

Yade for the non-geeks

J. Duriez, B. Chareyre, F.V. Donzé & F. Darve

Laboratoire Sols, Solides, Structures, Risques, Grenoble, France

jerome.duriez@hmg.inpg.fr bruno.chareyre@hmg.inpg.fr,

Frederic.Donze@hmg.inpg.fr Felix.Darve@inpg.fr

ABSTRACT

The first utilisation of Yade is described here : downloading the sources, compiling them, and launching a first simulation. By doing so, the major types of source files are presented with examples. The goal is to permit to a newbie to not feel lost if he wants to use Yade for his research activities. So great attention is paid to be as simple and clear as possible (with apologies to people who will think that this paper is full of obvious things).

INTRODUCTION

If you read these lines you have surely decided recently to try and use Yade for your numerical simulations. I was in this case two months ago, while I didn't know more or less anything on Linux, C++, or open source codes... So you can easily imagine that sometimes I had the feeling to try to solve puzzle with manuals redacted in chinese... I begin slowly to come back to the surface and I think that it would be a good thing (for each new user but for Yade itself) if the beginning of the use of this code could be not so disorientating for everyone. So here are a few paragraphs that are not intended to be very complicated but that, I hope, will help new users to see faster than me how coarsely Yade is built and how they will have to deal with it.

INDEX

INSTALLING AND BEGINNING TO USE YADE	P. G3
1. Compiling	p. G3
2. Launching Yade for the first time	p. G5
a) Line "Generator Name"	p. G7
b) The " <i>Parameters</i> " window	p. G7
c) The " <i>Generate</i> " button	p. G7
d) The .xml file	p. G8
3. Making a calculation	p. G10
THE PREPROCESSOR FILE	P. G11
1. The <i>generate</i> function	p. G11
2. The <i>createActors</i> function	p. G12
3. The <i>registerAttributes</i> function	p. G13
CONCLUSION	P. G14
REFERENCES	P. G15

INSTALLING AND BEGINNING TO USE YADE

Yade is developed originally on Linux, so (except if you have sufficient skills to compile the sources on another system) the first thing you have to do to use Yade is to quit Windows and jump into Linux, which could be a bit frightening if you are really a non geek. But sincerely under Linux, even if you could do all that you want with black screens with white letters (and probably many people would say you that it is the best thing to do), you would also have the possibility to use it so that you will not see so much differences with Windows.

When you are a little familiarized with Linux, you can download, all that Yade needs to be installed, on the web site : <http://yade.wikia.com/wiki/Yade> : in the rubrique **Installation** then **download** : <http://yade.wikia.com/wiki/Installation#Download>. You will so download a tar.gz file which is a compressed folder (like the .zip under Windows for example).

Once this file downloaded you have to decompress it, by right clicking on it and then making "**Extract**" into the location you want (as an example for me it is something like home/jerome/yade-0.11.1). You have then become a lot of files and folders which constitute all the "sources" of Yade. For those for who that would not be clear (if this happens), you have to understand that Yade has to be seen like a programming project (in C++ so) with a lot of source files that interact with each other and describe a lot of things like the geometry of your numerical model, the contact laws between the entities (in fact all the things that are used for the calculations). You have then to compile all these source files to create an executable and libraries based on what is written in these files.

1. Compiling

In fact to compile the source files all that you have to do is to **open a terminal** in the folder where you extracted the sources (home/jerome/yade-0.11.1 for me so), and then typing `scons`. This instruction, which will so create the "real" software Yade (the executable), uses the instructions (written in Python programming language) from the two files "*SConstruct*" and "*scons.config*" that are in this folder.

In fact, before typing `scons` you have to **edit `scons.config`** (which is by chance the simplest of the both...) where, for example, the folder where the executable will be installed (and few other things) is defined. You have to define this folder and you can do it on the first line beginning with "`PREFIX =`" (personally I choosed to name the folder of the executable : YADE, so I have me : "`PREFIX = '/home/jerome/YADE'` "). I can't really explain you all the others lines, I can only say that they allow you to set some options of installation, for example

the line " *exclude* = " which allows you not to install some modules of Yade. Yade being indeed constituted of different modules ("packages") (one for discrete element simulations, one for finite elements ones, and one for lattice ones for example), you could be interested in not to installing one of them : for example if you don't need the lattice module and if the compilation crushes only because of source files concerning this module, you can solve your problems by typing " *exclude* = '*lattice*' ", moreover it would save time. If you want all to be installed, type " *exclude* = '*none*' " !

