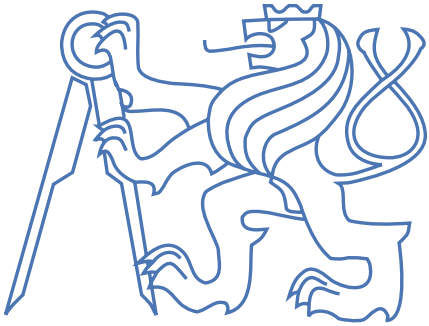
Czech Technical University in Prague



Doctoral Thesis Statement

Czech Technical University in Prague
Faculty of Civil Engineering
Department of Structural Mechanics

Václav Šmilauer

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

Ph.D. Programme: Civil engineering (P3607)
Branch of Study: Physical and material engineering (3911V005)

Doctoral thesis statement for obtaining the degree of "Doctor", abbreviated to "Ph.D."

Prague, June 2010

This thesis was elaborated in a full-time doctoral degree program jointly at the Department of Structural Mechanics, Faculty of Civil Engineering, Czech Technical University in Prague and Laboratoire 3S-R, Université Joseph Fourier, Grenoble.

Student:
Václav Šmilauer
Department of Mechanics
Thákurova 7/2077, CZ-16629 Prague 6

Supervisor:
Milan Jirásek
Department of Mechanics
Thákurova 7/2077, CZ-16629 Prague 6

Supervisor:
Laurent Daudeville
Laboratoire 3S-R
BP 53, F-38041 Grenoble Cedex 9

Contents

1 State of the art

1.1 Discrete models in general

Usage of particle models for mechanical problems originated in geomechanics in 1979, in a famous paper by Cundall & Strack named *A discrete numerical model for granular assemblies* [3]. Granular medium is modeled in a discrete fashion: circular non-deformable particles representing granula can collide, exerting forces on one another, while being governed by Newton's laws of dynamic equilibrium; these laws are integrated using an explicit scheme, proceeding by a given Δt at each step. Particles have both translational and rotational degrees of freedom. Forces coming from collision of particles are computed using penalty functions, which express simple spring-like contact, elastic in the normal sense (connecting both spheres' centers) and elasto-plastic with Mohr-Coulomb criterion in the perpendicular plane.

The Discrete Element Method (DEM) differs from other discrete models, namely

Mass-spring models, where nodes have only 3 degrees of freedom and their contacts only transmit normal force. Mass is lumped into nodes without any associated volume, without collision detection and creation of new contacts; initial contacts are pre-determined. Such models were used to model solid fracture (where dynamic effects were predominant [24]) or elastic cloth behavior [19].

Rigid body-spring model (RBSM), where polygonal/polyhedral particles are connected with multiple spring elements across the neighbor's contact sides/areas; particles have no deformability on their own, their elasticity is represented by said spring elements [9]; an implicit integration scheme is employed. This method is similar to FEM with zero-thickness interface elements [1], but leads to a smaller stiffness matrix, since displacements of any point belonging to a particle are uniquely determined from displacements/rotations of the particle itself. Nagai et al. [16] uses elaborate plasticity functions for shear loading.

Lattice models family, where nodes are connected with truss or beam elements. Typically, nodes carry no mass and static equilibrium

is sought; they do not occupy volume either, hence no new contacts between nodes will be created. Both regular and irregular lattices were studied. Properties of connecting elements are determined from local configuration, such as geometry of the Voronoï cell of each node and local material heterogeneity (e.g. mortar vs. aggregates in concrete). Originally, lattice was representing elastic continuum; the equivalence was established for both truss [8] and beam [21] elements. Later, obvious enhancements such as brittle beam failure were introduced. Lattice models nicely show the *emergence* of relatively complex structural behavior, although fairly simple formulas govern local processes. Some models find themselves on the border between DEM and lattice models, e.g. by considering sphere packing for finding initial contacts, but only finding a static solution later [5].

Cundall's original formulation of DEM was since then substantially enhanced in many ways, which are summarized in Bićanić [1] as follows:

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

Particle geometry Discs (2d) and spheres (3d) were first choices for the ease of contact detection. Approximating more complex shapes by spheres can be done by building up rigid aggregates ("clumps"), which might try to approximate real surfaces [18].

At further development stages, elliptical shapes, general quadrics and implicit superquadrics all have been used. Polygons/polyhedra with explicit vertices are also frequently used, although exact detection of contact might be tricky.

Surface singularities at vertices can be problematic, since direction of the repulsive force is not clearly defined. Several solutions are employed: rounding edges and vertices, replacing them with aligned spheres and cylinders; formulating the repulsive force volumetrically based on the overlap volume; using some more detailed knowledge about the particles in question, such as tracking the *common plane* defined by arrangement of vertices and faces

during movement of particles [17].

