C++/Boost::Python programming
Example with Yade-DEM

Bruno Chareyre, Grenoble INP, 3SR

On debian/ubuntu and connected to internet?
$ sudo apt-get install yade (~70MB)
$ yade
You can reproduce the example in a minute
Yade-DEM.org

- Open platform for the simulation of mechanical systems (DEM)
- Started(*) and hosted(**) at lab. 3SR / GitHub
- Developed natively on Debian/Ubuntu systems
- Compiles on CentOS, Red Hat, MS Windows (yes!),...
- Deployed on various servers (incl. Gricad/Froggy, Amazon EC,...)
- Pre-compiled packages available for Debian/Ubuntu (>2011)

(*) by Frédéric Donzé (2006)
(**) thanks to Rémi Cailletaud
Yade-DEM.org community

Users (>100/year)
- applications in mechanics, physics, process/chemical/civil engineering...
- typically little to no time/experience for advanced programming

Google Scholar citations

V. Šmilauer et al. (2015), *Yade Documentation 2nd ed.*
DOI 10.5281/zenodo.34073
Yade-DEM.org community

Developpers (~15/year, ~50 from begining)

In a Nutshell, Yade...

... has had 5,418 commits made by 58 contributors representing 109,905 lines of code

... is mostly written in C++ with a low number of source code comments

... has a well established, mature codebase maintained by a large development team with stable Y-O-Y commits

... took an estimated 28 years of effort (COCOMO model) starting with its first commit in January, 2005 ending with its most recent commit 5 days ago

<table>
<thead>
<tr>
<th>Statistic</th>
<th>All Time</th>
<th>12 Month</th>
<th>30 Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits:</td>
<td>5418</td>
<td>213</td>
<td>29</td>
</tr>
<tr>
<td>Contributors:</td>
<td>57</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>Files Modified:</td>
<td>13544</td>
<td>197</td>
<td>42</td>
</tr>
<tr>
<td>Lines Added:</td>
<td>1810716</td>
<td>29589</td>
<td>585</td>
</tr>
<tr>
<td>Lines Removed:</td>
<td>1651298</td>
<td>8169</td>
<td>347</td>
</tr>
</tbody>
</table>

Stats from OpenHub.net
Yade-DEM.org community

Developpers (~10/year, ~40 from begining)

Stats from GitHub.com

Users

Core devs.

Casual devs.

Lines of Code

Languages

C++ 57%
Python 32%
C 6%
7 Other 5%

Stats from GitHub.com
Scene & interface(s)

A “Scene” is mainly three lists (of c++ objects) with transition rules (see live example)

- **Bodies** (data)
  - position
  - velocity
  - physical properties

- **Engines** (act once per iteration)
  - laws of physics
  - boundary/field conditions
  - contact detection
  - recorders
  ...

- **Interactions** (auto-updated data)
  - physical state: deformation, forces, ...
Scene & interface(s)

Interfaces to a c++ (DEM) code
1) Hard-code
   change the source code + recompile,
   i.e. no interface.
**Scene & interface(s)**

**Interfaces to a c++ code**

1) Hard-code (i.e. no interface...)

→ maximizes flexibility, but:
- vertical learning curve
- difficult to debug / no interactivity
- no batch execution
- tends to mix user-specific code with actual source code (dev=user...)
- smart hacks are difficult to share with others
Scene & interface(s)

Interfaces
1) Hardcode
2) Write input files
3) +Read output files
Scene & interface(s)
Scene & interface(s)

Interfaces
1) Hardcode
2) Write input files
3) +Read output files

- no flexibility
- no extensibility
- no feedback loop
Scene & interface(s)

Interfaces
- Hardcode
- Write input files
- Read output files
- Graphical user interface (GUI)

Qt Controller

QGLView

Scene

Engines

Data

Bodies

Interactions
Scene & interface(s)

Interfaces
- Hardcode
- Write input files
- Read output files
- Graphical user interface (GUI)

Qt Controller

QGLView

Scene

Engines

Data

Bodies

Interactions
Scene & interface(s)