Once the *scons.config* file set you can now type in the terminal :

scons

NB : in fact it is not absolutely necessary to edit directly *scons.config* : for setting the same things you can also type in the terminal something like :

```
scons exclude=lattice,mass-spring optimize=1 debug=0  
version=-trunk PREFIX=/opt
```

(and then these options will be remembered in *scons.config*)

Then the only thing you have to do is to pray that all works well and that you will not have to give a look to the *SConstruct* file (if this is very unfortunately the case then you can use the yade mail list to ask a – precise - question by sending the error message printed)...

To finish about the compilation I have to say you that there are some other important files that allow a good compilation. In each folder of *pkg/*, you can find *SConscript* files (one in *pkg/dem*, one in *pkg/common*...). These files describe the interactions between the source files : if you open for example ***pkg/dem/SConscript*** you will find a lot of paragraphs looking like this :

```
env.SharedLibrary('TriaxialTest',  
  ['PreProcessor/TriaxialTest.cpp'],  
  LIBS=env['LIBS']+['SDECLinkGeometry',  
    'ElasticContactLaw',  
    'AveragePositionRecorder',  
    ...  
    'PositionOrientationRecorder']),
```

In fact for each class described in *pkg/dem* (and that you want to be compiled in the aim to use it in the executable for the simulations), you must have one paragraph. As you can see the paragraph begins with the name of the class (here it is " *TriaxialTest* ", we will speak about it in few lines), its location, and then you have all the list of the classes necessary to the one considered (*TriaxialTest* here) to work well. To see fast which classes are "necessary" you would normally only have to give a look to which classes are included in the source file considered (which is so " *pkg/dem/PreProcessor/TriaxialTest.cpp* " here)

If it is your very first compilation you should not have to give interest to these *SConscript* files. But later when you will modify some source files, you will have to add the corresponding lines in the corresponding paragraph. For example if you write a new class

named for example *LdC26*, that describes a new law of contact, and that you want use it in the existing file who describe your simulation (let's say that its name is " *MySimulation* "), you will have to

- add in the paragraph *env.SharedLibrary('MySimulation',...* a line '*LdC26*'

- and create a paragraph

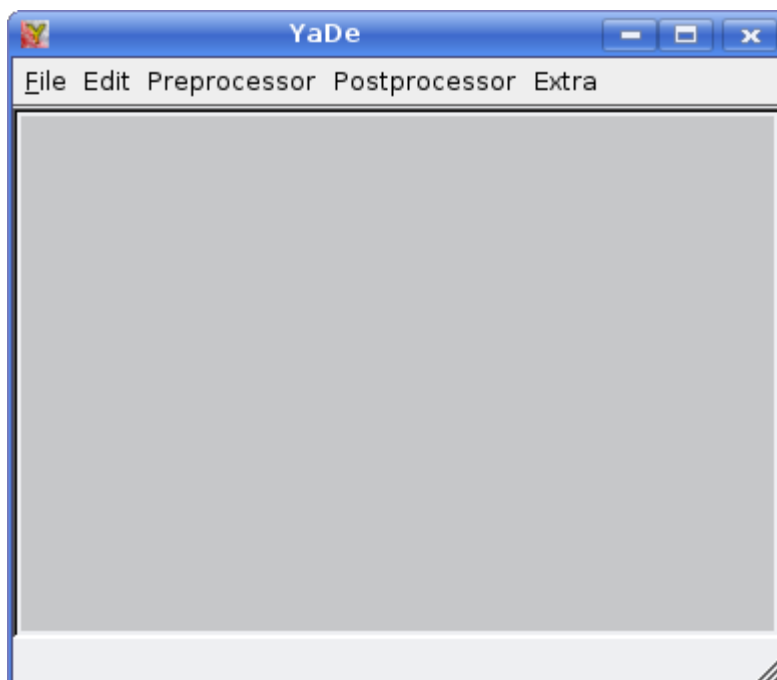
```
env.SharedLibrary('LdC26',  
    ['location of LdC26'],  
    LIBS=env['LIBS']+['all the classes corresponding to the objects you used for  
creating LdC26...
```