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

Contact detection algorithm Naïve checking of all possible couples soon leads to performance issues with increasing number of particles, having $\mathcal{O}(n^2)$ complexity. Moreover, for complex shapes exact contact evaluation can be complicated. Therefore, the detection is generally done in 2 passes. First, possible contacts based on approximate volume representation are sought (most non-trivial algorithms are $\mathcal{O}(n \log n)$, although $\mathcal{O}(n)$ algorithms exist [13, 15]). Second, contacts are evaluated considering the exact geometry of particles (in this pass, all possible combinations of particle geometries must be handled).

Boundary conditions Boundaries can be imposed at space level or at particle level. Space boundaries are, in particular, periodic boundaries, where particles leaving the periodic cell on one side enter on the other side. Particle-level boundaries may be as simple as fixing some particles in space; other boundaries, which aim at a more faithful representation of experimental setups, might be flexible (chain of particles tied together) or hydrostatic (forces exerted on boundary particles).

Particle deformability Early attempts at deformability considered discrete elements as deformable quadrilaterals, while several further development branches were followed later: *Combined finite/discrete element method* (FDEM) [14, 12] discretizes each discrete element internally into multiple finite elements and *Discontinuous deformation analysis* (DDA) [22] superimposes polynomial approximation of the strain field on the movement of the rigid body centroid. *Non-rigid aggregates* account for deformable particles by clustering primitive, rigid particles using a special type of cohesive bonds, creating a lattice-like deformable solid representation.

Cohesion and fracturing Cohesive interactions ("bonds") between particles have been used to represent non-granular media. Fracturing can take place between particle (inter-particle) or through particles, if they are internally discretized using e.g. FEM (intra-

particle).

1.2 Discrete models of concrete

Lattice models of concrete aim usually at meso-scale modeling, where aggregates and mortar are distinguished by differing beam parameters. Beam configuration can be regular (leading to failure anisotropy), random [11], or generated to match observed statistical distribution [10]. Lilliu and van Mier [11] uses brittle beam behavior, simply removing broken beams from simulation; Leite et al. [10] uses tensile softening to avoid dependence of global softening (which is observed even with brittle behavior of beams) on beam size. Neither of these models focuses on capturing compression behavior. Cusatis et al. [4] presents a rather sophisticated model, in which properties of lattice beams are computed from the geometry of Voronoï cells of each node. Granulometry is supposed to have substantial influence on confinement behavior; as it is not fully considered by the model, the beam-transversal stress history influences shear rigidity instead. Compression behavior is captured fairly well.

Discrete element models of concrete are rare, most come from teams around Frédéric V. Donzé. His work first targeted at 2D DEM[2]; later, exploiting the dynamic nature of DEM led to fast concrete dynamics in 3D [6, 7] and impact simulation [23]. In order to reduce computational costs, elastic FEM for the non-damaged subdomain is used, with some tricks to avoid spurious dynamic effects on the DEM/FEM boundary [20].

With both lattice and DEM models, arriving at reasonable compression behavior is non-trivial; it seems to stem from the fact that 1D elements (beams in lattice, bonds in DEM) have no notion of an overall stress state at their vicinity, but only insofar as it is manifested in the direction of the respective element. Cusatis et al. [4] works around this by explicitly introducing the influence of transversal stress. Hentz [6, sect. 5.3], on the other hand, blocks rotations of spherical elements to achieve a higher and more realistic f_c/f_t ratio, but it is questionable whether there is physically sound reason for such an adjustment.

2 Objectives

The research project on cohesive particle model is situated in the field of computational mechanics, which comprises theoretical mechanics and software engineering.

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

Yade, software platform targeting mainly DEM was chosen for the implementation of the concrete model. During my work, it had to be substantially enhanced. Since many of these aspects are interesting from software engineering point of view, they make organic part of this thesis statement and of the thesis itself. The focus was usability, documentation and performance, so that it could become the platform of choice for DEM research due to its flexibility.

The simulations to be run are subject to industrial confidentiality contract and are not shown in the following text. Only the material model itself is presented.

3 Result I: Material model

Computing displacements (and strains) within DEM is described in detail in the thesis. Here we only briefly show formulation of the particle model.

3.1 Normal stress

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

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

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

$$\omega = g(\kappa) = 1 - \frac{\varepsilon_f}{\kappa} \exp\left(-\frac{\kappa - \varepsilon_0}{\varepsilon_f}\right) \tag{2}$$

$$\kappa = \max \tilde{\varepsilon} \tag{3}$$

$$\tilde{\varepsilon} = \langle \varepsilon_N \rangle, \tag{4}$$

where $\tilde{\varepsilon}$ is the equivalent strain responsible for damage ($\langle \varepsilon_N \rangle$ signifies the positive part of ε_N).

The ε_0 parameter is the limit elastic strain, and the product $K_T \varepsilon_0$ corresponds to the local tensile strength at the level of one contact. The ε_f parameter is related to the slope of the softening part of the normal strain-stress diagram (fig. 2) and must be larger than ε_0. To better capture confinement effect, plasticity in compression is introduced. The strain-stress diagram then takes bilinear form in compression, and uses two additional parameters, limit compressive elastic strain $\varepsilon_s < 0$ and relative hardening modulus $\tilde{K}_s$.

