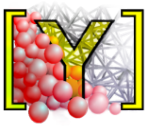


Reference Manual

Václav Šmilauer, Emanuele Catalano, Bruno Chareyre, Sergei Dorofeenko, Jérôme Duriez, Nolan Dyck, Jan Eliáš, Burak Er, Alexander Eulitz, Anton Gladky, Christian Jakob, François Kneib, Janek Kozicki, Donia Marzougui, Raphaël Maurin, Chiara Modenese, Luc Scholtès, Luc Sibille, Jan Stránský, Thomas Sweijen, Klaus Thoeni, Chao Yuan

Yade Documentation 2nd edition, 2015
based on Yade 1.14.0

Citing this document:

Šmilauer V. et al. (2015). Reference Manual. In: *Yade Documentation 2nd ed.* doi:10.5281/zenodo.34045. http://yade-dem.org
See also http://yade-dem/doc/citing.html.

Contents

Chapter 1

Class reference (yade.wrapper module)

1.1 Bodies

1.1.1 Body

class `yade.wrapper.Body`(*(object)arg1*)

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`)

`chain`
: Returns Id of chain to which the body belongs.

`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.

`dict`() → dict
: Return dictionary of attributes.

`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.

iterBorn
Returns step number at which the body was added to simulation.

mask
Shorthand for Body::groupMask

mat
Shorthand for Body::material

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

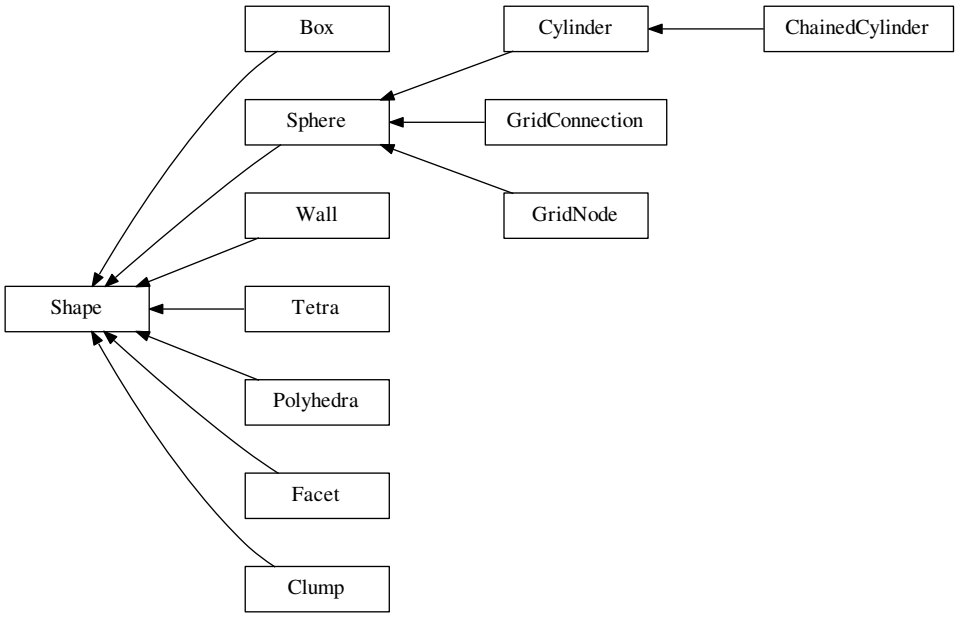
shape(=*uninitalized*)
Geometrical Shape.

state(=*new State*)
Physical state.

timeBorn
Returns time at which the body was added to simulation.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.1.2 Shape



class `yade.wrapper.Shape`(*(object)arg1*)
Geometry of a body

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`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.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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`(*(object)arg1*)
Box (cuboid) particle geometry. (Avoid using in new code, prefer Facet instead.

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`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.

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

`highlight`(=*false*)
Whether this Shape will be highlighted when rendered.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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.ChainedCylinder`(*(object)arg1*)
Geometry of a deformable chained cylinder, using geometry Cylinder.

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

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

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.

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

length(=*NaN*)
Length [m]

radius(=*NaN*)
Radius [m]

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**Clump**(*(object)arg1*)
Rigid aggregate of bodies

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**Cylinder**(*(object)arg1*)
Geometry of a cylinder, as Minkowski sum of line and sphere.

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

length(=*NaN*)
Length [m]

radius(=*NaN*)
Radius [m]

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**Facet**(*(object)arg1*)
Facet (triangular particle) geometry.

area(=*NaN*)
Facet's area

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

normal(=*Vector3r(NaN, NaN, NaN)*)
Facet's normal (in local coordinate system)

setVertices(*(Vector3)arg2, (Vector3)arg3, (Vector3)arg4*) → None
TODO

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

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.**GridConnection**(*(object)arg1*)
GridConnection shape. Component of a grid designed to link two GridNodes. It's highly recommended to use utils.gridConnection(...) to generate correct GridConnections.

cellDist(=*Vector3i(0, 0, 0)*)
missing doc :(

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

node1(=*uninitalized*)
First Body the GridConnection is connected to.

node2(=*uninitalized*)
Second Body the GridConnection is connected to.

periodic(=*false*)
true if two nodes from different periods are connected.

radius(=*NaN*)
Radius [m]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**GridNode**(*(object)arg1*)
GridNode shape, component of a grid. To create a Grid, place the nodes first, they will define the spacial discretisation of it. It's highly recommended to use utils.gridNode(...) to generate correct GridNodes. Note that the GridNodes should only be in an Interaction with other GridNodes. The Sphere-Grid contact is only handled by the GridConnections.

ConnList(=*uninitalized*)
List of GridConnections the GridNode is connected to.

addConnection(*(Body)Body*) → None
Add a GridConnection to the GridNode.

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

radius(=*NaN*)
Radius [m]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**Polyhedra**(*(object)arg1*)
Polyhedral (convex) geometry.

GetCentroid() → Vector3
return polyhedra's centroid

GetInertia() → Vector3
return polyhedra's inertia tensor

GetOri() → Quaternion
return polyhedra's orientation

GetSurfaceTriangulation() → object
triangulation of facets (for plotting)

GetSurfaces() → object
get indices of surfaces' vertices (for postprocessing)

GetVolume() → float
return polyhedra's volume

Initialize() → None
Initialization

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

seed(=*time(__null)*)
Seed for random generator.

setVertices(*(object)arg2*) → None
set vertices and update receiver

size(=*Vector3r(1., 1., 1.)*)
Size of the grain in meters - x,y,z - before random rotation

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

v(=*uninitalized*)
Tetrahedron vertices in global coordinate system.

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.Sphere`(*(object)arg1*)
Geometry of spherical particle.

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`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.

`radius`(=*NaN*)
Radius [m]

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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.Tetra`(*(object)arg1*)
Tetrahedron geometry.

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`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.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

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

`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.Wall`(*(object)arg1*)
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)

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

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.

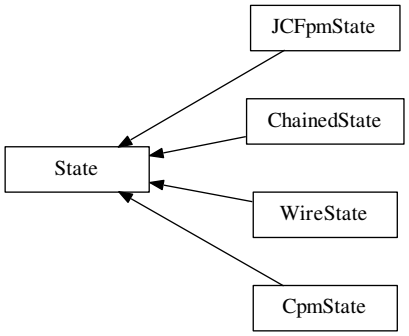
highlight(=*false*)
Whether this Shape will be highlighted when rendered.

sense(=*0*)
Which side of the wall interacts: -1 for negative only, 0 for both, +1 for positive only

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

1.1.3 State



class yade.wrapper.**State**(*(object)arg1*)
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).

densityScaling(=*1*)
(auto-updated) see GlobalStiffnessTimeStepper::targetDt.

dict() → dict
Return dictionary of attributes.

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.

isDamped(=*true*)
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class yade.wrapper.**ChainedState**(*(object)arg1*)
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 rank and chainNumber.

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

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

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

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

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).

chainNumber(=*0*)
chain id.

currentChain = 0

densityScaling(=*1*)
(auto-updated) see GlobalStiffnessTimeStepper::targetDt.

dict() → dict
Return dictionary of attributes.

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.

isDamped(=*true*)
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

mass(=*0*)
Mass of this body

ori
Current orientation.

pos
Current position.

rank(=*0*)
rank in the chain.

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class yade.wrapper.**CpmState**(*(object)arg1*)
State information about body use by cpm-model.

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

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).

damageTensor(=*Matrix3r::Zero()*)
Damage tensor computed with microplane theory averaging. state.damageTensor.trace() = state.normDmg

densityScaling(=*1*)
(auto-updated) see GlobalStiffnessTimeStepper::targetDt.

dict() → dict
Return dictionary of attributes.

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)

epsVolumetric(=*0*)
Volumetric strain around this body (unused for now)

inertia(=*Vector3r::Zero()*)
Inertia of associated body, in local coordinate system.

isDamped(=*true*)
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

mass(=*0*)
Mass of this body

normDmg(=*0*)
Average damage including already deleted contacts (it is really not damage, but 1-relResidualStrength now)

numBrokenCohesive(=*0*)
Number of (cohesive) contacts that damaged completely

numContacts(=*0*)
Number of contacts with 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.

stress(=*Matrix3r::Zero()*)
Stress tensor of the spherical particle (under assumption that particle volume = pi*r*r*r*4/3.) for packing fraction 0.62

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class yade.wrapper.**JCFpmState**(*(object)arg1*)
JCFpm state information about each body.

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).

densityScaling(=*1*)
(auto-updated) see GlobalStiffnessTimeStepper::targetDt.

dict() → dict
Return dictionary of attributes.

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.

isDamped(=*true*)
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

joint(=*0*)
Indicates the number of joint surfaces to which the particle belongs (0-> no joint, 1->1 joint, etc..). [-]

jointNormal1(=*Vector3r::Zero()*)
Specifies the normal direction to the joint plane 1. Rk: the ideal here would be to create a vector of vector wich size is defined by the joint integer (as much joint normals as joints). However, it needs to make the pushback function works with python since joint detection is done through a python script. lines 272 to 312 of cpp file should therefore be adapted. [-]

jointNormal2(=*Vector3r::Zero()*)
Specifies the normal direction to the joint plane 2. [-]

jointNormal3(=*Vector3r::Zero()*)
Specifies the normal direction to the joint plane 3. [-]

mass(=*0*)
Mass of this body

noIniLinks(=*0*)
Number of initial cohesive interactions. [-]

onJoint(=*false*)
Identifies if the particle is on a joint surface.

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.

shearBreak(=*0*)
Number of shear breakages. [-]

shearBreakRel(=*0*)
Relative number (in [0;1], compared with noIniLinks) of shear breakages. [-]

tensBreak(=*0*)
Number of tensile breakages. [-]

tensBreakRel(=*0*)
Relative number (in [0;1], compared with noIniLinks) of tensile breakages. [-]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class yade.wrapper.**WireState**(*(object)arg1*)
Wire state information of each body.

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

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).

densityScaling(=*1*)
(auto-updated) see GlobalStiffnessTimeStepper::targetDt.

dict() → dict
Return dictionary of attributes.

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.

isDamped(=*true*)
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

mass(=*0*)
Mass of this body

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

ori
Current orientation.

pos
Current position.

refOri(=*Quaternionr::Identity()*)
Reference orientation

refPos(=*Vector3r::Zero()*)
Reference position

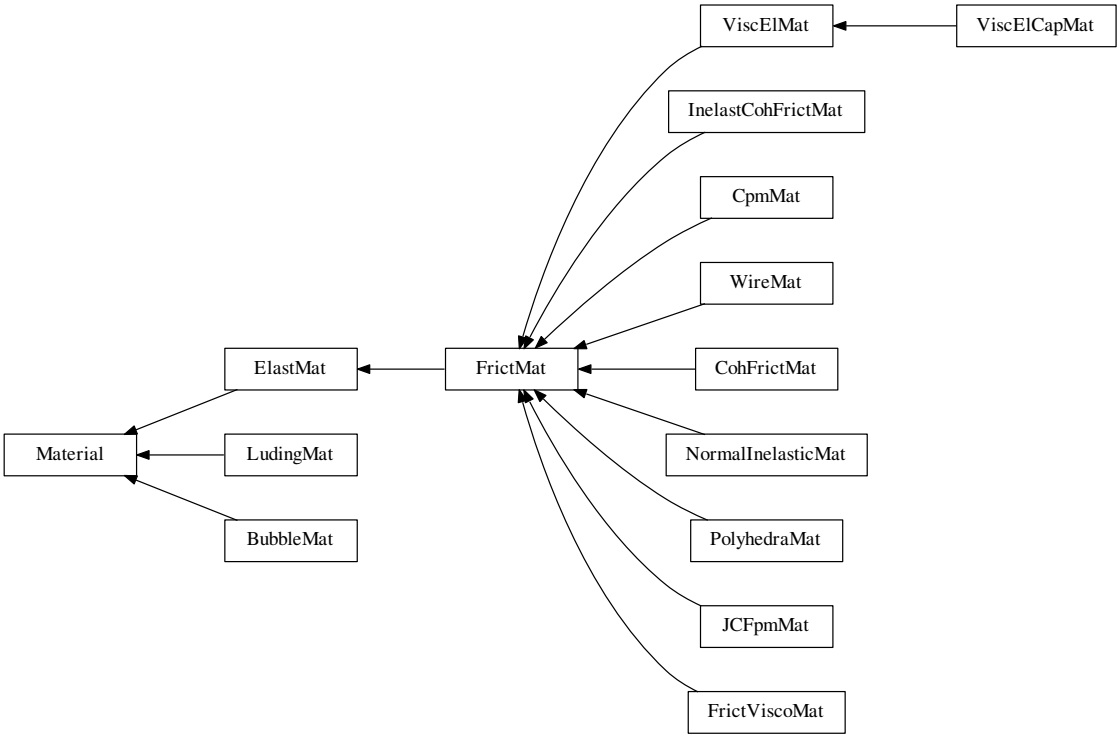
rot() → Vector3
Rotation from reference orientation (as rotation vector)

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

1.1.4 Material



class `yade.wrapper.Material`(*(object)arg1*)

Material properties of a body.

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

`dict`() → dict
: Return dictionary of attributes.

`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).

`updateAttrs`(*(dict)arg2*) → None

Update object attributes from given dictionary

class yade.wrapper.**BubbleMat**(*(object)arg1*)
material for bubble interactions, for use with other Bubble classes

density(*=1000*)
Density of the material [kg/m³]

dict() → dict
Return dictionary of attributes.

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).

surfaceTension(*=0.07197*)
The surface tension in the fluid surrounding the bubbles. The default value is that of water at 25 degrees Celcius.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CohFrictMat**(*(object)arg1*)

alphaKr(*=2.0*)
Dimensionless rolling stiffness.

alphaKtw(*=2.0*)
Dimensionless twist stiffness.

density(*=1000*)
Density of the material [kg/m³]

dict() → dict
Return dictionary of attributes.

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.

etaRoll(*=-1.*)
Dimensionless rolling (aka 'bending') strength. If negative, rolling moment will be elastic.

etaTwist(*=-1.*)
Dimensionless twisting strength. If negative, twist moment will be elastic.

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

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

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)

isCohesive(*=true*)

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.

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.

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).

normalCohesion(*=-1*)
Tensile strength, homogeneous to a pressure. If negative the normal force is purely elastic.

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

shearCohesion(*=-1*)
Shear strength, homogeneous to a pressure. If negative the shear force is purely elastic.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**CpmMat**(*(object)arg1*)
Concrete material, for use with other Cpm classes.

Note: Density is initialized to 4800 kgm−3automatically, which gives approximate 2800 kgm−3 on 0.5 density packing.

Concrete Particle Model (CPM)

CpmMat is particle material, Ip2_CpmMat_CpmMat_CpmPhys averages two particles' materials, creating CpmPhys, which is then used in interaction resultion by Law2_ScGeom_CpmPhys_Cpm. CpmState is associated to CpmMat and keeps state defined on particles rather than interactions (such as number of completely damaged interactions).

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.

damLaw(*=1*)
Law for damage evolution in uniaxial tension. 0 for linear stress-strain softening branch, 1 (default) for exponential damage evolution law

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

dict() → dict
Return dictionary of attributes.

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.

dmgRateExp(*=0*)
Exponent for normal viscosity function. [-]

dmgTau(*=-1*, *deactivated if negative*)
Characteristic time for normal viscosity. [s]

epsCrackOnset(*=NaN*)
Limit elastic strain [-]

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

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)

isoPrestress(*=0*)
Isotropic prestress of the whole specimen. [Pa]

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.

neverDamage(*=false*)
If true, no damage will occur (for testing only).

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).

plRateExp(*=0*)
Exponent for visco-plasticity function. [-]

plTau(*=-1*, *deactivated if negative*)
Characteristic time for visco-plasticity. [s]

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

relDuctility(*=NaN*)
relative ductility of bonds in normal direction

sigmaT(=*NaN*)
Initial cohesion [Pa]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**ElastMat**(*(object)arg1*)
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.

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

dict() → dict
Return dictionary of attributes.

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).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**FrictMat**(*(object)arg1*)
Elastic material with contact friction. See also ElastMat.

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

dict() → dict
Return dictionary of attributes.

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.

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

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).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**FrictViscoMat**(*(object)arg1*)
Material for use with the FrictViscoPM classes

betan(=*0.*)
Fraction of the viscous damping coefficient in normal direction equal to $\frac{c_n}{C_{n,crit}}$.

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

dict() → dict
Return dictionary of attributes.

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.

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

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).

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**InelastCohFrictMat**(*(object)arg1*)

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

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

compressionModulus(*=0.0*)
Compresion elasticity modulus

creepBending(*=0.0*)
Bending creeping coefficient. Usual values between 0 and 1.

creepTension(*=0.0*)
Tension/compression creeping coefficient. Usual values between 0 and 1.

creepTwist(*=0.0*)
Twist creeping coefficient. Usual values between 0 and 1.

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

dict() → dict
Return dictionary of attributes.

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.

epsilonMaxCompression(*=0.0*)
Maximal plastic strain compression

epsilonMaxTension(*=0.0*)
Maximal plastic strain tension

etaMaxBending(*=0.0*)
Maximal plastic bending strain

etaMaxTwist(*=0.0*)
Maximal plastic twist strain

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

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).

nuBending(=*0.0*)
Bending elastic stress limit

nuTwist(=*0.0*)
Twist elastic stress limit

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

shearCohesion(=*0.0*)
Shear elastic stress limit

shearModulus(=*0.0*)
shear elasticity modulus

sigmaCompression(=*0.0*)
Compression elastic stress limit

sigmaTension(=*0.0*)
Tension elastic stress limit

tensionModulus(=*0.0*)
Tension elasticity modulus

unloadBending(=*0.0*)
Bending plastic unload coefficient. Usual values between 0 and +infinity.

unloadTension(=*0.0*)
Tension/compression plastic unload coefficient. Usual values between 0 and +infinity.

unloadTwist(=*0.0*)
Twist plastic unload coefficient. Usual values between 0 and +infinity.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**JCFpmMat**(*(object)arg1*)
Possibly jointed, cohesive frictional material, for use with other JCFpm classes

cohesion(=*0.*)
Defines the maximum admissible tangential force in shear, for Fn=0, in the matrix (FsMax = cohesion * crossSection). [Pa]

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

dict() → dict
Return dictionary of attributes.

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.

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

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)

jointCohesion(*=0.*)
Defines the maximum admissible tangential force in shear, for Fn=0, on the joint surface. [Pa]

jointDilationAngle(*=0*)
Defines the dilatancy of the joint surface (only valid for smooth contact logic). [rad]

jointFrictionAngle(*=-1*)
Defines Coulomb friction on the joint surface. [rad]

jointNormalStiffness(*=0.*)
Defines the normal stiffness on the joint surface. [Pa/m]

jointShearStiffness(*=0.*)
Defines the shear stiffness on the joint surface. [Pa/m]

jointTensileStrength(*=0.*)
Defines the maximum admissible normal force in traction on the joint surface. [Pa]

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).

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

tensileStrength(*=0.*)
Defines the maximum admissible normal force in traction in the matrix (FnMax = tensileStrength * crossSection). [Pa]

type(*=0*)
If particles of two different types interact, it will be with friction only (no cohesion).[-]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**LudingMat**(*(object)arg1*)
Material for simple Ludning's model of contact.

G0(*=NaN*)
Viscous damping

PhiF(*=NaN*)
Dimensionless plasticity depth

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

dict() → dict
Return dictionary of attributes.

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.

frictionAngle(*=NaN*)
Friction angle [rad]

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)

k1(*=NaN*)
Slope of loading plastic branch

kc(*=NaN*)
Slope of irreversible, tensile adhesive branch

kp(*=NaN*)
Slope of unloading and reloading limit elastic branch

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).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NormalInelasticMat**(*(object)arg1*)
Material class for particles whose contact obey to a normal inelasticity (governed by this coeff_dech).

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

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

dict() → dict
Return dictionary of attributes.

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.

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

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).

`poisson`(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`young`(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class `yade.wrapper.PolyhedraMat`(*(object)arg1*)
Elastic material with Coulomb friction.

`IsSplitable`(=*0*)
To be splitted ... or not

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

`dict`() → dict
Return dictionary of attributes.

`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.

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

`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).

`poisson`(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

`strength`(=*100*)
Stress at which polyhedra of volume 4/3*pi [mm] breaks.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e8*)
TODO

class yade.wrapper.**ViscElCapMat**(*(object)arg1*)
Material for extended viscoelastic model of contact with capillary parameters.

Capillar(=*false*)
True, if capillar forces need to be added.

CapillarType(=*"“*)
Different types of capillar interaction: Willett_numeric, Willett_analytic [Willett2000] , Weigert [Weigert1999] , Rabinovich [Rabinov2005] , Lambert (simplified, corrected Rabinovich model) [Lambert2008]

Vb(=*0.0*)
Liquid bridge volume [m^3]

cn(=*NaN*)
Normal viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

cs(=*NaN*)
Shear viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

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

dict() → dict
Return dictionary of attributes.

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.

en(=*NaN*)
Restitution coefficient in normal direction

et(=*NaN*)
Restitution coefficient in tangential direction

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

gamma(=*0.0*)
Surface tension [N/m]

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)

kn(=*NaN*)
Normal elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

ks(=*NaN*)
Shear elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.

mR(=*0.0*)
Rolling resistance, see [Zhou1999536].

mRtype(*=1*)
Rolling resistance type, see [Zhou1999536]. mRtype=1 - equation (3) in [Zhou1999536]; mRtype=2 - equation (4) in [Zhou1999536].

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).

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

tc(*=NaN*)
Contact time

theta(*=0.0*)
Contact angle [°]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**ViscElMat**(*(object)arg1*)
Material for simple viscoelastic model of contact from analytical solution of a pair spheres interaction problem [Pournin2001] .

cn(*=NaN*)
Normal viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

cs(*=NaN*)
Shear viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

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

dict() → dict
Return dictionary of attributes.

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.

en(*=NaN*)
Restitution coefficient in normal direction

et(*=NaN*)
Restitution coefficient in tangential direction

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

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)

kn(*=NaN*)
Normal elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

ks(=*NaN*)
Shear elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.

mR(=*0.0*)
Rolling resistance, see [Zhou1999536].

mRtype(=*1*)
Rolling resistance type, see [Zhou1999536]. mRtype=1 - equation (3) in [Zhou1999536]; mRtype=2 - equation (4) in [Zhou1999536].

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).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

tc(=*NaN*)
Contact time

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**WireMat**(*(object)arg1*)
Material for use with the Wire classes

as(=*0.*)
Cross-section area of a single wire used to transform stress into force. [m^2]

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

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

dict() → dict
Return dictionary of attributes.

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.

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

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)

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.

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.

lambdaEps(=*0.47*)
Parameter between 0 and 1 to reduce strain at failure of a double-twisted wire (as used by [Bertrand2008]). [-]

lambdaF(=*1.0*)
Parameter between 0 and 1 introduced by [Thoeni2013] which defines where the shifted force-displacement curve intersects with the new initial stiffness: $F^* = \lambda_F F_{elastic}$. [-]

lambdak(=*0.73*)
Parameter between 0 and 1 to compute the elastic stiffness of a double-twisted wire (as used by [Bertrand2008]): $k^D = 2(\lambda_k k_h + (1-\lambda_k)k^S)$. [-]

lambdau(=*0.2*)
Parameter between 0 and 1 introduced by [Thoeni2013] which defines the maximum shift of the force-displacement curve in order to take an additional initial elongation (e.g. wire distortion/imperfections, slipping, system flexibility) into account: $\Delta l^* = \lambda_u l_0 rnd(seed)$. [-]

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).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

seed(=*12345*)
Integer used to initialize the random number generator for the calculation of the distortion. If the integer is equal to 0 a internal seed number based on the time is computed. [-]

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! NOTE: Vector needs to be initialized!

strainStressValuesDT(=*uninitalized*)
Piecewise linear definition of the stress-strain curve by set of points (strain[-]>0,stress[Pa]>0) for the double twist. Tension only is considered and the point (0,0) is not needed! If this value is given the calculation will be based on two different stress-strain curves without considering the parameter introduced by [Bertrand2008] (see [Thoeni2013]).

type
Three different types are considered:

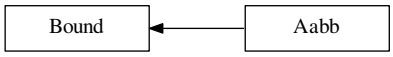
0	Corresponds to Bertrand's approach (see [Bertrand2008]): only one stress-strain curve is used
1	New approach: two separate stress-strain curves can be used (see [Thoeni2013])
2	New approach with stochastically distorted contact model: two separate stress-strain curves with changed initial stiffness and horizontal shift (shift is random if seed ≥ 0, for more details see [Thoeni2013])

By default the type is 0.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

1.1.5 Bound



class `yade.wrapper.Bound`(*(object)arg1*)

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

`dict`() → dict
: Return dictionary of attributes.

`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.

`lastUpdateIter`(=*0*)
: record iteration of last reference position update *(auto-updated)*

`max`(=*Vector3r(NaN, NaN, NaN)*)
: Upper 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)

`refPos`(=*Vector3r(NaN, NaN, NaN)*)
: Reference position, updated at current body position each time the bound dispatcher update bounds *(auto-updated)*

`sweepLength`(=*0*)
: The length used to increase the bounding boxe size, can be adjusted on the basis of previous displacement if BoundDispatcher::targetInterv>0. *(auto-updated)*

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.Aabb`(*(object)arg1*)

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.)

`color`(=*Vector3r(1, 1, 1)*)
: Color for rendering this object

`dict`() → dict
: Return dictionary of attributes.

`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.

`lastUpdateIter`(=*0*)
: record iteration of last reference position update *(auto-updated)*

`max`(=*Vector3r(NaN, NaN, NaN)*)
Upper 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)

`refPos`(=*Vector3r(NaN, NaN, NaN)*)
Reference position, updated at current body position each time the bound dispatcher update bounds *(auto-updated)*

`sweepLength`(=*0*)
The length used to increase the bounding boxe size, can be adjusted on the basis of previous displacement if BoundDispatcher::targetInterv>0. *(auto-updated)*

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

1.2 Interactions

1.2.1 Interaction

class `yade.wrapper.Interaction`(*(object)arg1*)
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 :-)

`dict`() → dict
Return dictionary of attributes.

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

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

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

`isActive`
True if this interaction is active. Otherwise the forces from this interaction will not be taken into account. True by default.

`isReal`
True if this interaction has both geom and phys; False otherwise.

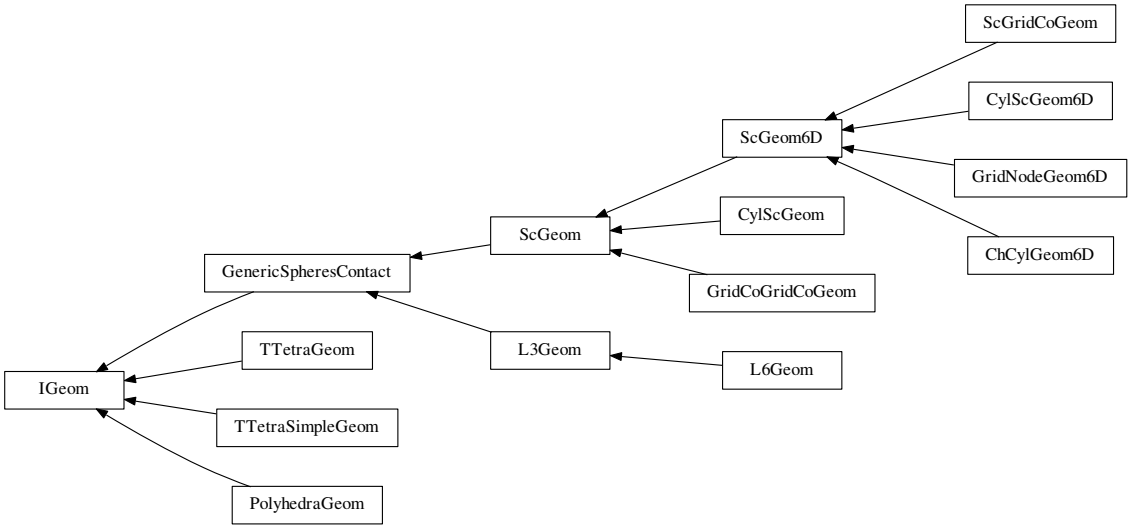
`iterBorn`(=*-1*)
Step number at which the interaction was added to simulation.

`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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.2.2 IGeom



class yade.wrapper.**IGeom**(*(object)arg1*)
Geometrical configuration of interaction

dict() → dict
Return dictionary of attributes.

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ChCylGeom6D**(*(object)arg1*)
Test

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

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

dict() → dict
Return dictionary of attributes.

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.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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)*

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

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

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

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

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

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

twist(=*0*)
Elastic twist angle (around normal axis) 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)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CylScGeom**(*(object)arg1*)
Geometry of a cylinder-sphere contact.

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

dict() → dict
Return dictionary of attributes.

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.

end(=*Vector3r::Zero()*)
position of 2nd node *(auto-updated)*

id3(=*0*)
id of next chained cylinder *(auto-updated)*

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

isDuplicate(=*0*)
this flag is turned true (1) automatically 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.

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

onNode(=*false*)
contact on node?

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

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

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

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

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

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

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CylScGeom6D**(*(object)arg1*)
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).

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

dict() → dict
Return dictionary of attributes.

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.

end(=*Vector3r::Zero()*)
position of 2nd node *(auto-updated)*

id3(=*0*)
id of next chained cylinder *(auto-updated)*

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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)*

isDuplicate(=*0*)
this flag is turned true (1) automatically 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.

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

onNode(=*false*)
contact on node?

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

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

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

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

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

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

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.

twist(=*0*)
Elastic twist angle (around normal axis) 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)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GenericSpheresContact**(*(object)arg1*)
Class uniting ScGeom and L3Geom, for the purposes of GlobalStiffnessTimeStepper. (It might be removed in the future). Do not use this class directly.

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

dict() → dict
Return dictionary of attributes.

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.

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)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GridCoGridCoGeom**(*(object)arg1*)
Geometry of a GridConnection-GridConnection contact.

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

dict() → dict
Return dictionary of attributes.

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.

incidentVel(*(Interaction)i*, *(bool)avoidGranularRatcheting=True*) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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

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

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

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

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

relPos1(=*0*)
position of the contact on the first connection (0: node-, 1:node+) *(auto-updated)*

relPos2(=*0*)
position of the contact on the first connection (0: node-, 1:node+) *(auto-updated)*

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GridNodeGeom6D**(*(object)arg1*)
Geometry of a GridNode-GridNode contact. Inherits almost everything from ScGeom6D.

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

connectionBody(=*uninitalized*)
Reference to the GridNode Body who is linking the two GridNodes.

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

dict() → dict
Return dictionary of attributes.

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.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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)*

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

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

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

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

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

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

twist(=*0*)
Elastic twist angle (around normal axis) 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)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**L3Geom**(*(object)arg1*)
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]

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

dict() → dict
Return dictionary of attributes.

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.

`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)*

`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.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.L6Geom`(*(object)arg1*)
Geometric of contact in local coordinates with 6 degrees of freedom. [experimental]

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

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

`dict`() → dict
Return dictionary of attributes.

`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.

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

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.

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

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

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.

---

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**PolyhedraGeom**(*(object)arg1*)
Geometry of interaction between 2 vector, including volumetric characteristics

contactPoint(=*Vector3r::Zero()*)
Contact point (global coords), centriod of the overlapping polyhedron

dict() → dict
Return dictionary of attributes.

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.

equivalentCrossSection(=*NaN*)
Cross-section area of the overlap (perpendicular to the normal) - not used

equivalentPenetrationDepth(=*NaN*)
volume / equivalentCrossSection - not used

normal(=*Vector3r::Zero()*)
Normal direction of the interaction

orthonormal_axis(=*Vector3r::Zero()*)

penetrationVolume(=*NaN*)
Volume of overlap [m³]

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

twist_axis(=*Vector3r::Zero()*)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ScGeom**(*(object)arg1*)
Class representing geometry of a contact point between two bodies. It is more general than sphere-sphere contact even though it is primarily focused on spheres interactions (reason for the 'Sc' namming); it is also used for representing contacts of a Sphere with non-spherical bodies (Facet, Plane, Box, ChainedCylinder), or between two non-spherical bodies (ChainedCylinder). The contact has 3 DOFs (normal and 2×shear) and uses incremental algorithm for updating shear.

We use symbols $\mathbf{x}$, $\boldsymbol{v}$, $\boldsymbol{\omega}$ respectively for position, linear and angular velocities (all in global coordinates) and r for particles radii; subscripted with 1 or 2 to distinguish 2 spheres in contact. Then we define branch length and unit contact normal

$$l = \|\mathbf{x}_2 - \mathbf{x}_1\|, \mathbf{n} = \frac{\mathbf{x}_2 - \mathbf{x}_1}{\|\mathbf{x}_2 - \mathbf{x}_1\|}$$

The relative velocity of the spheres is then

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

where the fraction multiplying translational velocities is to make the definition objective and avoid ratcheting effects (see Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting). The shear component is

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

Tangential displacement increment over last step then reads

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

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

dict() → dict
Return dictionary of attributes.

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.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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

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

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

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

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

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ScGeom6D**(*(object)arg1*)
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).

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

dict() → dict
Return dictionary of attributes.

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.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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)*

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

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

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

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

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

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

twist(=*0*)
Elastic twist angle (around normal axis) 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)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ScGridCoGeom**(*(object)arg1*)
Geometry of a GridConnection-Sphere contact.

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

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

dict() → dict
Return dictionary of attributes.

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.

id3(=*0*)
id of the first GridNode. *(auto-updated)*

id4(=*0*)
id of the second GridNode. *(auto-updated)*

incidentVel(*(Interaction)i*, *(bool)avoidGranularRatcheting=True*) → Vector3
Return incident velocity of the interaction (see also Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting for explanation of the ratcheting argument).

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)*

isDuplicate(=*0*)
this flag is turned true (1) automatically if the contact is shared between two Connections. 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.

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

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

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

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

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

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

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

trueInt(=*-1*)
Defines the body id of the GridConnection where the contact is real, when ScGridCoGeom::isDuplicate>0.

twist(=*0*)
Elastic twist angle (around normal axis) 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)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**TTetraGeom**(*(object)arg1*)
Geometry of interaction between 2 tetrahedra, including volumetric characteristics

contactPoint(=*uninitalized*)
Contact point (global coords)

dict() → dict
Return dictionary of attributes.

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.

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]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**TTetraSimpleGeom**(*(object)arg1*)
EXPERIMENTAL. Geometry of interaction between 2 tetrahedra

contactPoint(=*uninitalized*)
Contact point (global coords)

dict() → dict
Return dictionary of attributes.

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.

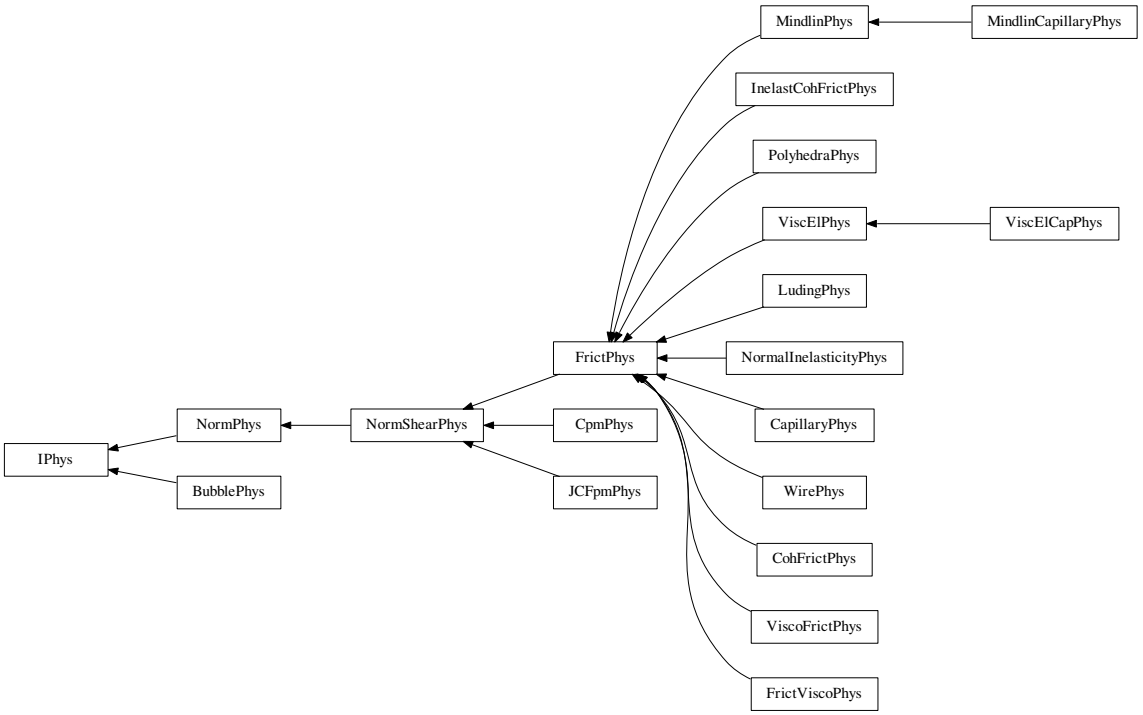
flag(*=0*)
TODO

normal(*=uninitalized*)
Normal of the interaction TODO

penetrationVolume(*=NaN*)
Volume of overlap [m³]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.2.3 IPhys



class yade.wrapper.**IPhys**(*(object)arg1*)
Physical (material) properties of interaction.

dict() → dict
Return dictionary of attributes.