2. Launching Yade for the first time

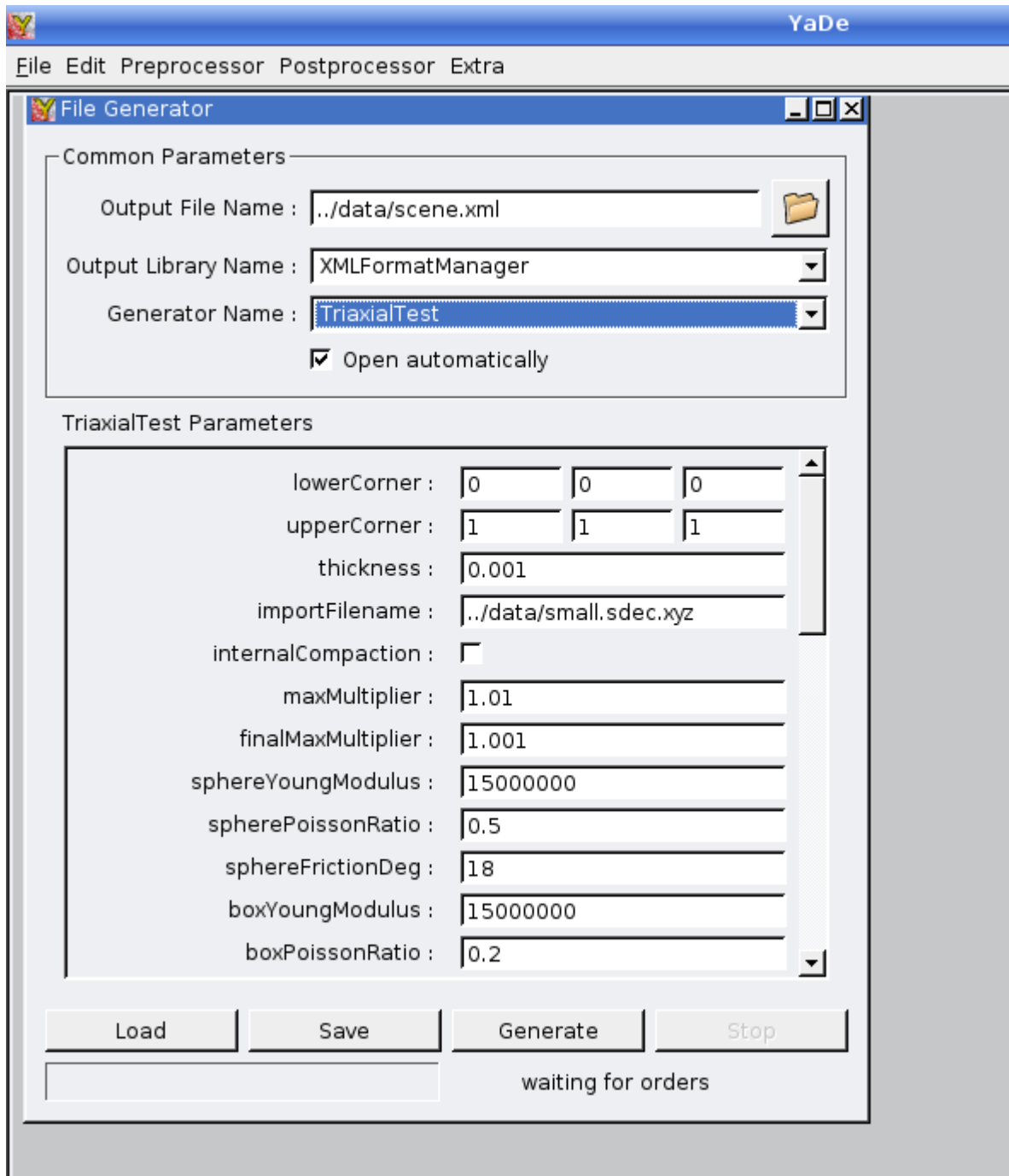
Once the compilation has could do its job (maybe around a few days after having downloaded the sources...) you can now go into the folder you indicated in *scons.config*. Folder in which you should find folders named *bin*, *data*, *include*, *lib*. In the *bin* one you will find the executable : you can launch it by opening a terminal in this folder and so typing

```
./yade-0.11.1
```

(if your executable has the same name as mine). Here is the window you should obtain :



Great, isn't it ? For making a few things you could click on *Preprocessor*, then *FileGenerator*, and you obtain this :



This window allows you to open a preexisting example of simulation. There are various examples, but because I will take often this example, let's say a few words about the *TriaxialTest*.

This preprocessor simulates a ... triaxial test by the discrete element method. The numerical model is composed of some spheres (simulating the physical sample) that are contained into a box constituted itself by six walls. By launching the calculation, the six walls move so that the numerical sample is confined, after that you could impose a deviatoric stress.

