removes Face f from this Surface s.

Signature: s.remove(f)

rotate
Rotates Surface s about vector dx,dy,dz and angle a. The sense of the rotation is given by the right-hand-rule.

Signature: s.rotate(dx,dy,dz,a)

scale
Scales Surface s by vector dx,dy,dz.

Signature: s.scale(dx=1,dy=1,dz=1)

split
Splits a surface into a tuple of connected and manifold components.

Signature: s.split()

stats
Returns statistics for this Surface f in a dict. The stats include n_faces, n_incompatible_faces,, n_boundary_edges, n_non_manifold_edges, and the statisics {min, max, sum, sum2, mean, stddev, and n} for populations of edges_per_vertex and faces_per_edge. Each of these names are dictionary keys.

Signature: s.stats()

strip
Returns a tuple of strips, where each strip is a tuple of Faces that are successive and have one edge in common.

Signature: s.split()

tessellate
Tessellate each face of this Surface s with 4 triangles. The number of triangles is increased by a factor of 4.

Signature: s.tessellate()

translate
Translates Surface s by vector dx,dy,dz.

Signature: s.translate(dx=0,dy=0,dz=0)

union
Returns the union of this Surface s1 with Surface s2.

Signature: s1.union(s2)

vertices
Returns a tuple containing the vertices of Surface s.

Signature: s.vertices()

volume
Returns the signed volume of the domain bounded by the Surface s.

Signature: s.volume()

write
Saves Surface s to File f in GTS ascii format. All the lines beginning with #! are ignored.

Signature: s.write(f)

write_oogl
Saves Surface s to File f in OOGL (Geomview) format.

Signature: s.write_oogl(f)

write_oogl_boundary
Saves boundary of Surface s to File f in OOGL (Geomview) format.

Signature: s.write_oogl_boundary(f)

write_vtk
Saves Surface s to File f in VTK format.

Signature: s.write_vtk(f)

class gts.Triangle(*inherits Object → object*)

Bases: gts.Object

Triangle object

__init__

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

angle

Returns the angle (radians) between Triangles t1 and t2

Signature: t1.angle(t2)

area

Returns the area of Triangle t.

Signature: t.area()

circumcenter

Returns a Vertex at the center of the circumscribing circle of this Triangle t, or None if the circumscribing circle is not defined.

Signature: t.circumcircle_center()

common_edge

Returns Edge common to both this Triangle t1 and other t2. Returns None if the triangles do not share an Edge.

Signature: t1.common_edge(t2)

e1

Edge 1

e2

Edge 2

e3

Edge 3

interpolate_height

Returns the height of the plane defined by Triangle t at Point p. Only the x- and y-coordinates of p are considered.

Signature: t.interpolate_height(p)

is_compatible

True if this triangle t1 and other t2 are compatible; otherwise False.

Checks if this triangle t1 and other t2, which share a common Edge, can be part of the same surface without conflict in the surface normal orientation.

Signature: t1.is_compatible(t2)

is_ok

True if this Triangle t is non-degenerate and non-duplicate. False otherwise.

Signature: t.is_ok()

is_stabbed

Returns the component of this Triangle t that is stabbed by a ray projecting from Point p to z=infinity. The result can be this Triangle t, one of its Edges or Vertices, or None. If the ray is contained in the plan of this Triangle then None is also returned.

Signature: t.is_stabbed(p)

normal

Returns a tuple of coordinates of the oriented normal of Triangle t as the cross-product of two edges, using the left-hand rule. The normal is not normalized. If this triangle is part of a closed and oriented surface, the normal points to the outside of the surface.

Signature: t.normal()

opposite
Returns Vertex opposite to Edge e or Edge opposite to Vertex v for this Triangle t.

Signature: t.opposite(e) or t.opposite(v)

orientation
Determines orientation of the plane (x,y) projection of Triangle t

Signature: t.orientation()

Returns a positive value if Points p1, p2 and p3 in Triangle t appear in counterclockwise order, a negative value if they appear in clockwise order and zero if they are colinear.

perimeter
Returns the perimeter of Triangle t.

Signature: t.perimeter()

quality
Returns the quality of Triangle t.