Interfaces
- Hardcode
- Write input files
- Read output files
- Graphical user interface (GUI)
  - no flexibility
  - no extensibility
  - no feedback loop
  - I/O files + complex design

Qt Controller

QGLView

Scene

Engines

Data

Bodies

Interactions
Scene & interface(s)

Interfaces
- Hardecode
- Write input files
- Read output files
- Graphical user interface (GUI)
- Command line interface (CLI)

![Qt Controller](image)

![QGLView](image)

Scene
- Engines
- Bodies
- Interactions
- Data

Scene & interface(s)
Interfaces
- Hardcode
- Write input files
- Read output files
- Graphical user interface (GUI)
- Command line interface (CLI)

State of the art in DEM softwares
- Most in-house codes stuck in the I/O files paradigm
- In the 90's Itasca© started developing the “FISH” language for their DEM softwares (coded in C++)
- ~2004 it was possible to pass arguments to FISH functions and to declare local variables...
- ~2014 Itasca© started considering Python!
A challenging development problem:

YADE-DEM is a bazaar

A challenging development problem

Users
little to no time/experience in programming
Needs:
- documentation
- computational efficiency
- simplicity of usage
- flexibility
- interactivity

Casual devs.
want to implement something new (contact model, particle shape,...)
Needs:
- simplicity of implementation
- low commit barrier
- will hardly learn new programing techniques

Core devs.
Needs:
- minimize workload
boost::python

Example
Consider this piece of C++ code that we want to use in python:

```cpp
vector<int> myRange(int n)
{
    vector<int> list;
    for (int k=0; k<n; n++) list.push_back(k);
    return list;
}
```
boost::python

Example
Consider this piece of C++ code that we want to use in python:

```cpp
vector<int> myRange(int n)
{
    vector<int> list;
    for (int k=0; k<n; n++) list.push_back(k);
    return list;
}
```

It is enough to append:

```cpp
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(myModule)
{
    boost::python::def("myRange", myRange);
}
```
boost::python

Example

```cpp
vector<int> myRange(int n) {
    vector<int> list;
    for (int k=0; k<n; n++) list.push_back(k);
    return list;
}

#include <boost/python.hpp>
BOOST_PYTHON_MODULE(myModule) {
    boost::python::def("myRange", myRange);
}
```

Compilation produces a dynamic library which python can import as a module:

```python
>>> from myModule import *
>>> x=myRange(10)
>>> print x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```
Wrapping classes is also possible

```cpp
BOOST_PYTHON_MODULE(classes)
{
    class_<World>("World")
        .def("greet", &World::greet)
        .def("set", &World::set)
        .def("many", &World::many)
;
};
```
A challenging development problem

Users
little to no time/experience in programming
Needs:
- documentation
- computational efficiency
- simplicity of usage
- flexibility
- interactivity

Casual devs.
want to implement something new (contact model, particle shape,...)
Needs:
- simplicity of implementation
- low commit barrier
- will hardly learn new programming techniques

Core devs.
Needs:
- minimize workload

Lots of interactions
+ Incompleteness of the interface
A challenging development problem

**Users**
little to no time/experience in programming

**Needs:**
- documentation
- computational efficiency
- simplicity of usage
- flexibility
- interactivity

**Casual devs.**
want to implement something new (contact model, particle shape,...)

**Needs:**
- simplicity of implementation
- low commit barrier
- will hardly learn new programing techniques

**Core devs.**

**Needs:**
- minimize workload
YADE_CLASS macro

Without python wrapping the class declaration of “Sphere” would be:

```cpp
// Geometry of spherical particle
class Sphere: public Shape{
  public:
  // Radius [m]
  Real radius;
  // constructor
  Sphere (): radius(NaN) {createIndex();}
};
```

Yade is imposing a different form in which declaration, initialization, wrapping and documentation are simultaneous:

```cpp
class Sphere: public Shape{
  YADE_CLASS_BASE_DOC_ATTRSCTOR(Sphere,Shape,"Geometry of spherical particle.",
  (Real, radius, NaN, "Radius [m]")�
  createIndex(); /*ctor*/
};
```
YADE_CLASS macro

Functions as well (and much more):