You can choose the example you want to launch in the line " *Generator Name* ".

a) Line " *Generator Name* "

In the source files there are indeed some "preprocessor" files that describe some numerical models, like so the simulation of a triaxial test (" *TriaxialTest* "...), or the simulation of the impact of a ball in a wall of bricks (" *SDECImpactTest* "), and so on. Depending on the type of the numerical simulation (DEM, FEM, lattice...) you can find these files in folders looking like **yade-0.11.1/pkg/dem/PreProcessor** (if it is a DEM simulation).

For each numerical model you have two files : one .hpp and one .cpp. In **yade-0.11.1/pkg/dem/PreProcessor** you can so find for example the files **TriaxialTest.hpp** and **Triaxial.cpp**. If you know something to C++ you would be used to that, and if it is not the case, before strong advicing you to give a serious look to C++ language (for french speaking people I personally advice <http://www.siteduzero.com/tuto-3-5395-0-apprenez-a-programmer-en-c.html> which I found very well done and, so, quite simple to understand), I could explain you that in the .hpp file you will find the declaration of all the variables and functions (you can make the difference by this way : the names of functions contain parentheses, the names of variables don't) that will be used in .cpp file.

If you are lucky you could also find in this .hpp file maybe some comments that will explain you to what correspond these variables and functions. If you want, you can give a look to some of these preprocessor files, but we will speak about it in some paragraphs.

b) The " *Parameters* " window

Below is a " *Parameters* " window where you can modify some features of the numerical models, which were given default numerical values in the preprocessor file. For example, for the *TriaxialTest*, you could change the values of the dimension of the box containing the numerical sample, physical parameters of the spheres and the walls : *sphereYoungModulus*, *boxYoungModulus*, *sphereFrictionDeg*, names and locations for outputs files and so on... I don't explain you now the real meanings of all of these physical parameters, be just aware that some correspond to what you think (for example the friction degrees that are used to described the maximum tangential forces on the contacts), but some not : for example the Young modulus are all but real Young Modulus (all the entities are indeed rigid), but they are used to compute the normal rigidities on the contacts.

c) The " *Generate* " button

When you are ready to start a calculation you can click on **Generate**. If I understood well this will read the corresponding preprocessor file so that, for example for the *TriaxialTest*, all the balls (and the boxes) will be effectively created, located to the right places and will be affected their physical parameters (that's a very incomplete but, I hope, quite practical and clear example). Engines working on bodies are created too (see next paragraph about engines). Thanks to the "execution" of the preprocessor file a .xml or .bin file

is generated and saved (in the location indicated in the *Output FileName* line). Binary files are shorter but non editable format files, I will speak to you about the .xml file that you could read with a text editor.

d) The .xml file (IMPORTANT § BECAUSE OF EXPLANATIONS ABOUT ENGINES)

In fact all that you have done here was only to create this .xml file which will be directly used now when the calculation will run. So let us give a look to it.

The first lines will probably be always the same and look to this :

```
<Yade>
  <rootBody _className_=          ...
    <physicalParameters          ...
    <geometricalModel />
    <interactingGeometry          ...
    <boundingVolume              ...
```

These lines are always the same, so I won't tell you a lot about this (and also because I don't see what I could explain you...). But just after you can see an important paragraph beginning with

```
<engines size="a number">
  <engines _className_="...
```

I would say that it is the point of Yade : **engines** are "things" that act on the numerical model every timestep. They are of course all defined in source files you could find in folders like for example **yade-0.11.1/pkg/dem/Engine/...** for engines specifics to DEM simulations or in **yade-0.11.1/pkg/common/Engine** for engines that could be used for every type of simulation. Here are two examples you could find in your .xml file issued from the *TriaxialTest* preprocessor, so that you could maybe better imagine what these engines could be :

– the "*TriaxialCompressionEngine*"

As you might imagine, this engine allows to confine the numerical sample. You could find the sources for it in **pkg/dem/Engine/DeusExMachina**.

You can see in the xml file, that after `<engines _className_="TriaxialCompressionEngine"`, there are still a lot of things written : that's all the variables (with their values) specifics to the *TriaxialCompressionEngine*. These variables were declared in *TriaxialCompressionEngine.hpp* (and you can see how exactly they are used in *TriaxialCompressionEngine.cpp*), but there values were defined in the *TriaxialTest* preprocessor file. Fortunately the .hpp file is here well commented and you should see fast to

what they correspond without searching them in the .cpp and trying to understand their role (which is the general case you will have to do).

- the " *GravityEngine* "

You could maybe not see it in this .xml file but you would surely fast find it in other .xml files corresponding to others preprocessor files. The sources for this engine are in **pkg/common/Engine/DeusExMachina**.