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**BubblePhys**(*(object)arg1*)
Physics of bubble-bubble interactions, for use with BubbleMat

Dmax(*=NaN*)
Maximum penetrationDepth of the bubbles before the force displacement curve changes to an artificial exponential curve. Setting this value will have no effect. See Law2_ScGeom_-BubblePhys_Bubble::pctMaxForce for more information

computeForce(*(float)arg1*, *(float)arg2*, *(float)arg3*, *(int)arg4*, *(float)arg5*, *(float)arg6*, *(float)arg7*, *(BubblePhys)arg8*) → float :
Computes the normal force acting between the two interacting bubbles using the Newton-Rhapson method

dict() → dict
Return dictionary of attributes.

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.

fN(=*NaN*)
Contact normal force

newtonIter(=*50*)
Maximum number of force iterations allowed

newtonTol(=*1e-6*)
Convergence criteria for force iterations

normalForce(=*Vector3r::Zero()*)
Normal force

rAvg(=*NaN*)
Average radius of the two interacting bubbles

surfaceTension(=*NaN*)
Surface tension of the surrounding liquid

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CapillaryPhys**(*(object)arg1*)
Physics (of interaction) for Law2_ScGeom_CapillaryPhys_Capillarity.

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)

capillaryPressure(=*0.*)
Value of the capillary pressure Uc. Defined as Ugas-Uliquid, obtained from corresponding Law2 parameter

dict() → dict
Return dictionary of attributes.

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.

fCap(=*Vector3r::Zero()*)
Capillary force produced by the presence of the meniscus. This is the force acting on particle #2

fusionNumber(=*0.*)
Indicates the number of meniscii that overlap with this one

isBroken(=*false*)
Might be set to true by the user to make liquid bridge inactive (capillary force is zero)

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

meniscus(=*false*)
True when a meniscus with a non-zero liquid volume (vMeniscus) has been computed for this interaction

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vMeniscus(=*0.*)
Volume of the meniscus

class yade.wrapper.**CohFrictPhys**(*(object)arg1*)

cohesionBroken(=*true*)
is cohesion active? Set to false at the creation of a cohesive contact, and set to true 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].

dict() → dict
Return dictionary of attributes.

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.

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

initCohesion(=*false*)
Initialize the cohesive behaviour with current state as equilibrium state (same as Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow but acting on only one interaction)

kn(=*0*)
Normal stiffness

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

ks(=*0*)
Shear stiffness

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

maxRollPl(=*0.0*)
Coefficient of rolling friction (negative means elastic).

maxTwistPl(=*0.0*)
Coefficient of twisting friction (negative means elastic).

momentRotationLaw(=*false*)
use bending/twisting moment at contacts. See Law2_ScGeom6D_CohFrictPhys_CohesionMoment::always_use_moment_law for details.

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

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

normalAdhesion(=*0*)
tensile strength

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

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

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

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

unpMax(=*0*)
maximum value of plastic normal displacement (counted positively), after that the interaction breaks even if CohFrictPhys::fragile =false. A negative value (i.e. -1) means no maximum.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CpmPhys**(*(object)arg1*)
Representation of a single interaction of the Cpm type: storage for relevant parameters.

Evolution of the contact is governed by Law2_ScGeom_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 *(auto-updated)*

Fs
Magnitude of shear force *(auto-updated)*

G(=*NaN*)
shear modulus [Pa]

crossSection(=*NaN*)
equivalent cross-section associated with this contact [m^2]

cummBetaCount = 0

cummBetaIter = 0

damLaw(=*1*)
Law for softening part of uniaxial tension. 0 for linear, 1 for exponential (default)

dict() → dict
Return dictionary of attributes.

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.

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 at which the bond is fully broken [-]

epsN
Current normal strain *(auto-updated)*

epsNPl
normal plastic strain (initially zero) *(auto-updated)*

epsT
Current shear strain *(auto-updated)*

epsTPl
shear plastic strain (initially zero) *(auto-updated)*

funcG(*(float)kappaD*, *(float)epsCrackOnset*, *(float)epsFracture*[, *(bool)neverDamage=False*[, *(int)damLaw=1*]]) → 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). TODO

funcGInv(*(float)omega*, *(float)epsCrackOnset*, *(float)epsFracture*[, *(bool)neverDamage=False*[, *(int)damLaw=1*]]) → float :
Inversion of damage evolution law, evaluating the κ_D parameter. ω is damage, for other parameters see funcG function

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
Up to now maximum normal strain (semi-norm), non-decreasing in time *(auto-updated)*

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

neverDamage(*=false*)
the damage evolution function will always return virgin state

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

omega
Damage internal variable *(auto-updated)*

plRateExp(*=0*)
exponent in the rate-dependent viscoplasticity

plTau(*=-1*)
characteristic time for viscoplasticity (if non-positive, no rate-dependence for shear)

refLength(*=NaN*)
initial length of interaction [m]

refPD(*=NaN*)
initial penetration depth of interaction [m] (used with ScGeom)

relDuctility(*=NaN*)
Relative ductility of bonds in normal direction

relResidualStrength
Relative residual strength *(auto-updated)*

setDamage(*(float)arg2*) → None
TODO

setRelResidualStrength(*(float)arg2*) → None
TODO

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

sigmaN
Current normal stress *(auto-updated)*

sigmaT
Current shear stress *(auto-updated)*

tanFrictionAngle(*=NaN*)
tangens of internal friction angle [-]

undamagedCohesion(*=NaN*)
virgin material cohesion [Pa]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**FrictPhys**(*(object)arg1*)
The simple linear elastic-plastic interaction with friction angle, like in the traditional [Cundall-Strack1979]

dict() → dict
Return dictionary of attributes.

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.

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

class `yade.wrapper.FrictViscoPhys(`*(object)arg1*`)`
Representation of a single interaction of the FrictViscoPM type, storage for relevant parameters

`cn(`*=NaN*`)`
Normal viscous constant defined as $\mathtt{n} = \mathtt{c}_{\mathtt{n,crit}}\beta_{\mathtt{n}}$.

`cn_crit(`*=NaN*`)`
Normal viscous constant for ctitical damping defined as $\mathtt{n} = \mathtt{C}_{\mathtt{n,crit}}\beta_{\mathtt{n}}$.

`dict()` → dict
Return dictionary of attributes.

`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.

`kn(`*=0*`)`
Normal stiffness

`ks(`*=0*`)`
Shear stiffness

`normalForce(`*=Vector3r::Zero()*`)`
Normal force after previous step (in global coordinates).

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

`shearForce(`*=Vector3r::Zero()*`)`
Shear force after previous step (in global coordinates).

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

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

class `yade.wrapper.InelastCohFrictPhys(`*(object)arg1*`)`

`cohesionBroken(`*=false*`)`
is cohesion active? will be set false when a fragile contact is broken

`dict()` → dict
Return dictionary of attributes.

`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.

`isBroken(`*=false*`)`
true if compression plastic fracture achieved

`kDam(`*=0*`)`
Damage coefficient on bending, computed from maximum bending moment reached and pure

creep behaviour. Its values will vary between InelastCohFrictPhys::kr and InelastCohFrictPhys::kRCrp .

kRCrp(=*0.0*)
Bending creep stiffness

kRUnld(=*0.0*)
Bending plastic unload stiffness

kTCrp(=*0.0*)
Tension/compression creep stiffness

kTUnld(=*0.0*)
Tension/compression plastic unload stiffness

kTwCrp(=*0.0*)
Twist creep stiffness

kTwUnld(=*0.0*)
Twist plastic unload stiffness

kn(=*0*)
Normal stiffness

knC(=*0*)
compression stiffness

knT(=*0*)
tension stiffness

kr(=*0*)
bending stiffness

ks(=*0*)
shear stiffness

ktw(=*0*)
twist shear stiffness

maxBendMom(=*0.0*)
Plastic failure bending moment.

maxContract(=*0.0*)
Plastic failure contraction (shrinkage).

maxCrpRchdB(=*Vector3r(0, 0, 0)*)
maximal bending moment reached on plastic deformation.

maxCrpRchdC(=*Vector2r(0, 0)*)
maximal compression reached on plastic deformation. maxCrpRchdC[0] stores un and maxCrpRchdC[1] stores Fn.

maxCrpRchdT(=*Vector2r(0, 0)*)
maximal extension reached on plastic deformation. maxCrpRchdT[0] stores un and maxCrpRchdT[1] stores Fn.

maxCrpRchdTw(=*Vector2r(0, 0)*)
maximal twist reached on plastic deformation. maxCrpRchdTw[0] stores twist angle and maxCrpRchdTw[1] stores twist moment.

maxElB(=*0.0*)
Maximum bending elastic moment.

maxElC(=*0.0*)
Maximum compression elastic force.

maxElT(=*0.0*)
Maximum tension elastic force.

maxElTw(=*0.0*)
Maximum twist elastic moment.

maxExten(=*0.0*)
Plastic failure extension (stretching).

maxTwist(=*0.0*)
Plastic failure twist angle

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

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

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

onPlastB(=*false*)
true if plasticity achieved on bending

onPlastC(=*false*)
true if plasticity achieved on compression

onPlastT(=*false*)
true if plasticity achieved on traction

onPlastTw(=*false*)
true if plasticity achieved on twisting

pureCreep(=*Vector3r(0, 0, 0)*)
Pure creep curve, used for comparison in calculation.

shearAdhesion(=*0*)
Maximum elastic shear force (cohesion).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

twp(=*0*)
plastic twist penetration depth describing the equilibrium state.

unp(=*0*)
plastic normal penetration depth describing the equilibrium state.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**JCFpmPhys**(*(object)arg1*)
Representation of a single interaction of the JCFpm type, storage for relevant parameters

FnMax(=*0*)
positiv value computed from tensile strength (or joint variant) to define the maximum admissible normal force in traction: Fn >= -FnMax. [N]

FsMax(=*0*)
computed from cohesion (or jointCohesion) to define the maximum admissible tangential force in shear, for Fn=0. [N]

crackJointAperture(=*0*)
Relative displacement between 2 spheres (in case of a crack it is equivalent of the crack aperture)

crossSection(=*0*)
crossSection=pi*Rmin^2. [m2]

dict() → dict
Return dictionary of attributes.

dilation(=*0*)
defines the normal displacement in the joint after sliding treshold. [m]

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.

initD(*=0*)
equilibrium distance for interacting particles. Computed as the interparticular distance at first contact detection.

isBroken(*=false*)
flag for broken interactions

isCohesive(*=false*)
If false, particles interact in a frictional way. If true, particles are bonded regarding the given cohesion and tensile strength (or their jointed variants).

isOnJoint(*=false*)
defined as true when both interacting particles are on joint and are in opposite sides of the joint surface. In this case, mechanical parameters of the interaction are derived from the ''joint..." material properties of the particles. Furthermore, the normal of the interaction may be re-oriented (see Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM.smoothJoint).

jointCumulativeSliding(*=0*)
sliding distance for particles interacting on a joint. Used, when is true, to take into account dilatancy due to shearing. [-]

jointNormal(*=Vector3r::Zero()*)
normal direction to the joint, deduced from e.g. .

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

more(*=false*)
specifies if the interaction is crossed by more than 3 joints. If true, interaction is deleted (temporary solution).

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tanDilationAngle(*=0*)
tangent of the angle defining the dilatancy of the joint surface (auto. computed from JCFpmMat.jointDilationAngle). [-]

tanFrictionAngle(*=0*)
tangent of Coulomb friction angle for this interaction (auto. computed). [-]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**LudingPhys**(*(object)arg1*)
IPhys created from LudingMat, for use with Law2_ScGeom_LudingPhys_Basic.

DeltMax(*=NaN*)
Maximum overlap between particles for a collision

DeltMin(*=NaN*)
MinimalDelta value of delta

DeltNull(*=NaN*)
Force free overlap, plastic contact deformation

`DeltPMax(=NaN)`
Maximum overlap between particles for the limit case

`DeltPNull(=NaN)`
Max force free overlap, plastic contact deformation

`DeltPrev(=NaN)`
Previous value of delta

`G0(=NaN)`
Viscous damping

`PhiF(=NaN)`
Dimensionless plasticity depth

`dict()` → dict
Return dictionary of attributes.

`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.

`k1(=NaN)`
Slope of loading plastic branch

`k2(=NaN)`
Slope of unloading and reloading elastic branch

`kc(=NaN)`
Slope of irreversible, tensile adhesive branch

`kn(=0)`
Normal stiffness

`kp(=NaN)`
Slope of unloading and reloading limit elastic branch

`ks(=0)`
Shear stiffness

`normalForce(=Vector3r::Zero())`
Normal force after previous step (in global coordinates).

`shearForce(=Vector3r::Zero())`
Shear force after previous step (in global coordinates).

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

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

class `yade.wrapper.MindlinCapillaryPhys(`*(object)arg1*`)`
Adds capillary physics to Mindlin's interaction physics.

`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)

`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*)
Normal Damping Ratio. Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

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

capillaryPressure(*=0.*)
Value of the capillary pressure Uc. Defined as Ugas-Uliquid, obtained from corresponding Law2 parameter

dict() → dict
Return dictionary of attributes.

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.

fCap(*=Vector3r::Zero()*)
Capillary Force produces by the presence of the meniscus. This is the force acting on particle #2

fusionNumber(*=0.*)
Indicates the number of meniscii that overlap with this one

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

isBroken(*=false*)
Might be set to true by the user to make liquid bridge inactive (capillary force is zero)

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

kn(*=0*)
Normal stiffness

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

kr(*=0.0*)
Rotational stiffness

ks(*=0*)
Shear 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

meniscus(*=false*)
True when a meniscus with a non-zero liquid volume (vMeniscus) has been computed for this interaction

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)

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

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

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

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

vMeniscus(=*0.*)
Volume of the meniscus

class yade.wrapper.**MindlinPhys**(*(object)arg1*)
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*)
Normal Damping Ratio. Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

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

dict() → dict
Return dictionary of attributes.

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.

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)

kn(=*0*)
Normal stiffness

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

kr(=*0.0*)
Rotational stiffness

ks(=*0*)
Shear 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)

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

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

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

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

class yade.wrapper.**NormPhys**(*(object)arg1*)
Abstract class for interactions that have normal stiffness.

dict() → dict
Return dictionary of attributes.

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.

kn(=*0*)
Normal stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NormShearPhys**(*(object)arg1*)
Abstract class for interactions that have shear stiffnesses, in addition to normal stiffness. This class is used in the PFC3d-style stiffness timestepper.

dict() → dict
Return dictionary of attributes.

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.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NormalInelasticityPhys**(*(object)arg1*)
Physics (of interaction) for using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity : with inelastic unloadings

dict() → dict
Return dictionary of attributes.

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.

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

kn(=*0*)
Normal stiffness

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

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

ks(=*0*)
Shear stiffness

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

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

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

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

unMax(=*0.0*)
the maximum value of penetration depth of the history of this interaction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**PolyhedraPhys**(*(object)arg1*)
Simple elastic material with friction for volumetric constitutive laws

dict() → dict
Return dictionary of attributes.

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.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ViscElCapPhys**(*(object)arg1*)
IPhys created from ViscElCapMat, for use with Law2_ScGeom_ViscElCapPhys_Basic.

Capillar(=*false*)
True, if capillar forces need to be added.

CapillarType(=*None_Capillar*)
Different types of capillar interaction: Willett_numeric, Willett_analytic, Weigert, Rabinovich, Lambert, Soulie

Vb(=*0.0*)
Liquid bridge volume [m^3]

cn(=*NaN*)
Normal viscous constant

cs(=*NaN*)
Shear viscous constant

dict() → dict
Return dictionary of attributes.

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.

gamma(=*0.0*)
Surface tension [N/m]

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

liqBridgeActive(=*false*)
Whether liquid bridge is active at the moment

liqBridgeCreated(=*false*)
Whether liquid bridge was created, only after a normal contact of spheres

mR(=*0.0*)
Rolling resistance, see [Zhou1999536].

mRtype(=*1*)
Rolling resistance type, see [Zhou1999536]. mRtype=1 - equation (3) in [Zhou1999536]; mRtype=2 - equation (4) in [Zhou1999536]

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

sCrit(=*false*)
Critical bridge length [m]

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

theta(=*0.0*)
Contact angle [rad]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ViscElPhys**(*(object)arg1*)
IPhys created from ViscElMat, for use with Law2_ScGeom_ViscElPhys_Basic.

cn(=*NaN*)
Normal viscous constant

cs(=*NaN*)
Shear viscous constant

dict() → dict
Return dictionary of attributes.

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.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

mR(=*0.0*)
Rolling resistance, see [Zhou1999536].

mRtype(=*1*)
Rolling resistance type, see [Zhou1999536]. mRtype=1 - equation (3) in [Zhou1999536]; mRtype=2 - equation (4) in [Zhou1999536]

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ViscoFrictPhys**(*(object)arg1*)
Temporary version of FrictPhys for compatibility with e.g. Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

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

dict() → dict
Return dictionary of attributes.

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.

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**WirePhys**(*(object)arg1*)
Representation of a single interaction of the WirePM type, storage for relevant parameters

dL(*=0.*)
Additional wire length for considering the distortion for WireMat type=2 (see [Thoeni2013]).

dict() → dict
Return dictionary of attributes.

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.

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).

isShifted(*=false*)
If true WireMat type=2 and the force-displacement curve will be shifted.

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

limitFactor(*=0.*)
This value indicates on how far from failing the wire is, e.g. actual normal displacement divided by admissible normal displacement.

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

`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.

`shearForce`(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

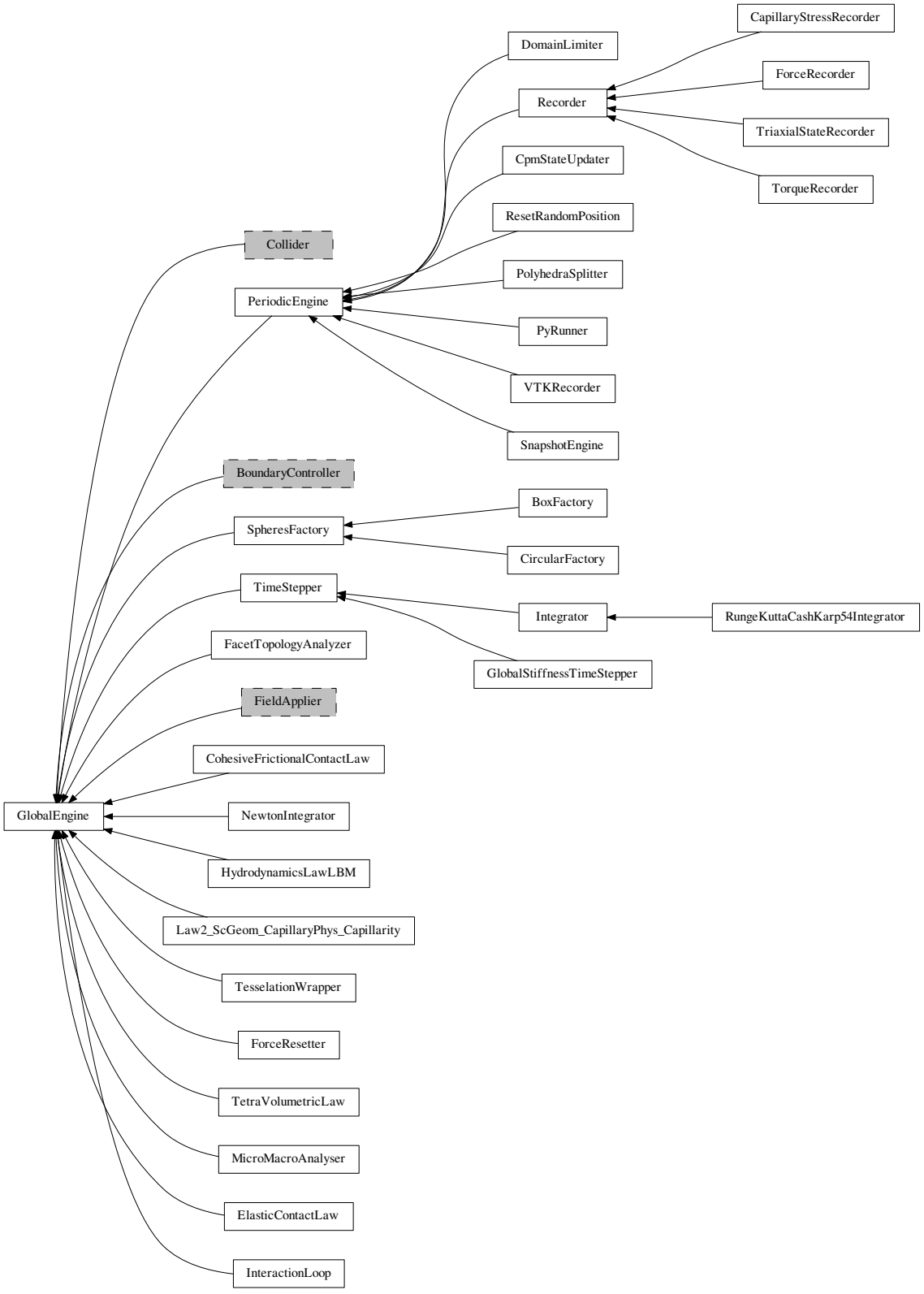
`stiffnessValues`(=*uninitalized*)
Defines the values for the various stiffnesses (the elastic stiffness is stored as kn).

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

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

1.3 Global engines

1.3.1 GlobalEngine



class `yade.wrapper.GlobalEngine`(*(object)arg1*)

Engine that will generally affect the whole simulation (contrary to PartialEngine).

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.

dict() → dict
Return dictionary of attributes.

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.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**BoxFactory**(*(object)arg1*)
Box geometry of the SpheresFactory region, given by extents and center

PSDcalculateMass(=*true*)
PSD-Input is in mass (true), otherwise the number of particles will be considered.

PSDcum(=*uninitalized*)
PSD-dispersion, cumulative procent meanings [-]

PSDsizes(=*uninitalized*)
PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(=*"“*)
Blocked degress of freedom

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

color(=*Vector3r(-1, -1, -1)*)
Use the color for newly created particles, if specified

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.

dict() → dict
Return dictionary of attributes.

exactDiam(=*true*)
If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

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).

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

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

ids(=*uninitalized*)
ids of created bodies

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*-1*)
groupMask to apply for newly created spheres

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.

maxMass(=*-1*)
Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(=*100*)
The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(=*Vector3r(NaN, NaN, NaN)*)
Orientation of the region's geometry, direction of particle's velocites if normalVel is not set.

normalVel(=*Vector3r(NaN, NaN, NaN)*)
Direction of particle's velocites.

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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).

stopIfFailed(=*true*)
If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

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

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

class yade.wrapper.**CapillaryStressRecorder**(*(object)arg1*)
Records information from capillary meniscii on samples submitted to triaxial compressions. Classical sign convention (tension positiv) is used for capillary stresses. -> New formalism needs to be tested!!!

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)

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.

dict() → dict
Return dictionary of attributes.

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).

file(=*uninitalized*)
Name of file to save to; must not be empty.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

truncate(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**CircularFactory**(*(object)arg1*)
Circular geometry of the SpheresFactory region. It can be disk (given by radius and center), or cylinder (given by radius, length and center).

PSDcalculateMass(=*true*)
PSD-Input is in mass (true), otherwise the number of particles will be considered.

PSDcum(=*uninitalized*)
PSD-dispersion, cumulative procent meanings [-]

PSDsizes(=*uninitalized*)
PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(=*"“*)
Blocked degress of freedom

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

color(=*Vector3r(-1*, *-1*, *-1)*)
Use the color for newly created particles, if specified

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.

dict() → dict
Return dictionary of attributes.

exactDiam(=*true*)
If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

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`).

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

ids(=*uninitalized*)
ids of created bodies

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

mask(=*-1*)
groupMask to apply for newly created spheres

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.

maxMass(=*-1*)
Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(=*100*)
The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(=*Vector3r(NaN, NaN, NaN)*)
Orientation of the region's geometry, direction of particle's velocites if normalVel is not set.

normalVel(=*Vector3r(NaN, NaN, NaN)*)
Direction of particle's velocites.

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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

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

radius(=*NaN*)
Radius of the region

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

stopIfFailed(=*true*)
If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

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

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

class yade.wrapper.**CohesiveFrictionalContactLaw**(*(object)arg1*)
[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.

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

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.

dict() → dict
Return dictionary of attributes.

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.

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)

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

shear_creep(=*false*)
activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CpmStateUpdater**(*(object)arg1*)
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.

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.

dict() → dict
Return dictionary of attributes.

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).

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxOmega(=*NaN*)
Globally maximum damage parameter at last run.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`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.DomainLimiter(`*(object)arg1*`)`
Delete particles that are out of axis-aligned box given by *lo* and *hi*.

`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.

`dict()` → dict
Return dictionary of attributes.

`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`).

`firstIterRun(=0)`
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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

`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)

`label(=uninitalized)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

`mDeleted(=0)`
Mass of deleted particles.

`mask(=-1)`
If mask is defined, only particles with corresponding groupMask will be deleted.

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

`nDo(=-1, deactivated)`
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vDeleted(=*0*)
Volume of deleted particles.

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.**ElasticContactLaw**(*(object)arg1*)
[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.

---

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.

dict() → dict
Return dictionary of attributes.

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.

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)

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.FacetTopologyAnalyzer`(*(object)arg1*)
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)*

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.ForceRecorder`(*(object)arg1*)
Engine saves the resultant force affecting to bodies, listed in *ids*. For instance, can be useful for defining the forces, which affects to _buldozer_ during its work.

`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)

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.

dict() → dict
Return dictionary of attributes.

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`).

file(=*uninitalized*)
Name of file to save to; must not be empty.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

totalForce(=*Vector3r::Zero()*)
Resultant force, returning by the function.

truncate(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`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.ForceResetter`(*(object)arg1*)
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.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlobalStiffnessTimeStepper`(*(object)arg1*)
An engine assigning the time-step as a fraction of the minimum eigen-period in the problem. The derivation is detailed in the chapter on DEM formulation. The viscEl option enables to evaluate the timestep in a similar way for the visco-elastic contact law Law2_ScGeom_ViscElPhys_Basic, more detail in GlobalStiffnessTimestepper::viscEl.

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

`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.

`defaultDt`(*=-1*)
used as the initial value of the timestep (especially useful in the first steps when no contact exist). If negative, it will be defined by utils.PWaveTimeStep * GlobalStiffnessTimeStepper::timestepSafetyCoefficient

`densityScaling`(*=false*)
(auto-updated) don't modify this value if you don't plan to modify the scaling factor manually for some bodies. In most cases, it is enough to set NewtonIntegrator::densityScaling and let this one be adjusted automatically.

`dict()` → dict
Return dictionary of attributes.

`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.

`maxDt`(=*Mathr::MAX_REAL*)
if positive, used as max value of the timestep whatever the computed value

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`previousDt`(=*1*)
last computed dt *(auto-updated)*

`targetDt`(=*1*)
if NewtonIntegrator::densityScaling is active, this value will be used as the simulation timestep and the scaling will use this value of dt as the target value. The value of targetDt is arbitrary and should have no effect in the result in general. However if some bodies have imposed velocities, for instance, they will move more or less per each step depending on this value.

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

`timestepSafetyCoefficient`(=*0.8*)
safety factor between the minimum eigen-period and the final assigned dt (less than 1)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`viscEl`(=*false*)
To use with ViscElPhys. if True, evaluate separetly the minimum eigen-period in the problem considering only the elastic contribution on one hand (spring only), and only the viscous contribution on the other hand (dashpot only). Take then the minimum of the two and use the safety coefficient GlobalStiffnessTimestepper::timestepSafetyCoefficient to take into account the possible coupling between the two contribution.

class `yade.wrapper.HydrodynamicsLawLBM`(*(object)arg1*)
Engine to simulate fluid flow (with the lattice Boltzmann method) with a coupling with the discrete element method. If you use this Engine, please cite and refer to F. Lominé et al. International Journal For Numerical and Analytical Method in Geomechanics, 2012, doi: 10.1002/nag.1109

`ConvergenceThreshold`(=*0.000001*)

`CstBodyForce`(=*Vector3r::Zero()*)
A constant body force (=that does not vary in time or space, otherwise the implementation introduces errors)

`DemIterLbmIterRatio`(=*-1*)
Ratio between DEM and LBM iterations for subcycling

`EndTime`(=*-1*)
: the time to stop the simulation

`EngineIsActivated`(=*true*)
: To activate (or not) the engine

`IterMax`(=*1*)
: This variable can be used to do several LBM iterations during one DEM iteration.

`IterPrint`(=*1*)
: Print info on screen every IterPrint iterations

`IterSave`(=*100*)
: Data are saved every IterSave LBM iteration (or see TimeSave)

`IterSubCyclingStart`(=*-1*)
: Iteration number when the subcycling process starts

`LBMSavedData`(=" ")
: a list of data that will be saved. Can use velocity,velXY,forces,rho,bodies,nodeBD,newNode,observedptc,observednode,contacts,spheres,bz2

`Nu`(=*0.000001*)
: Fluid kinematic viscosity

`Nx`(=*1000*)
: The number of grid division in x direction

`ObservedNode`(=*-1*)
: The identifier of the node that will be observed (-1 means none)

`ObservedPtc`(=*-1*)
: The identifier of the particle that will be observed (-1 means the first one)

`RadFactor`(=*1.0*)
: The radius of DEM particules seen by the LBM is the real radius of particules*RadFactor

`Rho`(=*1000.*)
: Fluid density

`SaveGridRatio`(=*1*)
: Grid data are saved every SaveGridRatio * IterSave LBM iteration (with SaveMode=1)

`SaveMode`(=*1*)
: Save Mode (1-> default, 2-> in time (not yet implemented)

`TimeSave`(=*-1*)
: Data are saved at constant time interval (or see IterSave)

`VbCutOff`(=*-1*)
: the minimum boundary velocity that is taken into account

`VelocityThreshold`(=*-1.*)
: Velocity threshold when removingCriterion=2

`WallXm_id`(=*2*)
: Identifier of the X- wall

`WallXp_id`(=*3*)
: Identifier of the X+ wall

`WallYm_id`(=*0*)
: Identifier of the Y- wall

`WallYp_id`(=*1*)
: Identifier of the Y+ wall

`WallZm_id`(=*4*)
: Identifier of the Z- wall

`WallZp_id(=5)`
Identifier of the Z+ wall

`XmBCType(=1)`
Boundary condition for the wall in Xm (-1: unused, 1: pressure condition, 2: velocity condition).

`XmBcRho(=-1)`
(!!! not fully implemented !!) The density imposed at the boundary

`XmBcVel(=Vector3r::Zero())`
(!!! not fully implemented !!) The velocity imposed at the boundary

`XmYmZmBCType(=-1)`
Boundary condition for the corner node XmYmZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XmYmZpBCType(=2)`
Boundary condition for the corner node XmYmZp (-1: unused, 1: pressure condition, 2: velocity condition).

`XmYpZmBCType(=-1)`
Boundary condition for the corner node XmYpZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XmYpZpBCType(=2)`
Boundary condition for the corner node XmYpZp (-1: unused, 1: pressure condition, 2: velocity condition).

`XpBCType(=1)`
Boundary condition for the wall in Xp (-1: unused, 1: pressure condition, 2: velocity condition).

`XpBcRho(=-1)`
(!!! not fully implemented !!) The density imposed at the boundary

`XpBcVel(=Vector3r::Zero())`
(!!! not fully implemented !!) The velocity imposed at the boundary

`XpYmZmBCType(=-1)`
Boundary condition for the corner node XpYmZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XpYmZpBCType(=2)`
Boundary condition for the corner node XpYmZp (-1: unused, 1: pressure condition, 2: velocity condition).

`XpYpZmBCType(=-1)`
Boundary condition for the corner node XpYpZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XpYpZpBCType(=2)`
Boundary condition for the corner node XpYpZp (-1: unused, 1: pressure condition, 2: velocity condition).

`YmBCType(=2)`
Boundary condition for the wall in Ym (-1: unused, 1: pressure condition, 2: velocity condition).

`YmBcRho(=-1)`
(!!! not fully implemented !!) The density imposed at the boundary

`YmBcVel(=Vector3r::Zero())`
(!!! not fully implemented !!) The velocity imposed at the boundary

`YpBCType(=2)`
Boundary condition for the wall in Yp (-1: unused, 1: pressure condition, 2: velocity condition).

YpBcRho(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

YpBcVel(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

ZmBCType(=*-1*)
Boundary condition for the wall in Zm (-1: unused, 1: pressure condition, 2: velocity condition).

ZmBcRho(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

ZmBcVel(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

ZpBCType(=*-1*)
Boundary condition for the wall in Zp (-1: unused, 1: pressure condition, 2: velocity condition).

ZpBcVel(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

applyForcesAndTorques(=*true*)
Switch to apply forces and torques

bc(=*" "*)
Boundary condition

dP(=*Vector3r(0., 0., 0.)*)
Pressure difference between input and output

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.

defaultLbmInitMode(=*0*)
Switch between the two initialisation methods

dict() → dict
Return dictionary of attributes.

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.

model(=*"d2q9"*)
The LB model. Until now only d2q9 is implemented

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

periodicity(=*" "*)
periodicity

removingCriterion(=*0*)
Criterion to remove a sphere (1->based on particle position, 2->based on particle velocity

tau(*=0.6*)
Relaxation time

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

useWallXm(*=false*)
Set true if you want that the LBM see the wall in Xm

useWallXp(*=false*)
Set true if you want that the LBM see the wall in Xp

useWallYm(*=true*)
Set true if you want that the LBM see the wall in Ym

useWallYp(*=true*)
Set true if you want that the LBM see the wall in Yp

useWallZm(*=false*)
Set true if you want that the LBM see the wall in Zm

useWallZp(*=false*)
Set true if you want that the LBM see the wall in Zp

zpBcRho(*=-1*)
(!!! not fully implemented !!) The density imposed at the boundary

class yade.wrapper.**Integrator**(*(object)arg1*)
Integration Engine Interface.

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

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.

dict() → dict
Return dictionary of attributes.

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).

integrationsteps(*=uninitalized*)
all integrationsteps count as all succesfull substeps

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxVelocitySq(*=NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

slaves
List of lists of Engines to calculate the force acting on the particles; to obtain the derivatives of the states, engines inside will be run sequentially.

timeStepUpdateInterval(*=1*)
dt update interval

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**InteractionLoop**(*(object)arg1*)
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.

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.

dict() → dict
Return dictionary of attributes.

eraseIntsInLoop(*=false*)
Defines if the interaction loop should erase pending interactions, else the collider takes care of that alone (depends on what collider is used).

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`).

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

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_CapillaryPhys_Capillarity**(*(object)arg1*)
This law allows one to take into account capillary forces/effects between spheres coming from the presence of interparticular liquid bridges (menisci).

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).

References: in english [Scholtes2009b]; more detailed, but in french [Scholtes2009d].

The law needs ascii files M(r=i) with i=R1/R2 to work (see https://yade-dem.org/wiki/CapillaryTriaxialTest). These ASCII files contain a set of results from the resolution of the Laplace-Young equation for different configurations of the interacting geometry, assuming a null wetting angle.

In order to allow capillary forces between distant spheres, it is necessary to enlarge the bounding boxes using Bo1_Sphere_Aabb::aabbEnlargeFactor and make the Ig2 define define distant interactions via interactionDetectionFactor. It is also necessary to disable interactions removal by the constitutive law (Law2). The only combinations of laws supported are currently capillary law + Law2_ScGeom_FrictPhys_CundallStrack and capillary law + Law2_ScGeom_MindlinPhys_Mindlin (and the other variants of Hertz-Mindlin).

See CapillaryPhys-example.py for an example script.

binaryFusion(=*true*)
If true, capillary forces are set to zero as soon as, at least, 1 overlap (menisci fusion) is detected. Otherwise fCap = fCap / (fusionNumber + 1)

capillaryPressure(=*0.*)
Value of the capillary pressure Uc defined as Uc=Ugas-Uliquid

createDistantMeniscii(=*false*)
Generate meniscii between distant spheres? Else only maintain the existing ones. For modeling a wetting path this flag should always be false. For a drying path it should be true for one step (initialization) then false, as in the logic of [Scholtes2009c]

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.

dict() → dict
Return dictionary of attributes.

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`).

fusionDetection(=*false*)
If true potential menisci overlaps are checked, computing fusionNumber for each capillary interaction, and reducing fCap according to binaryFusion

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.MicroMacroAnalyser`(*(object)arg1*)
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).

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

`incrtNumber`(=*1*)

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nonSphereAsFictious`(=*true*)
bodies that are not spheres will be used to defines bounds (else just skipped).