The quality of a triangle is defined as the ratio of the square root of its surface area to its perimeter relative to this same ratio for an equilateral triangle with the same area. The quality is then one for an equilateral triangle and tends to zero for a very stretched triangle.
Signature: t.quality()

revert
Changes the orientation of triangle t, turning it inside out.

Signature: t.revert()

vertex
Returns the Vertex of this Triangle t not in t.e1.

Signature: t.vertex()

vertices
Returns the three oriented set of vertices in Triangle t.

Signature: t.vertices()

class gts.**Vertex**(*inherits Point → Object → object*)
Bases: gts.Point

Vertex object

__init__
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

contacts
Returns the number of sets of connected Triangles sharing this Vertex v.

Signature: v.contacts().

If sever is True (default: False) and v is a contact vertex then the vertex is replaced in each Triangle with clones.

encroaches
Returns True if this Vertex v is strictly contained in the diametral circle of Edge e. False otherwise.

Only the projection onto the x-y plane is considered.

Signature: v.encroaches(e)

faces
Returns a tuple of Faces that have this Vertex v.

If a Surface s is given, only Vertices on s are considered.

Signature: v.faces() or v.faces(s).

`is_boundary`
: True if this Vertex v is used by a boundary Edge of Surface s.

  Signature: v.is_boundary().

`is_connected`
: Return True if this Vertex v1 is connected to Vertex v2 by a Segment.

  Signature: v1.is_connected().

`is_ok`
: True if this Vertex v is OK. False otherwise. This method is useful for unit testing and debugging.

  Signature: v.is_ok().

`is_unattached`
: True if this Vertex v is not the endpoint of any Segment.

  Signature: v.is_unattached().

`neighbors`
: Returns a tuple of Vertices attached to this Vertex v by a Segment.

  If a Surface s is given, only Vertices on s are considered.

  Signature: v.neighbors() or v.neighbors(s).

`replace`
: Replaces this Vertex v1 with Vertex v2 in all Segments that have v1. Vertex v1 itself is left unchanged.

  Signature: v1.replace(v2).

`triangles`
: Returns a list of Triangles that have this Vertex v.

  Signature: v.triangles()

Bibliography

[Camborde2000a] F. Camborde, C. Mariotti, F.V. Donzé (2000), **Numerical study of rock and concrete behaviour by discrete element modelling**. *Computers and Geotechnics* (27), pages 225–247.

[Chen2007a] Feng Chen, Eric. C. Drumm, Georges Guiochon (2007), **Prediction/verification of particle motion in one dimension with the discrete-element method**. *International Journal of Geomechanics, ASCE* (7), pages 344–352. DOI 10.1061/(ASCE)1532-3641(2007)7:5(344)

[Dang2010] H. K. Dang, M. A. Meguid (2010), **Algorithm to generate a discrete element specimen with predefined properties**. *International Journal of Geomechanics* (10), pages 85-91. DOI 10.1061/(ASCE)GM.1943-5622.0000028

[Donze1994a] F.V. Donzé, P. Mora, S.A. Magnier (1994), **Numerical simulation of faults and shear zones**. *Geophys. J. Int.* (116), pages 46–52.

[Donze1995a] F.V. Donzé, S.A. Magnier (1995), **Formulation of a three-dimensional numerical model of brittle behavior**. *Geophys. J. Int.* (122), pages 790–802.

[Donze1999a] F.V. Donzé, S.A. Magnier, L. Daudeville, C. Mariotti, L. Davenne (1999), **Study of the behavior of concrete at high strain rate compressions by a discrete element method**. *ASCE J. of Eng. Mech* (125), pages 1154–1163. DOI 10.1016/S0266-352X(00)00013-6

[Donze2004a] F.V. Donzé, P. Bernasconi (2004), **Simulation of the blasting patterns in shaft sinking using a discrete element method**. *Electronic Journal of Geotechnical Engineering* (9), pages 1–44.

[Duriez2010] J. Duriez, F.Darve, F.-V.Donze (2010), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences*. DOI 10.1016/j.ijrmms.2010.09.008 (in press)

[Harthong2009] Harthong, B., Jerier, J. F., Dorémus, P., Imbault, D., Donzé, F. V. (2009), **Modeling of high-density compaction of granular materials by the discrete element method**. *International Journal of Solids and Structures* (46), pages 3357–3364. DOI 10.1016/j.ijsolstr.2009.05.008

[Hassan2010] A. Hassan, B. Chareyre, F. Darve, J. Meyssonier, F. Flin (2010 (submitted)), **Microtomography-based discrete element modelling of creep in snow**. *Granular Matter*.

