

# pythonOCC Parametric Application Framework (PAF) tutorial r0.4

Thomas Paviot (tpaviot@gmail.com)

February 27, 2010

## **Abstract**

This guide aims at introducing in a few stages the pythonOCC package intended to parametric modeling.

## License

This document is distributed under the terms of the Creative Common BY-NC-SA 3.0 license. In a few words:

You are free:

- to Share
- to copy, distribute and transmit the work to Remix
- to adapt the work

Under the following conditions:

- Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial — You may not use this work for commercial purposes.
- Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

With the understanding that:

- Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights — In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author’s moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this<sup>1</sup> web page.

---

<sup>1</sup><http://creativecommons.org/licenses/by-nc-sa/3.0/>

# Contents

<b>1</b>	<b>What is PAF and Why Might I Need It?</b>	<b>4</b>
<b>2</b>	<b>What is needed to run the PAF?</b>	<b>4</b>
<b>3</b>	<b>The pin tutorial</b>	<b>4</b>
3.1	Creation of a parameterized cylinder . . . . .	4
3.1.1	Step 1 : initialization of the framework. . . . .	5
3.1.2	Step 2: creation of a set of parameters . . . . .	5
3.1.3	Step 3: creation of a ParametricModelingContext . . . . .	5
3.1.4	Creation of the parameters . . . . .	5
3.1.5	Step 5: creation of the geometry . . . . .	5
3.2	Let's play with ipython . . . . .	6
3.3	Creation of a relation (the Relation class) . . . . .	7
3.3.1	Step 1 : import the Relation class . . . . .	7
3.3.2	Step 2 : cast parameters to sympy symbols . . . . .	7
3.3.3	Step 3: write the relation . . . . .	8
3.3.4	Step 4: link the relation and the targeted parameter . . . . .	8
3.4	Adding a rule (the Rule class) . . . . .	9
3.4.1	Step 1 : import the Rule class . . . . .	9
3.4.2	Step 2 : define the python function . . . . .	10
3.4.3	Step 3 : creation of the Rule to connect a parameter to the python function . . . . .	10

## 1 What is PAF and Why Might I Need It?

The Parametric Application Framework (PAF) is a subpackage of the pythonOCC<sup>2</sup> development library. parametric modeling application framework based upon the OpenCASCADE 3D modeling kernel<sup>3</sup> and the SalomeGEOM<sup>4</sup> libraries. PAF is part of the pythonOCC distribution since the release 0.3.

## 2 What is needed to run the PAF?

In order to run PAF, you only have to: download and install pythonOCC 0.3 or higher. be sure that the SalomeGEOM libraries are installed The PAF makes possible to modify the parameters values and see the propagation of these modification to the geometry 'on the fly'. The best way to get all the power of PAF is certainly to use the ipython<sup>5</sup> console. When your PAF script is ready (let's name it test\_paf.py'), then just: `ipython -wthread test_paf.py`

## 3 The pin tutorial

You may know that a simple optimization computation (that can easily made by hand) lead to the fact that given a cylindrical pin, the maximal volume/minimal steel surface compromise is obtained if the diameter of the pin equals its height. The aim of this tutorial is then to have a 3D model of this pin compliant with this knowledge rule, and get a 3D shape with  $height = 2 * radius$  whenever a change occurs in the radius parameter. We'll first create 2 parameters *height* and *radius*, then create the geometry associative with these parameters.

### Notes about this tutorial

- In this tutorial, the following formatting convention is used: python code is formatted like:

```
import sys
print sys.path
```

- parameters are *italic* formatted. For instance: *radius*, *height*, *max\_mass*, etc.

### 3.1 Creation of a parameterized cylinder

Let's get started with the most simple example we could imagine.

<sup>2</sup><http://www.pythonocc.org>

<sup>3</sup><http://www.opencascade.org>

<sup>4</sup><http://sf.net/projects/salomegeometry>

<sup>5</sup><http://ipython.scipy.org/>

### 3.1.1 Step 1 : initialization of the framework.

You have to first import the classes needed whenever you want to use the PAF :

```
from OCC.PAF.Context import ParametricModelingContext
from OCC.PAF.Parametric import Parameters
```

We'll see in details these classes in the next part of this tutorial.