This engine allows to apply ... the gravity to the bodies of the numerical model !! Give a look to the sources, and you will see that there is here only a variable : a vector of three real numbers (cf the type *Vector3r* which is also defined somewhere in the sources) named *gravity*. In xml files where this engine appears you should see so a line looking to :

```
<engines _className_="GravityEngine" gravity="{0 -9.81 0}" />
```

To conclude these some words about the engines, I will ask you to continue to look into the sources of these engines (or others) and I'll point your attention about the fact that they all have a "action" or a "applycondition" function (if this doesn't appear directly, these functions are herited from a mother class). In fact that's how the calculation progresses : at each timestep the " action " function of all of the engines which appear in the list in the .xml file is executed.

NB1 : to be sure of that you can see the files **core/Metabody.hpp** and **.cpp** where the function *moveToNextTimeStep()* (whose you could imagine that it is executed at each timestep...) is defined. If you are familiar (and if you're not that's not too bad for the moment) with C++ you would see what I mean.

NB2 : if later you will modify this .xml file, by adding for example engines, be careful that if you want to use 15 engines, you have to have on the first line of this paragraph : `<engines size="15">` !

That's so how, at each timestep, the spheres move, the contact forces are calculated, the values of what you want to be stored are stored and so on...

To finish with the .xml file, you will see that after the § devoted to the engines there is a one where you could find the list of all of the numerical entities, named "*bodies*", constituting the numerical sample. This paragraph begins with :

```
<bodies _className_="BodyRedirectionVector" >
  <body size="a number">
    <body _className_="Body" id="0" ...
```

In the second line you can so see ... the number of bodies present in the model. Then comes the list of all these bodies with informations about each one. They have all an identity number (from 0 to number of bodies – 1), that you can see in ... *id*=.

After this number of identity, you can find *isDynamic="1" or "0"*. Depending on this, this body will (if *isDynamic="1"...*) or not be able to move. To be sure of that, you can open

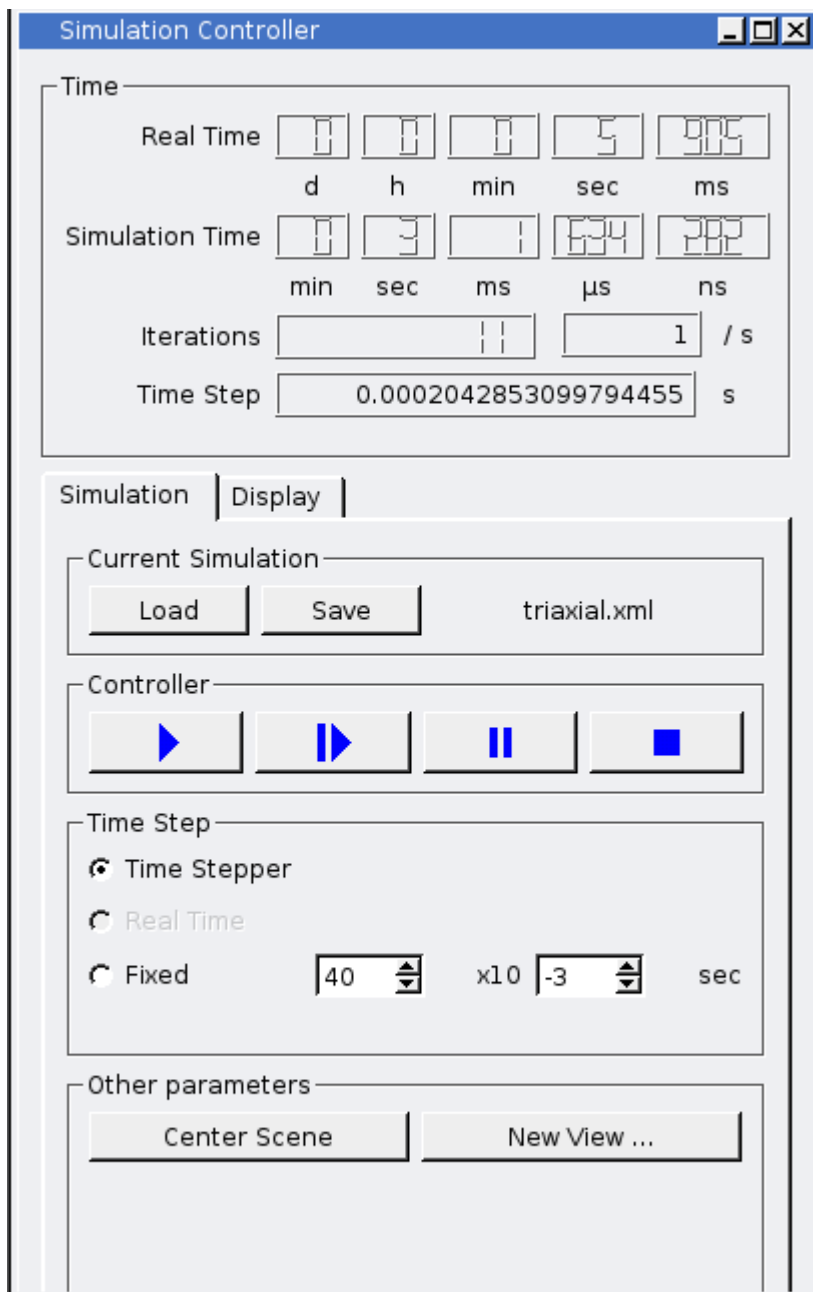
the file `pkg/common/Engine/EngineUnit/LeapFrogPositionIntegrator.cpp`. In this file is described how the velocities and positions of bodies are updated, knowing their acceleration. And as you can see, all these lines won't be executed if the body is not dynamic (cf line 18 : `if(!body->isDynamic) return;`). So it is a simple mean to have fixed bodies : set for them `isDynamic="0"`, and you will be sure that there will never move !