[Hentz2004a] S. Hentz, F.V. Donzé, L.Daudeville (2004), **Discrete element modelling of concrete submitted to dynamic loading at high strain rates**. *Computers and Structures* (82), pages 2509–2524. DOI 10.1016/j.compstruc.2004.05.016

[Hentz2004b] S. Hentz, L. Daudeville, F.V. Donzé (2004), **Identification and validation of a discrete element model for concrete**. *ASCE Journal of Engineering Mechanics* (130), pages 709–719. DOI 10.1061/(ASCE)0733-9399(2004)130:6(709)

[Hentz2005a] S. Hentz, F.V. Donzé, L.Daudeville (2005), **Discrete elements modeling of a reinforced concrete structure submitted to a rock impact**. *Italian Geotechnical Journal* (XXXIX), pages 83–94.

[Jerier2009] Jerier, Jean-François, Imbault, Didier, Donzé, Fréederic-Victor, Doremus, Pierre (2009), **A geometric algorithm based on tetrahedral meshes to generate a dense polydisperse sphere packing**. *Granular Matter* (11). DOI 10.1007/s10035-008-0116-0

[Jerier2010] Jerier, Jean-François, Richefeu, Vincent, Imbault, Didier, Donzé, Fréderic-Victor (2010), **Packing spherical discrete elements for large scale simulations**. *Computer Methods in Applied Mechanics and Engineering*. DOI 10.1016/j.cma.2010.01.016

[Jerier2010b] J.-F. Jerier, B. Hathong, V. Richefeu, B. Chareyre, D. Imbault, F.-V. Donze, P. Doremus (2010), **Study of cold powder compaction by using the discrete element method**. *Powder Technology* (In Press). DOI DOI: 10.1016/j.powtec.2010.08.056

[Kozicki2005a] J. Kozicki (2005), **Discrete lattice model used to describe the fracture process of concrete**. *Discrete Element Group for Risk Mitigation Annual Report 1, Grenoble University of Joseph Fourier, France*, pages 95–101. (fulltext)

[Kozicki2006a] J. Kozicki, J. Tejchman (2006), **2d lattice model for fracture in brittle materials**. *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88. (fulltext)

[Kozicki2007a] J. Kozicki, J. Tejchman (2007), **Effect of aggregate structure on fracture process in concrete using 2d lattice model"**. *Archives of Mechanics* (59), pages 365–384. (fulltext)

[Kozicki2008] J. Kozicki, F.V. Donzé (2008), **A new open-source software developed for numerical simulations using discrete modeling methods**. *Computer Methods in Applied Mechanics and Engineering* (197), pages 4429–4443. DOI 10.1016/j.cma.2008.05.023 (fulltext)

[Kozicki2009] J. Kozicki, F.V. Donzé (2009), **Yade-open dem: an open-source software using a discrete element method to simulate granular material**. *Engineering Computations* (26), pages 786–805. DOI 10.1108/02644400910985170 (fulltext)

[Magnier1998a] S.A. Magnier, F.V. Donzé (1998), **Numerical simulation of impacts using a discrete element method**. *Mech. Cohes.-frict. Mater.* (3), pages 257–276. DOI 10.1002/(SICI)1099-1484(199807)3:3<257::AID-CFM50>3.0.CO;2-Z

[Nicot2007a] Nicot, F., L. Sibille, F.V. Donzé, F. Darve (2007), **From microscopic to macroscopic second-order work in granular assemblies**. *Int. J. Mech. Mater.* (39), pages 664–684. DOI 10.1016/j.mechmat.2006.10.003

[Scholtes2009a] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Micromechanics of granular materials with capillary effects**. *International Journal of Engineering Science* (47), pages 64–75. DOI 10.1016/j.ijengsci.2008.07.002

[Scholtes2009b] Scholtès, L., Hicher, P.-Y., Chareyre, B., Nicot, F., Darve, F. (2009), **On the capillary stress tensor in wet granular materials**. *International Journal for Numerical and Analytical Methods in Geomechanics* (33), pages 1289–1313. DOI 10.1002/nag.767