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NewtonIntegrator**(*(object)arg1*)
Engine integrating newtonian motion equations.

damping(*=0.2*)
damping coefficient for Cundall's non viscous damping (see numerical damping and [Chareyre2005])

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.

densityScaling
if True, then density scaling [Pfc3dManual30] will be applied in order to have a critical timestep equal to GlobalStiffnessTimeStepper::targetDt for all bodies. This option makes the simulation unrealistic from a dynamic point of view, but may speedup quasistatic simulations. In rare situations, it could be useful to not set the scalling factor automatically for each body (which the time-stepper does). In such case revert GlobalStiffnessTimeStepper.densityScaling to False.

dict() → dict
Return dictionary of attributes.

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

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`).

gravity(*=Vector3r::Zero()*)
Gravitational acceleration (effectively replaces GravityEngine).

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

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(*=-1*)
If mask defined and the bitwise AND between mask and body's groupMask gives 0, the body will not move/rotate. Velocities and accelerations will be calculated not paying attention to this parameter.

maxVelocitySq(*=NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

prevVelGrad(*=Matrix3r::Zero()*)
Store previous velocity gradient (Cell::velGrad) to track acceleration. *(auto-updated)*

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**PeriodicEngine**(*(object)arg1*)
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 not 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 O.iter')
```

will print iteration number every 10000 iterations or every 5 seconds of wall clock time, whiever comes first since it was last run.

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.

dict() → dict
Return dictionary of attributes.

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).

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**PolyhedraSplitter**(*(object)arg1*)
Engine that splits polyhedras.

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.

dict() → dict
Return dictionary of attributes.

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`).

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone(=0)`
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads(=-1)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`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(`*(object)arg1*`)`
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.

`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.

`dict()` → dict
Return dictionary of attributes.

`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`).

`firstIterRun(=0)`
Sets the step number, at each an engine should be executed for the first time (disabled by default).

`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)

`label(=uninitalized)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo(=-1, deactivated)`
Limit number of executions by this number (deactivated if negative)

`nDone(=0)`
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads(=-1)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`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.Recorder(`*(object)arg1*`)`
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)

`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.

`dict()` → dict
Return dictionary of attributes.

`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`).

`file(=uninitalized)`
Name of file to save to; must not be empty.

`firstIterRun(=0)`
Sets the step number, at each an engine should be executed for the first time (disabled by default).

`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)

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`truncate`(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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.ResetRandomPosition`(*(object)arg1*)
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.

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

`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.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

normal(=*Vector3r(0, 1, 0)*)
??

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

point(=*Vector3r::Zero()*)
??

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)

subscribedBodies(=*uninitalized*)
Affected bodies.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.

virtLast(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

`volumeSection`(=*false*, *define factory by facets.*)
Create new spheres inside factory volume rather than on its surface.

class `yade.wrapper.RungeKuttaCashKarp54Integrator`(*(object)arg1*)
RungeKuttaCashKarp54Integrator engine.

`__init__`() → None
object ___init___(tuple args, dict kwds)

___init___((list)arg2) → object : Construct from (possibly nested) list of slaves.

`a_dxdt`(=*1.0*)

`a_x`(=*1.0*)

`abs_err`(=*1e-6*)
Relative integration tolerance

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

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

`integrationsteps`(=*uninitalized*)
all integrationsteps count as all succesfull substeps

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`maxVelocitySq`(=*NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rel_err`(=*1e-6*)
Absolute integration tolerance

`slaves`
List of lists of Engines to calculate the force acting on the particles; to obtain the derivatives of the states, engines inside will be run sequentially.

`stepsize`(=*1e-6*)
It is not important for an adaptive integration but important for the observer for setting the found states after integration

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

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.SnapshotEngine`(*(object)arg1*)
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)*

`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.

`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]

`dict`() → dict
Return dictionary of attributes.

`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`).

`fileBase`(=*""*)
Basename for snapshots

`firstIterRun`(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

`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.

`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)

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`msecSleep`(=*0*)
number of msec to sleep after snapshot (to prevent 3d hw problems) [ms]

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

plot(=*uninitalized*)
Name of field in plot.imgData to which taken snapshots will be appended automatically.

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)

snapshots(=*uninitalized*)
Files that have been created so far

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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.**SpheresFactory**(*(object)arg1*)
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 overridden SpheresFactory::pickRandomPosition().

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

PSDcalculateMass(=*true*)
PSD-Input is in mass (true), otherwise the number of particles will be considered.

PSDcum(=*uninitalized*)
PSD-dispersion, cumulative procent meanings [-]

PSDsizes(=*uninitalized*)
PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(="")
Blocked degress of freedom

color(=*Vector3r(-1, -1, -1)*)
Use the color for newly created particles, if specified

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.

dict() → dict
Return dictionary of attributes.

exactDiam(=*true*)
If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

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).

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

ids(=*uninitalized*)
ids of created bodies

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*-1*)
groupMask to apply for newly created spheres

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.

maxMass(=*-1*)
Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(=*100*)
The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(=*Vector3r(NaN, NaN, NaN)*)
Orientation of the region's geometry, direction of particle's velocites if normalVel is not set.

normalVel(=*Vector3r(NaN, NaN, NaN)*)
Direction of particle's velocites.

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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).

stopIfFailed(=*true*)
If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

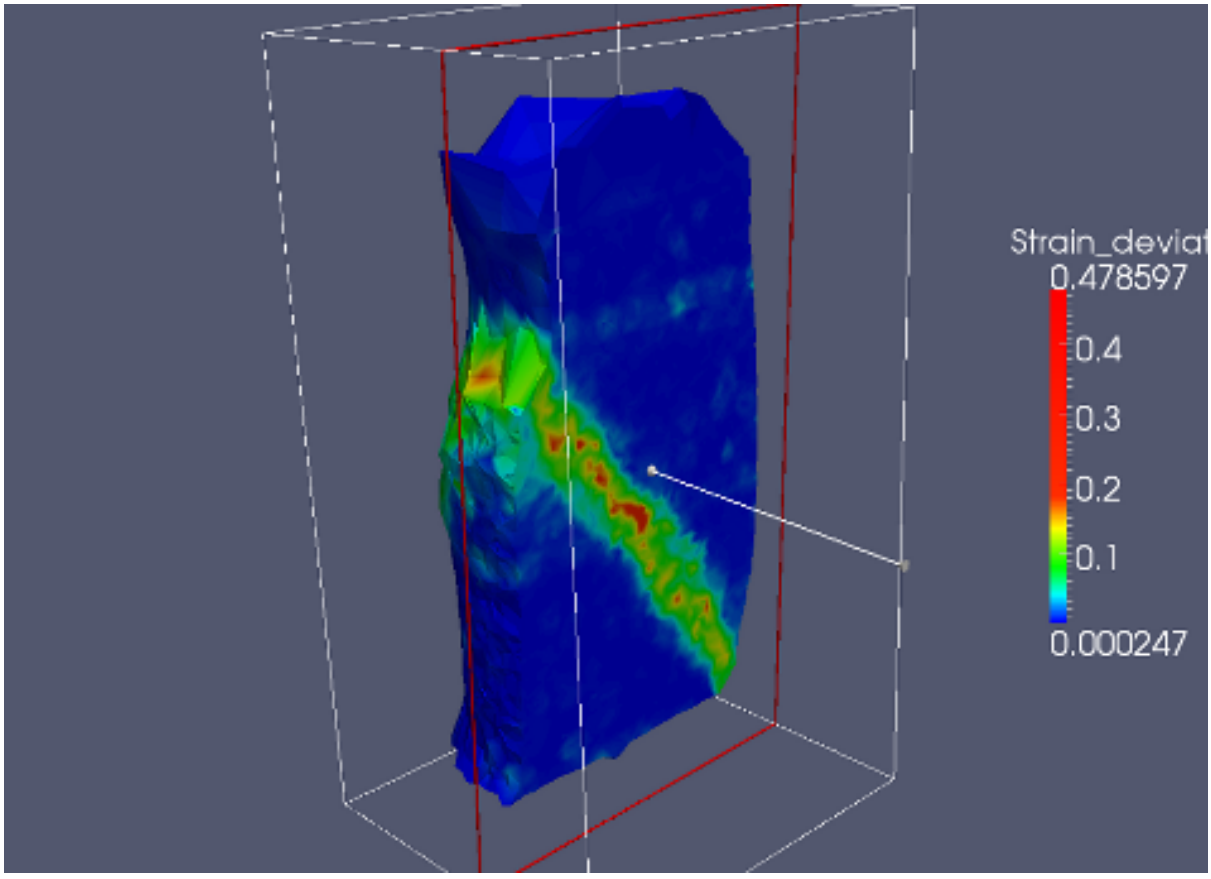
vMax(=*NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

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

class yade.wrapper.**TesselationWrapper**(*(object)arg1*)
Handle the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities (see also the dedicated section in user manual). The calculation of microstrain is explained in [Catalano2014a]

See example usage in script example/tesselationWrapper/tesselationWrapper.py.

Below is an output of the defToVtk function visualized with paraview (in this case Yade's TesselationWrapper was used to process experimental data obtained on sand by Edward Ando at Grenoble University, 3SR lab.)



computeDeformations() → None
compute per-particle deformation. Get it with TesselationWrapper::deformation (id,i,j).

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

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.

defToVtk([*(str)outputFile='def.vtk'*]) → None
Write local deformations in vtk format from states 0 and 1.

defToVtkFromPositions([*(str)input1='pos1'*[, *(str)input2='pos2'*[, *(str)outputFile='def.vtk'*[, *(bool)bz2=False*]]]]) → None
Write local deformations in vtk format from positions files (one sphere per line, with x,y,z,rad separated by spaces).

defToVtkFromStates([*(str)input1='state1'*[, *(str)input2='state2'*[, *(str)outputFile='def.vtk'*[, *(bool)bz2=True*]]]]) → None
Write local deformations in vtk format from state files (since the file format is very special, consider using defToVtkFromPositions if the input files were not generated by TesselationWrapper).

deformation(*(int)id*, *(int)i*, *(int)j*) → float
Get particle deformation

dict() → dict
Return dictionary of attributes.

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`).

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).

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

loadState([*(str)inputFile='state'*[, *(bool)state=0*[, *(bool)bz2=True*]]]) → None
Load a file with positions to define state 0 or 1.

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

saveState([*(str)outputFile='state'*[, *(bool)state=0*[, *(bool)bz2=True*]]]) → None
Save a file with positions, can be later reloaded in order to define state 0 or 1.

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

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

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class `yade.wrapper.TetraVolumetricLaw`(*(object)arg1*)
Calculate physical response of 2 tetrahedra in interaction, based on penetration configuration given by TTetraGeom.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.TimeStepper`(*(object)arg1*)
Engine defining time-step (fundamental class)

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

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.TorqueRecorder`(*(object)arg1*)
Engine saves the total torque according to the given axis and ZeroPoint, the force is taken from bodies, listed in *ids* For instance, can be useful for defining the torque, which affects on ball mill during its work.

`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)

`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.

`dict`() → dict
: Return dictionary of attributes.

`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`).

`file`(=*uninitalized*)
: Name of file to save to; must not be empty.

`firstIterRun`(=*0*)
: Sets the step number, at each an engine should be executed for the first time (disabled by default).

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

`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)

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
: Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
: Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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)

`rotationAxis`(=*Vector3r::UnitX()*)
: Rotation axis

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`totalTorque`(=*0*)
: Resultant torque, returning by the function.

`truncate`(=*false*)
: Whether to delete current file contents, if any, when opening (false by default)

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`virtLast`(=*0*)
: Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
: Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

`zeroPoint`(=*Vector3r::Zero()*)
: Point of rotation center

class `yade.wrapper.TriaxialStateRecorder`(*(object)arg1*)
: 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).

`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)

`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.

`dict`() → dict
: Return dictionary of attributes.

`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`).

`file`(=*uninitalized*)
: Name of file to save to; must not be empty.

`firstIterRun`(=*0*)
: Sets the step number, at each an engine should be executed for the first time (disabled by default).

`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)

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`porosity`(=*1*)
porosity of the packing [-]

`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)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`truncate`(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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.VTKRecorder`(*(object)arg1*)
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.

`Key`(=*""*)
Necessary if recorders contains 'cracks'. A string specifying the name of file 'cracks____.txt' that is considered in this case (see corresponding attribute).

`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].

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.

dict() → dict
Return dictionary of attributes.

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`).

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.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

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)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be exported. If 0, all bodies will be exported.

multiblock(=*false*)
Use multi-block (`.vtm`) files to store data, rather than separate `.vtu` files.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)

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).

`boxes` Save boxes positions (edges).

`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.

`mass` Saves masses (field `mass`) 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.

`coordNumber` Saves coordination number (number of neighbours) 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.

`force` Saves force and torque of spheres, facets and boxes as Vector3 and length (norm); only active if `spheres`, `facets` or `boxes` are activated.

`pericell` Saves the shape of the cell (simulation has to be periodic).

`bstresses` Saves per-particle principal stresses (sigI >= sigII >= sigIII) and associated principal directions (dirI/II/III). Per-particle stress tensors are given by bodyStressTensors (positive values for tensile states).

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), `cpmStress` (stress on particle); `intr` is activated automatically by `cpm`

`wpm` Saves data pertaining to the wire particle model: `wpmForceNFactor` shows the loading factor for the wire, e.g. normal force divided by threshold normal force.

`jcfpm` Saves data pertaining to the rock (smooth)-jointed model: `damage` is defined by JCFpmState.tensBreak + JCFpmState.shearBreak; `intr` is activated automatically by `jcfpm`, and on joint or cohesive interactions can be vizualized.

`cracks` Saves other data pertaining to the rock model: `cracks` shows locations where cohesive bonds failed during the simulation, with their types (0/1 for tensile/shear breakages), their sizes (0.5*(R1+R2)), and their normal directions. The corresponding attribute has to be activated, and Key attributes have to be consistent.

`skipFacetIntr`(=*true*)
Skip interactions that are not of sphere-sphere type (e.g. sphere-facet, sphere-box...), when saving interactions

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

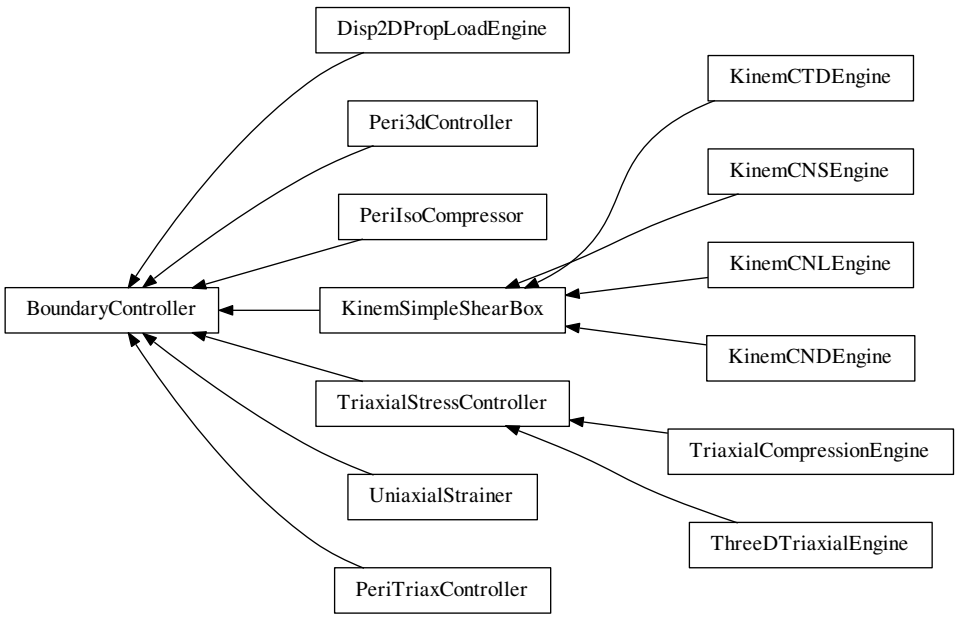
`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`virtLast`(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

1.3.2 BoundaryController



class `yade.wrapper.BoundaryController`(*(object)arg1*)
Base for engines controlling boundary conditions of simulations. Not to be used directly.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Disp2DPropLoadEngine**(*(object)arg1*)
Disturbs a simple shear sample in a given displacement direction

This engine allows one 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

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.

dict() → dict
Return dictionary of attributes.

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).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nbre_iter`(=*0*)
the number of iterations of loading to perform

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`theta`(=*0.0*)
the angle, in a (gamma,h=-u) plane from the gamma - axis to the perturbation vector (trigo wise) [degrees]

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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`(*(object)arg1*)
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 one to perform a constant normal displacement shear, by translating horizontally the upper plate, while the lateral ones rotate so that they always keep contact with the lower and upper walls.

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

`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. Has to be saved, but not to be changed by the user.

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

`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]. Has to be saved, but 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). Has to be saved, but not to be changed by the user.

gamma(=*0.0*)
the current value of the tangential displacement

gamma_save(=*uninitalized*)
vector with the values of gamma at which a save of the simulation is performed [m]

gammalim(=*0.0*)
the value of the tangential displacement at wich the displacement is stopped [m]

id_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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

max_vel(=*1.0*)
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.KinemCNLEngine`(*(object)arg1*)
: 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 one 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.

  `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. Has to be saved, but not to be changed by the user.

  `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.

  `dict`() → dict
  : Return dictionary of attributes.

  `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`).

  `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]. Has to be saved, but 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). Has to be saved, but not to be changed by the user.

  `gamma`(=*0.0*)
  : current value of tangential displacement [m]

gamma_save(=*uninitalized*)
vector with the values of gamma at which a save of the simulation is performed [m]

gammalim(=*0.0*)
the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

id_boxback(=*4*)
the id of the wall at the back of the sample

id_boxbas(=*1*)
the id of the lower wall

id_boxfront(=*5*)
the id of the wall in front of the sample

id_boxleft(=*0*)
the id of the left wall

id_boxright(=*2*)
the id of the right wall

id_topbox(=*3*)
the id of the upper wall

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

max_vel(=*1.0*)
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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]. Has to be saved, but not to be changed by the user.

class yade.wrapper.**KinemCNSEngine**(*(object)arg1*)
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 one 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

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

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

`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. Has to be saved, but not to be changed by the user.

`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.

`dict() → dict`
Return dictionary of attributes.

`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`).

`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]. Has to be saved, but 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). Has to be saved, but not to be changed by the user.

`gamma(=0.0)`
current value of tangential displacement [m]

`gammalim(=0.0)`
the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

`id_boxback(=4)`
the id of the wall at the back of the sample

`id_boxbas(=1)`
the id of the lower wall

`id_boxfront(=5)`
the id of the wall in front of the sample

`id_boxleft`(*=0*)
: the id of the left wall

`id_boxright`(*=2*)
: the id of the right wall

`id_topbox`(*=3*)
: the id of the upper wall

`label`(*=uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel`(*=1.0*)
: to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads`(*=-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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

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

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`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]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.KinemCTDEngine`(*(object)arg1*)
: 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

`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. Has to be saved, but not to be changed by the user.

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

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.

dict() → dict
Return dictionary of attributes.

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).

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]. Has to be saved, but 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). Has to be saved, but 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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

max_vel(=*1.0*)
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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]

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). Has to be saved, but not to be changed by the user.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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]. Has to be saved, but not to be changed by the user.

class yade.wrapper.**KinemSimpleShearBox**(*(object)arg1*)
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. Has to be saved, but not to be changed by the user.

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.

dict() → dict
Return dictionary of attributes.

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).

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]. Has to be saved, but 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). Has to be saved, but 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

`label`(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel`(*=1.0*)
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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). Has to be saved, but not to be changed by the user.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.Peri3dController`(*(object)arg1*)
Class for controlling independently all 6 components of "engineering" stress and strain of periodic Cell. goal are the goal values, while stressMask determines which components prescribe stress and which prescribe strain.

If the strain is prescribed, appropriate 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 $\mathcal{T} = \mathbf{U}\mathbf{P}$, 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 $dt \cdot \dot{\varepsilon}$ is properly rotated to cell's local coordinates and added to $\mathbf{P}$

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

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 examples/test/peri3dController_example1.py and examples/test/peri3dController_triaxialCompression.py explain usage and inputs of Peri3dController, examples/test/peri3dController_shear.py is an example of using shear components and also simulation on rotated cell.

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.

dict() → dict
: Return dictionary of attributes.

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

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`).

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

label(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

lenPe(=*0*)
: Peri3dController internal variable

lenPs(=*0*)
: Peri3dController internal variable

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.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

pathSizes(=*Vector6i::Zero()*)
Peri3dController internal variable

pathsCounter(=*Vector6i::Zero()*)
Peri3dController internal variable

pe(=*Vector6i::Zero()*)
Peri3dController internal variable

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

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

ps(=*Vector6i::Zero()*)
Peri3dController internal variable

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

strainGoal(=*Vector6r::Zero()*)
Peri3dController internal variable

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|.

stressGoal(=*Vector6r::Zero()*)
Peri3dController internal variable

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).

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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**(*(object)arg1*)
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

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.

dict() → dict
Return dictionary of attributes.

doneHook(=*""*)
Python command to be run when reaching the last specified stress

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`).

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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,

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`sigma`
Current stress value

`state`(=*0*)
Where are we at in the stress series

`stresses`(=*uninitalized*)
Stresses that should be reached, one after another

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.PeriTriaxController`(*(object)arg1*)
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)*

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`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`).

`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.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

prevGrow(=*Vector3r::Zero()*)
previous cell grow

relStressTol(=*3e-5*)
Relative stress tolerance

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)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ThreeDTriaxialEngine**(*(object)arg1*)
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.

Warning: This engine is deprecated, please switch to TriaxialStressController if you expect long term support.

`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

`boxVolume`
Total packing volume.

`computeStressStrainInterval(=10)`

`currentStrainRate1(=0)`
current strain rate in direction 1 - converging to ThreeDTriaxialEngine::strainRate1 (./s)

`currentStrainRate2(=0)`
current strain rate in direction 2 - converging to ThreeDTriaxialEngine::strainRate2 (./s)

`currentStrainRate3(=0)`
current strain rate in direction 3 - converging to ThreeDTriaxialEngine::strainRate3 (./s)

`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.

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

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

`dict() → dict`
Return dictionary of attributes.

`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`).

`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)

`frictionAngleDegree(=-1)`
Value of friction used in the simulation if (updateFrictionAngle)

`goal1(=0)`
prescribed stress/strain rate on axis 1, as defined by TriaxialStressController::stressMask

`goal2(=0)`
prescribed stress/strain rate on axis 2, as defined by TriaxialStressController::stressMask

`goal3(=0)`
prescribed stress/strain rate on axis 3, as defined by TriaxialStressController::stressMask

`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.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (initial fast increase - TriaxialStressController::finalMaxMultiplier is used in a second stage)

max_vel(=*1*)
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)*

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

particlesVolume
Total volume of particles (clumps and spheres).

porosity
Porosity of the packing.

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

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

radiusControlInterval(=*10*)

setContactProperties(*(float)arg2*) → None
Assign a new friction angle (degrees) to dynamic bodies and relative interactions

spheresVolume
Shorthand for TriaxialStressController::particlesVolume

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

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

strainDamping(=*0.9997*)
factor used for smoothing changes in effective strain rate. If target rate is TR, then (1-damping)*(TR-currentRate) will be added at each iteration. With damping=0, rate=target all the time. With damping=1, it doesn't change.

strainRate
Current strain rate in a vector d/dt(exx,eyy,ezz).

strainRate1(=*0*)
target strain rate in direction 1 (./s, >0 for compression)

strainRate2(=*0*)
target strain rate in direction 2 (./s, >0 for compression)

strainRate3(=*0*)
target strain rate in direction 3 (./s, >0 for compression)

stress(*(int)id*) → Vector3
Returns the average stress on boundary 'id'. Here, 'id' refers to the internal numbering of boundaries, between 0 and 5.

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

stressDamping(=*0.25*)
wall damping coefficient for the stress control - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

stressMask(=*7*)
Bitmask determining wether the imposed TriaxialStressController::goal's are stresses (0 for none, 7 for all, 1 for direction 1, 5 for directions 1 and 3, etc. :ydefault:'7

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateFrictionAngle(=*false*)
Switch to activate the update of the intergranular frictionto the value ThreeDTriaxialEngine::frictionAngleDegree.

updatePorosity(=*false*)
If true porosity calculation will be updated once (will automatically reset to false after one calculation step). Can be used, when volume of particles changes during the simulation (e.g. when particles are erased or when clumps are created).

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

wall_back_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_back_id(=*4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_bottom_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_bottom_id(=*2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_front_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_front_id(=*5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_left_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_left_id(=*0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_right_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_right_id(=*1*)
id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_top_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_top_id(=*3*)
id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

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

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

class yade.wrapper.**TriaxialCompressionEngine**(*(object)arg1*)
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].

Warning: This engine is deprecated, please switch to TriaxialStressController if you expect long term support.

`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(=false)`
Stop the simulation when the sample reach STATE_LIMBO, or keep running

`autoUnload(=true)`
Auto-switch from isotropic compaction to unloading

`boxVolume`
Total packing volume.

`computeStressStrainInterval(=10)`

`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)

`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.

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

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

`dict() → dict`
Return dictionary of attributes.

`epsilonMax(=0.5)`
Value of axial deformation for which the loading must stop

`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`).

`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)

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

goal1(=*0*)
prescribed stress/strain rate on axis 1, as defined by TriaxialStressController::stressMask

goal2(=*0*)
prescribed stress/strain rate on axis 2, as defined by TriaxialStressController::stressMask

goal3(=*0*)
prescribed stress/strain rate on axis 3, as defined by TriaxialStressController::stressMask

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(=*false*)
if true, sigma_iso is assigned to sigma1, 2 and 3 (applies at each iteration and overrides user-set values of s1,2,3)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (initial fast increase - TriaxialStressController::finalMaxMultiplier is used in a second stage)

maxStress(=*0*)
Max absolute value of axial stress during the simulation (for post-processing)

max_vel(=*1*)
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)*

noFiles(=*false*)
If true, no files will be generated (*.xml, *.spheres,...)

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

particlesVolume
Total volume of particles (clumps and spheres).

porosity
Porosity of the packing.

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

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)

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

radiusControlInterval(*=10*)

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 (< 0 for real - compressive - compaction)

sigmaLateralConfinement(*=1*)
Prescribed confining pressure in the deviatoric loading (< 0 for classical compressive cases); might be different from TriaxialCompressionEngine::sigmaIsoCompaction

sigma_iso(*=0*)
prescribed confining stress (see :yref:TriaxialCompressionEngine::isAxisymetric‘)

spheresVolume
Shorthand for TriaxialStressController::particlesVolume

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

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

strainDamping(*=0.99*)
coefficient used for smoother transitions in the strain rate. The rate reaches the target value like $\mathtt{d}^{\mathtt{n}}$ reaches 0, where $\mathtt{d}$ is the damping coefficient and $\mathtt{n}$ is the number of steps

strainRate(*=0*)
target strain rate (./s, >0 for compression)

stress(*(int)id*) → Vector3
Returns the average stress on boundary ‘id’. Here, ‘id’ refers to the internal numbering of boundaries, between 0 and 5.

stressDamping(*=0.25*)
wall damping coefficient for the stress control - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

stressMask(*=7*)
Bitmask determining wether the imposed TriaxialStressController::goal‘s are stresses (0 for none, 7 for all, 1 for direction 1, 5 for directions 1 and 3, etc. :ydefault:‘7

testEquilibriumInterval(*=20*)
interval of checks for transition between phases, higher than 1 saves computation time.

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

translationAxis(=*TriaxialStressController::normal[wall_bottom]*)
compression axis

uniaxialEpsilonCurr(=*1*)
Current value of axial deformation during confined loading (is reference to strain[1])

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatePorosity(=*false*)
If true porosity calculation will be updated once (will automatically reset to false after one calculation step). Can be used, when volume of particles changes during the simulation (e.g. when particles are erased or when clumps are created).

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

wall_back_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_back_id(=*4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_bottom_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_bottom_id(=*2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_front_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_front_id(=*5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_left_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_left_id(=*0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_right_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_right_id(=*1*)
id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_top_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_top_id(=*3*)
id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

warn(=*0*)
counter used for sending a deprecation warning once

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

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

class `yade.wrapper.TriaxialStressController`(*(object)arg1*)
An engine maintaining constant stresses or constant strain rates on some boundaries of a parallepipedic packing. The stress/strain control is defined for each axis using TriaxialStressController::stressMask (a bitMask) and target values are defined by goal1,goal2, and goal3. The sign conventions of continuum mechanics are used for strains and stresses (positive traction).

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*)

`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.

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

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

`dict`() → dict
Return dictionary of attributes.

`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`).

`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)

`goal1`(=*0*)
prescribed stress/strain rate on axis 1, as defined by TriaxialStressController::stressMask

`goal2`(=*0*)
prescribed stress/strain rate on axis 2, as defined by TriaxialStressController::stressMask

`goal3`(=*0*)
prescribed stress/strain rate on axis 3, as defined by TriaxialStressController::stressMask

`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.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxMultiplier(*=1.001*)
max multiplier of diameters during internal compaction (initial fast increase - TriaxialStressController::finalMaxMultiplier is used in a second stage)

max_vel(*=1*)
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)*

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

particlesVolume
Total volume of particles (clumps and spheres).

porosity
Porosity of the packing.

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

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

radiusControlInterval(*=10*)

spheresVolume
Shorthand for TriaxialStressController::particlesVolume

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

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

strainDamping(*=0.99*)
coefficient used for smoother transitions in the strain rate. The rate reaches the target value like $\mathtt{d}^{\mathtt{n}}$ reaches 0, where $\mathtt{d}$ is the damping coefficient and $\mathtt{n}$ is the number of steps

strainRate
Current strain rate in a vector d/dt(exx,eyy,ezz).

stress(*(int)id*) → Vector3
Returns the average stress on boundary 'id'. Here, 'id' refers to the internal numbering of boundaries, between 0 and 5.

stressDamping(*=0.25*)
wall damping coefficient for the stress control - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

stressMask(=*7*)
Bitmask determining wether the imposed TriaxialStressController::goal's are stresses (0 for none, 7 for all, 1 for direction 1, 5 for directions 1 and 3, etc. :ydefault:'7

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatePorosity(=*false*)
If true porosity calculation will be updated once (will automatically reset to false after one calculation step). Can be used, when volume of particles changes during the simulation (e.g. when particles are erased or when clumps are created).

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

wall_back_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_back_id(=*4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_bottom_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_bottom_id(=*2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_front_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_front_id(=*5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_left_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_left_id(=*0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_right_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_right_id(=*1*)
id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_top_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_top_id(=*3*)
id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

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

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

class `yade.wrapper.UniaxialStrainer`(*(object)arg1*)

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 or 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)*

`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.

`dict`() → dict
: Return dictionary of attributes.

`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`).

`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]

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`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).

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)* [-]

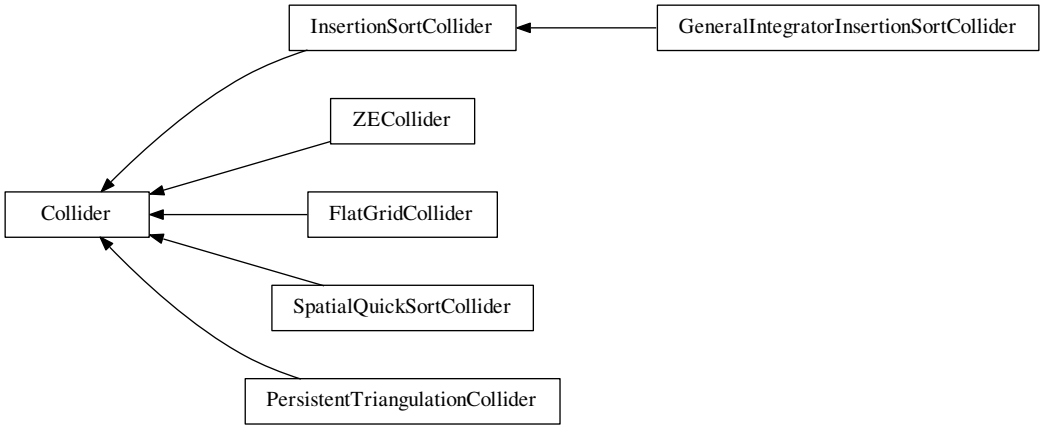
strainRate(=*NaN*)
Rate of strain, starting at 0, linearly raising to strainRate. [-]

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

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.3.3 Collider



class yade.wrapper.**Collider**(*(object)arg1*)
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.

`avoidSelfInteractionMask`
: This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
: BoundDispatcher object that is used for creating bounds on collider's request as necessary.

`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.

`dict`() → dict
: Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.FlatGridCollider`(*(object)arg1*)

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: objects Body::bound 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.

`avoidSelfInteractionMask`
: This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
: BoundDispatcher object that is used for creating bounds on collider's request as necessary.

`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.

`dict`() → dict
: Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`step`(=*0*)
: Step in the grid (cell size)

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`verletDist`(=*0*)
: Length by which enlarge space occupied by each particle; avoids running collision detection at every step.

class `yade.wrapper.GeneralIntegratorInsertionSortCollider`(*(object)arg1*)
This class is the adaptive version of the InsertionSortCollider and changes the NewtonIntegrator dependency of the collider algorithms to the Integrator interface which is more general.

`allowBiggerThanPeriod`
: If true, tests on bodies sizes will be disabled, and the simulation will run normaly even if bodies larger than period are found. It can be useful when the periodic problem include e.g. a floor modelized with wall/box/facet. Be sure you know what you are doing if you touch this flag. The result is undefined if one large body moves out of the (0,0,0) period.

`avoidSelfInteractionMask`
: This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
: BoundDispatcher object that is used for creating bounds on collider's request as necessary.

`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.

`dict()` → dict
Return dictionary of attributes.

`doSort(=false)`
Do forced resorting of interactions.

`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.

`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`).

`fastestBodyMaxDist(=-1)`
Normalized maximum displacement of the fastest body since last run; if >= 1, we could get out of bboxes and will trigger full run. *(auto-updated)*

`label(=uninitalized)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`minSweepDistFactor(=0.1)`
Minimal distance by which enlarge all bounding boxes; superseeds computed value of verletDist when lower that (minSweepDistFactor x verletDist).

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

`ompThreads(=-1)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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)*

`targetInterv(=50)`
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`updatingDispFactor`(*=-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

`useless`(*=uninitalized*)
for compatibility of scripts defining the old collider's attributes - see deprecated attributes

`verletDist`(*=-.5*, *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. The actual length added to one bound can be only a fraction of verletDist when InsertionSortCollider::targetInterv is > 0.

class `yade.wrapper.InsertionSortCollider`(*(object)arg1*)
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 this (sortAxis) axis, 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 if it does and gets in interaction. One way to explicitly by-pass this restriction is offered by `allowBiggerThanPeriod`, which can be turned on to insert a floor in the form of a very large box for instance (see examples/periodicSandPile.py).
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 displacements 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 `targetInterv` is >1, not all particles will have their bound enlarged by `verletDist`; instead, they will have bounds increased by a length in order to trigger a new colliding after `targetInterv` iteration, assuming they move at almost constant velocity. Ideally in this method, all particles would reach their bounds at the sime iteration. This is of course not the case as soon as velocities fluctuate in time. Bound::sweepLength is tuned on the basis of the displacement recorded between the last two runs of the collider. In this situation, `verletDist` defines the maximum sweep length.

`allowBiggerThanPeriod`
If true, tests on bodies sizes will be disabled, and the simulation will run normaly even if bodies larger than period are found. It can be useful when the periodic problem include e.g. a floor modelized with wall/box/facet. Be sure you know what you are doing if you touch this flag. The result is undefined if one large body moves out of the (0,0,0) period.

avoidSelfInteractionMask
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

boundDispatcher(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating bounds on collider's request as necessary.

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.

dict() → dict
Return dictionary of attributes.