NB : this file has a brother : **LeapFrogOrientationIntegrator** (in the same folder) which does the same work but for the rotations and orientations of bodies.

Then you have `se3`, which is a 7 real vector : the three first numbers represent the position of the center of the body, and the four last are a quaternion which describes the orientation of the body. The other parameters should be easy to understand. Please just note that there are some of them which are used for the screen display.

3. Making a calculation

Now that you have already learned some things about Yade, you could be envious to see how it calculates... After having clicked on Generate, two new windows should have open



one named *Simulation Controller* and one *Primary View*.

In the *Primary View* one is represented the numerical model. You could zoom, move (right click) and rotate (left click) the view with the mouse.

The *Simulation Controller* window looks like this :

You can so see different counters :

One for the "real time" : it is the time since you launched the calculations (it would so indicate 1 h if one day you launch a calculation at 12h before going eat and if you come back at 13 h).

One counter for the simulation time : it is the

"numerical time" : the product of the time step (which could evolve) by the number of iterations.

One counter for the number of iterations being done (with, right, the rate of iterations per second)

And finally the current value of the time step !!

Under you have "*load*" and "*save*" buttons. The "*save*" one allows to create a .xml file corresponding to the current state of the simulation. Compared to the one issued from the generation of the preprocessor file, the differences would surely be the positions and speeds of bodies for example...

The "*load*" one is to restore a .xml file (issued from saving or generating). Note that sometimes it doesn't work for me so I have to close this window (click just left of "Simulation Controller"), and close also the "File Generator" and then make **File->New Simulation**, then this Load button works better (don't ask me why !)

Then come buttons to control the running of the simulation (their meanings are normally quite clear...), and finally you can choose to use time step calculated by a time stepper, or use a constant time step that you fix.

THE PREPROCESSOR FILE

Now I'm afraid it is time to jump into the sources files, and especially into the preprocessor ones, to see from where all this stuff comes. Remind you just that, thanks to the .xml file and the "*Load simulation*" button you could make all your calculations without preprocessor files but, for doing that, you would have to be able to write yourself your xml file starting from nothing... I'm not sure it would be easier !

If you are always no familiar with C++ I'm afraid that this sweet period should finish now : to speak about my particular case it's perfectly clear that I don't know C++ as good as it would be necessary to re write Yade (and from very very very very far), but I'm satisfied to know the very few things that allow me not to feel in the worst unknown and dangerous jungle when I open a source file...

I propose you to continue to focus on the same example. So please now open **pkg/dem/PreProcessor/TriaxialTest.cpp** (and **.hpp**). You could open it with any text editor, but it would surely be better to open it with something especially devoting to programming, like KDevelop for example.

This preprocessor file, like the others, has the following functions : *generate*, *createBox*, *createSphere*, *createActors*, *positionRootBody*, *postProcessAttributes*, *registerAttributes*. I can't explain you all of these but here is what I understood (or was explained).

1. The *generate* function

I said you in the previous chapter that, when we click on " *Generate* " in *FileGenerator* window, the corresponding preprocessor file was executed. In fact we could imagine that, more precisely, this is this function which is executed (after surely also the constructor of the class). So let us hope that this function contains all that we want for doing our simulations...