[Scholtes2009c] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Discrete modelling of capillary mechanisms in multi-phase granular media**. *Computer Modeling in Engineering and Sciences* (52), pages 297–318. DOI ‘ <http://dx.doi.org/>‘_

[Sibille2007a] L. Sibille, F. Nicot, F.V. Donzé, F. Darve (2007), **Material instability in granular assemblies from fundamentally different models**. *International Journal For Numerical and Analytical Methods in Geomechanics* (31), pages 457–481. DOI 10.1002/nag.591

[Sibille2008a] L. Sibille, F.-V. Donzé, F. Nicot, B. Chareyre, F. Darve (2008), **From bifurcation to failure in a granular material: a dem analysis**. *Acta Geotechnica* (3), pages 15–24. DOI 10.1007/s11440-007-0035-y

[Smilauer2006] Václav Šmilauer (2006), **The splendors and miseries of yade design**. *Annual Report of Discrete Element Group for Hazard Mitigation.* (fulltext)

[Catalano2008a] E. Catalano (2008), **Infiltration effects on a partially saturated slope - an application of the discrete element method and its implementation in the open-source software yade**. Master thesis at *UJF-Grenoble.* (fulltext)

[Duriez2009a] J. Duriez (2009), **Stabilité des massifs rocheux : une approche mécanique**. PhD thesis at *Institut polytechnique de Grenoble.* (fulltext)

[Kozicki2007b] J. Kozicki (2007), **Application of discrete models to describe the fracture process in brittle materials**. PhD thesis at *Gdansk University of Technology.* (fulltext)

[Scholtes2009d] Luc Scholtès (2009), **modélisation micromécanique des milieux granulaires partiellement saturés**. PhD thesis at *Institut National Polytechnique de Grenoble.* (fulltext)

[Smilauer2010b] Václav Šmilauer (2010), **Cohesive particle model using the discrete element method on the yade platform**. PhD thesis at *Czech Technical University in Prague, Faculty of Civil Engineering & Université Grenoble I – Joseph Fourier, École doctorale I-MEP2*. (fulltext) (LaTeX sources)

[Smilauer2010c] Václav Šmilauer (2010), **Doctoral thesis statement**. *(PhD thesis summary)*. (fulltext) (LaTeX sources)

[Chareyre2009] Chareyre B.„ Scholtès L. (2009), **Micro-statics and micro-kinematics of capillary phenomena in dense granular materials**. In *Powders and Grains 2009 (Golden, USA)*.

[Chen2008a] Chen, F., Drumm, E.C., Guiochon, G., Suzuki, K (2008), **Discrete element simulation of 1d upward seepage flow with particle-fluid interaction using coupled open source software**. In *Proceedings of The 12th International Conference of the International Association for Computer Methods and Advances in Geomechanics (IACMAG) Goa, India*.

[Chen2009] Chen, F., Drumm, E.C., Guiochon, G. (2009), **3d dem analysis of graded rock fill sinkhole repair: particle size effects on the probability of stability**. In *Transportation Research Board Conference (Washington DC)*.

[Dang2008a] Dang, H.K., Mohamed, M.A. (2008), **An algorithm to generate a specimen for discrete element simulations with a predefined grain size distribution.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta*.

[Dang2008b] Dang, H.K., Mohamed, M.A. (2008), **3d simulation of the trap door problem using the discrete element method.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta*.

[Gillibert2009] Gillibert L., Flin F., Rolland du Roscoat S., Chareyre B., Philip A., Lesaffre B., Meyssonier J. (2009), **Curvature-driven grain segmentation: application to snow images from x-ray microtomography**. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Miami, USA)*.

[Hicher2009] Hicher P.-Y., Scholtès L., Chareyre B., Nicot F., Darve F. (2008), **On the capillary stress tensor in wet granular materials**. In *Inaugural International Conference of the Engineering Mechanics Institute (EM08) - (Minneapolis, USA)*.

[Kozicki2003a] J. Kozicki, J. Tejchman (2003), **Discrete methods to describe the behaviour of quasi-brittle and granular materials**. In *16th Engineering Mechanics Conference, University of Washington, Seattle, CD–ROM*.