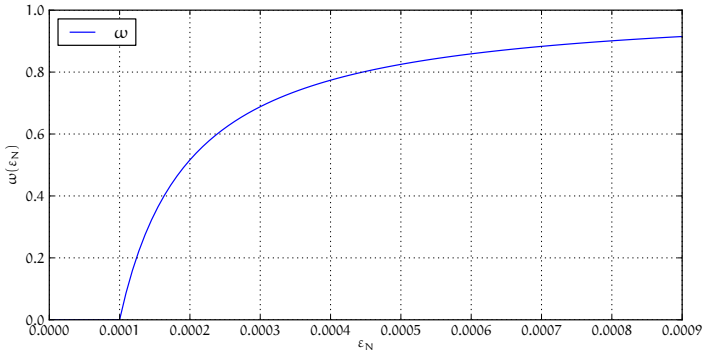


Figure 1: Damage ω evolution function $\omega = g(\kappa_D)$, where $\kappa_D = \max \varepsilon_N$ (using $\varepsilon_0 = 0.0001$, $\varepsilon_f = 30\varepsilon_0$).

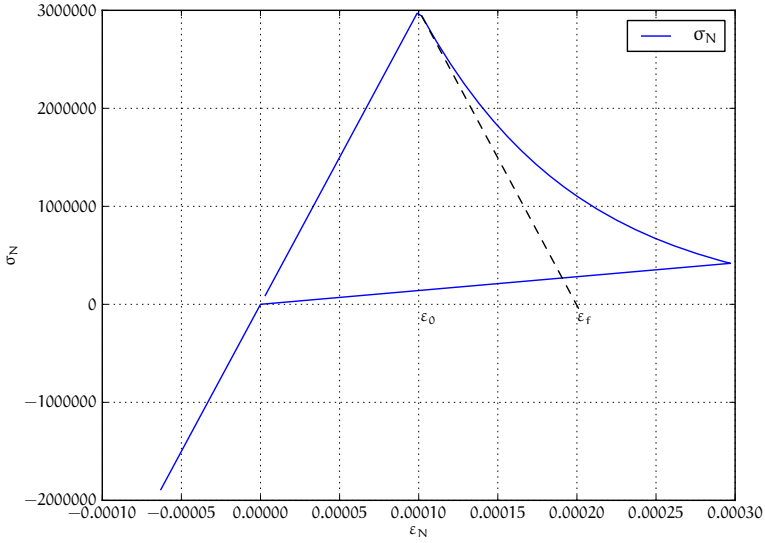


Figure 2: Strain-stress diagram in the normal direction.

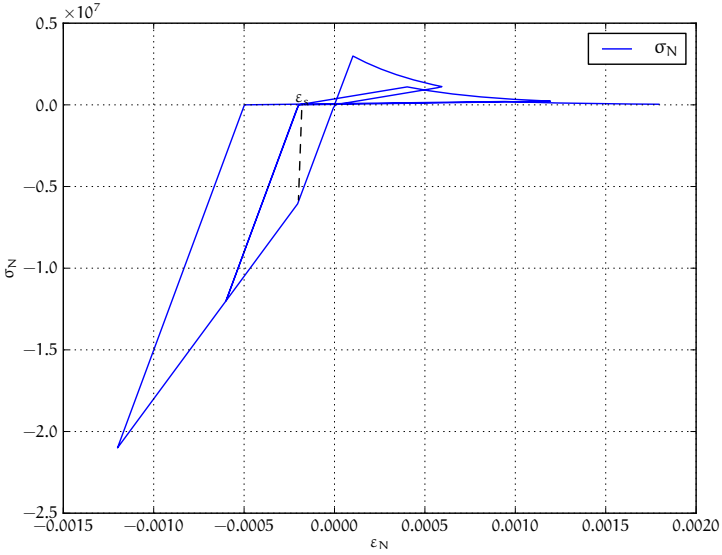


Figure 3: Strain-stress diagram in normal direction, loaded cyclically in tension and compression; it shows (1) damage in tension, no damage in compression (2) plasticity in compression, starting at strain ε_s; reduced (hardening) modulus is $\tilde{K}_s k_N$.

3.1.1 Visco-damage

In order to model time-dependent phenomena, viscosity is introduced in tension by adding viscous overstress $\sigma_{N\nu}$ to (1). As we suppose it to be related to a limited rate of crack propagation, it cannot depend on total strain rate; rather, we split total strain into the elastic strain ε_e and the damage part ε_d. Since $\varepsilon_e = \sigma_N / k_N$, we have

$$\varepsilon_d = \varepsilon_N - \frac{\sigma_N}{k_N} \tag{5}$$

We then postulate the overstress in the form

$$\sigma_{N\nu}(\dot{\varepsilon}_d) = k_N \varepsilon_0 \langle \tau_d \dot{\varepsilon}_{Nd} \rangle^{M_d}, \tag{6}$$

where $k_N \varepsilon_0$ is rate-independent tensile strength (introduced for the sake of dimensionality), τ_d is characteristic time for visco-damage and M_d is a dimensionless exponent. The normal stress equation then reads

$$\sigma_N = [1 - \omega H(\varepsilon_N)]\, k_N \varepsilon_N + \sigma_{N\nu}(\dot{\varepsilon}_{Nd}). \tag{7}$$

Due to mutual dependence of $\dot{\varepsilon}_d$, σ_N and $\sigma_{N\nu}$, the value of $\sigma_{N\nu}$ satisfying the above equations must be found using an iterative procedure.