This function begins with the call of the definition of a pointer towards a "*Metabody*", and whose name is *rootBody*. You could find the sources which define what a *Metabody* is exactly in **core/**, but I advice you for the moment to be satisfied with the idea that a *Metabody*, and, so, our object *rootBody*, represents the whole simulation, that is to say : all the discrete entities with their positions, parameters..., with all the engines that will act (and surely a lot of other things).

Then you have the call to the functions *createActors* (we will speak about it later) and *positionRootBody*.

Then come few lines to define some variables of the object *rootBody* :

```
rootBody->persistentInteractions    =    shared_ptr<InteractionContainer>(new  
InteractionHashMap);
```

```
rootBody->transientInteractions     =    shared_ptr<InteractionContainer>(new  
InteractionHashMap);
```

...

They seem to be always the same, and that's good because I can't really explain you them.

Finally is generally the creation of the numerical sample, thanks to the use of the functions *createSphere* and *createBox* for example.

2. The *createActors* function

I would say that this is a very important part of the file. For one time let's begin by the end of the paragraph and, exactly by these lines :

```
rootBody->engines.clear();
```

```
rootBody->engines.push_back(...
```

We saw in the .xml file that there were a list of engines that run during the calculation. **This list of engines is defined here.** As you see this list is stored in the *rootBody* object, and after having cleared it (to be sure of what we do), all the engines are, one after the other, added in the list, thanks to the C++ function *push_back*. In fact what are added are pointers towards "engine" objects. All these pointers were defined (with the values of their variables) in the lines above.

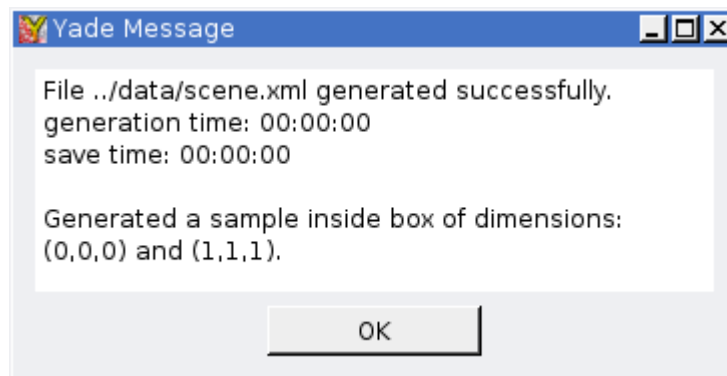
So if you make for example calculations described by a preprocessor file, but if you want focus on some special things (the force acting on a body) that are not stored in the "default" form of the preprocessor file, that's the place where you have to add the right things : that is to say the definition of a pointer towards the engine that will record what you want (you can find "recorder engines" in **pkg/dem/Engine/StandAloneEngine/** for example), the affectation of good values for its variables (for example the name of the file where the values were stored is a variable of such engines), and finally a line looking like that :

```
rootBody->engines.push_back(my_pointer_towards_a_engine_recorder_objet);
```

NB : do not forget also, if necessary, to make the right *# include* in the preprocessor file, and to modify the *SConscript* files, so that the compilation works.

3. The *registerAttributes* function

Let us finish by an other important function : the *registerAttributes*. When you generated your *TriaxialTest* for example, you obtained this window :



As it is suggested by the "save time" a save was done. But what was exactly saved, and why ? That's the role of the *registerAttributes* functions : they allow to save in memory for example values of variables that were defined in the preprocessor file (and which were, so, defined only once : when the simulation was "generated"). In the case of variables which would be used every time step (for example variables of engines), if the values of the variables would not be registered, the engine would have forgot these values ! (and in the less worse case the values of the constructor of the engine would so be taken).

So, for all things you wish Yade to keep in memory you have to write somewhere (in the implementation of the *registerAttributes* of your class in fact) :

```
REGISTER_ATTRIBUTE(my_thing_to_keep_in_memory);
```

To help you to understand, here is an example from *TriaxialTest*. You should have normally in the *TriaxialTest* preprocessor file, the definition and the use of pointers towards *VelocityRecorder* engines. You should find indeed lines looking like that :

```
velocityRecorder = shared_ptr<VelocityRecorder>(new VelocityRecorder);
```

With then the affectation of correct values to the attributes of this pointer, for example, the name of the output file :

```
velocityRecorder-> outputFile = velocityRecordFile;
```

You see here that a string (*velocityRecordFile*) is used to affect this value. The string is defined in fact in the constructor of the *TriaxialTest* (he is taken equal to *"../data/velocities"* for example), and you could see that this string is stored thanks to the line *REGISTER_ATTRIBUTE(velocityRecordFile);* which appears in the implementation of the *registerAttributes* function of *TriaxialTest*.