[Kozicki2003c] J. Kozicki, J. Tejchman (2003), **Lattice method to describe the behaviour of quasi-brittle materials**. In *CURE Workshop, Effective use of building materials, Sopot*.

[Kozicki2004a] J. Kozicki, J. Tejchman (2004), **Study of fracture process in concrete using a discrete lattice model**. In *CURE Workshop, Simulations in Urban Engineering, Gdansk*.

[Kozicki2005b] J. Kozicki, J. Tejchman (2005), **Simulations of fracture in concrete elements using a discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2005), Czestochowa, Poland*.

[Kozicki2005c] J. Kozicki, J. Tejchman (2005), **Simulation of the crack propagation in concrete with a discrete lattice model**. In *Proc. Conf. Analytical Models and New Concepts in Concrete and Masonry Structures (AMCM 2005), Gliwice, Poland*.

[Kozicki2006b] J. Kozicki, J. Tejchman (2006), **Modelling of fracture process in brittle materials using a lattice model**. In *Computational Modelling of Concrete Structures, EURO-C (eds.: G. Meschke, R. de Borst, H. Mang and N. Bicanic), Taylor anf Francis*.

[Kozicki2006c] J. Kozicki, J. Tejchman (2006), **Lattice type fracture model for brittle materials**. In *35th Solid Mechanics Conference (SOLMECH 2006), Krakow*.

[Kozicki2007c] J. Kozicki, J. Tejchman (2007), **Simulations of fracture processes in concrete using a 3d lattice model**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes*. (fulltext)

[Kozicki2007d] J. Kozicki, J. Tejchman (2007), **Effect of aggregate density on fracture process in concrete using 2d discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala*.

[Kozicki2007e] J. Kozicki, J. Tejchman (2007), **Modelling of a direct shear test in granular bodies with a continuum and a discrete approach**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala*.

[Kozicki2007f] J. Kozicki, J. Tejchman (2007), **Investigations of size effect in tensile fracture of concrete using a lattice model**. In *Proc. Conf. Modelling of Heterogeneous Materials with Applications in Construction and Biomedical Engineering (MHM 2007), Prague*.

[Scholtes2007a] L. Scholtès, B. Chareyre, F. Nicot, F. Darve (2007), **Micromechanical modelling of unsaturated granular media**. In *Proceedings ECCOMAS-MHM07, Prague*.

[Scholtes2008a] L. Scholtès, B. Chareyre, F. Nicot, F. Darve (2008), **Capillary effects modelling in unsaturated granular materials**. In *8th World Congress on Computational Mechanics - 5th European Congress on Computational Methods in Applied Sciences and Engineering, Venice*.

[Scholtes2008b] L. Scholtès, P.-Y. Hicher, F.Nicot, B. Chareyre, F. Darve (2008), **On the capillary stress tensor in unsaturated granular materials**. In *EM08: Inaugural International Conference of the Engineering Mechanics Institute, Minneapolis*.

[Scholtes2009e] Scholtes L, Chareyre B, Darve F (2009), **Micromechanics of partialy saturated granular material**. In *Int. Conf. on Particle Based Methods, ECCOMAS-Particles*.

[Shiu2007a] W. Shiu, F.V. Donze, L. Daudeville (2007), **Discrete element modelling of missile impacts on a reinforced concrete target**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes*.

[Smilauer2007a] V. Šmilauer (2007), **Discrete and hybrid models: applications to concrete damage**. In *Unpublished*. (fulltext)

[Smilauer2008] Václav Šmilauer (2008), **Commanding c++ with python**. In *ALERT Doctoral school talk*. (fulltext)

[Smilauer2010a] Václav Šmilauer (2010), **Yade: past, present, future**. In *Internal seminary in Laboratoire 3S-R, Grenoble*. (fulltext) (LaTeX sources)

[Stransky2010] Jan Stránský, Milan Jirásek, Václav Šmilauer (2010), **Macroscopic elastic properties of particle models**. In *Proceedings of the International Conference on Modelling and Simulation 2010, Prague*. (fulltext)

[yade:background] V. Šmilauer, B. Chareyre (2010), **Yade dem formulation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (http://yade-dem.org/doc/formulation.html)