### 3.1.2 Step 2: creation of a set of parameters

Let's first create a container that handle each new parameter needed:

```
p = Parameters()
```

### 3.1.3 Step 3: creation of a ParametricModelingContext

The ParametricModelingContext is the container for all the relation, rules, parameters. It provides a set of methods to build geometry. It has to be initialized with a set of parameters:

```
tutorial_context = ParametricModelingContext(p)
```

let's tell the context that a graphic window has to be set up in order to display the geometry:

```
tutorial_context.init_display()
```

### 3.1.4 Creation of the parameters

The two parameters height and radius must be initialized with float (or integer) values. In this first step, we initialize parameters with arbitrary (silly) values. We'll see later how to insert a relation between the parameters.

```
p.height = 43.3
p.radius = 12.9
```

### 3.1.5 Step 5: creation of the geometry

We'll use the MakeCylinderRH method from the GEOMImpl\_I3DPrimOperations class. Before that, let's register the prim\_operations in to the context.

```
tc.register_operations(tutorial_context.prim_operations)
my_box = tc.prim_operations.MakeCylinderRH(p.radius, \
p.height, name="cylinder1", show=True)
```

- Note 1: the name must be provided to the context

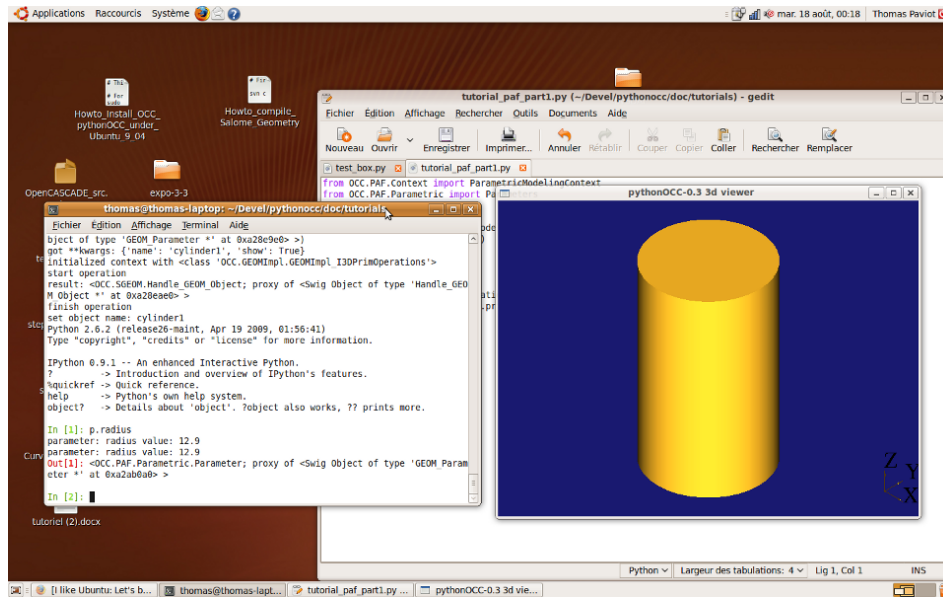


Figure 1: parametric cylinder created from an ipython console

- Note 2: the `show=True` optional python parameter tells the context to display the geometry in the graphic Window.

Here we are. You have a parameterized model of a cylinder! Let's first make a few tests from this (small!) script.

### 3.2 Let's play with ipython

We now have the following code:

```

from OCC.PAF.Context import ParametricModelingContext
from OCC.PAF.Parametric import Parameters
p = Parameters()
tc = ParametricModelingContext(p)
tc.init_display()
p.height = 43.3
p.radius = 12.9
tc.register_operations(tutorial_context.prim_operations)
my_cylinder = tc.prim_operations.MakeCylinderRH(p.radius, \
p.height, name="cylinder1", show=True)

```

saved in a `tutorial_paf_part1.py` Let's enter an ipython session: `ipython -wthread tutorial_paf_part1.py` You should have the following window: the cylinder is displayed (cf. figure 1).