doSort(=*false*)
Do forced resorting of interactions.

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.

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`).

fastestBodyMaxDist(=*-1*)
Normalized maximum displacement of the fastest body since last run; if >= 1, we could get out of bboxes and will trigger full run. *(auto-updated)*

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

minSweepDistFactor(=*0.1*)
Minimal distance by which enlarge all bounding boxes; superseeds computed value of verletDist when lower that (minSweepDistFactor x verletDist).

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)*

`targetInterv`(=*50*)
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`updatingDispFactor`(=*-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

`useless`(=*uninitalized*)
for compatibility of scripts defining the old collider's attributes - see deprecated attributes

`verletDist`(=*-.5*, *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. The actual length added to one bound can be only a fraction of verletDist when InsertionSortCollider::targetInterv is > 0.

class `yade.wrapper.PersistentTriangulationCollider`(*(object)arg1*)
Collision detection engine based on regular triangulation. Handles spheres and flat boundaries (considered as infinite-sized bounding spheres).

`avoidSelfInteractionMask`
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating bounds on collider's request as necessary.

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

`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.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.SpatialQuickSortCollider`(*(object)arg1*)
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.

`avoidSelfInteractionMask`
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating bounds on collider's request as necessary.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.ZECollider`(*(object)arg1*)
Collider with O(n log(n)) complexity, using a CGAL algorithm from Zomorodian and Edelsbrunner [Kettner2011] (http://www.cgal.org/Manual/beta/doc_html/cgal_manual/Box_intersection_d/Chapter_main.html)

`avoidSelfInteractionMask`
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating bounds on collider's request as necessary.

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.

dict() → dict
Return dictionary of attributes.

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`).

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)*

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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)*

targetInterv(=*30*)
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

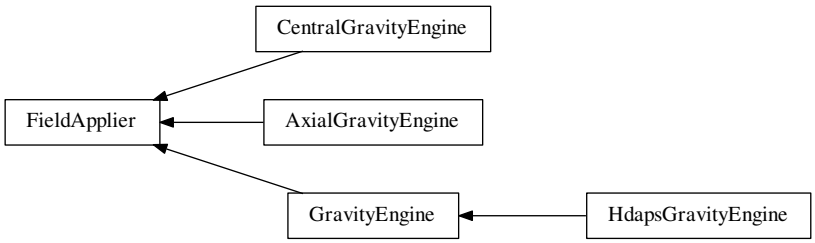
updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatingDispFactor(=*-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

verletDist(=*-.15*, *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.

1.3.4 FieldApplier



class `yade.wrapper.FieldApplier`(*(object)arg1*)
Base for engines applying force files on particles. Not to be used directly.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.AxialGravityEngine`(*(object)arg1*)
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.

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.

dict() → dict
Return dictionary of attributes.

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.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CentralGravityEngine**(*(object)arg1*)
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.

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.

dict() → dict
Return dictionary of attributes.

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.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can

depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

reciprocal(=*false*)
If true, acceleration will be applied on the central body as well.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GravityEngine**(*(object)arg1*)
Engine applying constant acceleration to all bodies. DEPRECATED, use Newton::gravity unless you need energy tracking or selective gravity application using groupMask).

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.

dict() → dict
Return dictionary of attributes.

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`).

gravity(=*Vector3r::Zero()*)
Acceleration [kgms−2]

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

warnOnce(=*true*)
For deprecation warning once.

class yade.wrapper.**HdapsGravityEngine**(*(object)arg1*)
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*.

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.

dict() → dict
Return dictionary of attributes.

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`).

gravity(=*Vector3r::Zero()*)
Acceleration [kgms−²]

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

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

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

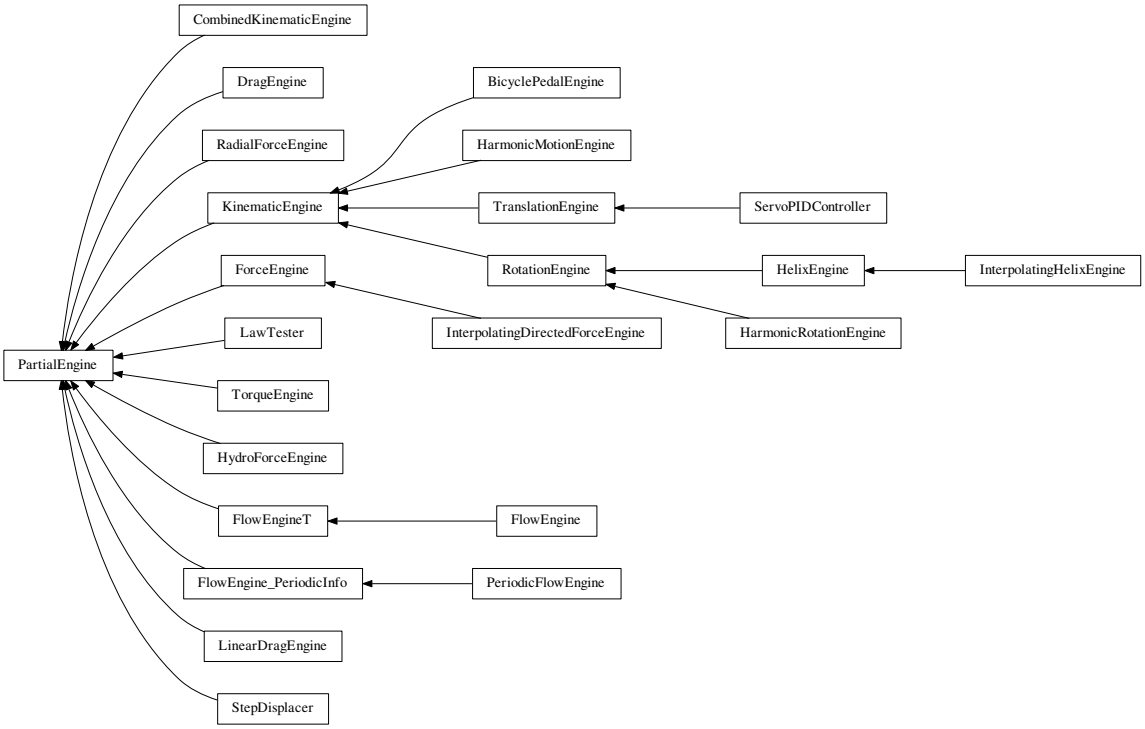
updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

warnOnce(=*true*)
For deprecation warning once.

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

1.4 Partial engines



class `yade.wrapper.PartialEngine`(*(object)arg1*)

Engine affecting only particular bodies in the simulation, defined by *ids*.

`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.

`dict()` → dict
: Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.BicyclePedalEngine`(*(object)arg1*)
Engine applying the linear motion of `bicycle pedal` e.g. moving points around the axis without rotation

`angularVelocity`(*=0*)
Angular velocity. [rad/s]

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

`fi`(*=Mathr::PI/2.0*)
Initial phase [radians]

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

`label`(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`radius`(*=-1.0*)
Rotation radius. [m]

`rotationAxis`(*=Vector3r::UnitX()*)
Axis of rotation (direction); will be normalized automatically.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.CombinedKinematicEngine`(*(object)arg1*)
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.

`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.

`dict`() → dict
Return dictionary of attributes.

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).

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**DragEngine**(*(object)arg1*)
Apply drag force on some particles at each step, decelerating them proportionally to their linear velocities. The applied force reads

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

where ρ is the medium density (density), $\boldsymbol{v}$ 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, listed in ids.

Cd(=*0.47*)
Drag coefficient <http://en.wikipedia.org/wiki/Drag_coefficient>'_.

Rho(=*1.225*)
Density of the medium (fluid or air), by default - the density of the air.

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.

dict() → dict
Return dictionary of attributes.

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).

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**FlowEngine**(*(object)arg1*)
An engine to solve flow problem in saturated granular media. Model description can be found in [Chareyre2012a] and [Catalano2014a]. See the example script FluidCouplingPFV/oedometer.py. More documentation to come.

OSI() → float
Return the number of interactions only between spheres.

avFlVelOnSph(*(int)idSph*) → object
compute a sphere-centered average fluid velocity

averagePressure() → float
Measure averaged pore pressure in the entire volume

averageSlicePressure(*(float)posY*) → float
Measure slice-averaged pore pressure at height posY

averageVelocity() → Vector3
measure the mean velocity in the period

blockCell(*(int)id*, *(bool)blockPressure*) → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(=*""*)
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also TemplateFlowEngine_FlowEngineT.blockCell), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6*, *false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with FlowEngine::xmin and friends.

bndCondValue(=*vector<double>(6*, *0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also FlowEngineT::boundaryXPos

boundaryUseMaxMin(=*vector<bool>(6*, *true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using FlowEngine::setBoundaryVel

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also FlowEngineT::boundaryPressure

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

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

compTessVolumes() → None
Like TesselationWrapper::computeVolumes()

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.

debug(=*false*)
Activate debug messages

defTolerance(=*0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of FlowEngine::meshUpdateInterval

dict() → dict
Return dictionary of attributes.

doInterpolate(=*false*)
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

dt(=*0*)
timestep [s]

edgeSize() → float
Return the number of interactions.

emulateAction() → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

eps(=*0.00001*)
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

epsVolMax(=*0*)
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

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`).

exportMatrix(*[(str)filename='matrix']*) → None
Export system matrix to a file with all entries (even zeros will displayed).

`exportTriplets`(*[(str)filename='triplets']*) → None
Export system matrix to a file with only non-zero entries.

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

`fluidBulkModulus`(=*0.*)
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

`fluidForce`(*(int)idSph*) → Vector3
Return the fluid force on sphere idSph.

`forceMetis`
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

`getBoundaryFlux`(*(int)boundary*) → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

`getCell`(*(float)arg2*, *(float)arg3*, *(float)pos*) → int
get id of the cell containing (X,Y,Z).

`getCellBarycenter`(*(int)id*) → Vector3
get barycenter of cell 'id'.

`getCellCenter`(*(int)id*) → Vector3
get voronoi center of cell 'id'.

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

`getCellPImposed`(*(int)id*) → bool
get the status of cell 'id' wrt imposed pressure.

`getCellPressure`(*(int)id*) → float
get pressure in cell 'id'.

`getConstrictions`(*[(bool)all=True]*) → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

`getConstrictionsFull`(*[(bool)all=True]*) → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

`getPorePressure`(*(Vector3)pos*) → float
Measure pore pressure in position pos[0],pos[1],pos[2]

`getVertices`(*(int)id*) → list
get the vertices of a cell

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

`ignoredBody`(=*-1*)
Id of a sphere to exclude from the triangulation.)

imposeFlux(*(Vector3)pos*, *(float)p*) → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

imposePressure(*(Vector3)pos*, *(float)p*) → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

imposePressureFromId(*(int)id*, *(float)p*) → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

isActivated(=*true*)
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see FlowEngine::clampKValues)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(=*1000*)
Maximum number of timesteps between re-triangulation events. See also FlowEngine::defTolerance.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see FlowEngine::clampKValues)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(=*1*)
number of openblas threads in the factorization phase

numSolveThreads(=*1*)
number of openblas threads in the solve phase.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(*=0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(*=1.0*)
permability multiplier

permeabilityMap(*=false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(*=0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(*=true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(*=false*)
Compute pump torque applied on particles

relax(*=1.9*)
Gauss-Seidel relaxation

saveVtk(*[(str)folder='./VTK']*) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(*=false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(*=0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(*=0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

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

stiffness(*=10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

tolerance(=*1e-06*)
Gauss-Seidel tolerance

twistTorque(=*false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(=*0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on FlowEngine::defTolerance and FlowEngine::meshUpdateInterval. Of course, it costs CPU time.

useSolver(=*0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(=*1.0*)
viscosity of the fluid

viscousNormalBodyStress(=*false*)
compute normal viscous stress applied on each body

viscousShear(=*false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

viscousShearBodyStress(=*false*)
compute shear viscous stress applied on each body

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

wallIds(=*vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

wallThickness(=*0*)
Walls thickness

waveAction(=*false*)
Allow sinusoidal pressure condition to simulate ocean waves

xmax(=*1*)
See FlowEngine::xmin.

xmin(=*0*)
Index of the boundary $\mathtt{x}_{\mathtt{min}}$. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

ymax(=*3*)
See FlowEngine::xmin.

ymin(=*2*)
See FlowEngine::xmin.

zmax(=*5*)
See FlowEngine::xmin.

zmin(=*4*)
See FlowEngine::xmin.

class yade.wrapper.**FlowEngineT**(*(object)arg1*)
A generic engine from wich more specialized engines can inherit. It is defined for the sole purpose of inserting the right data classes CellInfo and VertexInfo in the triangulation, and it should not

be used directly. Instead, look for specialized engines, e.g. FlowEngine, PeriodicFlowEngine, or DFNFlowEngine.

`OSI()` → float
Return the number of interactions only between spheres.

`avFlVelOnSph(`*(int)idSph*`)` → object
compute a sphere-centered average fluid velocity

`averagePressure()` → float
Measure averaged pore pressure in the entire volume

`averageSlicePressure(`*(float)posY*`)` → float
Measure slice-averaged pore pressure at height posY

`averageVelocity()` → Vector3
measure the mean velocity in the period

`blockCell(`*(int)id*, *(bool)blockPressure*`)` → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

`blockHook(`*=""*`)`
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also TemplateFlowEngine_FlowEngineT.blockCell), or apply exotic types of boundary conditions which need to visit the newly built mesh

`bndCondIsPressure(`*=vector<bool>(6, false)*`)`
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with FlowEngine::xmin and friends.

`bndCondValue(`*=vector<double>(6, 0)*`)`
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

`bodyNormalLubStress(`*(int)idSph*`)` → Matrix3
Return the normal lubrication stress on sphere idSph.

`bodyShearLubStress(`*(int)idSph*`)` → Matrix3
Return the shear lubrication stress on sphere idSph.

`boundaryPressure(`*=vector<Real>()*`)`
values defining pressure along x-axis for the top surface. See also FlowEngineT::boundaryXPos

`boundaryUseMaxMin(`*=vector<bool>(6, true)*`)`
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

`boundaryVelocity(`*=vector<Vector3r>(6, Vector3r::Zero())*`)`
velocity on top boundary, only change it using FlowEngine::setBoundaryVel

`boundaryXPos(`*=vector<Real>()*`)`
values of the x-coordinate for which pressure is defined. See also FlowEngineT::boundaryPressure

`cholmodStats()` → None
get statistics of cholmod solver activity

`clampKValues(`*=true*`)`
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

`clearImposedFlux()` → None
Clear the list of points with flux imposed.

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

`compTessVolumes()` → None
Like TesselationWrapper::computeVolumes()

`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.

`debug(=false)`
Activate debug messages

`defTolerance(=0.05)`
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of FlowEngine::meshUpdateInterval

`dict()` → dict
Return dictionary of attributes.

`doInterpolate(=false)`
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

`dt(=0)`
timestep [s]

`edgeSize()` → float
Return the number of interactions.

`emulateAction()` → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

`eps(=0.00001)`
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

`epsVolMax(=0)`
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

`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`).

`exportMatrix([(str)filename='matrix'])` → None
Export system matrix to a file with all entries (even zeros will displayed).

`exportTriplets([(str)filename='triplets'])` → None
Export system matrix to a file with only non-zero entries.

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

`fluidBulkModulus(=0.)`
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

`fluidForce((int)idSph)` → Vector3
Return the fluid force on sphere idSph.

`forceMetis`
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

`getBoundaryFlux(`*(int)boundary*`)` → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

`getCell(`*(float)arg2*, *(float)arg3*, *(float)pos*`)` → int
get id of the cell containing (X,Y,Z).

`getCellBarycenter(`*(int)id*`)` → Vector3
get barycenter of cell 'id'.

`getCellCenter(`*(int)id*`)` → Vector3
get voronoi center of cell 'id'.

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

`getCellPImposed(`*(int)id*`)` → bool
get the status of cell 'id' wrt imposed pressure.

`getCellPressure(`*(int)id*`)` → float
get pressure in cell 'id'.

`getConstrictions(`[*(bool)all=True*]`)` → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

`getConstrictionsFull(`[*(bool)all=True*]`)` → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

`getPorePressure(`*(Vector3)pos*`)` → float
Measure pore pressure in position pos[0],pos[1],pos[2]

`getVertices(`*(int)id*`)` → list
get the vertices of a cell

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

`ignoredBody(`=*-1*`)`
Id of a sphere to exclude from the triangulation.)

`imposeFlux(`*(Vector3)pos*, *(float)p*`)` → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

`imposePressure(`*(Vector3)pos*, *(float)p*`)` → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

`imposePressureFromId(`*(int)id*, *(float)p*`)` → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

`isActivated(`=*true*`)`
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see FlowEngine::clampKValues)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(=*1000*)
Maximum number of timesteps between re-triangulation events. See also FlowEngine::defTolerance.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see FlowEngine::clampKValues)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(=*1*)
number of openblas threads in the factorization phase

numSolveThreads(=*1*)
number of openblas threads in the solve phase.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(=*0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(=*1.0*)
permability multiplier

permeabilityMap(=*false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(=*0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(*=true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(*=false*)
Compute pump torque applied on particles

relax(*=1.9*)
Gauss-Seidel relaxation

saveVtk([*(str)folder='./VTK'*]) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(*=false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(*=0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(*=0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

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

stiffness(*=10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

tolerance(*=1e-06*)
Gauss-Seidel tolerance

twistTorque(*=false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(*=0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on FlowEngine::defTolerance and FlowEngine::meshUpdateInterval. Of course, it costs CPU time.

useSolver(*=0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(*=1.0*)
viscosity of the fluid

viscousNormalBodyStress(*=false*)
compute normal viscous stress applied on each body

viscousShear(*=false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

viscousShearBodyStress(*=false*)
compute shear viscous stress applied on each body

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

wallIds(*=vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

wallThickness(*=0*)
Walls thickness

waveAction(*=false*)
Allow sinusoidal pressure condition to simulate ocean waves

xmax(*=1*)
See FlowEngine::xmin.

xmin(*=0*)
Index of the boundary x_{min}. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

ymax(*=3*)
See FlowEngine::xmin.

ymin(*=2*)
See FlowEngine::xmin.

zmax(*=5*)
See FlowEngine::xmin.

zmin(*=4*)
See FlowEngine::xmin.

class yade.wrapper.**FlowEngine_PeriodicInfo**(*(object)arg1*)
A generic engine from wich more specialized engines can inherit. It is defined for the sole purpose of inserting the right data classes CellInfo and VertexInfo in the triangulation, and it should not be used directly. Instead, look for specialized engines, e.g. FlowEngine, PeriodicFlowEngine, or DFNFlowEngine.

OSI() → float
Return the number of interactions only between spheres.

avFlVelOnSph(*(int)idSph*) → object
compute a sphere-centered average fluid velocity

averagePressure() → float
Measure averaged pore pressure in the entire volume

averageSlicePressure(*(float)posY*) → float
Measure slice-averaged pore pressure at height posY

averageVelocity() → Vector3
measure the mean velocity in the period

blockCell(*(int)id*, *(bool)blockPressure*) → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(="")
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also TemplateFlowEngine_FlowEngine_PeriodicInfo.blockCell), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6, false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with FlowEngine::xmin and friends.

bndCondValue(=*vector<double>(6, 0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also FlowEngine_PeriodicInfo::boundaryXPos

boundaryUseMaxMin(=*vector<bool>(6, true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using FlowEngine::setBoundaryVel

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also FlowEngine_PeriodicInfo::boundaryPressure

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

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

compTessVolumes() → None
Like TesselationWrapper::computeVolumes()

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.

debug(=*false*)
Activate debug messages

defTolerance(*=0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of FlowEngine::meshUpdateInterval

dict() → dict
Return dictionary of attributes.

doInterpolate(*=false*)
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

dt(*=0*)
timestep [s]

edgeSize() → float
Return the number of interactions.

emulateAction() → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

eps(*=0.00001*)
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

epsVolMax(*=0*)
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

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`).

exportMatrix([*(str)filename='matrix'*]) → None
Export system matrix to a file with all entries (even zeros will displayed).

exportTriplets([*(str)filename='triplets'*]) → None
Export system matrix to a file with only non-zero entries.

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

fluidBulkModulus(*=0.*)
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

fluidForce(*(int)idSph*) → Vector3
Return the fluid force on sphere idSph.

forceMetis
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

getBoundaryFlux(*(int)boundary*) → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

---

getCell(*(float)arg2*, *(float)arg3*, *(float)pos*) → int
get id of the cell containing (X,Y,Z).

getCellBarycenter(*(int)id*) → Vector3
get barycenter of cell 'id'.

getCellCenter(*(int)id*) → Vector3
get voronoi center of cell 'id'.

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

getCellPImposed(*(int)id*) → bool
get the status of cell 'id' wrt imposed pressure.

getCellPressure(*(int)id*) → float
get pressure in cell 'id'.

getConstrictions([*(bool)all=True*]) → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

getConstrictionsFull([*(bool)all=True*]) → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

getPorePressure(*(Vector3)pos*) → float
Measure pore pressure in position pos[0],pos[1],pos[2]

getVertices(*(int)id*) → list
get the vertices of a cell

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

ignoredBody(*=-1*)
Id of a sphere to exclude from the triangulation.)

imposeFlux(*(Vector3)pos*, *(float)p*) → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

imposePressure(*(Vector3)pos*, *(float)p*) → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

imposePressureFromId(*(int)id*, *(float)p*) → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

isActivated(*=true*)
Activates Flow Engine

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(*=100*)
define the max K value (see FlowEngine::clampKValues)

meanKStat(*=false*)
report the local permeabilities' correction

meshUpdateInterval(*=1000*)
Maximum number of timesteps between re-triangulation events. See also FlowEngine::defTolerance.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see FlowEngine::clampKValues)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(=*1*)
number of openblas threads in the factorization phase

numSolveThreads(=*1*)
number of openblas threads in the solve phase.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(=*0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(=*1.0*)
permability multiplier

permeabilityMap(=*false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(=*0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(=*true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(=*false*)
Compute pump torque applied on particles

relax(=*1.9*)
Gauss-Seidel relaxation

saveVtk(*[(str)folder='./VTK']*) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(=*false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(=*0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(=*0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

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

stiffness(=*10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

tolerance(=*1e-06*)
Gauss-Seidel tolerance

twistTorque(=*false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(=*0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on FlowEngine::defTolerance and FlowEngine::meshUpdateInterval. Of course, it costs CPU time.

useSolver(=*0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(=*1.0*)
viscosity of the fluid

viscousNormalBodyStress(=*false*)
compute normal viscous stress applied on each body

viscousShear(=*false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

`viscousShearBodyStress`(=*false*)
compute shear viscous stress applied on each body

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

`wallIds`(=*vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

`wallThickness`(=*0*)
Walls thickness

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

`xmax`(=*1*)
See FlowEngine::xmin.

`xmin`(=*0*)
Index of the boundary x_{min}. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

`ymax`(=*3*)
See FlowEngine::xmin.

`ymin`(=*2*)
See FlowEngine::xmin.

`zmax`(=*5*)
See FlowEngine::xmin.

`zmin`(=*4*)
See FlowEngine::xmin.

class `yade.wrapper.ForceEngine`(*(object)arg1*)
Apply contact force on some particles at each step.

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

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

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**HarmonicMotionEngine**(*(object)arg1*)
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]

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.

dict() → dict
Return dictionary of attributes.

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).

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.

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**HarmonicRotationEngine**(*(object)arg1*)
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]

angularVelocity(=*0*)
Angular velocity. [rad/s]

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

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

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

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.HelixEngine`(*(object)arg1*)
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]

`angularVelocity`(=*0*)
Angular velocity. [rad/s]

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.HydroForceEngine`(*(object)arg1*)

Apply drag and lift due to a fluid flow vector (1D) to each sphere + the buoyant weight.
The applied drag force reads

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

where ρ is the medium density (densFluid), v is particle's velocity, v_f is the velocity of the fluid at the particle center(vxFluid), A is particle projected area (disc), C_d is the drag coefficient. The formulation of the drag coefficient depends on the local particle reynolds number and the solid volume fraction. The formulation of the drag is [Dallavalle1948] [RevilBaudard2013] with a correction of Richardson-Zaki [Richardson1954] to take into account the hindrance effect. This law is classical in sediment transport. It is possible to activate a fluctuation of the drag force for each particle which account for the turbulent fluctuation of the fluid velocity (velFluct). The model implemented for the turbulent velocity fluctuation is a simple discrete random walk which takes as input the Reynolds stress tensor R^f_{xz} as a function of the depth, and allows to recover the main property of the fluctuations by imposing $< u'_x u'_z > (z) =< R^f_{xz} > (z)/\rho^f$. It requires as input $< R^f_{xz} > (z)/\rho^f$ called simplifiedReynoldStresses in the code. The formulation of the lift is taken from [Wiberg1985] and is such that :

$$F_L = \frac{1}{2} C_L A \rho^f ((v_f - v)^2_{top} - (v_f - v)^2_{bottom})$$

Where the subscript top and bottom means evaluated at the top (respectively the bottom) of the sphere considered. This formulation of the lift account for the difference of pressure at the top and the bottom of the particle inside a turbulent shear flow. As this formulation is controversial when approaching the threshold of motion [Schmeeckle2007] it is possible to desactivate it with the variable lift. The buoyancy is taken into account through the buoyant weight :

$$F_{buoyancy} = -\rho^f V^p g$$

, where g is the gravity vector along the vertical, and V^p is the volume of the particle. This engine also evaluate the average particle velocity, solid volume fraction and drag force depth profiles, through the function averageProfile. This is done as the solid volume fraction depth profile is required for the drag calculation, and as the three are required for the independent fluid resolution.

`Cl`(=*0.2*)
Value of the lift coefficient taken from [Wiberg1985]

`activateAverage`(=*false*)
If true, activate the calculation of the average depth profiles of drag, solid volume fraction, and solid velocity for the application of the force (phiPart in hindrance function) and to use in python for the coupling with the fluid.

`averageDrag`(=*uninitalized*)
Discretized drag depth profile. No role in the engine, output parameter. For practical reason, it can be evaluated directly inside the engine, calling from python the averageProfile() method of the engine, or puting activateAverage to True.

`averageProfile`() → None
Compute and store the particle velocity (vxPart, vyPart, vzPart) and solid volume fraction (phiPart) depth profile. For each defined cell z, the k component of the average particle velocity reads:

$$< v_k >^z = \sum_p V^p v_k^p / \sum_p V^p,$$

where the sum is made over the particles contained in the cell, v_k^p is the k component of the velocity associated to particle p, and V^p is the part of the volume of the particle p contained inside the cell. This definition allows to smooth the averaging, and is equivalent to taking into account the center of the particles only when there is a lot of particles in each cell. As for the solid volume fraction, it is evaluated in the same way: for each defined cell z, it reads:

$< \varphi >^z = \frac{1}{V_{cell}} \sum_p V^p$, where V_{cell} is the volume of the cell considered, and V^p is the volume of particle p contained in cell z. This function gives depth profiles of average velocity and solid volume fraction, returning the average quantities in each cell of height dz, from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef<HydroForceEngine.zRef>‘+:yref:‘nCell<HydroForceEngine.zRef>‘x :yref:‘deltaZ (input parameters).

`bedElevation`(=*uninitalized*)
Elevation of the bed above which the fluid flow is turbulent and the particles undergo turbulent velocity fluctuation.

`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.

`deltaZ`(=*uninitalized*)
Height of the discretization cell.

`densFluid`(=*1000*)
Density of the fluid, by default - density of water

`dict`() → dict
Return dictionary of attributes.

`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`).

`expoRZ`(=*3.1*)
Value of the Richardson-Zaki exponent, for the drag correction due to hindrance

gravity(=*Vector3r(0, 0, -9.81)*)
Gravity vector (may depend on the slope).

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

lift(=*false*)
Option to activate or not the evaluation of the lift

nCell(=*uninitalized*)
Number of cell in the depth

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

phiPart(=*uninitalized*)
Discretized solid volume fraction depth profile. Can be taken as input parameter, or evaluated directly inside the engine, calling from python the averageProfile() function, or puting activateAverage to True.

simplifiedReynoldStresses(=*uninitalized*)
Vector of size equal to nCell containing the Reynolds stresses divided by the fluid density in function of the depth. simplifiedReynoldStresses(z) $=< u'_x u'_z > (z)^2$

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vCell(=*uninitalized*)
Volume of averaging cell

vFluctX(=*uninitalized*)
Vector associating a streamwise fluid velocity fluctuation to each particle. Fluctuation calculated in the C++ code from the discrete random walk model

vFluctZ(=*uninitalized*)
Vector associating a normal fluid velocity fluctuation to each particle. Fluctuation calculated in the C++ code from the discrete random walk model

velFluct(=*false*)
If true, activate the determination of turbulent fluid velocity fluctuation for the next time step only at the position of each particle, using a simple discrete random walk (DRW) model based on the Reynolds stresses profile (simplifiedReynoldStresses)

viscoDyn(=*1e-3*)
Dynamic viscosity of the fluid, by default - viscosity of water

vxFluid(=*uninitalized*)
Discretized streamwise fluid velocity depth profile

vxPart(=*uninitalized*)
Discretized streamwise solid velocity depth profile. Can be taken as input parameter, or evaluated directly inside the engine, calling from python the averageProfile() function, or puting activateAverage to True.

vyPart(=*uninitalized*)
Discretized spanwise solid velocity depth profile. No role in the engine, output parameter. For practical reason, it can be evaluated directly inside the engine, calling from python the averageProfile() method of the engine, or puting activateAverage to True.

vzPart(=*uninitalized*)
Discretized normal solid velocity depth profile. No role in the engine, output parameter. For practical reason, it can be evaluated directly inside the engine, calling from python the averageProfile() method of the engine, or puting activateAverage to True.

zRef(=*uninitalized*)
Position of the reference point which correspond to the first value of the fluid velocity, i.e. to the ground.

class yade.wrapper.**InterpolatingDirectedForceEngine**(*(object)arg1*)
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.

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.

dict() → dict
Return dictionary of attributes.

direction(=*Vector3r::UnitX()*)
Contact force direction (normalized automatically)

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).

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

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

magnitudes(=*uninitalized*)
Force magnitudes readings [N]

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

times(=*uninitalized*)
Time readings [s]

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wrap`(=*false*)
wrap to the beginning of the sequence if beyond the last time point

class `yade.wrapper.InterpolatingHelixEngine`(*(object)arg1*)
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).

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

`angularVelocities`(=*uninitalized*)
List of angular velocities; manadatorily of same length as times. [rad/s]

`angularVelocity`(=*0*)
Angular velocity. [rad/s]

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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.

`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]

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wrap`(=*false*)
Wrap t if t>times_n, i.e. t_wrapped=t-N*(times_n-times_0)

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.KinematicEngine`(*(object)arg1*)
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.

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.LawTester`(*(object)arg1*)
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

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.

`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.

`dict`() → dict
: Return dictionary of attributes.

`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.

`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`).

`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.

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class yade.wrapper.**LinearDragEngine**(*(object)arg1*)
Apply viscous resistance or linear drag on some particles at each step, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -b\boldsymbol{v}$$

where b is the linear drag, $\boldsymbol{v}$ is particle's velocity.

$$b = 6\pi\nu r$$

where ν is the medium viscosity, r is the Stokes radius of the particle (but in this case we accept it equal to sphere radius for simplification),

Note: linear drag is only applied to spherical particles, listed in ids.

`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.

`dict`() → dict
: Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nu`(=*0.001*)
: Viscosity of the medium.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.PeriodicFlowEngine`(*(object)arg1*)
A variant of FlowEngine implementing periodic boundary conditions. The API is very similar.

`OSI`() → float
: Return the number of interactions only between spheres.

`avFlVelOnSph`(*(int)idSph*) → object
: compute a sphere-centered average fluid velocity

`averagePressure`() → float
: Measure averaged pore pressure in the entire volume

`averageSlicePressure`(*(float)posY*) → float
: Measure slice-averaged pore pressure at height posY

`averageVelocity`() → Vector3
: measure the mean velocity in the period

`blockCell`(*(int)id*, *(bool)blockPressure*) → None
: block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all

incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(=”“)
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also TemplateFlowEngine_FlowEngine_PeriodicInfo.blockCell), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6, false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with FlowEngine::xmin and friends.

bndCondValue(=*vector<double>(6, 0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also FlowEngine_PeriodicInfo::boundaryXPos

boundaryUseMaxMin(=*vector<bool>(6, true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using FlowEngine::setBoundaryVel

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also FlowEngine_PeriodicInfo::boundaryPressure

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

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

compTessVolumes() → None
Like TesselationWrapper::computeVolumes()

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.

debug(=*false*)
Activate debug messages

defTolerance(=*0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of FlowEngine::meshUpdateInterval

`dict()` → dict
Return dictionary of attributes.

`doInterpolate(`=*false*`)`
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

`dt(`=*0*`)`
timestep [s]

`duplicateThreshold(`=*0.06*`)`
distance from cell borders that will triger periodic duplication in the triangulation *(auto-updated)*

`edgeSize()` → float
Return the number of interactions.

`emulateAction()` → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

`eps(`=*0.00001*`)`
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

`epsVolMax(`=*0*`)`
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

`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`).

`exportMatrix(`[*(str)filename='matrix'*]`)` → None
Export system matrix to a file with all entries (even zeros will displayed).

`exportTriplets(`[*(str)filename='triplets'*]`)` → None
Export system matrix to a file with only non-zero entries.

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

`fluidBulkModulus(`=*0.*`)`
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

`fluidForce(`*(int)idSph*`)` → Vector3
Return the fluid force on sphere idSph.

`forceMetis`
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

`getBoundaryFlux(`*(int)boundary*`)` → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

`getCell(`*(float)arg2*, *(float)arg3*, *(float)pos*`)` → int
get id of the cell containing (X,Y,Z).

`getCellBarycenter(`*(int)id*`)` → Vector3
get barycenter of cell 'id'.

getCellCenter(*(int)id*) → Vector3
get voronoi center of cell 'id'.

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

getCellPImposed(*(int)id*) → bool
get the status of cell 'id' wrt imposed pressure.

getCellPressure(*(int)id*) → float
get pressure in cell 'id'.

getConstrictions([*(bool)all=True*]) → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

getConstrictionsFull([*(bool)all=True*]) → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

getPorePressure(*(Vector3)pos*) → float
Measure pore pressure in position pos[0],pos[1],pos[2]

getVertices(*(int)id*) → list
get the vertices of a cell

gradP(=*Vector3r::Zero()*)
Macroscopic pressure gradient

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

ignoredBody(=*-1*)
Id of a sphere to exclude from the triangulation.)

imposeFlux(*(Vector3)pos*, *(float)p*) → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

imposePressure(*(Vector3)pos*, *(float)p*) → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

imposePressureFromId(*(int)id*, *(float)p*) → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

isActivated(=*true*)
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see FlowEngine::clampKValues)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(=*1000*)
Maximum number of timesteps between re-triangulation events. See also FlowEngine::defTolerance.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see FlowEngine::clampKValues)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(=*1*)
number of openblas threads in the factorization phase

numSolveThreads(=*1*)
number of openblas threads in the solve phase.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(=*0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(=*1.0*)
permability multiplier

permeabilityMap(=*false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(=*0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(=*true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(=*false*)
Compute pump torque applied on particles

relax(=*1.9*)
Gauss-Seidel relaxation

saveVtk(\[*(str)folder='./VTK'*\]) → None
Save pressure field in vtk format. Specify a folder name for output.

`setCellPImposed`(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

`setCellPressure`(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

`setImposedPressure`(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

`shearLubForce`(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

`shearLubTorque`(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

`shearLubrication`(=*false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

`shearVelocity`(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

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

`sineMagnitude`(=*0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

`slipBoundary`(=*true*)
Controls friction condition on lateral walls

`stiffness`(=*10000*)
equivalent contact stiffness used in the lubrication model

`surfaceDistanceParticle`(*(int)interaction*) → float
Return the distance between particles.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`tolerance`(=*1e-06*)
Gauss-Seidel tolerance

`twistTorque`(=*false*)
Compute twist torque applied on particles

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`updateBCs`() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

`updateTriangulation`(=*0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on FlowEngine::defTolerance and FlowEngine::meshUpdateInterval. Of course, it costs CPU time.

`useSolver`(=*0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

`viscosity`(=*1.0*)
viscosity of the fluid

`viscousNormalBodyStress`(=*false*)
compute normal viscous stress applied on each body

`viscousShear`(=*false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

viscousShearBodyStress(=*false*)
compute shear viscous stress applied on each body

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

wallIds(=*vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

wallThickness(=*0*)
Walls thickness

waveAction(=*false*)
Allow sinusoidal pressure condition to simulate ocean waves

xmax(=*1*)
See FlowEngine::xmin.

xmin(=*0*)
Index of the boundary x_{min}. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

ymax(=*3*)
See FlowEngine::xmin.

ymin(=*2*)
See FlowEngine::xmin.

zmax(=*5*)
See FlowEngine::xmin.

zmin(=*4*)
See FlowEngine::xmin.

class yade.wrapper.**RadialForceEngine**(*(object)arg1*)
Apply force of given magnitude directed away from spatial axis.

axisDir(=*Vector3r::UnitX()*)
Axis direction (normalized automatically)

axisPt(=*Vector3r::Zero()*)
Point on axis

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.

dict() → dict
Return dictionary of attributes.

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).

fNorm(=*0*)
Applied force magnitude

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

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be

typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**RotationEngine**(*(object)arg1*)
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]

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.

dict() → dict
Return dictionary of attributes.