[yade:doc] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Documentation**. The Yade Project. (http://yade-dem.org/doc/)

[yade:manual] V. Šmilauer, A. Gladky, J. Kozicki, C. Modenese, J. Stránský (2010), **Yade, using and programming**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/)

[yade:project] **Yade: open source discrete element method** (http://yade-dem.org)

[yade:reference] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Reference Documentation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (http://yade-dem.org/doc/)

[Addetta2001] G.~A. D'Addetta, F. Kun, E. Ramm, H.~J. Herrmann (2001), **From solids to granulates - Discrete element simulations of fracture and fragmentation processes in geomaterials.**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*. (fulltext)

[Allen1989] M. P. Allen, D. J. Tildesley (1989), **Computer simulation of liquids**. Clarendon Press.

[Alonso2004] F. Alonso-Marroqu? R. Garc?Rojo, H. J. Herrmann (2004), **Micro-mechanical investigation of the granular ratcheting**. In *Cyclic Behaviour of Soils and Liquefaction Phenomena*. (fulltext)

[Bertrand2005] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2005), **Modelling a geo-composite cell using discrete analysis**. *Computers and Geotechnics* (32), pages 564–577.

[Bertrand2008] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2008), **Discrete element method (dem) numerical modeling of double-twisted hexagonal mesh**. *Canadian Geotechnical Journal* (45), pages 1104–1117.

[Chareyre2002a] B. Chareyre, L. Briancon, P. Villard (2002), **Theoretical versus experimental modeling of the anchorage capacity of geotextiles in trenches.**. *Geosynthet. Int.* (9), pages 97–123.

[Chareyre2002b] B. Chareyre, P. Villard (2002), **Discrete element modeling of curved geosynthetic anchorages with known macro-properties.**. In *Proc., First Int. PFC Symposium, Gelsenkirchen, Germany*.

[Chareyre2003] Bruno Chareyre (2003), **Mod?sation du comportement d'ouvrages composites sol-g?ynth?que par ?ments discrets - application aux tranch? d'ancrage en t? de talus.**. PhD thesis at *Grenoble University*. (fulltext)

[Chareyre2005] Bruno Chareyre, Pascal Villard (2005), **Dynamic spar elements and discrete element methods in two dimensions for the modeling of soil-inclusion problems**. *Journal of Engineering Mechanics* (131), pages 689–698. DOI 10.1061/(ASCE)0733-9399(2005)131:7(689) (fulltext)

[CundallStrack1979] P.A. Cundall, O.D.L. Strack (1979), **A discrete numerical model for granular assemblies**. *Geotechnique* (), pages 47–65. DOI 10.1680/geot.1979.29.1.47

[DeghmReport2006] F. V. Donz?ed.), **Annual report 2006** (2006). *Discrete Element Group for Hazard Mitigation*. Universit?oseph Fourier, Grenoble (fulltext)

[Duriez2010] J. Duriez, F.Darve, F.-V.Donze (2010), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences*. (in press)

[GarciaRojo2004] R. Garc?Rojo, S. McNamara, H. J. Herrmann (2004), **Discrete element methods for the micro-mechanical investigation of granular ratcheting**. In *Proceedings ECCOMAS 2004*. (fulltext)

[Hentz2003] S?astien Hentz (2003), **Mod?sation d'une structure en b?n arm?oumise ?n choc par la m?ode des el?nts discrets**. PhD thesis at *Universit?renoble 1 – Joseph Fourier*.

[Hubbard1996] Philip M. Hubbard (1996), **Approximating polyhedra with spheres for time-critical collision detection**. *ACM Trans. Graph.* (15), pages 179–210. DOI 10.1145/231731.231732

[Johnson2008] Scott M. Johnson, John R. Williams, Benjamin K. Cook (2008), **Quaternion-based rigid body rotation integration algorithms for use in particle methods**. *International Journal for Numerical Methods in Engineering* (74), pages 1303–1313. DOI 10.1002/nme.2210

[Jung1997] Derek Jung, Kamal K. Gupta (1997), **Octree-based hierarchical distance maps for collision detection**. *Journal of Robotic Systems* (14), pages 789–806. DOI 10.1002/(SICI)1097-4563(199711)14:11<789::AID-ROB3>3.0.CO;2-Q