The effect of viscosity on damage for one contact is shown on fig. 4; calibration of the new parameters τ_d and M_d is described in the thesis.

3.2 Shear stresses

For the shear stress we use plastic constitutive law

$$\boldsymbol{\sigma}_T = k_T(\boldsymbol{\varepsilon}_T - \boldsymbol{\varepsilon}_{Tp}) \tag{8}$$

where ε_{Tp} is the plastic strain on the contact and k_T is shear contact modulus computed from k_N as the ratio k_T/k_N is fixed. The shear stress is limited by the yield function (fig. 5)

$$f(\sigma_N, \boldsymbol{\sigma}_T) = |\boldsymbol{\sigma}_T| - r_{pl} = |\boldsymbol{\sigma}_T - (c_T - \sigma_N \tan\varphi), \quad c_T = c_{T0}(1-\omega) \tag{9}$$

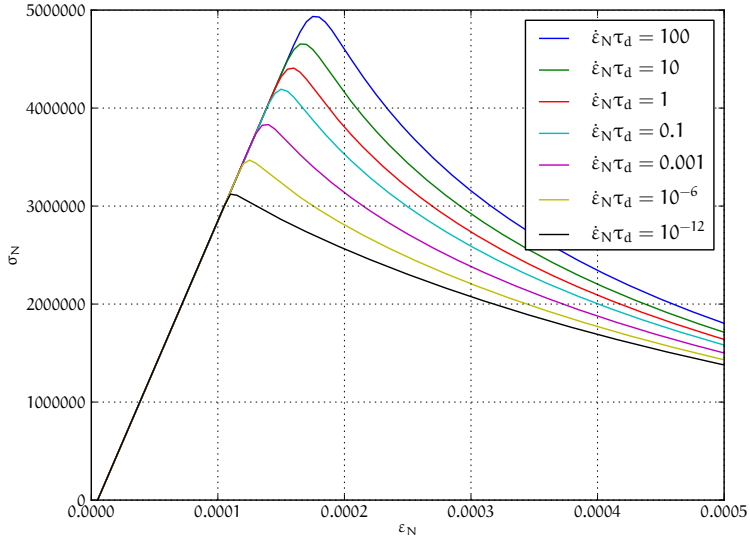


Figure 4: Strain-stress curve in tension with different rates of loading; the parameters used here are $\tau_d = 1\,\mathrm{s}$ and $M_d = 0.1$.

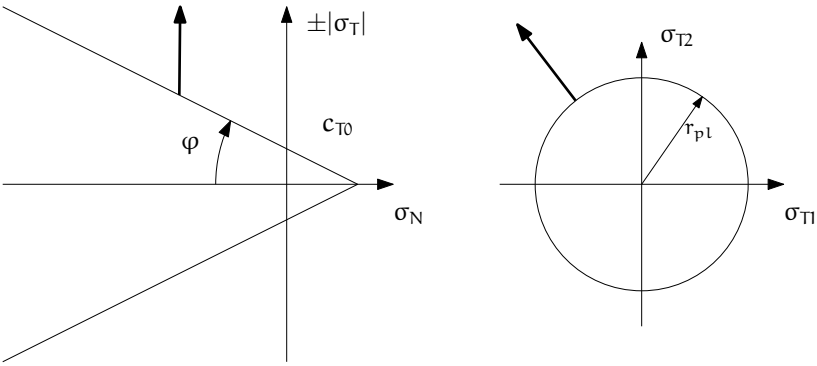


Figure 5: Linear yield surface and plastic flow rule.

where material parameters c_{T0} and φ are initial cohesion and internal friction angle, respectively. The initial cohesion c_{T0} is reduced to the current cohesion c_T using damage state variable ω. Note that we split the plasticity function in a part that depends on $\boldsymbol{\sigma}_T$ and another part which depends on already known values of ω and σ_N; the plastic flow rule makes us of this split, as it does not depend on σ_N, which does not have to be re-evaluated:

$$\dot{\boldsymbol{\varepsilon}}_{Tp} = \dot{\lambda} \frac{\boldsymbol{\sigma}_T}{|\boldsymbol{\sigma}_T|}, \tag{10}$$

λ being plastic multiplier, is associated in the plane of shear stresses (using simple radial stress return) but not with respect to the normal stress (fig. 5). For better simulation of confined conditions, the plastic surface takes more complex form (logarithm in compression), described in detail in the thesis.

3.2.1 Visco-plasticity

Visco-plasticity changes radius of plasticity surface based on rate of plastic slip. It is formally similar to the visco-damage formulation. During calibration, it had not beneficial effects on the results and is not used in the model.