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`).

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

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

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.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

zeroPoint(*=Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class yade.wrapper.**ServoPIDController**(*(object)arg1*)
PIDController servo-engine for applying prescribed force on bodies. http://en.wikipedia.org/wiki/PID_controller

axis(=*Vector3r::Zero()*)
Unit vector along which apply the velocity [-]

curVel(=*0.0*)
Current applied velocity [m/s]

current(=*Vector3r::Zero()*)
Current value for the controller [N]

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.

dict() → dict
Return dictionary of attributes.

errorCur(=*0.0*)
Current error [N]

errorPrev(=*0.0*)
Previous error [N]

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`).

iTerm(=*0.0*)
Integral term [N]

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

iterPeriod(=*100.0*)
Periodicity criterion of velocity correlation [-]

iterPrevStart(=*-1.0*)
Previous iteration of velocity correlation [-]

kD(=*0.0*)
Derivative gain/coefficient for the PID-controller [-]

kI(=*0.0*)
Integral gain/coefficient for the PID-controller [-]

kP(=*0.0*)
Proportional gain/coefficient for the PID-controller [-]

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxVelocity(=*0.0*)
Velocity [m/s]

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

target(=*0.0*)
Target value for the controller [N]

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`translationAxis`(=*uninitalized*)
Direction [Vector3]

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class `yade.wrapper.StepDisplacer`(*(object)arg1*)
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 another), but in this case, see rather LawTester

`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.

`dict`() → dict
Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

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

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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).

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.TorqueEngine`(*(object)arg1*)
Apply given torque (momentum) value at every subscribed particle, at every step.

`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.

`dict()` → dict
Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`moment`(=*Vector3r::Zero()*)
Torque value to be applied.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.TranslationEngine`(*(object)arg1*)
This engine is the base class for different engines, which require any kind of motion.

`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.

`dict()` → dict
Return dictionary of attributes.

`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`).

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

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

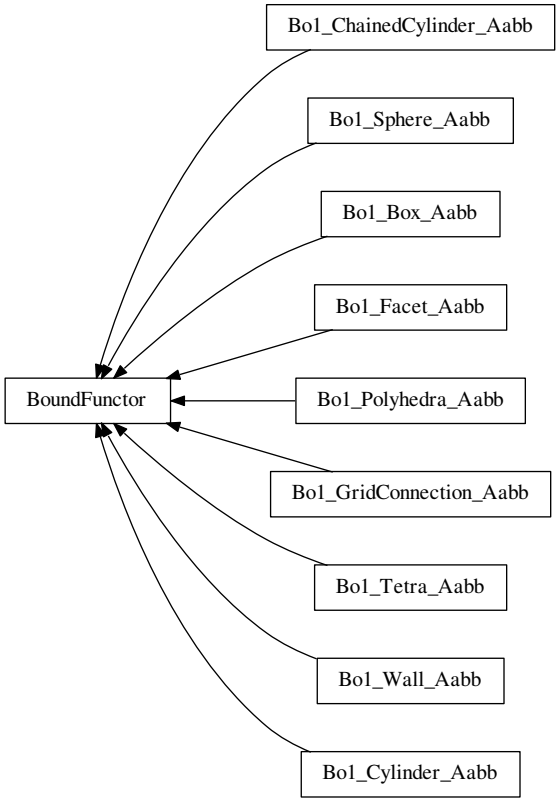
translationAxis(=*uninitalized*)
Direction [Vector3]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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

1.5 Bounding volume creation

1.5.1 BoundFunctor



class yade.wrapper.**BoundFunctor**(*(object)arg1*)
Functor for creating/updating Body::bound.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Box_Aabb(*(object)arg1*)
Create/update an Aabb of a Box.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_ChainedCylinder_Aabb(*(object)arg1*)
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_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

---

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Cylinder_Aabb(*(object)arg1*)
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_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

---

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Bo1_Facet_Aabb**(*(object)arg1*)
Creates/updates an Aabb of a Facet.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Bo1_GridConnection_Aabb**(*(object)arg1*)
Functor creating Aabb from a GridConnection.

aabbEnlargeFactor(=*-1*, *deactivated*)
Relative enlargement of the bounding box; deactivated if negative.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Bo1_Polyhedra_Aabb**(*(object)arg1*)
Create/update Aabb of a Polyhedra

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Sphere_Aabb(*(object)arg1*)
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_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

---

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Tetra_Aabb(*(object)arg1*)
Create/update Aabb of a Tetra

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Wall_Aabb(*(object)arg1*)
Creates/updates an Aabb of a Wall

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.5.2 BoundDispatcher

class `yade.wrapper.BoundDispatcher(`*(object)arg1*`)`
Dispatcher calling functors based on received argument type(s).

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

`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.

`dict()` → dict
Return dictionary of attributes.

`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.

`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`).

`functors`
Functors associated with this dispatcher.

`label(`*=uninitalized*`)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`minSweepDistFactor(`*=0.2*`)`
Minimal distance by which enlarge all bounding boxes; superseeds computed value of sweepDist when lower that (minSweepDistFactor x sweepDist). Updated by the collider. *(auto-updated)*.

`ompThreads(`*=-1*`)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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).

`targetInterv(`*=-1*`)`
see InsertionSortCollider::targetInterv *(auto-updated)*

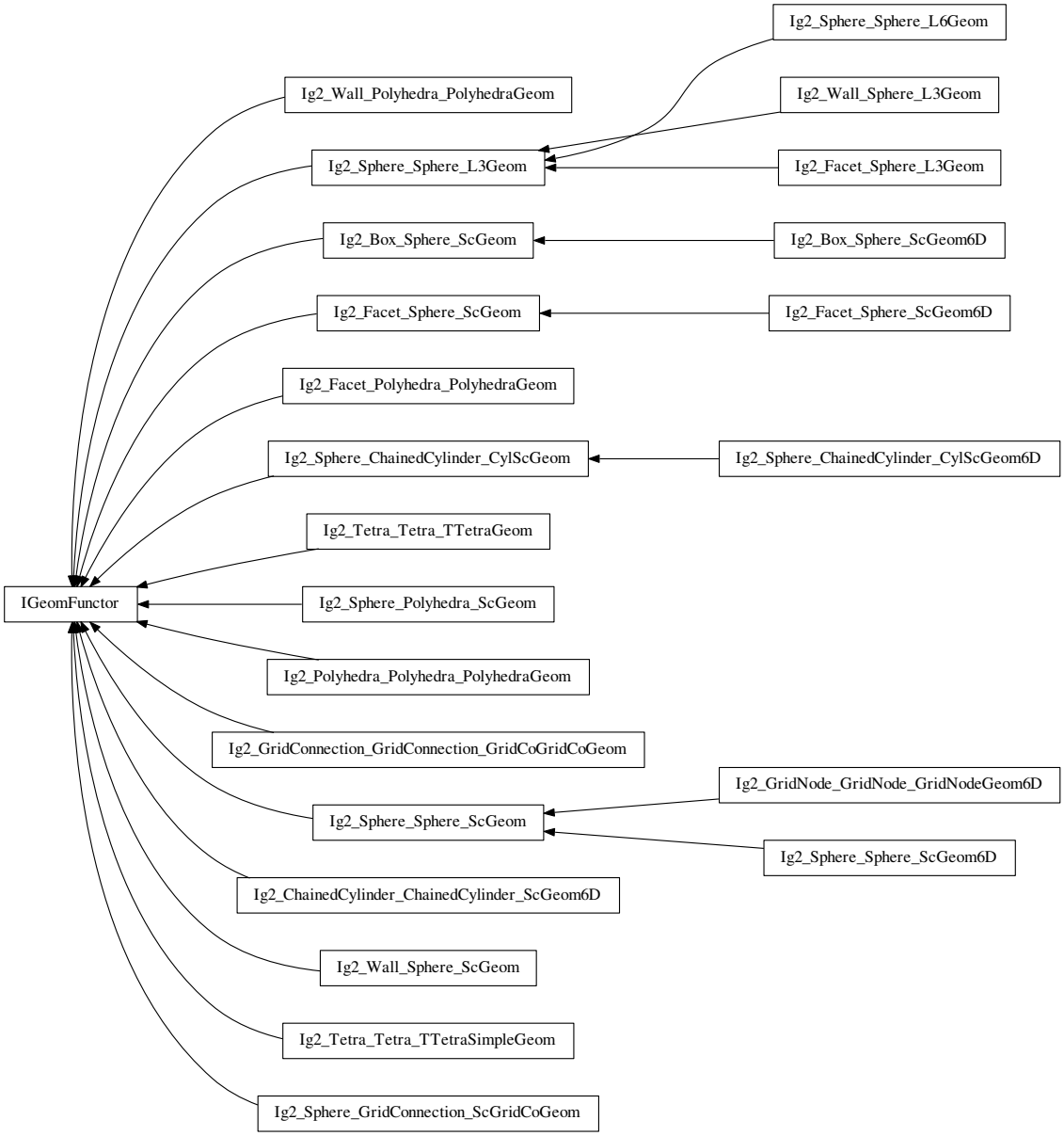
`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`updatingDispFactor(`*=-1*`)`
see InsertionSortCollider::updatingDispFactor *(auto-updated)*

1.6 Interaction Geometry creation

1.6.1 IGeomFunctor



class `yade.wrapper.IGeomFunctor`(*(object)arg1*)

Functor for creating/updating Interaction::geom objects.

`bases`
: Ordered list of types (as strings) this functor accepts.

`dict()` → dict
: Return dictionary of attributes.

`label`(=*uninitalized*)
: Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
: Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the

source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Box_Sphere_ScGeom**(*(object)arg1*)
Create an interaction geometry ScGeom from Box and Sphere, representing the box with a projected virtual sphere of same radius.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Box_Sphere_ScGeom6D**(*(object)arg1*)
Create an interaction geometry ScGeom6D from Box and Sphere, representing the box with a projected virtual sphere of same radius.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_ChainedCylinder_ChainedCylinder_ScGeom6D**(*(object)arg1*)
Create/update a ScGeom instance representing connexion between chained cylinders.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

halfLengthContacts(=*true*)
If True, Cylinders nodes interact like spheres of radius 0.5*length, else one node has size length while the other has size 0. The difference is mainly the locus of rotation definition.

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Facet_Polyhedra_PolyhedraGeom`(*(object)arg1*)
Create/update geometry of collision between Facet and Polyhedra

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Facet_Sphere_L3Geom`(*(object)arg1*)
Incrementally compute L3Geom for contact between Facet and Sphere. Uses attributes of Ig2_Sphere_Sphere_L3Geom.

`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.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`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).

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Facet_Sphere_ScGeom`(*(object)arg1*)
Create/update a ScGeom instance representing intersection of Facet and Sphere.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`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.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Facet_Sphere_ScGeom6D`(*(object)arg1*)
Create an interaction geometry ScGeom6D from Facet and Sphere, representing the Facet with a projected virtual sphere of same radius.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`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.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_GridConnection_GridConnection_GridCoGridCoGeom`(*(object)arg1*)
Create/update a GridCoGridCoGeom instance representing the geometry of a contact point between two GridConnection , including relative rotations.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_GridNode_GridNode_GridNodeGeom6D`(*(object)arg1*)
Create/update a GridNodeGeom6D instance representing the geometry of a contact point between two GridNode, including relative rotations.

`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. [McNamara2008], 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.).

`bases`
Ordered list of types (as strings) this functor accepts.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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_Polyhedra_Polyhedra_PolyhedraGeom`(*(object)arg1*)
Create/update geometry of collision between 2 Polyhedras

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom`(*(object)arg1*)
Create/update a ScGeom instance representing intersection of two Spheres.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

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

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom6D`(*(object)arg1*)
Create/update a ScGeom6D instance representing the geometry of a contact point between two Spheres, including relative rotations.

`bases`
Ordered list of types (as strings) this functor accepts.

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.

dict() → dict
Return dictionary of attributes.

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateRotations(=*false*)
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_Sphere_GridConnection_ScGridCoGeom**(*(object)arg1*)
Create/update a ScGridCoGeom6D instance representing the geometry of a contact point between a GricConnection and a Sphere including relative rotations.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Polyhedra_ScGeom**(*(object)arg1*)
Create/update geometry of collision between Sphere and Polyhedra

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

edgeCoeff(=*1.0*)
multiplier of penetrationDepth when sphere contacts edge (simulating smaller volume of actual intersection or when several polyhedrons has common edge)

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`vertexCoeff`(*=1.0*)
multiplier of penetrationDepth when sphere contacts vertex (simulating smaller volume of actual intersection or when several polyhedrons has common vertex)

class `yade.wrapper.Ig2_Sphere_Sphere_L3Geom`(*(object)arg1*)
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 $\mathbf{v}_i$, 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).

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

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

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.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Sphere_L6Geom**(*(object)arg1*)
Incrementally compute L6Geom for contact of 2 spheres.

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.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Sphere_ScGeom**(*(object)arg1*)
Create/update a ScGeom instance representing the geometry of a contact point between two Spheres 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. [McNamara2008], 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.).

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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.

---

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Sphere_ScGeom6D**(*(object)arg1*)
Create/update a ScGeom6D instance representing the geometry of a contact point between two Spheres, including relative rotations.

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. [McNamara2008], 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.).

bases
Ordered list of types (as strings) this functor accepts.

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.

dict() → dict
Return dictionary of attributes.

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.

---

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`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`(*(object)arg1*)
Create/update geometry of collision between 2 tetrahedra (TTetraGeom instance)

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Tetra_Tetra_TTetraSimpleGeom`(*(object)arg1*)
EXPERIMANTAL. Create/update geometry of collision between 2 tetrahedra (TTetraSimpleGeom instance)

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Wall_Polyhedra_PolyhedraGeom`(*(object)arg1*)
Create/update geometry of collision between Wall and Polyhedra

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Wall_Sphere_L3Geom`(*(object)arg1*)

Incrementally compute L3Geom for contact between Wall and Sphere. Uses attributes of Ig2_Sphere_Sphere_L3Geom.

`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.

`bases`

Ordered list of types (as strings) this functor accepts.

`dict()` → dict

Return dictionary of attributes.

`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).

`label`(*=uninitalized*)

Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`noRatch`(*=true*)

See Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting.

`timingDeltas`

Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`trsfRenorm`(*=100*)

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

`updateAttrs`(*(dict)arg2*) → None

Update object attributes from given dictionary

class `yade.wrapper.Ig2_Wall_Sphere_ScGeom`(*(object)arg1*)

Create/update a ScGeom instance representing intersection of Wall and Sphere.

`bases`

Ordered list of types (as strings) this functor accepts.

`dict()` → dict

Return dictionary of attributes.

`label`(*=uninitalized*)

Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`noRatch`(*=true*)

Avoid granular ratcheting

`timingDeltas`

Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None

Update object attributes from given dictionary

1.6.2 IGeomDispatcher

class `yade.wrapper.IGeomDispatcher`(*(object)arg1*)

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

`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.

`dict`() → dict
: Return dictionary of attributes.

`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.

`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`).

`functors`
: Functors associated with this dispatcher.

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

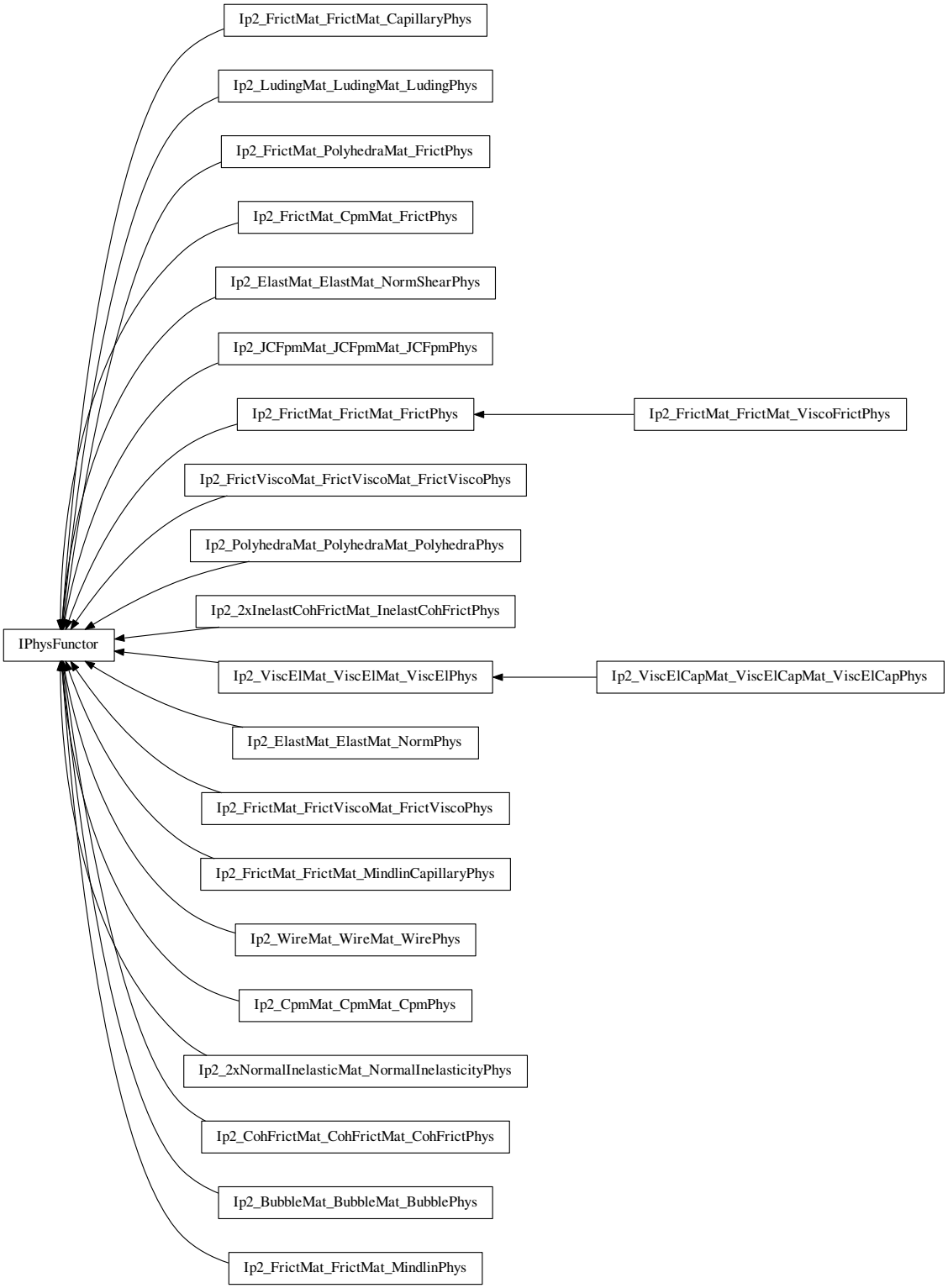
`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

1.7 Interaction Physics creation

1.7.1 IPhysFunctor



class `yade.wrapper.IPhysFunctor`(*(object)arg1*)

Functor for creating/updating Interaction::phys objects.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_2xInelastCohFrictMat_InelastCohFrictPhys**(*(object)arg1*)
Generates cohesive-frictional interactions with moments. Used in the contact law Law2_ScGeom6D_InelastCohFrictPhys_CohesionMoment.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_2xNormalInelasticMat_NormalInelasticityPhys**(*(object)arg1*)
Computes interaction attributes (of NormalInelasticityPhys type) from NormalInelasticMat material parameters. For simulations using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity. Note that, as for others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

bases
Ordered list of types (as strings) this functor accepts.

betaR(=*0.12*)
Parameter for computing the torque-stifness : T-stifness = betaR * Rmoy^2

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_BubbleMat_BubbleMat_BubblePhys**(*(object)arg1*)
Generates bubble interactions.Used in the contact law Law2_ScGeom_BubblePhys_Bubble.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_CohFrictMat_CohFrictMat_CohFrictPhys**(*(object)arg1*)
Generates cohesive-frictional interactions with moments, used in the contact law Law2_ScGeom6D_CohFrictPhys_CohesionMoment. The normal/shear stiffness and friction definitions are the same as in Ip2_FrictMat_FrictMat_FrictPhys, check the documentation there for details.

Adhesions related to the normal and the shear components are calculated form CohFrictMat::normalCohesion (C_n) and CohFrictMat::shearlCohesion (C_s). For particles of size R_1,R_2 the adhesion will be $a_i = C_i \min(R_1, R_2)^2$, $i = n\,s$.

Twist and rolling stiffnesses are proportional to the shear stiffness through dimensionless factors alphaKtw and alphaKr, such that the rotational stiffnesses are defined by $k_s \alpha_i R_1 R_2$, $i = t w r$

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_CpmMat_CpmMat_CpmPhys**(*(object)arg1*)
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.

bases
Ordered list of types (as strings) this functor accepts.

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).

dict() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_ElastMat_ElastMat_NormPhys`(*(object)arg1*)
Create a NormPhys from two ElastMats. TODO. EXPERIMENTAL

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_ElastMat_ElastMat_NormShearPhys`(*(object)arg1*)
Create a NormShearPhys from two ElastMats. TODO. EXPERIMENTAL

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_FrictMat_CpmMat_FrictPhys`(*(object)arg1*)
Convert CpmMat instance and FrictMat instance to FrictPhys with corresponding parameters (young, poisson, frictionAngle). Uses simple (arithmetic) averages if material parameters are different.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_CapillaryPhys**(*(object)arg1*)
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.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_FrictPhys**(*(object)arg1*)
Create a FrictPhys from two FrictMats. The compliance of one sphere under point load is defined here as $1/(E.D)$, with E the stiffness of the sphere and D its diameter. The compliance of the contact itself will be the sum of compliances from each sphere, i.e. $1/(E_1.D_1)+1/(E_2.D_2)$ in the general case, or $2/(E.D)$ in the special case of equal sizes and equal stiffness. Note that summing compliances corresponds to an harmonic average of stiffnesss (as in e.g. [Scholtes2009a]), which is how kn is actually computed in the Ip2_FrictMat_FrictMat_FrictPhys functor:

$k_n = \frac{E_1 D_1 * E_2 D_2}{E_1 D_1 + E_2 D_2} = \frac{k_1 * k_2}{k_1 + k_2}$, with $k_i = E_i D_i$.

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. In the case of a contact between a ViscElMat and a FrictMat, be sure to set FrictMat::young and FrictMat::poisson, otherwise the default value will be used.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_MindlinCapillaryPhys**(*(object)arg1*)
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.

bases
Ordered list of types (as strings) this functor accepts.

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

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

dict() → dict
Return dictionary of attributes.

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_MindlinPhys**(*(object)arg1*)
Calculate some physical parameters needed to obtain the normal and shear stiffnesses according to the Hertz-Mindlin formulation (as implemented in PFC).

Viscous parameters can be specified either using coefficients of restitution (e_n, e_s) or viscous damping ratio (β_n, β_s). The following rules apply: #. If the β_n (β_s) ratio 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.

bases
Ordered list of types (as strings) this functor accepts.

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

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

dict() → dict
Return dictionary of attributes.

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

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

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_ViscoFrictPhys**(*(object)arg1*)
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.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictViscoMat_FrictViscoPhys**(*(object)arg1*)
Converts a FrictMat and FrictViscoMat instance to FrictViscoPhys with corresponding parameters. Basically this functor corresponds to Ip2_FrictMat_FrictMat_FrictPhys with the only difference that damping in normal direction can be considered.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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

kRatio(=*uninitalized*)
Instance of MatchMaker determining how to compute interaction's shear contact stiffnesses. If this value is not given the elastic properties (i.e. poisson) of the two colliding materials are used to calculate the stiffness.

kn(=*uninitalized*)
Instance of MatchMaker determining how to compute interaction's normal contact stiffnesses. If this value is not given the elastic properties (i.e. young) of the two colliding materials are used to calculate the stiffness.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_PolyhedraMat_FrictPhys**(*(object)arg1*)

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictViscoMat_FrictViscoMat_FrictViscoPhys**(*(object)arg1*)
Converts 2 FrictViscoMat instances to FrictViscoPhys with corresponding parameters. Basically this functor corresponds to Ip2_FrictMat_FrictMat_FrictPhys with the only difference that damping in normal direction can be considered.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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

kRatio(=*uninitalized*)
Instance of MatchMaker determining how to compute interaction's shear contact stiffnesses. If this value is not given the elastic properties (i.e. poisson) of the two colliding materials are used to calculate the stiffness.

kn(=*uninitalized*)
Instance of MatchMaker determining how to compute interaction's normal contact stiffnesses. If this value is not given the elastic properties (i.e. young) of the two colliding materials are used to calculate the stiffness.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_JCFpmMat_JCFpmMat_JCFpmPhys**(*(object)arg1*)
Converts 2 JCFpmMat instances to one JCFpmPhys instance, with corresponding parameters.

bases
Ordered list of types (as strings) this functor accepts.

cohesiveTresholdIteration(=*1*)
should new contacts be cohesive? If strictly negativ, they will in any case. If positiv, they will before this iter, they won't afterward.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_LudingMat_LudingMat_LudingPhys**(*(object)arg1*)
Convert 2 instances of LudingMat to LudingPhys using the rule of consecutive connection.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_PolyhedraMat_PolyhedraMat_PolyhedraPhys**(*(object)arg1*)

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_ViscElCapMat_ViscElCapMat_ViscElCapPhys**(*(object)arg1*)
Convert 2 instances of ViscElCapMat to ViscElCapPhys using the rule of consecutive connection.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

en(=*uninitalized*)
Instance of MatchMaker determining restitution coefficient in normal direction

et(=*uninitalized*)
Instance of MatchMaker determining restitution coefficient in tangential direction

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

tc(=*uninitalized*)
Instance of MatchMaker determining contact time

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_ViscElMat_ViscElMat_ViscElPhys**(*(object)arg1*)
Convert 2 instances of ViscElMat to ViscElPhys using the rule of consecutive connection.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

en(=*uninitalized*)
Instance of MatchMaker determining restitution coefficient in normal direction

et(=*uninitalized*)
Instance of MatchMaker determining restitution coefficient in tangential direction

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

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

tc(=*uninitalized*)
Instance of MatchMaker determining contact time

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_WireMat_WireMat_WirePhys**(*(object)arg1*)
Converts 2 WireMat instances to WirePhys with corresponding parameters.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.7.2 IPhysDispatcher

class yade.wrapper.**IPhysDispatcher**(*(object)arg1*)
Dispatcher calling functors based on received argument type(s).

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.

dict() → dict
Return dictionary of attributes.

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.

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).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

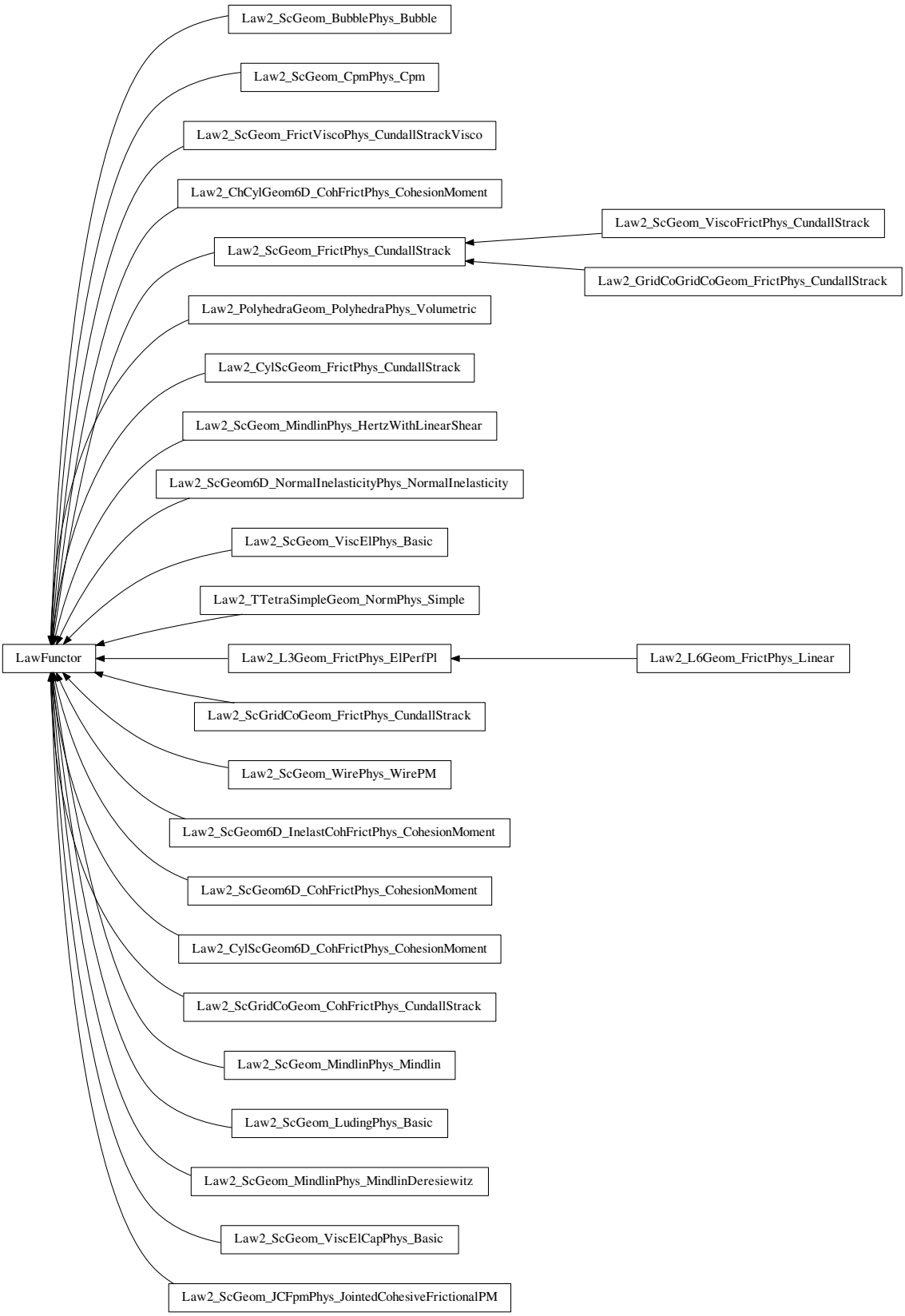
ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.8 Constitutive laws

1.8.1 LawFunctor



class yade.wrapper.**LawFunctor**(*(object)arg1*)

222 Functor for applying constitutive laws on interactions. Chapter 1. Class reference (yade.wrapper module)

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ChCylGeom6D_CohFrictPhys_CohesionMoment**(*(object)arg1*)
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 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.

---

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

bases
Ordered list of types (as strings) this functor accepts.

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

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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_CylScGeom6D_CohFrictPhys_CohesionMoment`(*(object)arg1*)
This law generalises Law2_CylScGeom_FrictPhys_CundallStrack by adding cohesion and moments at contact.

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

`bases`
Ordered list of types (as strings) this functor accepts.

`creep_viscosity`(=*1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_CohFrictPhys...

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`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.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`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_CylScGeom_FrictPhys_CundallStrack`(*(object)arg1*)
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.

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.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_GridCoGridCoGeom_FrictPhys_CundallStrack**(*(object)arg1*)
Frictional elastic contact law between two gridConnection . See Law2_ScGeom_FrictPhys_CundallStrack for more details.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_L3Geom_FrictPhys_ElPerfPl**(*(object)arg1*)
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`).

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

noSlip(=*false*)
No plastic slipping.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_L6Geom_FrictPhys_Linear**(*(object)arg1*)
Basic law for testing L6Geom – linear in both normal and shear sense, without slip or breakage.

bases
Ordered list of types (as strings) this functor accepts.

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

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

noSlip(=*false*)
No plastic slipping.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_PolyhedraGeom_PolyhedraPhys_Volumetric**(*(object)arg1*)
Calculate physical response of 2 vector in interaction, based on penetration configuration given by PolyhedraGeom. Normal force is proportional to the volume of intersection

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

plasticDissipation() → float
Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if Law2_PolyhedraGeom_PolyhedraPhys_Volumetric::traceEnergy is true.

shearForce(=*Vector3r::Zero()*)
Shear force from last step

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

volumePower(=*1.*)
Power of volume used in evaluation of normal force. Default is 1.0 - normal force is linearly proportional to volume. 1.0/3.0 would mean that normal force is proportional to the cube root of volume, approximation of penetration depth.

class yade.wrapper.**Law2_ScGeom6D_CohFrictPhys_CohesionMoment**(*(object)arg1*)
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_Cundall-Strack.

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_s 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, 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, 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 (details can be found in [Bourrier2013]). The maximum value of moments can be defined and takes the form of rolling friction. Cohesive -type moment may also be included 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 $d\Theta_{t,e}/dt = -M_t/\nu_t$.

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

bases
Ordered list of types (as strings) this functor accepts.

bendingElastEnergy() → float
Compute bending elastic energy.

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

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute total elastic energy.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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)

normElastEnergy() → float
Compute normal elastic energy.

shearElastEnergy() → float
Compute shear elastic energy.

shear_creep(=*false*)
activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

twistElastEnergy() → float
Compute twist elastic energy.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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_InelastCohFrictPhys_CohesionMoment**(*(object)arg1*)
This law is currently under developpement. Final version and documentation will come before the end of 2014.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

normElastEnergy() → float
Compute normal elastic energy.

shearElastEnergy() → float
Compute shear elastic energy.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity**(*(object)arg1*)
Contact law used to simulate granular filler in rock joints [Duriez2009a], [Duriez2011]. 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 examples/normalInelasticity-test.py

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

momentAlwaysElastic(=*false*)
boolean, true=> the part of the contact torque (caused by relative rotations, which is 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

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_BubblePhys_Bubble**(*(object)arg1*)
Constitutive law for Bubble model.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

pctMaxForce(=*0.1*)
Chan[2011] states the contact law is valid only for small interferences; therefore an exponential force-displacement curve models the contact stiffness outside that regime (large penetration). This artificial stiffening ensures that bubbles will not pass through eachother or completely overlap during the simulation. The maximum force is Fmax = (2*pi*surfaceTension*rAvg). pctMaxForce is the percentage of the maximum force dictates the separation threshold, Dmax, for each contact. Penetrations less than Dmax calculate the reaction force from the derived contact law, while penetrations equal to or greater than Dmax calculate the reaction force from the artificial exponential curve.

surfaceTension(=*0.07197*)
The surface tension in the liquid surrounding the bubbles. The default value is that of water at 25 degrees Celcius.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_CpmPhys_Cpm**(*(object)arg1*)
Constitutive law for the cpm-model.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "CpmPhys" contacts

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)

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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_ScGeom_FrictPhys_CundallStrack**(*(object)arg1*)
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.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

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).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_FrictViscoPhys_CundallStrackVisco**(*(object)arg1*)
Constitutive law for the FrictViscoPM. Corresponds to Law2_ScGeom_FrictPhys_CundallStrack with the only difference that viscous damping in normal direction can be considered.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "FrictViscoPhys" contacts

initPlasticDissipation(*(float)arg2*) → None
Initialize cummulated plastic dissipation to a value (0 by default).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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 :yref:Law2_ScGeom_FrictViscoPhys_CundallStrackVisco::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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM**(*(object)arg1*)
Interaction law for cohesive frictional material, e.g. rock, possibly presenting joint surfaces, that can be mechanically described with a smooth contact logic [Ivars2011] (implemented in Yade in [Scholtes2012]). See examples/jointedCohesiveFrictionalPM for script examples. Joint surface definitions (through stl meshes or direct definition with gts module) are illustrated there.

Key(=”“)
string specifying the name of saved file ‘cracks_____.txt’, when recordCracks is true.

bases
Ordered list of types (as strings) this functor accepts.

cracksFileExist(=*false*)
if true (and if recordCracks), data are appended to an existing ‘cracksKey’ text file; otherwise its content is reset.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene

recordCracks(=*false*)
if true, data about interactions that lose their cohesive feature are stored in a text file cracksKey.txt (see Key and cracksFileExist). It contains 9 columns: the break iteration, the 3 coordinates of the contact point, the type (1 means shear break, while 0 corresponds to tensile break), the ‘’cross section” (mean radius of the 2 spheres) and the 3 coordinates of the contact normal.

smoothJoint(=*false*)
if true, interactions of particles belonging to joint surface (JCFpmPhys.isOnJoint) are handled according to a smooth contact logic [Ivars2011], [Scholtes2012].

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_LudingPhys_Basic**(*(object)arg1*)
Linear viscoelastic model operating on ScGeom and LudingPhys.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_MindlinPhys_HertzWithLinearShear**(*(object)arg1*)
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.

---

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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)

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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_MindlinPhys_Mindlin**(*(object)arg1*)
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.

bases
Ordered list of types (as strings) this functor accepts.

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.

dict() → dict
Return dictionary of attributes.

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.)

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`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)

`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

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Law2_ScGeom_MindlinPhys_MindlinDeresiewitz`(*(object)arg1*)
Hertz-Mindlin contact law with partial slip solution, as described in [Thornton1991].

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`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)

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Law2_ScGeom_ViscElCapPhys_Basic`(*(object)arg1*)
Extended version of Linear viscoelastic model with capillary parameters.

`NLiqBridg`(=*uninitalized*)
The total number of liquid bridges

`VLiqBridg`(=*uninitalized*)
The total volume of liquid bridges

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_ViscElPhys_Basic**(*(object)arg1*)
Linear viscoelastic model operating on ScGeom and ViscElPhys. The contact law is visco-elastic in the normal direction, and visco-elastic frictional in the tangential direction. The normal contact is modelled as a spring of equivalent stiffness k_n, placed in parallel with a viscous damper of equivalent viscosity c_n. As for the tangential contact, it is made of a spring-dashpot system (in parallel with equivalent stiffness k_s and viscosity c_s) in serie with a slider of friction coefficient $\mu = \tan\varphi$.

The friction coefficient $\mu = \tan\varphi$ is always evaluated as $\tan(\min(\varphi_1, \varphi_2))$, where φ_1 and φ_2 are respectively the friction angle of particle 1 and 2. For the other parameters, depending on the material input, the equivalent parameters of the contact (K_n,C_n,K_s,C_s,φ) are evaluated differently. In the following, the quantities in parenthesis are the material constant which are precised for each particle. They are then associated to particle 1 and 2 (e.g. kn_1,kn_2,cn_1...), and should not be confused with the equivalent parameters of the contact (K_n,C_n,K_s,C_s,φ).

- If contact time (tc), normal and tangential restitution coefficient (en,et) are precised, the equivalent parameters are evaluated following the formulation of Pournin [Pournin2001].
- If normal and tangential stiffnesses (kn, ks) and damping constant (cn,cs) of each particle are precised, the equivalent stiffnesses and damping constants of each contact made of two particles 1 and 2 is made $A = 2\frac{a_1 a_2}{a_1+a_2}$, where A is K_n, K_s, C_n and C_s, and 1 and 2 refer to the value associated to particle 1 and 2.
- Alternatively it is possible to precise the Young modulus (young) and poisson's ratio (poisson) instead of the normal and spring constant (kn and ks). In this case, the equivalent parameters are evaluated the same way as the previous case with $kn_x = E_x d_x$, $ks_x = \nu_x kn_x$, where E_x, ν_x and d_x are Young modulus, poisson's ratio and diameter of particle x.
- If Yound modulus (young), poisson's ratio (poisson), normal and tangential restitution coefficient (en,et) are precised, the equivalent stiffnesses are evaluated as previously: $K_n = 2\frac{kn_1 kn_2}{kn_1+kn_2}$, $kn_x = E_x d_x$, $K_s = 2(ks_1 ks_2)/(ks_1 + ks_2)$, $ks_x = \nu kn_x$. The damping constant is computed at each contact in order to fulfill the normal restitution coefficient $e_n = (en_1 en_2)/(en_1 + en_2)$. This is achieved resolving numerically equation 21 of [Schwager2007] (There is in fact a mistake in the article from equation 18 to 19, so that there is a change in sign). Be careful in this configuration the tangential restitution coefficient is set to 1 (no tangential damping). This formulation imposes directly the normal restitution coefficient of the collisions instead of the damping constant.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_ViscoFrictPhys_CundallStrack**(*(object)arg1*)
Law similar to Law2_ScGeom_FrictPhys_CundallStrack with the addition of shear creep at contacts.

bases
Ordered list of types (as strings) this functor accepts.

creepStiffness(*=1*)

dict() → dict
Return dictionary of attributes.

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).

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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.

shearCreep(*=false*)

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.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

viscosity(*=1*)

class yade.wrapper.**Law2_ScGeom_WirePhys_WirePM**(*(object)arg1*)
Constitutive law for the wire model.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGridCoGeom_CohFrictPhys_CundallStrack**(*(object)arg1*)
Law between a cohesive frictional GridConnection and a cohesive frictional Sphere. Almost the same than Law2_ScGeom6D_CohFrictPhys_CohesionMoment, but THE ROTATIONAL MOMENTS ARE NOT COMPUTED.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGridCoGeom_FrictPhys_CundallStrack**(*(object)arg1*)
Law between a frictional GridConnection and a frictional Sphere. Almost the same than Law2_ScGeom_FrictPhys_CundallStrack, but the force is divided and applied on the two GridNodes only.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

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)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_TTetraSimpleGeom_NormPhys_Simple**(*(object)arg1*)
EXPERIMENTAL. TODO

bases
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(`=*uninitalized*`)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

1.8.2 LawDispatcher

class `yade.wrapper.LawDispatcher(`*(object)arg1*`)`
Dispatcher calling functors based on received argument type(s).

`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.

`dict()` → dict
Return dictionary of attributes.

`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.

`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`).

`functors`
Functors associated with this dispatcher.

`label(`=*uninitalized*`)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads(`=*-1*`)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

1.9 Callbacks

class `yade.wrapper.IntrCallback`(*(object)arg1*)
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.

`dict`() → dict
Return dictionary of attributes.

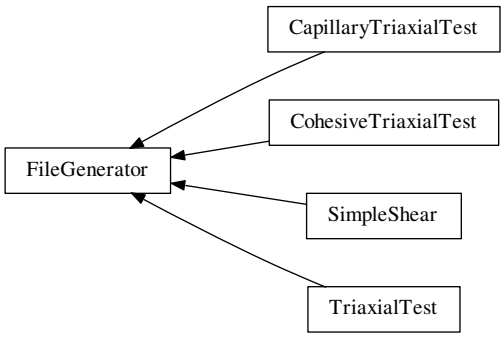
`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.SumIntrForcesCb`(*(object)arg1*)
Callback summing magnitudes of forces over all interactions. IPhys of interactions must derive from NormShearPhys (responsability fo the user).

`dict`() → dict
Return dictionary of attributes.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

1.10 Preprocessors



class `yade.wrapper.FileGenerator`(*(object)arg1*)
Base class for scene generators, preprocessors.

`dict`() → dict
Return dictionary of attributes.

`generate`(*(str)out*) → None
Generate scene, save to given file

`load()` → None
Generate scene, save to temporary file and load immediately

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

class `yade.wrapper.CapillaryTriaxialTest(`*(object)arg1*`)`
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.'

`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.

`capillaryPressure(`*=0*`)`
Define succion in the packing [Pa]. This is the value used in the capillary model.

`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

dict() → dict
Return dictionary of attributes.

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

generate(*(str)out*) → None
Generate scene, save to given file

importFilename(=*""*)
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

load() → None
Generate scene, save to temporary file and load immediately

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 (< 0 for real - compressive - compaction).

sigmaLateralConfinement(*=-50000*)
Lateral stress during triaxial loading (< 0 for classical compressive cases). 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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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**(*(object)arg1*)
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

dict() → dict
Return dictionary of attributes.

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?

generate(*(str)out*) → None
Generate scene, save to given file

importFilename(="")
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

load() → None
Generate scene, save to temporary file and load immediately

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 (< 0 for real - compressive - compaction).

sigmaLateralConfinement(=*-50000*)
Lateral stress during triaxial loading (< 0 for classical compressive cases). 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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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**(*(object)arg1*)
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 is generated by default via the same method used in TriaxialTest Preprocesor (=> see in source function GenerateCloud). But import of a list of spheres from a text file can be also performed after few changes in the source code.

  Launching this preprocessor will carry out an oedometric compression, until a value of normal stress equal to 2 MPa (and stable). But with others Engines KinemCNDEngine, KinemCNSEngine and KinemCNLEngine, respectively constant normal displacement, constant normal rigidity and constant normal stress paths can be carried out for such simple shear boxes.

NB about micro-parameters : their default values correspond to those used in [Duriez2009a] and [Duriez2011] to simulate infilled rock joints.

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]

dict() → dict
Return dictionary of attributes.

generate(*(str)out*) → None
Generate scene, save to given file

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 (if gravApplied) [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]

load() → None
Generate scene, save to temporary file and load immediately

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

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

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

class `yade.wrapper.TriaxialTest`(*(object)arg1*)
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 TriaxialCompressionEngine 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 inc**

   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

dict() → dict
Return dictionary of attributes.

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.

generate(*(str)out*) → None
Generate scene, save to given file

importFilename(=*""*)
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

load() → None
Generate scene, save to temporary file and load immediately

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 (< 0 for real - compressive - compaction).

sigmaLateralConfinement(=*-50000*)
Lateral stress during triaxial loading (< 0 for classical compressive cases). 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

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

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**(*(object)arg1*)
Class responsible for rendering scene on OpenGL devices.

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

bound(=*false*)
Render body Bound

cellColor(=*Vector3r(1, 1, 0)*)
Color of the periodic cell (RGB).

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

dict() → dict
Return dictionary of attributes.

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).

hideBody(*(int)id*) → None
Hide body from id (see OpenGLRenderer::showBody)

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.

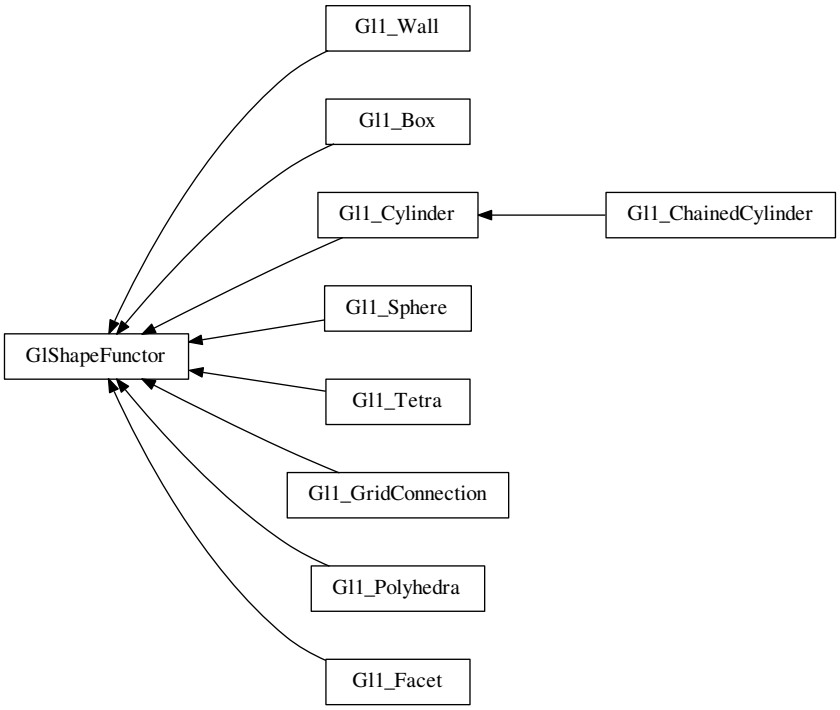
shape(=*true*)
Render body Shape

showBody(*(int)id*) → None
Make body visible (see OpenGLRenderer::hideBody)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Render all bodies with wire only (faster)

1.11.2 GlShapeFunctor



class yade.wrapper.**GlShapeFunctor**(*(object)arg1*)
Abstract functor for rendering Shape objects.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Gl1_Box**(*(object)arg1*)
Renders Box object

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_ChainedCylinder`(*(object)arg1*)
Renders ChainedCylinder object including a shift for compensating flexion.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`glutNormalize` = **True**

`glutSlices` = **8**

`glutStacks` = **4**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Cylinder`(*(object)arg1*)
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.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`glutNormalize` = **True**

`glutSlices` = **8**