[Klosowski1998] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, Karel Zikan (1998), **Efficient collision detection using bounding volume hierarchies of k-dops**. *IEEE Transactions on Visualization and Computer Graphics* (4), pages 21–36. (fulltext)

[Kuhl2001] E. Kuhl, G. A. D'Addetta, M. Leukart, E. Ramm (2001), **Microplane modelling and particle modelling of cohesive-frictional materials**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*. DOI 10.1007/3-540-44424-6_3 (fulltext)

[Lu1998] Ya Yan Lu (1998), **Computing the logarithm of a symmetric positive definite matrix**. *Appl. Numer. Math* (26), pages 483–496. DOI 10.1016/S0168-9274(97)00103-7 (fulltext)

[Luding2008] Stefan Luding (2008), **Introduction to discrete element methods**. In *European Journal of Environmental and Civil Engineering*.

[McNamara2008] S. McNamara, R. Garc?Rojo, H. J. Herrmann (2008), **Microscopic origin of granular ratcheting**. *Physical Review E* (77). DOI 11.1103/PhysRevE.77.031304

[Munjiza1998] A. Munjiza, K. R. F. Andrews (1998), **Nbs contact detection algorithm for bodies of similar size**. *International Journal for Numerical Methods in Engineering* (43), pages 131–149. DOI 10.1002/(SICI)1097-0207(19980915)43:1<131::AID-NME447>3.0.CO;2-S

[Munjiza2006] A. Munjiza, E. Rougier, N. W. M. John (2006), **Mr linear contact detection algorithm**. *International Journal for Numerical Methods in Engineering* (66), pages 46–71. DOI 10.1002/nme.1538

[Neto2006] Natale Neto, Luca Bellucci (2006), **A new algorithm for rigid body molecular dynamics**. *Chemical Physics* (328), pages 259–268. DOI 10.1016/j.chemphys.2006.07.009

[Omelyan1999] Igor P. Omelyan (1999), **A new leapfrog integrator of rotational motion. the revised angular-momentum approach**. *Molecular Simulation* (22). DOI 10.1080/08927029908022097 (fulltext)

[Pfc3dManual30] ICG (2003), **Pfc3d (particle flow code in 3d) theory and background manual, version 3.0**. Itasca Consulting Group.

[Pournin2001] L. Pournin, Th. M. Liebling, A. Mocellin (2001), **Molecular-dynamics force models for better control of energy dissipation in numerical simulations of dense granular media**. *Phys. Rev. E* (65), pages 011302. DOI 10.1103/PhysRevE.65.011302

[Price2007] Mathew Price, Vasile Murariu, Garry Morrison (2007), **Sphere clump generation and trajectory comparison for real particles**. In *Proceedings of Discrete Element Modelling 2007*. (fulltext)

[Satake1982] M. Satake (1982), **Fabric tensor in granular materials.**. In *Proc., IUTAM Symp. on Deformation and Failure of Granular materials, Delft, The Netherlands*.

[Thornton1991] Colin Thornton, K. K. Yin (1991), **Impact of elastic spheres with and without adhesion**. *Powder technology* (65), pages 153–166. DOI 10.1016/0032-5910(91)80178-L

[Thornton2000] Colin Thornton (2000), **Numerical simulations of deviatoric shear deformation of granular media**. *G?echnique* (50), pages 43–53. DOI 10.1680/geot.2000.50.1.43

[Verlet1967] Loup Verlet (1967), **Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules**. *Phys. Rev.* (159), pages 98. DOI 10.1103/PhysRev.159.98

[Villard2004a] P. Villard, B. Chareyre (2004), **Design methods for geosynthetic anchor trenches on the basis of true scale experiments and discrete element modelling**. *Canadian Geotechnical Journal* (41), pages 1193–1205.

[Wang2009] Yucang Wang (2009), **A new algorithm to model the dynamics of 3-d bonded rigid bodies with rotations**. *Acta Geotechnica* (4), pages 117–127. DOI 10.1007/s11440-008-0072-1 (fulltext)

[cgal] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, Mariette Yvinec (2002), **Triangulations in cgal**. *Computational Geometry: Theory and Applications* (22), pages 5–19.

Python Module Index