3.3 Calibration

The model comprises relatively many contact-level *parameters*, but among those one can clearly distinguish the elastic ones and non-elastic ones. Material to be simulated is determined by its macroscopic *properties*; they also fall in the elastic and non-elastic categories. We can therefore calibrate elastic parameters to obtain desired macroscopic elastic properties, then continue with non-elastic ones. The non-elastic parameters and properties have further relatively independent subgroups: damage+plasticity, confinement and rate-dependence, which are again calibrated separately. The calibration procedure is described in the thesis in detail, we give only parameter and properties overview at this place.

Model parameters can be summarized as follows:

1. geometry

   r sphere radius
   R_I interaction radius

2. elasticity

   k_N normal contact stiffness
   k_T/k_N relative shear contact stiffness

3. damage and plasticity

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

4. confinement

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

5. rate-dependence

   τ_d characteristic time for visco-damage

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

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

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

   E Young's modulus,
   ν Poisson's ratio

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

   f_t tensile strength
   f_c compressive strength
   G_f fracture energy

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

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

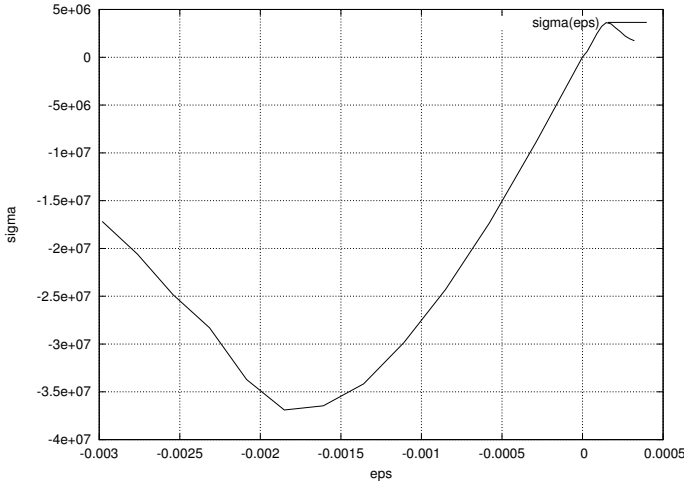


Figure 6: Strain-stress diagram for the calibrated concrete model.

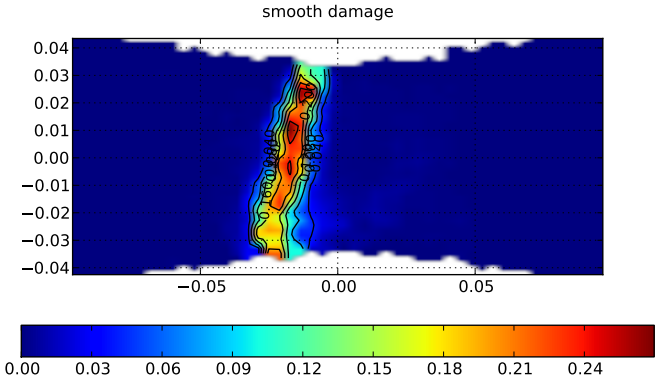


Figure 7: Localization on a dogbone specimen during the uniaxial tension test.

4 Result II: the Yade platform

This section briefly introduces Yade, the program, from the user's perspective. For detailed documentation, the reader is referred to https://www.yade-dem.org/sphinx/. Our work on Yade concentrated on the following:

Python scripting interface using boost::python was the most important and involved change and is shown below.

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

Documentation within the source code, setting up wiki.

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

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

4.1 Basic usage

Running Yade Some prior knowledge is required for operation.

- Basics of command line in your Linux system are necessary for running yade.
- Python language; we recommend the official Python tutorial.

Yade is being run primarily from terminal; the name of command is **yade**.

```
$ yade
Welcome to Yade bzr1984
```

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

The command-line is ipython, python shell with enhanced interactive capabilities; it features persistent history (remembers commands from your last sessions), searching and so on.

Typically, one will not type Yade commands by hand, but use *scripts*, python programs describing and running your simulations. This simple script will just print "Hello world!":

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

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

```
$ yade script.py
# ...
Hello world!
Yade [1]:
```

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

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

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

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

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

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

```
 ->  [5]: 206
```

Generators are regular yade objects that support attribute access.