`glutStacks` = **4**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Facet`(*(object)arg1*)
Renders Facet object

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

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`normals` = **False**

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_GridConnection`(*(object)arg1*)
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 cylinder slices.

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

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`glutNormalize` = **True**

`glutSlices` = **8**

`glutStacks` = **4**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Polyhedra`(*(object)arg1*)
Renders Polyhedra object

`wire`(=*false*) [**static**]
Only show wireframe

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Sphere`(*(object)arg1*)
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 spheres are displayed (Gl1_Sphere::stripes = False), quality mutiplies Gl1_Sphere::glutSlices and Gl1_Sphere::glutStacks. If striped spheres are displayed (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)

`circleView`(=*false*) [**static**]
For 2D simulations : display tori instead of spheres, so they will appear like circles if the viewer is looking in the right direction. In this case, remember to disable perspective by pressing "t"-key in the viewer.

`circleRelThickness`(=*0.2*) [**static**]
If Gl1_Sphere::circleView is enabled, this is the torus diameter relative to the sphere radius (i.e. the circle relative thickness).

`circleAllowedRotationAxis`(='z') [**static**]
If Gl1_Sphere::circleView is enabled, this is the only axis ('x', 'y' or 'z') along which rotation is allowed for the 2D simulation. It allows right orientation of the tori to appear like circles in the viewer. For example, if circleAllowedRotationAxis='x' is set, blockedDOFs="YZ" should also be set for all your particles.

`bases`
Ordered list of types (as strings) this functor accepts.

`circleAllowedRotationAxis` = **'z'**

`circleRelThickness` = **0.2**

`circleView` = **False**

`dict()` → dict
Return dictionary of attributes.

`glutSlices` = **12**

`glutStacks` = **6**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`localSpecView` = **True**

`quality` = **1.0**

`stripes` = **False**

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Tetra`(*(object)arg1*)
Renders Tetra object

`wire`(=*true*) [**static**]
TODO

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **True**

class `yade.wrapper.Gl1_Wall`(*(object)arg1*)
Renders Wall object

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

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`div` = **20**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.11.3 GlStateFunctor

class yade.wrapper.**GlStateFunctor**(*(object)arg1*)
Abstract functor for rendering State objects.

bases
Ordered list of types (as strings) this functor accepts.

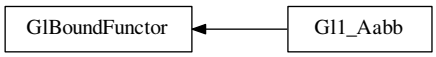
dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.11.4 GlBoundFunctor



class yade.wrapper.**GlBoundFunctor**(*(object)arg1*)
Abstract functor for rendering Bound objects.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Gl1_Aabb**(*(object)arg1*)
Render Axis-aligned bounding box (Aabb).

bases
Ordered list of types (as strings) this functor accepts.

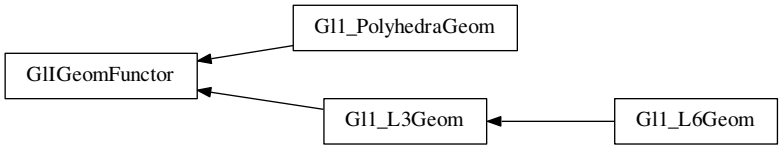
dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

1.11.5 GlIGeomFunctor



class `yade.wrapper.GlIGeomFunctor`(*(object)arg1*)
Abstract functor for rendering IGeom objects.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_L3Geom`(*(object)arg1*)
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.

`axesLabels` = **False**

`axesScale` = **1.0**

`axesWd` = **1.0**

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`uPhiWd = 2.0`

`uScale = 1.0`

`updateAttrs(dict)arg2)` → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_L6Geom(object)arg1)`
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.

`axesLabels = False`

`axesScale = 1.0`

`axesWd = 1.0`

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`phiScale = 1.0`

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`uPhiWd = 2.0`

`uScale = 1.0`

`updateAttrs(dict)arg2)` → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_PolyhedraGeom(object)arg1)`
Render PolyhedraGeom geometry.

`bases`
Ordered list of types (as strings) this functor accepts.

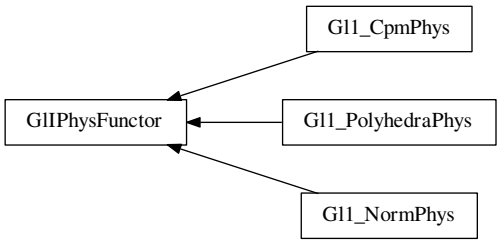
`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.11.6 GlIPhysFunctor



class yade.wrapper.**GlIPhysFunctor**(*(object)arg1*)
Abstract functor for rendering IPhys objects.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Gl1_CpmPhys**(*(object)arg1*)
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 ε_0 × colorStrainRatio (ε_0 = CpmPhys.epsCrackOnset). Otherwise, color based on the residual strength.

epsNLabel(=*false*) [**static**]
Numerically show normal strain

bases
Ordered list of types (as strings) this functor accepts.

`colorStrainRatio` = **-1.0**

`contactLine` = **True**

`dict()` → dict
Return dictionary of attributes.

`dmgLabel` = **True**

`dmgPlane` = **False**

`epsNLabel` = **False**

`epsT` = **False**

`epsTAxes` = **False**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`normal` = **False**

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_NormPhys`(*(object)arg1*)
Renders NormPhys objects as cylinders of which diameter and color depends on NormPhys.normalForce magnitude.

`maxFn`(=*0*) [**static**]
Value of NormPhys.normalForce corresponding to maxRadius. 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.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`maxFn = 0.0`

`maxRadius = -1.0`

`maxWeakFn = nan`

`refRadius = inf`

`signFilter = 0`

`slices = 6`

`stacks = 1`

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`weakFilter = 0`

`weakScale = 1.0`

class `yade.wrapper.Gl1_PolyhedraPhys(`*(object)arg1*`)`
Renders PolyhedraPhys objects as cylinders of which diameter and color depends on PolyhedraPhys::normForce magnitude.

`maxFn(=0)` [**static**]
Value of NormPhys.normalForce corresponding to maxDiameter. This value will be increased (but *not decreased*) automatically.

`refRadius(=std::numeric_limits<Real>::infinity())` [**static**]
Reference (minimum) particle radius

`signFilter(=0)` [**static**]
If non-zero, only display contacts with negative (-1) or positive (+1) normal forces; if zero, all contacts will be displayed.

`maxRadius(=-1)` [**static**]
Cylinder radius corresponding to the maximum normal force.

`slices(=6)` [**static**]
Number of sphere slices; (see glutCylinder reference)

`stacks(=1)` [**static**]
Number of sphere stacks; (see glutCylinder reference)

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`maxFn = 0.0`

`maxRadius = -1.0`

`refRadius = inf`

`signFilter = 0`

slices = 6

stacks = 1

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.12 Simulation data

1.12.1 Omega

class yade.wrapper.**Omega**(*(object)arg1*)

addScene() → int
Add new scene to Omega, returns its number

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.

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. The file should be saved in the same version of Yade, otherwise compatibility is not guaranteed.

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 scenes!).

resetAllScenes() → None
Reset all scenes.

resetCurrentScene() → None
Reset current 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 or .yade or .yade.gz). .xml files are bigger than .yade, but can be more or less easily (due to their size) opened and edited, e.g. with text editors. .bz2 and .gz correspond both to compressed versions. All saved files should be loaded in the same version of Yade, otherwise compatibility is not guaranteed.

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.

sceneToString() → str
Return the entire scene as a string. Equivalent to using O.save(...) except that the scene goes to a string instead of a file. (see also stringToScene())

speed
Return current calculation speed [iter/sec].

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.

stopAtTime
Get/set time after which the simulation will stop.

stringToScene(*(str)arg2*[, *(str)mark=''*]) → None
Load simulation from a string passed as argument (see also sceneToString).

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])

switchToScene(*(int)arg2*) → None
Switch to defined scene. Default scene has number 0, other scenes have to be created by addScene method.

tags
Tags (string=string dictionary) of the current simulation (container supporting string-index access/assignment)

thisScene
Return current scene's id.

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(`*(object)arg1*, *(BodyContainer)arg2*`)`

`__init__(`*(BodyContainer)arg2*`)` → None

`addToClump(`*(object)arg2*, *(int)arg3*[, *(int)discretization=0*]`)` → None
Add body b (or a list of bodies) to an existing clump c. c must be clump and b may not be a clump member of c. Clump masses and inertia are adapted automatically (for details see clump()).

See examples/clumps/addToClump-example.py for an example script.

Note: If b is a clump itself, then all members will be added to c and b will be deleted. If b is a clump member of clump d, then all members from d will be added to c and d will be deleted. If you need to add just clump member b, release this member from d first.

`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*[, *(int)discretization=0*]`)` → tuple
Append given list of bodies as a clump (rigid aggregate); returns a tuple of `(clumpId,[memberId1,memberId2,...])`. Clump masses and inertia are adapted automatically (for details see clump()).

`clear()` → None
Remove all bodies (interactions not checked)

`clump(`*(object)arg2*[, *(int)discretization=0*]`)` → int
Clump given bodies together (creating a rigid aggregate); returns `clumpId`. Clump masses and inertia are adapted automatically when discretization>0. If clump members are overlapping this is done by integration/summation over mass points using a regular grid of cells (grid cells length is defined as $\mathtt{R}_{\mathtt{min}}/\mathtt{discretization}$, where $\mathtt{R}_{\mathtt{min}}$ is minimum clump member radius). For non-overlapping members inertia of the clump is the sum of inertias from members. If discretization<=0 sum of inertias from members is used (faster, but inaccurate).

`erase(`*(int)arg2*[, *(bool)eraseClumpMembers=0*]`)` → bool
Erase body with the given id; all interaction will be deleted by InteractionLoop in the next step. If a clump is erased use *O.bodies.erase(clumpId,True)* to erase the clump AND its members.

`getRoundness(`[*(list)excludeList=*[]]`)` → float
Returns roundness coefficient RC = R2/R1. R1 is the equivalent sphere radius of a clump. R2 is the minimum radius of a sphere, that imbeds the clump. If just spheres are present

RC = 1. If clumps are present 0 < RC < 1. Bodies can be excluded from the calculation by giving a list of ids: *O.bodies.getRoundness([ids])*.

See examples/clumps/replaceByClumps-example.py for an example script.

`releaseFromClump`(*(int)arg2*, *(int)arg3*[, *(int)discretization=0*]) → None
Release body b from clump c. b must be a clump member of c. Clump masses and inertia are adapted automatically (for details see clump()).

See examples/clumps/releaseFromClump-example.py for an example script.

Note: If c contains only 2 members b will not be released and a warning will appear. In this case clump c should be erased.

`replace`(*(object)arg2*) → object

`replaceByClumps`(*(list)arg2*, *(object)arg3*[, *(int)discretization=0*]) → list
Replace spheres by clumps using a list of clump templates and a list of amounts; returns a list of tuples: `[(clumpId1,[memberId1,memberId2,...]),(clumpId2,[memberId1,memberId2,...]),...]`. A new clump will have the same volume as the sphere, that was replaced. Clump masses and inertia are adapted automatically (for details see clump()).

O.bodies.replaceByClumps([utils.clumpTemplate([1,1],[.5,.5])] , [.9]) #will replace 90 % of all standalone spheres by 'dyads'

See examples/clumps/replaceByClumps-example.py for an example script.

`updateClumpProperties`([*(list)excludeList=[]*[, *(int)discretization=5*]]) → None
Manually force Yade to update clump properties mass, volume and inertia (for details of 'discretization' value see clump()). Can be used, when clumps are modified or erased during a simulation. Clumps can be excluded from the calculation by giving a list of ids: *O.bodies.updateProperties([ids])*.

1.12.3 InteractionContainer

class `yade.wrapper.InteractionContainer`(*(object)arg1*, *(InteractionContainer)arg2*)
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, and invalidate persistent collider data (if the collider supports it).

`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`(*(object)arg1*, *(ForceContainer)arg2*)

`__init__`(*(ForceContainer)arg2*) → None

`addF`(*(int)id*, *(Vector3)f*[, *(bool)permanent=False*]) → None
Apply force on body (accumulates).

If permanent=false (default), the force applies for one iteration, then it is reset by ForceResetter. # If permanent=true, it persists over iterations, until it is overwritten by another call to addF(id,f,True) or removed by reset(resetAll=True). The permanent force on a body can be checked with permF(id).

`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*[, *(bool)permanent=False*]) → None
Apply torque on body (accumulates).

If permanent=false (default), the torque applies for one iteration, then it is reset by ForceResetter. # If permanent=true, it persists over iterations, until it is overwritten by another call to addT(id,f,True) or removed by reset(resetAll=True). The permanent torque on a body can be checked with permT(id).

`f`(*(int)id*[, *(bool)sync=False*]) → Vector3
Force applied on body. For clumps in openMP, synchronize the force container with sync=True, else the value will be wrong.

`getPermForceUsed`() → bool
Check wether permanent forces are present.

`m`(*(int)id*[, *(bool)sync=False*]) → Vector3
Deprecated alias for t (torque).

`move`(*(int)id*) → Vector3
Displacement applied on body.

`permF`(*(int)id*) → Vector3
read the value of permanent force on body (set with setPermF()).

`permT`(*(int)id*) → Vector3
read the value of permanent torque on body (set with setPermT()).

`reset`([*(bool)resetAll=True*]) → None
Reset the force container, including user defined permanent forces/torques. resetAll=False will keep permanent forces/torques unchanged.

`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*[, *(bool)sync=False*]) → Vector3
Torque applied on body. For clumps in openMP, synchronize the force container with sync=True, else the value will be wrong.

1.12.5 MaterialContainer

class `yade.wrapper.MaterialContainer`(*(object)arg1*, *(MaterialContainer)arg2*)
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`(*(object)arg1*)
Object comprising the whole simulation.

`compressionNegative`
Whether the convention is that compression has negative sign (set by IGeomFunctor).

`dict`() → dict
Return dictionary of attributes.

`doSort`(=*false*)
Used, when new body is added to the scene.

`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 IGeomFunctor).

`selectedBody`(=*-1*)
Id of body that is selected by the user

`speed`(=*0*)
Current calculation speed [iter/s]

`stopAtIter`(=*0*)
Iteration after which to stop the simulation.

`stopAtTime`(=*0*)
Time 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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

1.12.7 Cell

class yade.wrapper.**Cell**(*(object)arg1*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

dict() → dict
Return dictionary of attributes.

getDefGrad() → Matrix3
Returns deformation gradient tensor $\mathbf{F}$ of the cell deformation (http://en.wikipedia.org/wiki/Finite_strain_theory)

getEulerianAlmansiStrain() → Matrix3
Returns Eulerian-Almansi strain tensor $\boldsymbol{e} = \frac{1}{2}(\mathbf{I} - \mathbf{b}^{-1}) = \frac{1}{2}(\mathbf{I} - (\mathbf{F}\mathbf{F}^{\mathsf{T}})^{-1})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

getLCauchyGreenDef() → Matrix3
Returns left Cauchy-Green deformation tensor $\mathbf{b} = \mathbf{F}\mathbf{F}^{\mathsf{T}}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

getLagrangianStrain() → Matrix3
Returns Lagrangian strain tensor $\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}) = \frac{1}{2}(\mathbf{F}^{\mathsf{T}}\mathbf{F} - \mathbf{I}) = \frac{1}{2}(\mathbf{U}^2 - \mathbf{I})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

getLeftStretch() → Matrix3
Returns left (spatial) stretch tensor of the cell (matrix $\mathbf{U}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

getPolarDecOfDefGrad() → tuple
Returns orthogonal matrix $\mathbf{R}$ and symmetric positive semi-definite matrix $\mathbf{U}$ as polar decomposition of deformation gradient $\mathbf{F}$ of the cell ($\mathbf{F} = \mathbf{R}\mathbf{U}$)

getRCauchyGreenDef() → Matrix3
Returns right Cauchy-Green deformation tensor $\mathbf{C} = \mathbf{F}^{\mathsf{T}}\mathbf{F}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

getRightStretch() → Matrix3
Returns right (material) stretch tensor of the cell (matrix $\mathbf{V}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U} = \mathbf{V}\mathbf{R} \rightarrow \mathbf{V} = \mathbf{F}\mathbf{R}^{\mathsf{T}}$)

getRotation() → Matrix3
Returns rotation of the cell (orthogonal matrix $\mathbf{R}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

getSmallStrain() → Matrix3
Returns small strain tensor $\boldsymbol{\varepsilon} = \frac{1}{2}(\mathbf{F}+\mathbf{F}^{\mathsf{T}})-\mathbf{I}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

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 -1 × hSize.

homoDeform(=*true*)
Deform (velGrad) the cell homothetically, by adjusting positions and velocities of bodies. The velocity change is obtained by deriving the expression v= v.x, where v is the macroscopic velocity gradient, giving in an incremental form: Δv=Δ v x + v Δx. As a result, velocities are modified as soon as `velGrad` changes, according to the first term: Δv(t)=Δ v x(t), while the 2nd term reflects a convective term: Δv'= v v(t-dt/2).

nextVelGrad(=*Matrix3r::Zero()*)
see Cell.velGrad.

prevHSize(=*Matrix3r::Identity()*)
hSize from the previous step, used in the definition of relative velocity across periods.

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 `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)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`velGrad`
Velocity gradient of the transformation; used in NewtonIntegrator. Values of velGrad accumulate in trsf at every step.

NOTE: changing velGrad at the beginning of a simulation loop would lead to inaccurate integration for one step, as it should normaly be changed after the contact laws (but before Newton). To avoid this problem, assignment is deferred automatically. The target value typed in terminal is actually stored in Cell.nextVelGrad and will be applied right in time by Newton integrator.

Note: Assigning individual components of velGrad is not possible (it will not return any error but it will have no effect). Instead, you can assign to Cell.nextVelGrad, as in O.cell.nextVelGrad[1,2]=1.

`velGradChanged`(=*false*)
true when velGrad has been changed manually (see also Cell.nextVelGrad)

`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.TimingDeltas`(*(object)arg1*)

`data`
Get timing data as list of tuples (label, execTime[nsec], execCount) (one tuple per checkpoint)

`reset`() → None
Reset timing information

class `yade.wrapper.GlShapeDispatcher`(*(object)arg1*)
Dispatcher calling functors based on received argument type(s).

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`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`).

`functors`
Functors associated with this dispatcher.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.LBMlink`(*(object)arg1*)
Link class for Lattice Boltzmann Method

`DistMid`(=*Vector3r::Zero()*)
Distance between middle of the link and mass center of body

`PointingOutside`(=*false*)
True if it is a link pointing outside to the system (from a fluid or solid node)

`VbMid`(=*Vector3r::Zero()*)
Velocity of boundary at midpoint

`ct`(=*0.*)
Coupling term in modified bounce back rule

`dict`() → dict
Return dictionary of attributes.

`fid`(=*-1*)
Fluid node identifier

`i`(=*-1*)
direction index of the link

`idx_sigma_i`(=*-1*)
sigma_i direction index (Fluid->Solid)

`isBd`(=*false*)
True if it is a boundary link

`nid1`(=*-1*)
fixed node identifier

`nid2`(=*-1*)
fixed node identifier or -1 if node points outside

`sid`(=*-1*)
Solid node identifier

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlExtra_LawTester`(*(object)arg1*)
Find an instance of LawTester and show visually its data.

`dead`(=*false*)
Deactivate the object (on error/exception).

`dict`() → dict
Return dictionary of attributes.

`tester`(=*uninitalized*)
Associated LawTester object.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.MatchMaker`(*(object)arg1*)
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.

`dict`() → dict
Return dictionary of attributes.

`matches`(=*uninitalized*)
Array of `(id1,id2,value)` items; queries matching `id1` + `id2` or `id2` + `id1` will return `value`

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`val`(=*NaN*)
Constant value returned if there is no match and algo is `val`

class `yade.wrapper.GlBoundDispatcher`(*(object)arg1*)
Dispatcher calling functors based on received argument type(s).

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`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).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**EnergyTracker**(*(object)arg1*)
Storage for tracing energies. Only to be used if O.trackEnergy is True.

clear() → None
Clear all stored values.

dict() → dict
Return dictionary of attributes.

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.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Engine**(*(object)arg1*)
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.

dict() → dict
Return dictionary of attributes.

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.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**LBMnode**(*(object)arg1*)
Node class for Lattice Boltzmann Method

dict() → dict
Return dictionary of attributes.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GlIGeomDispatcher**(*(object)arg1*)
Dispatcher calling functors based on received argument type(s).

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.

dict() → dict
Return dictionary of attributes.

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.

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`).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.ParallelEngine`(*(object)arg1*)
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.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`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.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.LBMbody`(*(object)arg1*)
Body class for Lattice Boltzmann Method

`AVel`(=*Vector3r::Zero()*)
Angular velocity of body

`Fh`(=*Vector3r::Zero()*)
Hydrodynamical force on body

`Mh`(=*Vector3r::Zero()*)
Hydrodynamical momentum on body

`dict`() → dict
Return dictionary of attributes.

`fm`(=*Vector3r::Zero()*)
Hydrodynamic force (LB unit) at t-0.5dt

`force`(=*Vector3r::Zero()*)
Hydrodynamic force, need to be reinitialized (LB unit)

`fp`(=*Vector3r::Zero()*)
Hydrodynamic force (LB unit) at t+0.5dt

`isEroded`(=*false*)
Hydrodynamical force on body

mm(=*Vector3r::Zero()*)
Hydrodynamic momentum (LB unit) at t-0.5dt

momentum(=*Vector3r::Zero()*)
Hydrodynamic momentum,need to be reinitialized (LB unit)

mp(=*Vector3r::Zero()*)
Hydrodynamic momentum (LB unit) at t+0.5dt

pos(=*Vector3r::Zero()*)
Position of body

radius(=*-1000.*)
Radius of body (for sphere)

saveProperties(=*false*)
To save properties of the body

type(=*-1*)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Velocity of body

class yade.wrapper.**Functor**(*(object)arg1*)
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.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Serializable**(*(object)arg1*)

dict() → dict
Return dictionary of attributes.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GlStateDispatcher**(*(object)arg1*)
Dispatcher calling functors based on received argument type(s).

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.

dict() → dict
Return dictionary of attributes.

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.

`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`).

`functors`
Functors associated with this dispatcher.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlIPhysDispatcher`(*(object)arg1*)
Dispatcher calling functors based on received argument type(s).

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`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`).

`functors`
Functors associated with this dispatcher.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlExtra_OctreeCubes`(*(object)arg1*)
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).

`dead`(=*false*)
Deactivate the object (on error/exception).

`dict`() → dict
Return dictionary of attributes.

`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)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Dispatcher`(*(object)arg1*)
Engine dispatching control to its associated functors, based on types of argument it receives. This abstract base class provides no functionality in itself.

`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.

`dict`() → dict
Return dictionary of attributes.

`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.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. InteractionLoop). This attribute is mostly useful for experiments or when combining ParallelEngine with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Cell`(*(object)arg1*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`dict`() → dict
Return dictionary of attributes.

`getDefGrad`() → Matrix3
Returns deformation gradient tensor $\mathbf{F}$ of the cell deformation (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getEulerianAlmansiStrain`() → Matrix3
Returns Eulerian-Almansi strain tensor $\boldsymbol{e} = \frac{1}{2}(\mathbf{I} - \mathbf{b}^{-1}) = \frac{1}{2}(\mathbf{I} - (\mathbf{F}\mathbf{F}^{\mathsf{T}})^{-1})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLCauchyGreenDef`() → Matrix3
Returns left Cauchy-Green deformation tensor $\mathbf{b} = \mathbf{F}\mathbf{F}^{\mathsf{T}}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLagrangianStrain`() → Matrix3
Returns Lagrangian strain tensor $\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}) = \frac{1}{2}(\mathbf{F}^{\mathsf{T}}\mathbf{F} - \mathbf{I}) = \frac{1}{2}(\mathbf{U}^2 - \mathbf{I})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLeftStretch`() → Matrix3
Returns left (spatial) stretch tensor of the cell (matrix $\mathbf{U}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getPolarDecOfDefGrad`() → tuple
Returns orthogonal matrix $\mathbf{R}$ and symmetric positive semi-definite matrix $\mathbf{U}$ as polar decomposition of deformation gradient $\mathbf{F}$ of the cell ($\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getRCauchyGreenDef`() → Matrix3
Returns right Cauchy-Green deformation tensor $\mathbf{C} = \mathbf{F}^{\mathsf{T}}\mathbf{F}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getRightStretch`() → Matrix3
Returns right (material) stretch tensor of the cell (matrix $\mathbf{V}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U} = \mathbf{V}\mathbf{R} \rightarrow \mathbf{V} = \mathbf{F}\mathbf{R}^{\mathsf{T}}$)

`getRotation`() → Matrix3
Returns rotation of the cell (orthogonal matrix $\mathbf{R}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getSmallStrain`() → Matrix3
Returns small strain tensor $\boldsymbol{\varepsilon} = \frac{1}{2}(\mathbf{F}+\mathbf{F}^{\mathsf{T}})-\mathbf{I}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`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 $^{-1}$ × hSize.

`homoDeform`(=*true*)
Deform (velGrad) the cell homothetically, by adjusting positions and velocities of bodies. The velocity change is obtained by deriving the expression v= v.x, where v is the macroscopic velocity gradient, giving in an incremental form: Δv=Δ v x + v Δx. As a result, velocities are modified as soon as `velGrad` changes, according to the first term: Δv(t)=Δ v x(t), while the 2nd term reflects a convective term: Δv'= v v(t-dt/2).

`nextVelGrad`(=*Matrix3r::Zero()*)
see Cell.velGrad.

`prevHSize`(=*Matrix3r::Identity()*)
hSize from the previous step, used in the definition of relative velocity across periods.

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 `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)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

velGrad
Velocity gradient of the transformation; used in NewtonIntegrator. Values of velGrad accumulate in trsf at every step.

NOTE: changing velGrad at the beginning of a simulation loop would lead to inacurate integration for one step, as it should normaly be changed after the contact laws (but before Newton). To avoid this problem, assignment is deferred automatically. The target value typed in terminal is actually stored in Cell.nextVelGrad and will be applied right in time by Newton integrator.

Note: Assigning individual components of velGrad is not possible (it will not return any error but it will have no effect). Instead, you can assign to Cell.nextVelGrad, as in O.cell.nextVelGrad[1,2]=1.

---

velGradChanged(=*false*)
true when velGrad has been changed manually (see also Cell.nextVelGrad)

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.GlExtraDrawer`(*(object)arg1*)
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).

`dict`() → dict
Return dictionary of attributes.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

Chapter 2

Yade modules

2.1 yade.bodiesHandling module

Miscellaneous functions, which are useful for handling bodies.

`yade.bodiesHandling.facetsDimensions(`*idFacets=[]*, *mask=-1*`)`
The function accepts the list of facet id's or list of facets and calculates max and min dimensions, geometrical center.

Parameters

- **idFacets** (*list*) – list of spheres
- **mask** (*int*) – Body.mask for the checked bodies

Returns dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension facet Id, Vector3), `maxId` (maximal dimension facet Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `number` (number of facets, int),

`yade.bodiesHandling.sphereDuplicate(`*idSphere*`)`
The functions makes a copy of sphere

`yade.bodiesHandling.spheresModify(`*idSpheres=[]*, *mask=-1*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, *copy=False*`)`
The function accepts the list of spheres id's or list of bodies and modifies them: rotating, scaling, shifting. if copy=True copies bodies and modifies them. Also the mask can be given. If idSpheres not empty, the function affects only bodies, where the mask passes. If idSpheres is empty, the function search for bodies, where the mask passes.

Parameters

- **shift** (*Vector3*) – Vector3(X,Y,Z) parameter moves spheres.
- **scale** (*float*) – factor scales given spheres.
- **orientation** (*Quaternion*) – orientation of spheres
- **mask** (*int*) – Body.mask for the checked bodies

Returns list of bodies if copy=True, and Boolean value if copy=False

`yade.bodiesHandling.spheresPackDimensions(`*idSpheres=[]*, *mask=-1*`)`
The function accepts the list of spheres id's or list of bodies and calculates max and min dimensions, geometrical center.

Parameters

- **idSpheres** (*list*) – list of spheres
- **mask** (*int*) – Body.mask for the checked bodies

Returns dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension sphere Id, Vector3), `maxId` (maximal dimension sphere Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `volume` (volume of spheres, Real), `mass` (mass of spheres, Real), `number` (number of spheres, int),

2.2 yade.export module

Export (not only) geometry to various formats.

class `yade.export.VTKExporter`

Class for exporting data to VTK Simple Legacy File (for example if, for some reason, you are not able to use VTKRecorder). Export of spheres, facets, interactions and polyhedra is supported.

USAGE: create object vtkExporter = VTKExporter('baseFileName'), add to engines PyRunner with command='vtkExporter.exportSomething(params)' alternatively just use vtkExporter.exportSomething(...) at the end of the script for instance

Example: examples/test/vtk-exporter/vtkExporter.py, examples/test/unv-read/unvReadVTKExport.py.

Parameters

- **baseName** (*string*) – name of the exported files. The files would be named baseName-spheres-snapNb.vtk or baseName-facets-snapNb.vtk
- **startSnap** (*int*) – the numbering of files will start form startSnap

`exportContactPoints()`

exports constact points and defined properties.

:param [(int,int)] ids: see exportInteractions :param [tuple(2)] what: what to export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the CPs are labeled as i in this function (sccording to their interaction). Scalar, vector and tensor variables are supported. For example, to export stiffness difference from certain value (1e9) (named as dStiff) you should write: ... what=[('dStiff','i.phys.kn-1e9'), ... :param {Interaction:Vector3} useRef: if not specified, current position used. Otherwise use position from dict using interactions as keys. Interactions not in dict are not exported :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportFacets()`

exports facets (positions) and defined properties. Facets are exported with multiplicated nodes

:param [int]|"all" ids: if "all", then export all facets, otherwise only facets from integer list :param [tuple(2)] what: see exportSpheres :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportFacetsAsMesh()`

exports facets (positions) and defined properties. Facets are exported as mesh (not with multiplicated nodes). Therefore additional parameters connectivityTable is needed

:param [int]|"all" ids: if "all", then export all facets, otherwise only facets from integer list :param [tuple(2)] what: see exportSpheres :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used :param [(float,float,float)|Vector3] nodes: list of coordinates of nodes :param [(int,int,int)] connectivityTable: list of node ids of individual elements (facets)

`exportInteractions()`

exports interactions and defined properties.

:param [(int,int)]|"all" ids: if "all", then export all interactions, otherwise only interactions from (int,int) list :param [tuple(2)] what: what to export. parameter is list of couple

(name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the interactions are labeled as i in this function. Scalar, vector and tensor variables are supported. For example, to export stiffness difference from certain value (1e9) (named as dStiff) you should write: ... what=[('dStiff','i.phys.kn-1e9'), ... :param [tuple(2|3)] verticesWhat: what to export on connected bodies. Bodies are labeled as 'b' (or 'b1' and 'b2' if you need treat both bodies differently) :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportPeriodicCell()`
exports spheres (positions and radius) and defined properties.

:param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportPolyhedra()`
Exports polyhedrons and defined properties.

:param ids: if "all", then export all polyhedrons, otherwise only polyhedrons from integer list :type ids: [int] | "all" :param what: what other than then position to export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (with name particleVelocity) and the distance form point (0,0,0) (named as dist) you should write: ... what=[('particleVelocity','b.state.vel'),('dist','b.state.pos.norm()', ... :type what: [tuple(2)] :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportSpheres()`
exports spheres (positions and radius) and defined properties.

:param [int]|"all" ids: if "all", then export all spheres, otherwise only spheres from integer list :param [tuple(2)] what: what other than then position and radius export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (with name particleVelocity) and the distance form point (0,0,0) (named as dist) you should write: ... what=[('particleVelocity','b.state.vel'),('dist','b.state.pos.norm()', ... :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used :param bool useRef: if False (default), use current position of the spheres for export, use reference position otherwise

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.gmshGeo`(*filename*, *comment=''*, *mask=-1*, *accuracy=-1*)
Save spheres in geo-file for the following using in GMSH (http://www.geuz.org/gmsh/doc/texinfo/) program. The spheres can be there meshed.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask
- **accuracy** (*float*) – the accuracy parameter, which will be set for the poinst in geo-file. By default: 1./10. of the minimal sphere diameter.

Returns number of spheres which were exported.

Return type int

yade.export.text(*filename*, *mask=-1*)
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.
- **mask** (*int*) – export only spheres with the corresponding mask

Returns number of spheres which were written.

Return type int

yade.export.text2vtk(*inFileName*, *outFileName*)
Converts text file (created by export.textExt function) into vtk file. See examples/test/paraview-spheres-solid-section/export_text.py example

Parameters

- **inFileName** (*str*) – name of input text file
- **outFileName** (*str*) – name of output vtk file

yade.export.text2vtkSection(*inFileName*, *outFileName*, *point*, *normal=(1, 0, 0)*)
Converts section through spheres from text file (created by export.textExt function) into vtk file. See examples/test/paraview-spheres-solid-section/export_text.py example

Parameters

- **inFileName** (*str*) – name of input text file
- **outFileName** (*str*) – name of output vtk file
- **point** (*Vector3|(float,float,float)*) – coordinates of a point lying on the section plane
- **normal** (*Vector3|(float,float,float)*) – normal vector of the section plane

yade.export.textClumps(*filename*, *format='x_y_z_r_clumpId'*, *comment=''*, *mask=-1*)
Save clumps-members into a text file. Non-clumps members are bodies are silently skipped.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **comment** (*string*) – 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 '\n#' for next lines.
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask

Returns number of clumps, number of spheres which were written.

Return type int

yade.export.textExt(*filename*, *format='x_y_z_r'*, *comment=''*, *mask=-1*, *attrs=[]*)
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** (*string*) – the name of output format. Supported 'x_y_z_r'(default), 'x_y_z_r_matId', 'x_y_z_r_attrs' (use proper comment)

- **comment** (*string*) – 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 '\n#' for next lines. With 'x_y_z_r_attrs' format, the last (or only) line should consist of column headers of quantities passed as attrs (1 comment word for scalars, 3 comment words for vectors and 9 comment words for matrices)
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask
- **attrs** (*[str]*) – attributes to be exported with 'x_y_z_r_attrs' format. Each str in the list is evaluated for every body exported with body=b (i.e. 'b.state.pos.norm()' would stand for distance of body from coordinate system origin)

Returns number of spheres which were written.

Return type int

`yade.export.textPolyhedra`(*fileName*, *comment=''*, *mask=-1*, *explanationComment=True*, *attrs=[]*)

Save polyhedra into a text file. Non-polyhedra bodies are silently skipped.

Parameters

- **filename** (*string*) – the name of the output file
- **comment** (*string*) – 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 '\n#' for next lines.
- **mask** (*int*) – export only polyhedra with the corresponding mask
- **explanationComment** (*str*) – inclde explanation of format to the beginning of file

Returns number of polyhedra which were written.

Return type int

2.3 yade.geom module

Creates geometry objects from facets.

`yade.geom.facetBox`(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned box composed of facets, with given center, extents and orientation. If any of the box dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the box.

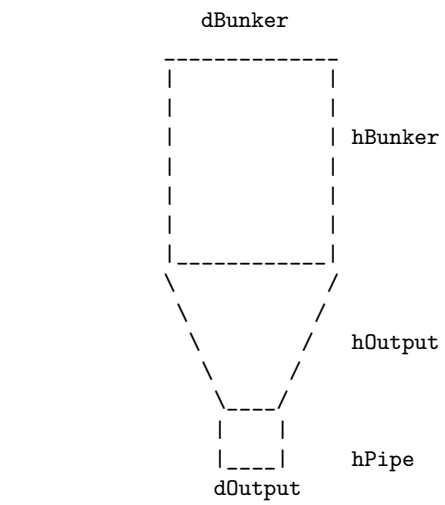
Parameters

- **center** (*Vector3*) – center of the box
- **extents** (*Vector3*) – lengths of the box sides
- **orientation** (*Quaternion*) – orientation of the box
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to utils.facet

Returns list of facets forming the box

`yade.geom.facetBunker`(*center*, *dBunker*, *dOutput*, *hBunker*, *hOutput*, *hPipe=0.0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=4*, *angleRange=None*, *closeGap=False*, ***kw*)

Create arbitrarily-aligned bunker, composed of facets, with given center, radii, heights and orientation. Return List of facets forming the bunker;



Parameters

- **center** (*Vector3*) – center of the created bunker
- **dBunker** (*float*) – bunker diameter, top
- **dOutput** (*float*) – bunker output diameter
- **hBunker** (*float*) – bunker height
- **hOutput** (*float*) – bunker output height
- **hPipe** (*float*) – bunker pipe height
- **orientation** (*Quaternion*) – orientation of the bunker; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the bunker surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of bunker by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetCone`(*center*, *radiusTop*, *radiusBottom*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Create arbitrarily-aligned cone composed of facets, with given center, radius, height and orientation. Return List of facets forming the cone;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusTop** (*float*) – cone top radius
- **radiusBottom** (*float*) – cone bottom radius
- **radiusTopInner** (*float*) – inner radius of cones top, -1 by default
- **radiusBottomInner** (*float*) – inner radius of cones bottom, -1 by default
- **height** (*float*) – cone height
- **orientation** (*Quaternion*) – orientation of the cone; the reference orientation has axis along the +x axis.

- **segmentsNumber** (*int*) – number of edges on the cone surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of cone by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetCylinder`(*center*, *radius*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Create arbitrarily-aligned cylinder composed of facets, with given center, radius, height and orientation. Return List of facets forming the cylinder;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radius** (*float*) – cylinder radius
- **height** (*float*) – cylinder height
- **radiusTopInner** (*float*) – inner radius of cylinders top, -1 by default
- **radiusBottomInner** (*float*) – inner radius of cylinders bottom, -1 by default
- **orientation** (*Quaternion*) – orientation of the cylinder; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the cylinder surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of bunker by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetCylinderConeGenerator`(*center*, *radiusTop*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusBottom=-1*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Please, do not use this function directly! Use geom.facetCylinder and geom.facetCone instead. This is the base function for generating cylinders and cones from facets. :param float radiusTop: top radius :param float radiusBottom: bottom radius :param **kw: (unused keyword arguments) passed to utils.facet;

`yade.geom.facetHelix`(*center*, *radiusOuter*, *pitch*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned helix composed of facets, with given center, radius (outer and inner), pitch and orientation. Return List of facets forming the helix;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)

- **orientation** (*Quaternion*) – orientation of the helix; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the helix surface (>=3)
- **angleRange** (*(ϑmin,Θmax)*) – range of angles; if `None`, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetParallelepiped(`*center*, *extents*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*`)`

Create arbitrarily-aligned Parallelepiped composed of facets, with given center, extents, height and orientation. If any of the parallelepiped dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the parallelepiped.

Parameters

- **center** (*Vector3*) – center of the parallelepiped
- **extents** (*Vector3*) – lengths of the parallelepiped sides
- **height** (*Real*) – height of the parallelepiped (along axis z)
- **orientation** (*Quaternion*) – orientation of the parallelepiped
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to utils.facet

Returns list of facets forming the parallelepiped

`yade.geom.facetPolygon(`*center*, *radiusOuter*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*`)`

Create arbitrarily-aligned polygon composed of facets, with given center, radius (outer and inner) and orientation. Return List of facets forming the polygon;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)
- **orientation** (*Quaternion*) – orientation of the polygon; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the polygon surface (>=3)
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of polygon by specifying range of angles; if `None`, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetPolygonHelixGenerator(`*center*, *radiusOuter*, *pitch=0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*`)`

Please, do not use this function directly! Use geom.facetPloygon and geom.facetHelix instead. This is the base function for generating polygons and helixes from facets.

`yade.geom.facetSphere(`*center*, *radius*, *thetaResolution=8*, *phiResolution=8*, *returnElementMap=False*, ***kw*`)`

Create arbitrarily-aligned sphere composed of facets, with given center, radius and orientation. Return List of facets forming the sphere. Parameters inspired by ParaView sphere glyph

Parameters

- **center** (*Vector3*) – center of the created sphere
- **radius** (*float*) – sphere radius
- **thetaResolution** (*int*) – number of facets around “equator”

- **phiResolution** (*int*) – number of facets between "poles" + 1
- **returnElementMap** (*bool*) – returns also tuple of nodes ((x1,y1,z1),(x2,y2,z2),...) and elements ((id01,id02,id03),(id11,id12,id13),...) if true, only facets otherwise
- ****kw** – (unused keyword arguments) passed to utils.facet;

2.4 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

2.5 yade.pack module

Creating packings and filling volumes defined by boundary representation or constructive solid geometry.

For examples, see

- 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/packs/packs.py
- examples/gts-horse/gts-horse.py
- examples/WireMatPM/wirepackings.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 (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
```