```cpp
class Sphere: public Shape{
  Real newFunction(const char* path);

  YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(Sphere,Shape,"Geometry of spherical particle.",
    ((Real, radius, NaN,"Radius [m]")),
    createIndex(); /*ctor*/,
    .def(newFunction, &Sphere::newFunction, boost::python::arg("folder")="./",
      "Write into a file. This is a cross-ref to :yref:`Body`")
};
```

Result:
Python wrapping is a mandatory part of the class declaration, it appears in all header files
YADE_CLASS macro

class Sphere: public Shape{
  Real newFunction(const char* path);

  YADE_CLASS_BASE_DOC_ATTRSCTOR_PY(Sphere, Shape, "Geometry of spherical particle.",
  ((Real, radius, NaN, "Radius [m]")),
  createIndex(); /*ctor*/,
  .def(newFunction, &Sphere::newFunction, boost::python::arg("folder")="./",
  "Write into a file. This is a cross-ref to :yref:`Body`")
};

In the Qt window:
In the online/pdf documentations (built with Sphinx):

Whether this Shape is rendered using color surfaces, or only wireframe (can still
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 instar
indexable at last. If names is true (default), return class names rather than numerica:
```

```
dispIndex
Return class index of this instance.
```

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

```
radius(=NaN)
Radius [m]
```
YADE_CLASS macro

Inline documentation and auto-completion (ipython):

Yade [2]: s=Sphere()
Yade [3]: s?
Type: Sphere
String Form:<Sphere instance at 0x354d800>
File: /usr/lib/x86_64-linux-gnu/yadedaily/py/yade/wrapper.so
Docstring: Geometry of spherical particle.

Yade [4]: s.
s.color s.dispHierarchy s.highlight s.updateAttrs
s.dict s.dispIndex s.radius s.wire

Yade [4]: s.radius?
Type: property
String Form:<property object at 0x7f61aae16db8>
Docstring: Radius [m] :ydefault:`NaN` :yattrtype:`Real` :yattrflags:`0`
Note: YADE itself is a python module

bchareyre@dt-med008:~$ yade
Welcome to Yade 1.07.0
TCP python prompt on localhost:9000, auth cookie `adkyus'
XMLRPC info provider on http://localhost:21000
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d
view (use h-key for showing help), F10 both, F9 generator,
F8 plot. ]]
Yade [1]:

Behind the scene:

~$ python
In [1]: #set custom ipython decorations and other things
...
In [N]: import yade
Yade [1]:
Conclusion

Advantages (among others)

- Nearly no limit to user's imagination
- Powerful (pre/post-)processing tools at no (development) cost
- Inline documentation
- Debugging scenes is much easier
- Couplings with other codes: OpenFoam, e-script (FEM), Yales2, Palabos,...
- Some task parallelism can be exploited at the python level (mpi4py for FEMxDEM)
- Online discussions and bug reports can come with Minimal Working Examples (MWE™)
- ...
Conclusion

Downside

- **Very** intrusive technique
- Compilation time skyrockets due to boost templates (~1h for fresh build on the average desktop)

Conclusions

- If you are starting an ambitious project in C++ better integrate python from the very beginning
- It may actually help for the development itself
- Yade-DEM could be used as a template project for such thing
Dependencies (some of them)

**IPython**
- CLI
- NumPy
- Matplotlib

**Eigen**
- Linear algebra

**CGAL**
- Comput. Geometry

**OpenBLAS**
- Optimized algebra

**SuiteSparse**
- Sparse linear solvers

**git**
- VCS

**Sphinx**
- Python doc

**Qt**
- GUI

**OpenGL**
- 3D rendering

**QGLViewer**
- Post-processing

**VTK**
- ParaView

**Kitware**
- Post-processing
Scene & interface(s)

A "Scene" is mainly three lists (of C++ objects) +

- **Engines**
  - boundary/field conditions
  - contact detection
  - recorders
  - ...

- **Bodies**
  - position
  - velocity
  - physical properties

- **Interactions (auto-update)**
  - physical state: deformation, forces, ...

---

Data & predicting (e.g. CFD, FEM, DAE)