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

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

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

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

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

Yade [10]: O.step()

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

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

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

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

```
Yade [14]: O.pause()

Yade [15]: O.iter
 ->  [15]: 1550

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

Yade [17]: O.iter
 ->  [17]: 101550

Yade [18]: O.stopAtIter=500000

Yade [19]: O.wait()

Yade [20]: O.iter
 ->  [20]: 101550
```

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

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

Yade [22]: O.reload()

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

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

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

Yade [26]: O.loadTmp()
```

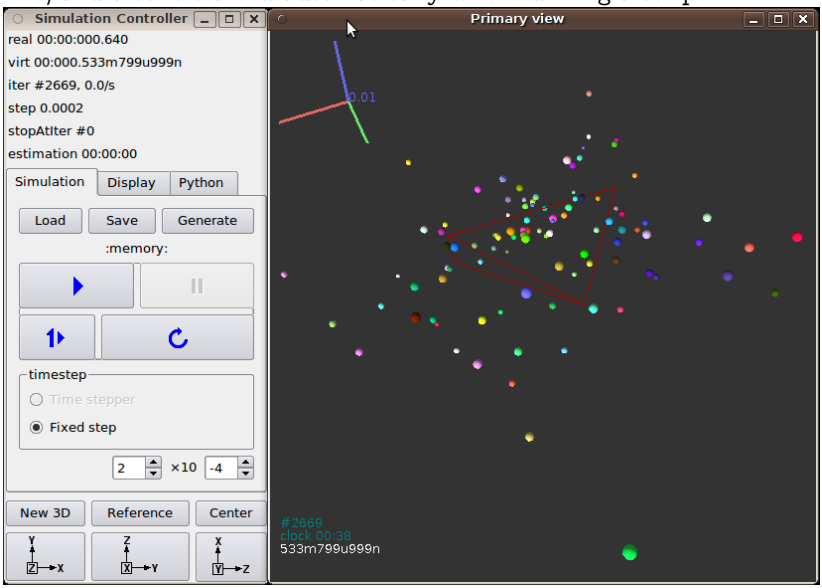
```
Yade [27]: O.saveTmp('init') ## named memory slot

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

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

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

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



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

4.2 Architecture overview

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

4.2.1 Data and functions

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

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

```
Yade [32]: O.bodies       # some data components
 ->  [32]: <yade.wrapper.BodyContainer object at 0x3c51ed8>

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

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

Data components

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

Each Body comprises the following:

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

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

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

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

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

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

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

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

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

Yade [36]: s.shape, s.state, s.mat, s.bound
 ->  [36]:
(<Sphere instance at 0x3da70f0>,
 <State instance at 0x3f50cc0>,
 <FrictMat instance at 0x40c46c0>,
 <Aabb instance at 0x3e1f5a0>)
```

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

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

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

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

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

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

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

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

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

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

IndexError: Body id out of range.
```

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

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

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

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

Bodies can be iterated over using standard python iteration syntax:

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

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

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

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

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

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

```
Yade [48]: O.interactions[0,1]
 ->  [48]: <Interaction instance at 0x3abe360>