yade.pack.**filterSpherePack**(*predicate*, *spherePack*, *returnSpherePack=None*, ***kw*)
Using given SpherePack instance, return spheres that satisfy predicate. It returns either a pack.SpherePack (if returnSpherePack) or a list. 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.

yade.pack.**hexaNet**(*radius*, *cornerCoord=[0, 0, 0]*, *xLength=1.0*, *yLength=0.5*, *mos=0.08*, *a=0.04*, *b=0.04*, *startAtCorner=True*, *isSymmetric=False*, ***kw*)
Definition of the particles for a hexagonal wire net in the x-y-plane for the WireMatPM.

Parameters

- **radius** – radius of the particle
- **cornerCoord** – coordinates of the lower left corner of the net
- **xLenght** – net length in x-direction
- **yLenght** – net length in y-direction
- **mos** – mesh opening size (horizontal distance between the double twists)
- **a** – length of double-twist
- **b** – height of single wire section
- **startAtCorner** – if true the generation starts with a double-twist at the lower left corner
- **isSymmetric** – defines if the net is symmetric with respect to the y-axis

Returns set of spheres which defines the net (net) and exact dimensions of the net (lx,ly).

note:: This packing works for the WireMatPM only. The particles at the corner are always generated first. For examples on how to use this packing see examples/WireMatPM. In order to create the proper interactions for the net the interaction radius has to be adapted in the simulation.

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()

center() → Vector3

dim() → Vector3

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=False*, *memoDbg=False*, *color=None*, *returnSpherePack=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)), with an uniform distribution. 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 (not 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 examples/gts-horse/gts-random-pack-obb.py
- **memoDbg** – show packings that are considered and reasons why they are rejected/accepted
- **returnSpherePack** – see the corresponding argument in pack.filterSpherePack

Returns SpherePack object with spheres, filtered by the predicate.

yade.pack.**randomPeriPack**(*radius*, *initSize*, *rRelFuzz=0.0*, *memoizeDb=None*, *noPrint=False*)
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`(*(object)arg1*[, *(list)list*])

Set of spheres represented as centers and radii. This class is returned by pack.randomDensePack, pack.randomPeriPack and others. The object supports iteration over spheres, as in

```
>>> sp=SpherePack()
>>> for center,radius in sp: print center,radius

>>> for sphere in sp: print sphere[0],sphere[1]   ## same, but without unpacking the tuple automatically

>>> for i in range(0,len(sp)): print sp[i][0], sp[i][1]   ## same, but accessing spheres by index
```

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.

`isPeriodic`

was the packing generated in periodic boundaries?

`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=0.65*[, *(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 (also works in 2D if minCorner[k]=maxCorner[k] for one coordinate). Sphere radius distribution can be specified using one of the following ways:

1. *rMean*, *rRelFuzz* and *num* gives uniform radius distribution in $rMean \times (1 \pm rRelFuzz)$. 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)seed=0*]]]) → 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 attempts to place a new clump randomly is attained. :param seed: number used to initialize the random number generator.

`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

Not working. Use makeCloud instead.

`particleSD2`(*(object)radii*, *(object)passing*, *(int)numSph*[, *(bool)periodic=False*[, *(float)cloudPorosity=0.8*[, *(int)seed=0*]]]) → int

Not working. Use makeCloud instead.

`particleSD_2d`(*(Vector2)minCorner*, *(Vector2)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=[]*[, *(object)passing=[]*[, *(bool)passingIsNotPercentageButCount=False*[, *(int)seed=0*]]]]) → int

Not working. Use makeCloud instead.

`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`.

`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 (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
```

translate(*(Vector3)arg2*) → None
Translate all spheres by given vector.

class yade._packSpheres.**SpherePackIterator**(*(object)arg1*, *(SpherePackIterator)arg2*)

__init__(*(SpherePackIterator)arg2*) → None

next() → tuple

Spatial predicates for volumes (defined analytically or by triangulation).

class yade._packPredicates.**Predicate**(*(object)arg1*)

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

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**PredicateDifference**(*(object)arg1*, *(object)arg2*, *(object)arg3*)
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`.

A

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class yade._packPredicates.**PredicateIntersection**(*(object)arg1*, *(object)arg2*, *(object)arg3*)
Intersection (conjunction) of 2 predicates. A point has to be inside both predicates. Can be constructed using the `&` operator on predicates: `pred1 & pred2`.

`A`

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class yade._packPredicates.**PredicateSymmetricDifference**(*(object)arg1*, *(object)arg2*, *(object)arg3*)
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`.

`A`

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class yade._packPredicates.**PredicateUnion**(*(object)arg1*, *(object)arg2*, *(object)arg3*)
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`.

`A`

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class yade._packPredicates.**inAlignedBox**(*(object)arg1*, *(Vector3)minAABB*, *(Vector3)maxAABB*)
Axis-aligned box predicate

`__init__`(*(Vector3)minAABB*, *(Vector3)maxAABB*) → None
Ctor taking minumum and maximum points of the box (as 3-tuples).

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class yade._packPredicates.**inCylinder**(*(object)arg1*, *(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*)

Cylinder predicate

__init__(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*) → None
Ctor taking centers of the lateral walls (as 3-tuples) and radius.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inEllipsoid**(*(object)arg1*, *(Vector3)centerPoint*, *(Vector3)abc*)

Ellipsoid predicate

__init__(*(Vector3)centerPoint*, *(Vector3)abc*) → None
Ctor taking center of the ellipsoid (3-tuple) and its 3 radii (3-tuple).

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inGtsSurface**(*(object)arg1*, *(object)surface*[, *(bool)noPad*])

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.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

surf
The associated gts.Surface object.

class yade._packPredicates.**inHyperboloid**(*(object)arg1*, *(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*, *(float)skirt*)

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).

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inParallelepiped**(*(object)arg1*, *(Vector3)o*, *(Vector3)a*, *(Vector3)b*, *(Vector3)c*)

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`.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

`dim()` → Vector3

class `yade._packPredicates.inSphere`(*(object)arg1*, *(Vector3)center*, *(float)radius*)
Sphere predicate.

`__init__`(*(Vector3)center*, *(float)radius*) → None
Ctor taking center (as a 3-tuple) and radius

`aabb()` → tuple
aabb((Predicate)arg1) → None

`center()` → Vector3

`dim()` → Vector3

class `yade._packPredicates.notInNotch`(*(object)arg1*, *(Vector3)centerPoint*, *(Vector3)edge*, *(Vector3)normal*, *(float)aperture*)
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.

`aabb()` → tuple
aabb((Predicate)arg1) → None

`center()` → Vector3

`dim()` → Vector3

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` = **{'force': [nan, nan, 1000.0], 'sigma': [12, nan, nan], 'eps': [0.0001, 0.001, nan]}**
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` = **{'i': ('t',), 'i ': ('z1', 'v1')}**
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` = **True**
Enable/disable live plot updating. Disabled by default for now, since it has a few rough edges.

`yade.plot.liveInterval` = **1**
Interval for the live plot updating, in seconds.

`yade.plot.autozoom` = **True**
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`

- note that your yade compilation should present qt4 feature so that figures can be displayed) 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
>>> os.remove('someFile.pdf')
```

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),warnOnce=False),
...    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.polyhedra_utils module

Auxiliary functions for polyhedra

yade.polyhedra_utils.**fillBox**(*mincoord, maxcoord, material, sizemin=[1, 1, 1], sizemax=[1, 1, 1], ratio=[0, 0, 0], seed=None, mask=1*)
fill box [mincoord, maxcoord] by non-overlaping polyhedrons with random geometry and sizes within the range (uniformly distributed) :param Vector3 mincoord: first corner :param Vector3 maxcoord: second corner :param Vector3 sizemin: minimal size of bodies :param Vector3 sizemax: maximal size of bodies :param Vector3 ratio: scaling ratio :param float seed: random seed

yade.polyhedra_utils.**fillBoxByBalls**(*mincoord, maxcoord, material, sizemin=[1, 1, 1], sizemax=[1, 1, 1], ratio=[0, 0, 0], seed=None, mask=1, numpoints=60*)

yade.polyhedra_utils.**polyhedra**(*material, size=Vector3(1,1,1), seed=None, v=[], mask=1, fixed=False, color=[-1, -1, -1]*)
create polyhedra, one can specify vertices directly, or leave it empty for random shape.

Parameters

- **material** (*Material*) – material of new body
- **size** (*Vector3*) – size of new body (see Polyhedra docs)
- **seed** (*float*) – seed for random operations
- **v** (*[Vector3]*) – list of body vertices (see Polyhedra docs)

yade.polyhedra_utils.**polyhedraSnubCube**(*radius, material, centre, mask=1*)

yade.polyhedra_utils.**polyhedraTruncIcosaHed**(*radius, material, centre, mask=1*)

yade.polyhedra_utils.**polyhedralBall**(*radius, N, material, center, mask=1*)
creates polyhedra having N vertices and resembling sphere

Parameters

- **radius** (*float*) – ball radius
- **N** (*int*) – number of vertices
- **material** (*Material*) – material of new body
- **center** (*Vector3*) – center of the new body

yade.polyhedra_utils.**randomColor**(*seed=None*)

yade._polyhedra_utils.**MaxCoord**(*(Shape)arg1, (State)arg2*) → Vector3
returns max coordinates

yade._polyhedra_utils.**MinCoord**(*(Shape)arg1, (State)arg2*) → Vector3
returns min coordinates

yade._polyhedra_utils.**PWaveTimeStep**() → float
Get timestep accoring to the velocity of P-Wave propagation; computed from sphere radii, rigidities and masses.

yade._polyhedra_utils.**PrintPolyhedra**(*(Shape)arg1*) → None
Print list of vertices sorted according to polyhedrons facets.

yade._polyhedra_utils.**PrintPolyhedraActualPos**(*(Shape)arg1, (State)arg2*) → None
Print list of vertices sorted according to polyhedrons facets.

yade._polyhedra_utils.**SieveCurve**() → None
save sieve curve coordinates into file

yade._polyhedra_utils.**SieveSize**(*(Shape)arg1*) → float
returns approximate sieve size of polyhedron

yade._polyhedra_utils.**SizeOfPolyhedra**(*(Shape)arg1*) → Vector3
returns max, middle an min size in perpendicular directions

yade._polyhedra_utils.SizeRatio() → None
save sizes of polyhedra into file

yade._polyhedra_utils.Split(*(Body)arg1*, *(Vector3)arg2*, *(Vector3)arg3*) → None
split polyhedron perpendicularly to given direction through given point

yade._polyhedra_utils.convexHull(*(object)arg1*) → bool

yade._polyhedra_utils.do_Polyhedras_Intersect(*(Shape)arg1*, *(Shape)arg2*, *(State)arg3*, *(State)arg4*) → bool
check polyhedras intersection

yade._polyhedra_utils.fillBoxByBalls_cpp(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*, *(Vector3)arg4*, *(Vector3)arg5*, *(int)arg6*, *(Material)arg7*, *(int)arg8*) → object
Generate non-overlaping 'spherical' polyhedrons in box

yade._polyhedra_utils.fillBox_cpp(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*, *(Vector3)arg4*, *(Vector3)arg5*, *(int)arg6*, *(Material)arg7*) → object
Generate non-overlaping polyhedrons in box

2.8 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.

2.8.1 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`).

2.8.2 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.

2.8.3 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.

2.9 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.bin(`*QTextStream*`)` → QTextStream

`yade.qt.hex(`*QTextStream*`)` → QTextStream

`yade.qt.oct(`*QTextStream*`)` → QTextStream

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(`[*(str)stateFilename='.qglviewer.xml'*]`)` → None
Load display parameters from file saved previously into.

`lookAt`
Point at which camera is directed.

`ortho`
Whether orthographic projection is used; if false, use perspective projection.

`saveSnapshot(`*(str)filename*`)` → None
Save the current view to image file

`saveState(`[*(str)stateFilename='.qglviewer.xml'*]`)` → None
Save display parameters into a file. 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.

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

2.10 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                                             102               2150us                0.02%
"collider"                                                  5              64200us                0.60%
InteractionLoop                                           102            10571887us               98.49%
"combEngine"                                              102               8362us                0.08%
"newton"                                                  102              73166us                0.68%
"cpmStateUpdater"                                           1               9605us                0.09%
PyRunner                                                    1                136us                0.00%
"plotDataCollector"                                         1                291us                0.00%
TOTAL                                                              10733564us               100.00%
```

sample output (compiled with -DENABLE_PROFILING=1 option):

```
Name                                                    Count                 Time            Rel. time
-------------------------------------------------------------------------------------------------------
ForceResetter                                             102               2150us                0.02%
"collider"                                                  5              64200us                0.60%
InteractionLoop                                           102            10571887us               98.49%
  Ig2_Sphere_Sphere_ScGeom                               1222186            1723168us               16.30%
    Ig2_Sphere_Sphere_ScGeom                             1222186            1723168us              100.00%
  Ig2_Facet_Sphere_ScGeom                                  753               1157us                0.01%
```

```
    Ig2_Facet_Sphere_ScGeom                753              1157us              100.00%
  Ip2_CpmMat_CpmMat_CpmPhys              11712             26015us               0.25%
    end of Ip2_CpmPhys                   11712             26015us             100.00%
  Ip2_FrictMat_CpmMat_FrictPhys              0                0us               0.00%
  Law2_ScGeom_CpmPhys_Cpm                3583872           4819289us              45.59%
    GO A                                 1194624           1423738us              29.54%
    GO B                                 1194624           1801250us              37.38%
    rest                                 1194624           1594300us              33.08%
    TOTAL                                3583872           4819289us             100.00%
  Law2_ScGeom_FrictPhys_CundallStrack        0                0us               0.00%
"combEngine"                             102               8362us               0.08%
"newton"                                 102              73166us               0.68%
"cpmStateUpdater"                          1               9605us               0.09%
PyRunner                                   1                136us               0.00%
"plotDataCollector"                        1                291us               0.00%
TOTAL                                                   10733564us             100.00%
```

2.11 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.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).

class `yade.utils.UnstructuredGrid`
EXPERIMENTAL. Class representing triangulated FEM-like unstructured grid. It is used for transfereing data from ad to YADE and external FEM program. The main purpose of this class is to store information about individual grid vertices/nodes coords (since facets stores only coordinates of vertices in local coords) and to avaluate and/or apply nodal forces from contact forces (from actual contact force and contact point the force is distributed to nodes using linear approximation).

TODO rewrite to C++ TODO better docs

Parameters

- **vertices** (*dict*) – dict of {internal vertex label:vertex}, e.g. {5:(0,0,0),22:(0,1,0),23:(1,0,0)}
- **connectivityTable** (*dict*) – dict of {internal element label:[indices of vertices]}, e.g. {88:[5,22,23]}

`build()`

`getForcesOfNodes()`
Computes forces for each vertex/node. The nodal force is computed from contact force and contact point using linear approximation

`resetForces()`

`setPositionsOfNodes()`
Sets new position of nodes and also updates all elements in the simulation

:param [Vector3] newPoss: list of new positions

`setup()`
Sets new information to receiver

:param dict vertices: see constructor for explanation :param dict connectivityTable: see constructor for explanation :param bool toSimulation: if new information should be inserted to Yade simulation (create new bodies or not) :param [[int]]|None bodies: list of list of bodies indices to be appended as clumps (thus no contact detection is done within one body)

`toSimulation()`
Insert all elements to Yade simulation

`updateElements()`
Updates positions of all elements in the simulation

`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.aabbExtrema2d`(*pts*)
Return 2d bounding box for a sequence of 2-tuples.

`yade.utils.aabbWalls`(*extrema=None*, *thickness=0*, *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.avgNumInteractions`(*cutoff=0.0*, *skipFree=False*, *considerClumps=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. When clumps are present, number of particles is the sum of standalone spheres plus the sum of clumps. Clumps are considered in the calculation if cutoff != 0 or skipFree = True. If cutoff=0 (default) and skipFree=False (default) one needs to set considerClumps=True to consider clumps in the calculation.

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.
- **considerClumps** – also consider clumps if cutoff=0 and skipFree=False; for further explanation see above.

`yade.utils.box`(*center*, *extents*, *orientation=Quaternion((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=0.2*, *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.

class `yade.utils.clumpTemplate`

Create a clump template by a list of relative radii and a list of relative positions. Both lists must have the same length.

Parameters

- **relRadii** (*[float,float,...]*) – list of relative radii (minimum length = 2)
- **relPositions** (*[Vector3,Vector3,...]*) – list of relative positions (minimum length = 2)

`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:

```
.. code-block:: python


FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5,label='defaultMat')
```

`yade.utils.facet`(*vertices*, *dynamic=None*, *fixed=True*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-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.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.gridConnection`(*id1*, *id2*, *radius*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*, *cellDist=None*)

`yade.utils.gridNode`(*center*, *radius*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*)

`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.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*, *sphSph=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. If sphSph only sphere-sphere interactions are considered for the 3 directions histograms.

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.polyhedron`(*vertices*, *dynamic=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)
Create polyhedron with given parameters.

Parameters **vertices** (*[[Vector3]]*) – coordinates of vertices in the global coordinate system.

See utils.sphere's documentation for meaning of other parameters.

`yade.utils.psd`(*bins=5*, *mass=True*, *mask=-1*)
Calculates particle size distribution.

Parameters

- **bins** (*int*) – number of bins
- **mass** (*bool*) – if true, the mass-PSD will be calculated
- **mask** (*int*) – Body.mask for the body

Returns

- binsSizes: list of bin's sizes
- binsProc: how much material (in percents) are in the bin, cumulative
- binsSumCum: how much material (in units) are in the bin, cumulative

binsSizes, binsProc, binsSumCum

`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:

```
>>> saveVars('something',a=1,b=2,c=3)
>>> from yade.params.something 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 ‘‘loadVars(‘something’)‘‘and they will be defined in the yade.params.*mark* module. The *loadNow* parameter calls utils.loadVars after saving automatically. If ‘something’ already exists, given variables will be inserted.

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
- **dynamic** (*float*) – deprecated, see “fixed”
- **fixed** (*float*) – generate the body with all DOFs blocked?
- **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
- **wire** – display as wire sphere?
- **highlight** – highlight this body in the viewer?
- **Vector3-or-None** – body’s color, as normalized RGB; random color will be assigned if `None`.

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.tetra(*vertices*, *strictCheck=True*, *dynamic=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)

Create tetrahedron with given parameters.

Parameters

- **vertices** (*[Vector3,Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.
- **strictCheck** (*bool*) – checks vertices order, raise RuntimeError for negative volume

See utils.sphere's documentation for meaning of other parameters.

yade.utils.tetraPoly(*vertices*, *dynamic=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)

Create tetrahedron (actually simple Polyhedra) with given parameters.

Parameters **vertices** (*[Vector3,Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.

See utils.sphere's documentation for meaning of other parameters.

yade.utils.trackPerfomance(*updateTime=5*)

Track perfomance of a simulation. (Experimental) Will create new thread to produce some plots. Useful for track perfomance of long run simulations (in bath mode for example).

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.**uniaxialTestFeatures**(*filename=None*, *areaSections=10*, *axis=-1*, *distFactor=2.2*, ***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.**voxelPorosityTriaxial**(*triax*, *resolution=200*, *offset=0*)

Calculate the porosity of a sample, given the TriaxialCompressionEngine.

A function utils.voxelPorosity is invoked, with the volume of a box enclosed by TriaxialCompressionEngine walls. The additional parameter offset allows using a smaller volume inside the box, where each side of the volume is at offset distance from the walls. By this way it is possible to find a more precise porosity of the sample, since at walls' contact the porosity is usually reduced.

A recommended value of offset is bigger or equal to the average radius of spheres inside.

The value of resolution depends on size of spheres used. It can be calibrated by invoking voxelPorosityTriaxial with offset=0 and comparing the result with TriaxialCompressionEngine.porosity. After calibration, the offset can be set to radius, or a bigger value, to get the result.

Parameters

- **triax** – the TriaxialCompressionEngine handle
- **resolution** – voxel grid resolution
- **offset** – offset distance

Returns the porosity of the sample inside given volume

Example invocation:

```
from yade import utils
rAvg=0.03
TriaxialTest(numberOfGrains=200,radiusMean=rAvg).load()
O.dt=-1
O.run(1000)
O.engines[4].porosity
0.44007807740143889
utils.voxelPorosityTriaxial(O.engines[4],200,0)
0.44055412500000002
```

```
utils.voxelPorosityTriaxial(O.engines[4],200,rAvg)
0.36798199999999998
```

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.**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.**RayleighWaveTimeStep**() → float
Determination of time step according to Rayleigh wave speed of force propagation.

yade._utils.**TetrahedronCentralInertiaTensor**(*(object)arg1*) → Matrix3
TODO

yade._utils.**TetrahedronInertiaTensor**(*(object)arg1*) → Matrix3
TODO

yade._utils.**TetrahedronSignedVolume**(*(object)arg1*) → float
TODO

yade._utils.**TetrahedronVolume**(*(object)arg1*) → float
TODO

yade._utils.**TetrahedronWithLocalAxesPrincipal**(*(Body)arg1*) → Quaternion
TODO

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.**angularMomentum**([*(Vector3)origin=Vector3(0, 0, 0)*]) → Vector3
TODO

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() → list

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: $\sigma_{ij} = \delta_{ik}\sigma_{kj} = x_{i,k}\sigma_{kj} = (x_i\sigma_{kj})_{,k} - x_i\sigma_{kj,k}$.

At equilibrium, the divergence of stress is null: $\sigma_{kj,k} = 0$. Consequently, after divergence theorem: $\frac{1}{V}\int_V \sigma_{ij}dV = \frac{1}{V}\int_V (x_i\sigma_{kj})_{,k}dV = \frac{1}{V}\int_{\partial V} x_i\sigma_{kj}n_k dS = \frac{1}{V}\sum_b x_i^b f_j^b$.

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: $\sigma_{ij}^{equ.} = (1-n)\sigma_{ij}^{solid}$.

This last relation may not be very useful if porosity is not homogeneous. If it happens, one can define the equivalent bulk stress a the particles scale by assigning a volume to each particle. This volume can be obtained from TesselationWrapper (see e.g. [Catalano2014a])

yade._utils.calm([*(int)mask=-1*]) → None

Set translational and rotational velocities of bodies to zero. Applied to all bodies by default. To calm only some bodies, use mask parameter, it will calm only bodies with groupMask compatible to given value

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.

Warning: This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

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 R^3 axes and transformed axes are as small as possible. This function relies on the fact that periodic cell defines by repetition or its corners regular grid of points in R^3; however, all cells generating identical grid are equivalent and can be flipped one over another. This necessiatates adjustment of Interaction.cellDist for interactions that cross boundary and didn't before (or vice versa), and re-initialization of collider. The *flip* argument can be used to specify desired flip: integers, each column for one axis; if zero matrix, best fit (minimizing the angles) is computed automatically.

In c++, this function is accessible as `Shop::flipCell`.

Warning: 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.**getBodyIdsContacts**([*(int)bodyID=0*]) → list

Get a list of body-ids, which contacts the given body.

yade._utils.**getCapillaryStress**([*(float)volume=0*[, *(bool)mindlin=False*]]) → Matrix3

Compute and return Love-Weber capillary stress tensor:

$\sigma_{ij}^{cap} = \frac{1}{V} \sum_b l_i^b f_j^{cap,b}$, where the sum is over all interactions, with l the branch vector (joining centers of the bodies) and f^{cap} is the capillary force. V can be passed to the function. If it is not, it will be equal to one in non-periodic cases, or equal to the volume of the cell in periodic cases. Only the CapillaryPhys interaction type is supported presently. Using this function with physics MindlinCapillaryPhys needs to pass True as second argument.

yade._utils.**getDepthProfiles**(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*) → tuple

Compute and return the particle velocity and solid volume fraction (porosity) depth profile. For each defined cell z, the k component of the average particle velocity reads:

$< v_k >^z = \sum_p V^p v_k^p / \sum_p V^p$,

where the sum is made over the particles contained in the cell, v_k^p is the k component of the velocity associated to particle p, and V^p is the part of the volume of the particle p contained inside the cell. This definition allows to smooth the averaging, and is equivalent to taking into account the center of the particles only when there is a lot of particles in each cell. As for the solid volume fraction, it is evaluated in the same way: for each defined cell z, it reads:

$< \varphi >^z = \frac{1}{V_{cell}} \sum_p V^p$, where V_{cell} is the volume of the cell considered, and V^p is the volume of particle p contained in cell z. This function gives depth profiles of average velocity and solid volume fraction, returning the average quantities in each cell of height dz, from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef+nCell*dz (input parameters).

yade._utils.**getSpheresMass**([*(int)mask=-1*]) → float

Compute the total mass of spheres in the simulation (might crash for now if dynamic bodies are not spheres), mask parameter is considered

yade._utils.**getSpheresVolume**([*(int)mask=-1*]) → float

Compute the total volume of spheres in the simulation (might crash for now if dynamic bodies are not spheres), mask parameter is considered

yade._utils.getSpheresVolume2D([*(int)mask=-1*]) → float
Compute the total volume of discs in the simulation (might crash for now if dynamic bodies are not discs), mask parameter is considered

yade._utils.getStress([*(float)volume=0*]) → Matrix3
Compute and return Love-Weber stress tensor:

$\sigma_{ij} = \frac{1}{V}\sum_b f_i^b l_j^b$, where the sum is over all interactions, with f the contact force and l the branch vector (joining centers of the bodies). Stress is negativ for repulsive contact forces, i.e. compression. V can be passed to the function. If it is not, it will be equal to the volume of the cell in periodic cases, or to the one deduced from utils.aabbDim() in non-periodic cases.

yade._utils.getStressAndTangent([*(float)volume=0*[, *(bool)symmetry=True*]]) → tuple
Compute overall stress of periodic cell using the same equation as function getStress. In addition, the tangent operator is calculated using the equation published in [Kruyt and Rothenburg1998]_:

$$S_{ijkl} = \frac{1}{V}\sum_c (k_n n_i l_j n_k l_l + k_t t_i l_j t_k l_l)$$

Parameters

- **volume** (*float*) – same as in function getStress
- **symmetry** (*bool*) – make the tensors symmetric.