Thanks to this, Yade will keep in memory that the output file of the *velocityRecorder* engine is not only a string named *velocityRecordFile*, but in fact *"../data/velocities"*.

NB : if you wonder why Yade is able to recall that the output file of the engine is the string *velocityRecordFile*, the answer is simple : give a look to the source of the engine (in **pkg/dem/Engine/StandAloneEngine/VelocityRecorder.cpp**, and you will see, in the function *registerAttributes*, the line *" REGISTER_ATTRIBUTE(outputFile); " !!!*

CONCLUSION

That's the end of this paper. As you saw I can't explain you all : first because explaining all the source files of Yade would take very much more than these few pages, and second and overall because I have not finished yet to ask me questions about Yade ! But I hope that it helped you to understand how to apprehend Yade and that it lets you understand and accept that now all that you have to do for using Yade is to open quite a lot of source files, find in them all the variables or functions about which you ask questions, and seek for their declaration and their use to understand what is their role (and after when you will write your own files and submit them to the Yade community, please add comments !!!)

ACKNOWLEDGEMENTS

The first author wants to acknowledge the people (PhD students and users of Yade of his laboratory, Yade developpers...) who accepted to take of their time to explain him simply and clearly various things about Yade.

REFERENCES

J. KOZICKI, F.V. DONZÉ „Applying an open--source software for numerical simulations using finite element or discrete modelling methods”. *Computer Methods in Applied Mechanics and Engineering*, (submitted) **2008**

J. KOZICKI, F.V. DONZÉ „YADE-OPEN DEM: an open--source software using a discrete element method to simulate granular material”. *Engineering Computations*, (submitted) **2008**

FENG CHEN, ERIC. C. DRUMM, AND GEORGES GUIOCHON „Prediction/Verification of Particle Motion in One Dimension with the Discrete-Element Method”. *International Journal of Geomechanics, ASCE*, Volume 7, Issue 5, pp. 344-352 **2007**

NICOT, F., L. SIBILLE, F.V. DONZÉ, F. DARVE „From microscopic to macroscopic second-order work in granular assemblies”. *Int. J. Mech. Mater.*, 39 (7), 664-684 **2007**

SIBILLE, F. NICOT, F.V. DONZÉ, F. DARVE „Material instability in granular assemblies from fundamentally different models”. *International Journal For Numerical and Analytical Methods in Geomechanics*, vol:31, 457-481 **2007**

J. KOZICKI, J. TEJCHMAN „Effect of aggregate structure on fracture process in concrete using 2D lattice model”. *Archives of Mechanics*, Vol. 59, No 4–5, pages 365–384 **2007**

S. HENTZ, F.V. DONZÉ, L.DAUDEVILLE „Discrete element modelling of concrete submitted to dynamic loading at high strain rates”. *Computers and Structures*, vol: 82 No 29-30 pp. 2509-2524 **2004**

S. HENTZ, L. DAUDEVILLE, F.V. DONZÉ „Identification and Validation of a Discrete Element Model for Concrete”. *ASCE Journal of Engineering Mechanics*, vol:130 No 6 pp.709-719 **2004**

F. CAMBORDE, C. MARIOTTI, F.V. DONZÉ „Numerical study of rock and Concrete behaviour by discrete element modelling”. *Computers and Geotechnics*, 27 : 225-247 **2000**

F.V. DONZÉ, S.A. MAGNIER, L. DAUDEVILLE, C. MARIOTTI, L. DAVENNE „Study of the behavior of concrete at high strain rate compressions by a discrete element method”. *ASCE J. of Eng. Mech*, 125 (10) : 1154-1163 **1999**

S.A. MAGNIER, F.V. DONZÉ „Numerical simulation of impacts using a discrete element method”. *Mech. Cohes.-frict. Mater.*, 3, 257-276 **1998**

F.V. DONZÉ, S.A. MAGNIER „Formulation of a three-dimensional numerical model of brittle behavior”. *Geophys. J. Int.*, 122, 790-802 **1995**

F.V. DONZÉ, P. MORA, S.A. MAGNIER „Numerical simulation of faults and shear zones”. *Geophys. J. Int.*, 116, 46-52 **1994**