Yade [49]: O.interactions[1,0]     # order of ids is not important
```

```
 ->  [49]: <Interaction instance at 0x3abe360>

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

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

Yade [52]: i.geom
 ->  [52]: <Dem3DofGeom_SphereSphere instance at 0x421b9e0>

Yade [53]: i.phys
 ->  [53]: <FrictPhys instance at 0x3fb66b0>

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

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

IndexError: No such interaction
```

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

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

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

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

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

4.2.2 Function components

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

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

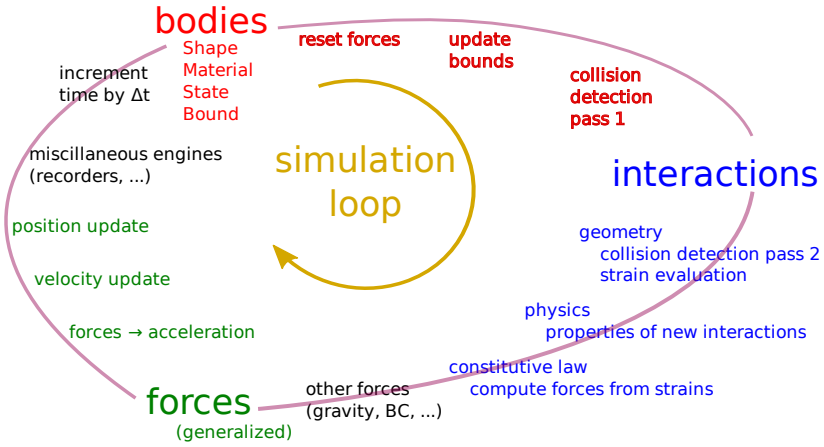


Figure 8: Typical simulation loop; each step begins at body-cented bit at 11 o'clock, continues with interaction bit, force application bit, miscillanea and ends with time update.

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

Engines Simulation loop, shown at img. img-yade-iter-loop, can be described as follows in Python (details will be explained later); each of

the **O.engine** items is instance of a type deriving from Engine:

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

There are 3 fundamental types of Engines:

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

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

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

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

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

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

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

The next part, reading

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

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

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

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

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

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

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

Ip2 functors are derived from InteractionPhysicsFunctor.

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

Law2 functors all inherit from LawFunctor.

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

5 Conclusion

A new particle model of concrete was developed within the DEM framework, based on continuum formulations. It was tested on standard setups and calibration procedures are described in detail. Because the formulation is rather general, the model itself is suitable for use for other cohesive materials by changing numerical parameters. For confidentiality reasons, we did not show any applications of the model. It was shown that particle-based model, although only formulated locally on one-dimensional contact, can capture behavior found in experiments; although there is still long journey towards complete description of continuum behavior, the basis has shown to be solid and worth confidence. Lot of meaningful work can also be done in the analytical description of macroscopic properties based on local parameters, which we only touched lightly; this should permit to better establish mathematical relationship with continuum-based models.

The description of Yade from user and programmer perspective in the thesis (as well as online) are the first comprehensive documents of this kind. Yade API documentation is unique in the sense that it is identical for c++ and Python, the languages used in Yade. Yade as a platform for DEM has seen important rise in usage at several universities during last 3 years. We attribute it mostly to documentation, consistent API and quality of the core code.

There are many possible ways for future development. On the cohesive particle model side, lot of work can be done on analytical (statistical) description of the relationship between dense interaction network and continuum, including transitions between particle-based and continuum-based models; this was actually part of this project, but was only finished in part and the partial results will be handed down to the posterity separately. Concerning Yade as software, the most challenging is to steer individual developers away from the desire for fast results towards a more responsible attitude of community development. Community development is sustainable, while individual ad-hoc solutions are not.

6 Summary

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

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

7 Résumé (in Czech)

Disertační práce popisuje částicový model betonu využívající metodu diskrétních prvků (DEM) a implementovaný na platformě Yade. DEM diskretizuje simulovanou oblast uspořádáním (kulových) částic, jejichž pohyb je určen lokálními kontaktními zákony a Newtonovými rovnicemi. Soudržnost materiálu je modelována prostorově předurčenými kohezivními kontakty mezi částicemi a ztráta soudržnosti vznikne poškozením jednotlivých kontaktů. Beton se uvažuje jako homogenní materiál, přičemž částice jsou pouze diskretizační jednotky (nereprezentují kamenivo, cementovou pastu nebo póry); lokální kontaktní zákon zahrnuje poškození, plasticitu a viskositu, přičemž kalibrace je detailně popsána.

Tento model byl implementován na softwarové platformě Yade, během práce podstatně rozšířené; v disertaci je předložen její první kompletní popis. Program Yade je primárně určen pro explicitní dynamické simulace (zejména DEM) a je navržen jako vysoce modulární stavebnice různých algoritmů. Výpočetní část je napsána v c++ kvůli výkonu a podporuje paralelní výpočet (sdílená paměť). Populární skriptovací jazyk Python se používá k úspornému a efektivnímu popisu simulace, pro její ovládání a post-processing; díky Pythonu je také možno přistupovat k většině interních dat. Platforma vyžaduje určitou programovací kulturu (zejména dokumentaci), což by jí mělo zajistit udržitelný rozvoj do budoucna.

References

[1] N. Bićanić. Discrete Element Methods. In E. Stein, R. de Borst, and T.J.R. Hughes, editors, *Encyclopedia of Computational Mechanics: Fundametals*, pages 311–337. Wiley and Sons, 2004.