Returns macroscopic stress tensor and tangent operator as py::tuple

yade._utils.getStressProfile(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*, *(object)vPartAverageX*, *(object)vPartAverageY*, *(object)vPartAverageZ*) → tuple
Compute and return the stress tensor depth profile, including the contribution from Love-Weber stress tensor and the dynamic stress tensor taking into account the effect of particles inertia. For each defined cell z, the stress tensor reads:

$\sigma_{ij}^z = \frac{1}{V}\sum_c f_i^c l_j^{c,z} - \frac{1}{V}\sum_p m^p u_i'^p u_j'^p$,

where the first sum is made over the contacts which are contained or cross the cell z, f^c is the contact force from particle 1 to particle 2, and l^{c,z} is the part of the branch vector from particle 2 to particle 1, contained in the cell. The second sum is made over the particles, and u'^p is the velocity fluctuations of the particle p with respect to the spatial averaged particle velocity at this point (given as input parameters). The expression of the stress tensor is the same as the one given in getStress plus the inertial contribution. Apart from that, the main difference with getStress stands in the fact that it gives a depth profile of stress tensor, i.e. from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef+nCell*dz (input parameters), it is computing the stress tensor for each cell of height dz. For the love-Weber stress contribution, the branch vector taken into account in the calculations is only the part of the branch vector contained in the cell considered. To validate the formulation, it has been checked that activating only the Love-Weber stress tensor, and suming all the contributions at the different altitude, we recover the same stress tensor as when using getStress. For my own use, I have troubles with strong overlap between fixed object, so that I made a condition to exclude the contribution to the stress tensor of the fixed objects, this can be desactivated easily if needed (and should be desactivated for the comparison with getStress).

yade._utils.getViscoelasticFromSpheresInteraction(*(float)tc*, *(float)en*, *(float)es*) → dict
Attention! The function is deprecated! Compute viscoelastic interaction parameters from analytical

solution of a pair spheres collision problem:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$
$$c_n = -\frac{2m}{t_c}\ln e_n$$
$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$
$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t$$

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.growParticle`(*(int)bodyID*, *(float)multiplier*[, *(bool)updateMass=True*]) → None
Change the size of a single sphere (to be implemented: single clump). If updateMass=True, then the mass is updated.

`yade._utils.growParticles`(*(float)multiplier*[, *(bool)updateMass=True*[, *(bool)dynamicOnly=True*]]) → None
Change the size of spheres and sphere clumps by the multiplier. If updateMass=True, then the mass and inertia are updated. dynamicOnly=True will select dynamic bodies.

`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*, *(bool)sphSph*, *(float)minProjLen*) → tuple

`yade._utils.intrsOfEachBody`() → list
returns list of lists of interactions of each body

`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.momentum`() → Vector3
TODO

`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*[, *(bool)splitNormalTensor=False*[, *(float)thresholdForce=nan*]]]) → 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.

Parameters

- **splitNormalTensor** (*bool*) – if true the function returns normal stress tensor split into two parts according to the two subnetworks of strong an weak forces.
- **thresholdForce** (*Real*) – threshold value according to which the normal stress tensor can be split (e.g. a zero value would make distinction between tensile and compressive forces).

yade._utils.**numIntrsOfEachBody**() → list

returns list of number of interactions of each body

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 V. For periodic simulations, current volume of the Cell is used. For aperiodic simulations, the value deduced from utils.aabbDim() is used. For compatibility reasons, positive values passed by the user are also accepted in this case.

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.**setContactFriction**(*(float)angleRad*) → None

Modify the friction angle (in radians) inside the material classes and existing contacts. The friction for non-dynamic bodies is not modified.

yade._utils.**setNewVerticesOfFacet**(*(Body)b*, *(Vector3)v1*, *(Vector3)v2*, *(Vector3)v3*) → None

Sets new vertices (in global coordinates) to given facet.

yade._utils.**setRefSe3**() → None

Set reference positions and orientations of all bodies equal to their current positions and orientations.

yade._utils.**shiftBodies**(*(list)ids*, *(Vector3)shift*) → float

Shifts bodies listed in ids without updating their velocities.

yade._utils.**spiralProject**(*(Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]) → tuple

yade._utils.**sumFacetNormalForces**(*(object)ids*[, *(int)axis=-1*]) → float

Sum force magnitudes on given bodies (must have shape of the Facet type), considering only part of forces perpendicular to each facet's face; if *axis* has positive value, then the specified axis (0=x, 1=y, 2=z) will be used instead of facet's normals.

yade._utils.**sumForces**(*(list)ids*, *(Vector3)direction*) → float

Return summary force on bodies with given *ids*, projected on the *direction* vector.

yade._utils.**sumTorques**(*(list)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 mean 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.**voidratio2D**([*(float)zlen=1*]) → float
Compute 2D packing void ratio $\frac{V-V_s}{V_s}$ where V is overall volume and V_s is volume of disks.

Parameters **zlen** (*float*) – length in the third direction.

yade._utils.**voxelPorosity**([*(int)resolution=200*[, *(Vector3)start=Vector3(0, 0, 0)*[, *(Vector3)end=Vector3(0, 0, 0)*]]]) → float
Compute packing porosity $\frac{V-V_v}{V}$ where V is a specified volume (from start to end) and V_v is volume of voxels that fall inside any sphere. The calculation method is to divide whole volume into a dense grid of voxels (at given resolution), and count the voxels that fall inside any of the spheres. This method allows one to calculate porosity in any given sub-volume of a whole sample. It is properly excluding part of a sphere that does not fall inside a specified volume.

Parameters

- **resolution** (*int*) – voxel grid resolution, values bigger than resolution=1600 require a 64 bit operating system, because more than 4GB of RAM is used, a resolution=800 will use 500MB of RAM.
- **start** (*Vector3*) – start corner of the volume.
- **end** (*Vector3*) – end corner of the volume.

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.

2.12 yade.ymport module

Import geometry from various formats ('import' is python keyword, hence the name 'ymport').

yade.ymport.**ele**(*nodeFileName*, *eleFileName*, *shift=(0, 0, 0)*, *scale=1.0*, ***kw*)
Import tetrahedral mesh from .ele file, return list of created tetrahedrons.

Parameters

- **nodeFileName** (*string*) – name of .node file
- **eleFileName** (*string*) – name of .ele file
- **shift** (*(float,float,float)/Vector3*) – (X,Y,Z) parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to utils.polyhedron

yade.ymport.**gengeo**(*mntable*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)
Imports geometry from LSMGenGeo library and creates spheres. Since 2012 the package is available in Debian/Ubuntu and known as python-demgengeo http://packages.qa.debian.org/p/python-demgengeo.html

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 one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/packs/packs.py, usage of LSMGenGeo library in examples/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. Since 2012 the package is available in Debian/Ubuntu and known as python-demgengeo http://packages.qa.debian.org/p/python-demgengeo.html

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 one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/packs/packs.py, usage of LSMGenGeo library in examples/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.iges(*fileName*, *shift=(0, 0, 0)*, *scale=1.0*, *returnConnectivityTable=False*, ***kw*)
Import triangular mesh from .igs file, return list of created facets.

Parameters

- **fileName** (*string*) – name of iges file
- **shift** (*(float,float,float)|Vector3*) – (X,Y,Z) parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to utils.facet
- **returnConnectivityTable** (*bool*) – if True, apart from facets returns also nodes (list of (x,y,z) nodes coordinates) and elements (list of (id1,id2,id3) element nodes ids). If False (default), returns only 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, returns a list of corresponding bodies; that may be inserted to the simulation with O.bodies.append().

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.textClumps(*fileName*, *shift=Vector3(0, 0, 0)*, *discretization=0*, *orientation=Quaternion((1, 0, 0), 0)*, *scale=1.0*, ***kw*)
Load clumps-members from file, insert them to the simulation.

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_clumpId*
- **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, returns a list of corresponding bodies; that may be inserted to the simulation with O.bodies.append().

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – 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

yade.ymport.**unv**(*fileName*, *shift=(0, 0, 0)*, *scale=1.0*, *returnConnectivityTable=False*, ***kw*)

Import geometry from unv file, return list of created facets.

param string fileName name of unv file

param (float,float,float)|Vector3 shift (X,Y,Z) parameter moves the specimen.

param float scale factor scales the given data.

param **kw (unused keyword arguments) is passed to utils.facet

param bool returnConnectivityTable if True, apart from facets returns also nodes (list of (x,y,z) nodes coordinates) and elements (list of (id1,id2,id3) element nodes ids). If False (default), returns only facets

unv files are mainly used for FEM analyses (are used by OOFEM and Abaqus), but triangular elements can be imported as facets. These files cen be created e.g. with open-source free software Salome.

Example: examples/test/unv-read/unvRead.py.

Bibliography

[yade:background] V. Šmilauer, B. Chareyre (2010), (Yade dem formulation). In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (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: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. (fulltext) (http://yade-dem.org/doc/)

[Bance2014] Bance, S., Fischbacher, J., Schrefl, T., Zins, I., Rieger, G., Cassignol, C. (2014), (Micromagnetics of shape anisotropy based permanent magnets). *Journal of Magnetism and Magnetic Materials* (363), pages 121–124.

[Bonilla2015] Bonilla-Sierra, V., Scholtès, L., Donzé, F.V., Elmouttie, M.K. (2015), (Rock slope stability analysis using photogrammetric data and dfn–dem modelling). *Acta Geotechnica*, pages 1-15. DOI 10.1007/s11440-015-0374-z (fulltext)

[Boon2012a] Boon,C.W., Houlsby, G.T., Utili, S. (2012), (A new algorithm for contact detection between convex polygonal and polyhedral particles in the discrete element method). *Computers and Geotechnics* (44), pages 73 - 82. DOI 10.1016/j.compgeo.2012.03.012 (fulltext)

[Boon2012b] Boon,C.W., Houlsby, G.T., Utili, S. (2013), (A new contact detection algorithm for three-dimensional non-spherical particles). *Powder Technology*. DOI 10.1016/j.powtec.2012.12.040 (fulltext)

[Boon2014] Boon, C.W., Houlsby, G.T., Utili, S. (2014), (New insights into the 1963 vajont slide using 2d and 3d distinct-element method analyses). *Géotechnique* (64), pages 800–816.

[Boon2015] Boon, C.W., Houlsby, G.T., Utili, S. (2015), (A new rock slicing method based on linear programming). *Computers and Geotechnics* (65), pages 12–29.

[Bourrier2013] Bourrier, F., Kneib, F., Chareyre, B., Fourcaud, T. (2013), (Discrete modeling of granular soils reinforcement by plant roots). *Ecological Engineering*. DOI 10.1016/j.ecoleng.2013.05.002 (fulltext)

[Bourrier2015] Bourrier, F., Lambert, S., Baroth, J. (2015), (A reliability-based approach for the design of rockfall protection fences). *Rock Mechanics and Rock Engineering* (48), pages 247–259.

[Catalano2014a] Catalano, E., Chareyre, B., Barthélémy, E. (2014), (Pore-scale modeling of fluid-particles interaction and emerging poromechanical effects). *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 51–71. DOI 10.1002/nag.2198 (fulltext) (http://arxiv.org/pdf/1304.4895.pdf)

[Chareyre2012a] Chareyre, B., Cortis, A., Catalano, E., Barthélemy, E. (2012), (Pore-scale modeling of viscous flow and induced forces in dense sphere packings). *Transport in Porous Media* (92), pages 473-493. DOI 10.1007/s11242-011-9915-6 (fulltext)

[Chen2007] Chen, F., Drumm, E. C., Guiochon, G. (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)

[Chen2011a] Chen, F., Drumm, E., Guiochon G. (2011), (Coupled discrete element and finite volume solution of two classical soil mechanics problems). *Computers and Geotechnics*. DOI 10.1016/j.compgeo.2011.03.009 (fulltext)

[Chen2012] Chen, Jingsong, Huang, Baoshan, Chen, Feng, Shu, Xiang (2012), (Application of discrete element method to superpave gyratory compaction). *Road Materials and Pavement Design* (13), pages 480-500. DOI 10.1080/14680629.2012.694160 (fulltext)

[Chen2014] Chen, J., Huang, B., Shu, X., Hu, C. (2014), (Dem simulation of laboratory compaction of asphalt mixtures using an open source code). *Journal of Materials in Civil Engineering*.

[Dang2010a] Dang, H. K., Meguid, M. A. (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

[Dang2010b] Dang, H. K., Meguid, M. A. (2010), (Evaluating the performance of an explicit dynamic relaxation technique in analyzing non-linear geotechnical engineering problems). *Computers and Geotechnics* (37), pages 125 - 131. DOI 10.1016/j.compgeo.2009.08.004

[Donze2008] Donzé, F.V. (2008), (Impacts on cohesive frictional geomaterials). *European Journal of Environmental and Civil Engineering* (12), pages 967–985.

[Duriez2011] Duriez,J., Darve, F., Donzé, F.V. (2011), (A discrete modeling-based constitutive relation for infilled rock joints). *International Journal of Rock Mechanics & Mining Sciences* (48), pages 458–468. DOI 10.1016/j.ijrmms.2010.09.008

[Duriez2013] Duriez, J., Darve, F., Donzé, F.V. (2013), (Incrementally non-linear plasticity applied to rock joint modelling). *International Journal for Numerical and Analytical Methods in Geomechanics* (37), pages 453–477. DOI 10.1002/nag.1105 (fulltext)

[Dyck2015] Dyck, N. J, Straatman, A.G. (2015), (A new approach to digital generation of spherical void phase porous media microstructures). *International Journal of Heat and Mass Transfer* (81), pages 470–477.

[Elias2014] Jan Elias (2014), (Simulation of railway ballast using crushable polyhedral particles). *Powder Technology* (264), pages 458–465. DOI 10.1016/j.powtec.2014.05.052

[Epifancev2013] Epifancev, K., Nikulin, A., Kovshov, S., Mozer, S., Brigadnov, I. (2013), (Modeling of peat mass process formation based on 3d analysis of the screw machine by the code yade). *American Journal of Mechanical Engineering* (1), pages 73–75. DOI 10.12691/ajme-1-3-3 (fulltext)

[Epifantsev2012] Epifantsev, K., Mikhailov, A., Gladky, A. (2012), (Proizvodstvo kuskovogo torfa, ekstrudirovanie, forma zakhodnoi i kalibriruyushchei chasti fil'ery matritsy, metod diskretnykh elementov [rus]). *Mining informational and analytical bulletin (scientific and technical journal)*, pages 212-219.

[Favier2009a] Favier, L., Daudon, D., Donzé, F.V., Mazars, J. (2009), (Predicting the drag coefficient of a granular flow using the discrete element method). *Journal of Statistical Mechanics: Theory and Experiment* (2009), pages P06012.

[Favier2012] Favier, L., Daudon, D., Donzé, F.V. (2012), (Rigid obstacle impacted by a supercritical cohesive granular flow using a 3d discrete element model). *Cold Regions Science and Technology* (85), pages 232–241. (fulltext)

[Gladky2014] Gladkyy, Anton, Schwarze, Rüdiger (2014), (Comparison of different capillary bridge models for application in the discrete element method). *Granular Matter*, pages 1-10. DOI 10.1007/s10035-014-0527-z (fulltext)

[Grujicic2013] Grujicic, M, Snipes, JS, Ramaswami, S, Yavari, R (2013), (Discrete element modeling and analysis of structural collapse/survivability of a building subjected to improvised explosive device (ied) attack). *Advances in Materials Science and Applications* (2), pages 9–24.

[Guo2014] Guo, Ning, Zhao, Jidong (2014), (A coupled fem/dem approach for hierarchical multiscale modelling of granular media). *International Journal for Numerical Methods in Engineering* (99), pages 789–818. DOI 10.1002/nme.4702 (fulltext)

[Guo2015] N. Guo, J. Zhao (2015), (Multiscale insights into classical geomechanics problems). *International Journal for Numerical and Analytical Methods in Geomechanics*. (under review)

[Gusenbauer2012] Gusenbauer, M., Kovacs, A., Reichel, F., Exl, L., Bance, S., Özelt, H., Schrefl, T. (2012), (Self-organizing magnetic beads for biomedical applications). *Journal of Magnetism and Magnetic Materials* (324), pages 977–982.

[Gusenbauer2014] Gusenbauer, M., Nguyen, H., Reichel, F., Exl, L., Bance, S., Fischbacher, J., Özelt, H., Kovacs, A., Brandl, M., Schrefl, T. (2014), (Guided self-assembly of magnetic beads for biomedical applications). *Physica B: Condensed Matter* (435), pages 21–24.

[Hadda2013] Hadda, Nejib, Nicot, François, Bourrier, Franck, Sibille, Luc, Radjai, Farhang, Darve, Félix (2013), (Micromechanical analysis of second order work in granular media). *Granular Matter* (15), pages 221–235. DOI 10.1007/s10035-013-0402-3 (fulltext)

[Hadda2015] Hadda, N., Nicot, F., Wan, R., Darve, F. (2015), (Microstructural self-organization in granular materials during failure). *Comptes Rendus Mécanique*.

[Harthong2009] Harthong, B., Jerier, J.F., Doremus, 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

[Harthong2012b] Harthong, B., Jerier, J.-F., Richefeu, V., Chareyre, B., Doremus, P., Imbault, D., Donzé, F.V. (2012), (Contact impingement in packings of elastic–plastic spheres, application to powder compaction). *International Journal of Mechanical Sciences* (61), pages 32–43.

[Hartong2012a] Harthong, B., Scholtès, L., Donzé, F.-V. (2012), (Strength characterization of rock masses, using a coupled dem–dfn model). *Geophysical Journal International* (191), pages 467–480. DOI 10.1111/j.1365-246X.2012.05642.x (fulltext)

[Hassan2010] Hassan, A., Chareyre, B., Darve, F., Meyssonier, J., Flin, F. (2010 (submitted)), (Microtomography-based discrete element modelling of creep in snow). *Granular Matter*.

[Hilton2013] Hilton, J. E., Tordesillas, A. (2013), (Drag force on a spherical intruder in a granular bed at low froude number). *Phys. Rev. E* (88), pages 062203. DOI 10.1103/PhysRevE.88.062203 (fulltext)

[Jerier2009] Jerier, J.-F., Imbault, D.and Donzé, F.V., Doremus, P. (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

[Jerier2010a] Jerier, J.-F., Richefeu, V., Imbault, D., Donzé, F.V. (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] Jerier, J.-F., Hathong, B., Richefeu, V., Chareyre, B., Imbault, D., Donzé, F.-V., Doremus, P. (2010), (Study of cold powder compaction by using the discrete element method). *Powder Technology* (In Press). DOI 10.1016/j.powtec.2010.08.056

[Kozicki2006a] Kozicki, J., Tejchman, J. (2006), (2D lattice model for fracture in brittle materials). *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88. (fulltext)

[Kozicki2007a] Kozicki, J., Tejchman, J. (2007), (Effect of aggregate structure on fracture process in concrete using 2d lattice model"). *Archives of Mechanics* (59), pages 365–384. (fulltext)

[Kozicki2008] Kozicki, J., Donzé, F.V. (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] Kozicki, J., Donzé, F.V. (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)

[Lomine2013] Lominé, F., Scholtès, L., Sibille, L., Poullain, P. (2013), (Modelling of fluid-solid interaction in granular media with coupled lb/de methods: application to piping erosion). *Interna-*

tional Journal for Numerical and Analytical Methods in Geomechanics (37), pages 577-596. DOI 10.1002/nag.1109

[Nicot2011] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), (Failure mechanisms in granular media: a discrete element analysis). *Granular Matter* (13), pages 255-260. DOI 10.1007/s10035-010-0242-3

[Nicot2012] Nicot, F., Sibille, L., Darve, F. (2012), (Failure in rate-independent granular materials as a bifurcation toward a dynamic regime). *International Journal of Plasticity* (29), pages 136-154. DOI 10.1016/j.ijplas.2011.08.002

[Nicot2013a] Nicot, F., Hadda, N., Darve, F. (2013), (Second-order work analysis for granular materials using a multiscale approach). *International Journal for Numerical and Analytical Methods in Geomechanics*. DOI 10.1002/nag.2175

[Nicot2013b] Nicot, F., Hadda, N., Guessasma, M., Fortin, J., Millet, O. (2013), (On the definition of the stress tensor in granular media). *International Journal of Solids and Structures*. DOI 10.1016/j.ijsolstr.2013.04.001 (fulltext)

[Nitka2015] Nitka, M., Tejchman, J. (2015), (Modelling of concrete behaviour in uniaxial compression and tension with dem). *Granular Matter*, pages 1–20.

[Puckett2011] Puckett, J.G., Lechenault, F., Daniels, K.E. (2011), (Local origins of volume fraction fluctuations in dense granular materials). *Physical Review E* (83), pages 041301. DOI 10.1103/PhysRevE.83.041301 (fulltext)

[Sayeed2011] Sayeed, M.A., Suzuki, K., Rahman, M.M., Mohamad, W.H.W., Razlan, M.A., Ahmad, Z., Thumrongvut, J., Seangatith, S., Sobhan, MA, Mofiz, SA, others (2011), (Strength and deformation characteristics of granular materials under extremely low to high confining pressures in triaxial compression). *International Journal of Civil & Environmental Engineering IJCEE-IJENS* (11).

[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 (fulltext)

[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 (fulltext)

[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. (fulltext)

[Scholtes2010] Scholtès, L., Hicher, P.-Y., Sibille, L. (2010), (Multiscale approaches to describe mechanical responses induced by particle removal in granular materials). *Comptes Rendus Mécanique* (338), pages 627–638. DOI 10.1016/j.crme.2010.10.003 (fulltext)

[Scholtes2011] Scholtès, L., Donzé, F.V., Khanal, M. (2011), (Scale effects on strength of geomaterials, case study: coal). *Journal of the Mechanics and Physics of Solids* (59), pages 1131–1146. DOI 10.1016/j.jmps.2011.01.009 (fulltext)

[Scholtes2012] Scholtès, L., Donzé, F.V. (2012), (Modelling progressive failure in fractured rock masses using a 3d discrete element method). *International Journal of Rock Mechanics and Mining Sciences* (52), pages 18–30. DOI 10.1016/j.ijrmms.2012.02.009 (fulltext)

[Scholtes2013] Scholtès, L., Donzé, F.V. (2013), (A DEM model for soft and hard rocks: role of grain interlocking on strength). *Journal of the Mechanics and Physics of Solids* (61), pages 352–369. DOI 10.1016/j.jmps.2012.10.005 (fulltext)

[Scholtes2015a] Scholtès, L., Chareyre, B., Michallet, H., Catalano, E., Marzougui, D. (2015), (Modeling wave-induced pore pressure and effective stress in a granular seabed). *Continuum Mechanics and Thermodynamics* (27), pages 305–323. DOI http://dx.doi.org/10.1007/s00161-014-0377-2

[Scholtes2015b] Scholtès, L., Donzé, F., V. (2015), (A dem analysis of step-path failure in jointed rock slopes). *Comptes rendus - Mécanique* (343), pages 155–165. DOI http://dx.doi.org/10.1016/j.crme.2014.11.002

[Shiu2008] Shiu, W., Donzé, F.V., Daudeville, L. (2008), (Compaction process in concrete during missile impact: a dem analysis). *Computers and Concrete* (5), pages 329–342.

[Shiu2009] Shiu, W., Donzé, F.V., Daudeville, L. (2009), (Discrete element modelling of missile impacts on a reinforced concrete target). *International Journal of Computer Applications in Technology* (34), pages 33–41.

[Sibille2014] Sibille, L., Lominé, F., Poullain, P., Sail, Y., Marot, D. (2014), (Internal erosion in granular media: direct numerical simulations and energy interpretation). *Hydrological Processes*. DOI 10.1002/hyp.10351 (fulltext) (First published online Oct. 2014)

[Sibille2015] Sibille, L., Hadda, N., Nicot, F., Tordesillas, A., Darve, F. (2015), (Granular plasticity, a contribution from discrete mechanics). *Journal of the Mechanics and Physics of Solids* (75), pages 119–139. DOI 10.1016/j.jmps.2014.09.010 (fulltext)

[Smilauer2006] Václav Šmilauer (2006), (The splendors and miseries of yade design). *Annual Report of Discrete Element Group for Hazard Mitigation*. (fulltext)

[Thoeni2013] K. Thoeni, C. Lambert, A. Giacomini, S.W. Sloan (2013), (Discrete modelling of hexagonal wire meshes with a stochastically distorted contact model). *Computers and Geotechnics* (49), pages 158–169. DOI 10.1016/j.compgeo.2012.10.014 (fulltext)

[Thoeni2014] K. Thoeni, A. Giacomini, C. Lambert, S.W. Sloan, J.P. Carter (2014), (A 3D discrete element modelling approach for rockfall analysis with drapery systems). *International Journal of Rock Mechanics and Mining Sciences* (68), pages 107–119. DOI 10.1016/j.ijrmms.2014.02.008 (fulltext)

[Tong2012] Tong, A.-T., Catalano, E., Chareyre, B. (2012), (Pore-scale flow simulations: model predictions compared with experiments on bi-dispersed granular assemblies). *Oil & Gas Science and Technology - Rev. IFP Energies nouvelles*. DOI 10.2516/ogst/2012032 (fulltext)

[Tran2011] Tran, V.T., Donzé, F.V., Marin, P. (2011), (A discrete element model of concrete under high triaxial loading). *Cement and Concrete Composites*.

[Tran2012] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), (An algorithm for the propagation of uncertainty in soils using the discrete element method). *The Electronic Journal of Geotechnical Engineering*. (fulltext)

[Tran2012c] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), (Discrete element and experimental investigations of the earth pressure distribution on cylindrical shafts). *International Journal of Geomechanics*. DOI 10.1061/(ASCE)GM.1943-5622.0000277

[Tran2013] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2013), (A finite–discrete element framework for the 3d modeling of geogrid–soil interaction under pullout loading conditions). *Geotextiles and Geomembranes* (37), pages 1-9. DOI 10.1016/j.geotexmem.2013.01.003

[Tran2014] Tran, VDH, Meguid, MA, Chouinard, LE (2014), (Three-dimensional analysis of geogrid-reinforced soil using a finite-discrete element framework). *International Journal of Geomechanics*.

[Wan2014] Wan, R, Khosravani, S, Pouragha, M (2014), (Micromechanical analysis of force transport in wet granular soils). *Vadose Zone Journal* (13).

[Wang2014] Wang, XiaoLiang, Li, JiaChun (2014), (Simulation of triaxial response of granular materials by modified dem). *Science China Physics, Mechanics & Astronomy* (57), pages 2297–2308.

[Zhao2015] J. Zhao, N. Guo (2015), (The interplay between anisotropy and strain localisation in granular soils: a multiscale insight). *Géotechnique*. (under review)

[kozicki2014] Kozicki, Jan, Tejchman, Jacek, Mühlhaus, Hans-Bernd (2014), (Discrete simulations of a triaxial compression test for sand by dem). *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 1923–1952.

[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)

[Catalano2012] Emanuele Catalano (2012), (A pore-scale coupled hydromechanical model for biphasic granular media). PhD thesis at *Université de Grenoble*. (fulltext)

[Charlas2013] Benoit Charlas (2013), (Etude du comportement mécanique d'un hydrure intermétallique utilisé pour le stockage d'hydrogène). PhD thesis at *Université de Grenoble*. (fulltext)

[Chen2009a] Chen, F. (2009), (Coupled flow discrete element method application in granular porous media using open source codes). PhD thesis at *University of Tennessee, Knoxville*. (fulltext)

[Chen2011b] Chen, J. (2011), (Discrete element method (dem) analyses for hot-mix asphalt (hma) mixture compaction). PhD thesis at *University of Tennessee, Knoxville*. (fulltext)

[Duriez2009a] J. Duriez (2009), (Stabilité des massifs rocheux : une approche mécanique). PhD thesis at *Institut polytechnique de Grenoble*. (fulltext)

[Favier2009c] Favier, L. (2009), (Approche numérique par éléments discrets 3d de la sollicitation d'un écoulement granulaire sur un obstacle). PhD thesis at *Université Grenoble I – Joseph Fourier*.

[Guo2014c] N. Guo (2014), (Multiscale characterization of the shear behavior of granular media). PhD thesis at *The Hong Kong University of Science and Technology*.

[Jerier2009b] Jerier, J.F. (2009), (Modélisation de la compression haute densité des poudres métalliques ductiles par la méthode des éléments discrets (in french)). PhD thesis at *Université Grenoble I – Joseph Fourier*. (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)

[Marzougui2011] Marzougui, D. (2011), (Hydromechanical modeling of the transport and deformation in bed load sediment with discrete elements and finite volume). Master thesis at *Ecole Nationale d'Ingénieur de Tunis*. (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)

[Tran2011b] Van Tieng TRAN (2011), (Structures en béton soumises à des chargements mécaniques extrêmes: modélisation de la réponse locale par la méthode des éléments discrets (in french)). PhD thesis at *Université Grenoble I – Joseph Fourier*. (fulltext)

[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-Marroquin, R. Garcia-Rojo, H.J. Herrmann (2004), (Micro-mechanical investigation of the granular ratcheting). In *Cyclic Behaviour of Soils and Liquefaction Phenomena*. (fulltext)

[Antypov2011] D. Antypov, J. A. Elliott (2011), (On an analytical solution for the damped hertzian spring). *EPL (Europhysics Letters)* (94), pages 50004. (fulltext)

[Bagi2006] Katalin Bagi (2006), (Analysis of microstructural strain tensors for granular assemblies). *International Journal of Solids and Structures* (43), pages 3166 - 3184. DOI 10.1016/j.ijsolstr.2005.07.016

[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.

[Calvetti1997] Calvetti, F., Combe, G., Lanier, J. (1997), (Experimental micromechanical analysis of a 2d granular material: relation between structure evolution and loading path). *Mechanics of Cohesive-frictional Materials* (2), pages 121–163.

[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.

[Chan2011] D. Chan, E. Klaseboer, R. Manica (2011), (Film drainage and coalescence between deformable drops and bubbles.). *Soft Matter* (7), pages 2235-2264.

[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), (Modélisation du comportement d'ouvrages composites sol-géosynthétique par éléments discrets - application aux tranchées d'ancrage en tête 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

[Dallavalle1948] J. M. DallaValle (1948), (Micrometrics : the technology of fine particles). Pitman Pub. Corp.

[DeghmReport2006] F. V. Donzé (ed.), (Annual report 2006) (2006). *Discrete Element Group for Hazard Mitigation*. Université Joseph Fourier, Grenoble (fulltext)

[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.

[GarciaRojo2004] R. García-Rojo, S. McNamara, H. J. Herrmann (2004), (Discrete element methods for the micro-mechanical investigation of granular ratcheting). In *Proceedings ECCOMAS 2004*. (fulltext)

[Hentz2003] Séebastien Hentz (2003), (Modélisation d'une structure en béton armé soumise à un choc par la méthode des eléments discrets). PhD thesis at *Université Grenoble 1 – Joseph Fourier*.

[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.

[Herminghaus2005] Herminghaus, S. (2005), (Dynamics of wet granular matter). *Advances in Physics* (54), pages 221-261. DOI 10.1080/00018730500167855 (fulltext)

[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

[Ivars2011] Diego Mas Ivars, Matthew E. Pierce, Caroline Darcel, Juan Reyes-Montes, David O. Potyondy, R. Paul Young, Peter A. Cundall (2011), (The synthetic rock mass approach for jointed rock mass modelling). *International Journal of Rock Mechanics and Mining Sciences* (48), pages 219 - 244. DOI 10.1016/j.ijrmms.2010.11.014

[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

[Kettner2011] Lutz Kettner, Andreas Meyer, Afra Zomorodian (2011), (Intersecting sequences of dD iso-oriented boxes). In *CGAL User and Reference Manual*. (fulltext)

[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)

[Lambert2008] Lambert, Pierre, Chau, Alexandre, Delchambre, Alain, Régnier, Stéphane (2008), (Comparison between two capillary forces models). *Langmuir* (24), pages 3157–3163.

[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)

[Lucy1977] Lucy, L.~B. (1977), (A numerical approach to the testing of the fission hypothesis). *aj* (82), pages 1013-1024. DOI 10.1086/112164 (fulltext)

[Luding2008] Stefan Luding (2008), (Introduction to discrete element methods). In *European Journal of Environmental and Civil Engineering*.

[Luding2008b] Luding, Stefan (2008), (Cohesive, frictional powders: contact models for tension). *Granular Matter* (10), pages 235-246. DOI 10.1007/s10035-008-0099-x (fulltext)

[Magnier1998a] S.A. Magnier, F.V. Donzé (1998), (Numerical simulation of impacts using a discrete element method). *Mech. Cohes.-fric. Mater.* (3), pages 257–276. DOI 10.1002/(SICI)1099-1484(199807)3:3<257::AID-CFM50>3.0.CO;2-Z

[Mani2013] Mani, Roman, Kadau, Dirk, Herrmann, HansJ. (2013), (Liquid migration in sheared unsaturated granular media). *Granular Matter* (15), pages 447-454. DOI 10.1007/s10035-012-0387-3 (fulltext)

[McNamara2008] S. McNamara, R. García-Rojo, H. J. Herrmann (2008), (Microscopic origin of granular ratcheting). *Physical Review E* (77). DOI 11.1103/PhysRevE.77.031304

[Monaghan1985] Monaghan, J.~J., Lattanzio, J.~C. (1985), (A refined particle method for astrophysical problems). *aap* (149), pages 135-143. (fulltext)

[Monaghan1992] Monaghan, J.~J. (1992), (Smoothed particle hydrodynamics). *araa* (30), pages 543-574. DOI 10.1146/annurev.aa.30.090192.002551

[Morris1997] (1997), (Modeling low reynolds number incompressible flows using {sph}). *Journal of Computational Physics* (136), pages 214 - 226. DOI http://dx.doi.org/10.1006/jcph.1997.5776 (fulltext) ()

[Mueller2003] Müller, Matthias, Charypar, David, Gross, Markus (2003), (Particle-based fluid simulation for interactive applications). In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. (fulltext)

[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.

[Pion2011] Sylvain Pion, Monique Teillaud (2011), (3D triangulations). In *CGAL User and Reference Manual*. (fulltext)

[Potyondy2004] D.O. Potyondy, P.A. Cundall (2004), (A bonded-particle model for rock). *International Journal of Rock Mechanics and Mining Sciences* (41), pages 1329 - 1364. DOI 10.1016/j.ijrmms.2004.09.011

[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)

[Rabinov2005] RABINOVICH Yakov I., ESAYANUR Madhavan S., MOUDGIL Brij M. (2005), (Capillary forces between two spheres with a fixed volume liquid bridge : theory and experiment). *Langmuir* (21), pages 10992–10997. (fulltext) (eng)

[Radjai2011] Radjai, F., Dubois, F. (2011), (Discrete-element modeling of granular materials). John Wiley & Sons. (fulltext)

[RevilBaudard2013] Revil-Baudard, T., Chauchat, J. (2013), (A two-phase model for sheet flow regime based on dense granular flow rheology). *Journal of Geophysical Research: Oceans* (118), pages 619–634.

[Richarson1954] Richardson, J. F.„ W. N. Zaki (1954), (Sedimentation and fluidization: part i). *Trans. Instn. Chem. Engrs* (32).

[Satake1982] M. Satake (1982), (Fabric tensor in granular materials.). In *Proc., IUTAM Symp. on Deformation and Failure of Granular materials, Delft, The Netherlands*.

[Schmeeckle2007] Schmeeckle, Mark W., Nelson, Jonathan M., Shreve, Ronald L. (2007), (Forces on stationary particles in near-bed turbulent flows). *Journal of Geophysical Research: Earth Surface* (112). DOI 10.1029/2006JF000536 (fulltext)

[Schwager2007] Schwager, Thomas, Pöschel, Thorsten (2007), (Coefficient of restitution and linear–dashpot model revisited). *Granular Matter* (9), pages 465-469. DOI 10.1007/s10035-007-0065-z (fulltext)

[Soulie2006] Soulié, F., Cherblanc, F., El Youssoufi, M.S., Saix, C. (2006), (Influence of liquid bridges on the mechanical behaviour of polydisperse granular materials). *International Journal for Numerical and Analytical Methods in Geomechanics* (30), pages 213–228. DOI 10.1002/nag.476 (fulltext)

[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). *Géotechnique* (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)

[Weigert1999] Weigert, Tom, Ripperger, Siegfried (1999), (Calculation of the liquid bridge volume and bulk saturation from the half-filling angle). *Particle & Particle Systems Characterization* (16), pages 238–242. DOI 10.1002/(SICI)1521-4117(199910)16:5<238::AID-PPSC238>3.0.CO;2-E (fulltext)

[Wiberg1985] Wiberg, Patricia L., Smith, J. Dungan (1985), (A theoretical model for saltating grains in water). *Journal of Geophysical Research: Oceans* (90), pages 7341–7354.

[Willett2000] Willett, Christopher D., Adams, Michael J., Johnson, Simon A., Seville, Jonathan P. K. (2000), (Capillary bridges between two spherical bodies). *Langmuir* (16), pages 9396-9405. DOI 10.1021/la000657y (fulltext)

[Zhou1999536] Y.C. Zhou, B.D. Wright, R.Y. Yang, B.H. Xu, A.B. Yu (1999), (Rolling friction in the dynamic simulation of sandpile formation). *Physica A: Statistical Mechanics and its Applications* (269), pages 536–553. DOI 10.1016/S0378-4371(99)00183-1 (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