[2] F. Camborde, C. Mariotti, and F. V. Donzé. Numerical study of rock and concrete behaviour by discrete element modelling. *Computers and Geotechnics*, 27(4):225–247, 2000. URL http://dx.doi.org/10.1016/S0266-352X(00)00013-6.

[3] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*, 29(1):47–65, 1979.

[4] Gianluca Cusatis, Zdenek P. Bažant, and Luigi Cedolin. Confinement-shear lattice model for concrete damage in tension and compression: I. theory. *Journal of Engineering Mechanics*, 129(12):1439–1448, 2003. URL http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=JENMDT000129000012001439000001&idtype=cvips&gifs=yes.

[5] D. V. Griffiths and G. G. W. Mustoe. Modelling of elastic continua using a grillage of structural elements based on discrete element concepts. *International Journal for Numerical Methods in Engineering*, 50(7):1759–1775, 2001. doi: 10.1002/nme.99. URL http://80.www3.interscience.wiley.com.dialog.cvut.cz/journal/76509666/abstract.

[6] Sébastien Hentz. *Modélisation d'une Structure en Béton Armé Soumise à un Choc par la méthode des Éléments Discrets*. PhD thesis, Université Grenoble 1 – Joseph Fourier, October 2003.

[7] Sébastien Hentz, Laurent Daudeville, and Frédéric V. Donzé. Identification and validation of a discrete element model for concrete. *Journal of Engineering Mechanics*, 130(6):709–719, June 2004.

[8] Alexander Hrennikoff. Solution of problems of elasticity by the frame-work method. *ASME Journal of Applied Mechanics*, (8): A619–A715, 1941.

[9] Tadahiko Kawai. New element models in discrete structural analysis. *Journal of the Society of Naval Architects of Japan*, (141):

174–180, 19770600. ISSN 05148499. URL http://ci.nii.ac.jp/naid/110003878089/en/.

[10] J. P. B. Leite, V. Slowik, and H. Mihashi. Computer simulation of fracture processes of concrete using mesolevel models of lattice structures. *Cement and Concrete Research*, 34(6):1025–1033, 2004. doi: DOI:10.1016/j.cemconres.2003.11.011. URL http://www.sciencedirect.com/science/article/B6TWG-4B8X294-2/2/51f72ac6eb39cfeaf744c5980dd2fc2f.

[11] G. Lilliu and J. G. M. van Mier. 3d lattice type fracture model for concrete. *Engineering Fracture Mechanics*, 70(7–8):927–941, 2003. ISSN 0013-7944. doi: DOI:10.1016/S0013-7944(02)00158-3. URL http://www.sciencedirect.com/science/article/B6V2R-47DM661-2/2/b3ec6fb13217ef0e8f7a854f6aa166de.

[12] A. Munjiza. *The Combined Finite-Discrete Element Method*. John Wiley & Sons, Ltd, 2004.

[13] A. Munjiza and K. R. F. Andrews. NBS contact detection algorithm for bodies of similar size. *International Journal for Numerical Methods in Engineering*, 43(1):131–149, 1998. doi: 10.1002/(SICI)1097-0207(19980915)43:1<131::AID-NME447>3.0.CO;2-S. URL http://www3.interscience.wiley.com/journal/10005234/abstract.

[14] A. Munjiza, D. R. J. Owen, and N. Bićanić. A combined finite-discrete element method in transient dynamics of fracuring solids. *Engineering Computations*, 12:145–174, 1995.

[15] A. Munjiza, E. Rougier, and N. W. M. John. MR linear contact detection algorithm. *International Journal for Numerical Methods in Engineering*, 66(1):46–71, 2006. doi: 10.1002/nme.1538. URL http://dx.doi.org/10.1002/nme.1538.

[16] Kouhei Nagai, Yasuhiko Sato, Tamon Ueda, and Yoshio Kakuta. Numerical simulation of fracture process of concrete model by rigid body spring method. コンクリート工学年次論文集, 24(2):163–168, 2002. URL http://211.10.28.144/data_pdf/24/024-01-2028.pdf.

[17] Erfan G. Nezami, Youssef M.A. Hashash, Dawei Zhao, and Jamshid Ghaboussi. A fast contact detection algorithm for 3-d discrete element method. *Computers and Geotechnics*, 31(7):

575–587, 2004. ISSN 0266-352X. doi: 10.1016/j.compgeo.2004.08.002. URL http://www.sciencedirect.com/science/article/B6V2C-4DMW3PT-1/2/d109e9e249daf37d294e6a10d24f8d31.

[18] Mathew Price, Vasile Murariu, and Garry Morrison. Sphere clump generation and trajectory comparison for real particles. In *Proceedings of Discrete Element Modelling 2007*, 2007. URL http://www.cogency.co.za/images/info/dem2007_sphereclump.pdf.

[19] Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 147–154. Canadian Human-Computer Communications Society, 1995. URL http://citeseer.ist.psu.edu/provot96deformation.html.

[20] Jessice Rousseau, Emmanuel Frangin, Phiippe Marin, and Laurent Daudeville. Multidomain finite and discrete elements method for impact analysis of a concrete structure. *Engineering structures*, 43(1–2):2735–2743, 2009. URL http://geo.hmg.inpg.fr/%7Edaudevil/publis/engstruct2.pdf.

[21] E. Schlangen and E. J. Garboczi. New method for simulating fracture using an elastically uniform random geometry lattice. *International Journal of Engineering Science*, 34(10):1131–1144, 1996. URL http://www.fire.nist.gov/bfrlpubs/build96/PDF/b96022.pdf.

[22] Gen-Hua Shi. Discontinuous deformation analysis: a new numerical model for the statics and dynamics of deformable block structures. *Engineering computations*, 9:157–168, 1992.

[23] W. J. Shiu, F. V. Donzé, and L. Daudeville. Compaction process in concrete during missile impact: a dem analysis. *Computers and Concrete*, 5(4):329–342, 2008. URL http://geo.hmg.inpg.fr/%7Edaudevil/publis/Computers&Concrete2.pdf.

[24] J. G. Williams. The analysis of dynamic fracture using lumped mass-spring models. *International Journal of Fracture*, 33(1):47–59, January 1987. ISSN 0376-9429 (Print) 1573-2673 (Online). doi: 10.1007/BF00034898. URL http://www.springerlink.com/content/h31089636u20h601/.

Publications

[25] Jan Stránský, Milan Jirásek, and Václav Šmilauer (25%). Macroscopic elastic properties of particle models. In *Proceedings of the Interaction Conference on Modelling and Simulation 2010, Prague*. preprint, June 2010.

[26] Václav Šmilauer. The splendors and miseries of yade design. *Annual Report of Discrete Element Group for Hazard Mitigation*, 2006. URL https://yade-dem.org/w/images/a/a6/Smilauer-the_splendors_and_miseries_of_yade_design-2007.pdf.

[27] Václav Šmilauer. Commanding c++ with python. 2008. URL https://yade-dem.org/w/images/4/40/Yade-python-aussois-2008.pdf.

[28] Václav Šmilauer. Yade: past, present, future. 3 2010. URL https://yade-dem.org/w/images/5/59/Eudoxos2010-yade-past-present-future.pdf.