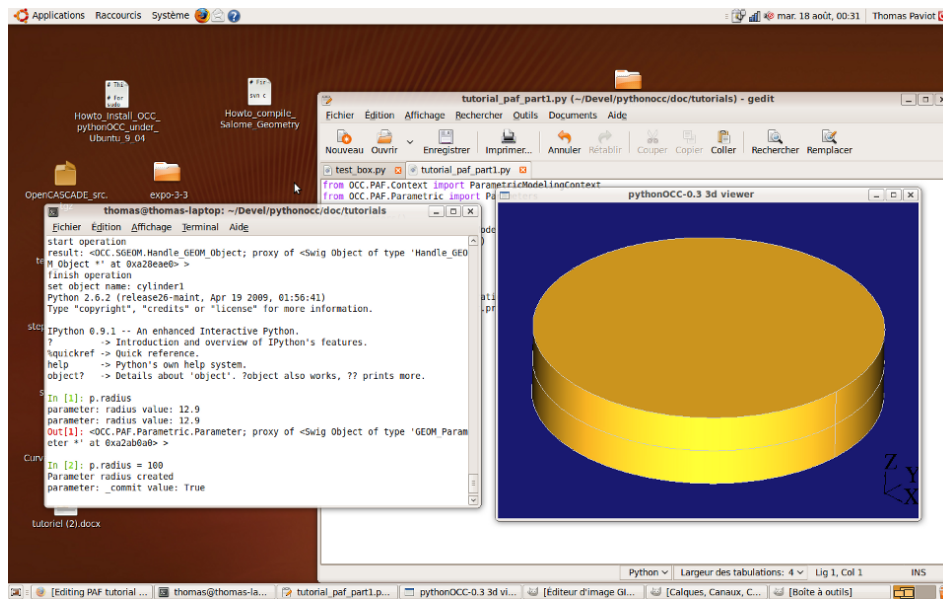


Figure 2: Impact of a parameter modification

Now just change the value of the *radius* parameter from 12.9 to 100. You'll see the modified geometry displayed in the graphic window (cf. figure 2).

### 3.3 Creation of a relation (the Relation class)

We aim now to have a kind of relation that constraints the height to equal the diameter, that is,  $height = 2 * diameter$ . The PAF uses the symbolic library `sympy`<sup>6</sup> for the computation of relations.

#### 3.3.1 Step 1 : import the Relation class

related code

```
from OCC.PAF.Parametric import Relation
```

#### 3.3.2 Step 2 : cast parameters to sympy symbols

```
symbols.radius = p.radius.symbol
height = p.height.symbol
```

<sup>6</sup><http://code.google.com/p/sympy/>

### 3.3.3 Step 3: write the relation

Write here the second member of the relation.

```
my_relation = 2*radius
```

### 3.3.4 Step 4: link the relation and the targeted parameter

```
Relation( p, "height", my_relation )
```

Take care to not forget the "" for the parameter *height*.

The complete code now looks like:

```
from OCC.PAF.Context import ParametricModelingContext
from OCC.PAF.Parametric import Parameters
from OCC.PAF.Parametric import Relation

p = Parameters()

tc = ParametricModelingContext(p)
tc.init_display()

p.height = 43.3
p.radius = 12.9

tc.register_operations(tutorial_context.prim_operations)
my_cylinder = tc.prim_operations.MakeCylinderRH(p.radius, \
p.height, name="cylinder1", show=True)

radius = p.radius.symbol
height = p.height.symbol

my_relation = 2*radius
Relation(p, "height", my_relation)
```

From an ipython session, change the value of the *radius* parameter from 12.9 to 30:

```
ipython -wthread tutorial_paf_part3.py
>> p.radius = 30
```

You'll see that the new value of the parameter *height* is 60 and have the following screen (cf. figure 3).

Note: if you change the parameter *height* value from 60.0 to another value, for instance 70.0, you will have no error message, but you will see that the height parameter has still the value 60.0. This parameter is driven by radius. Each time a parameter is changes, all relations are computed again.

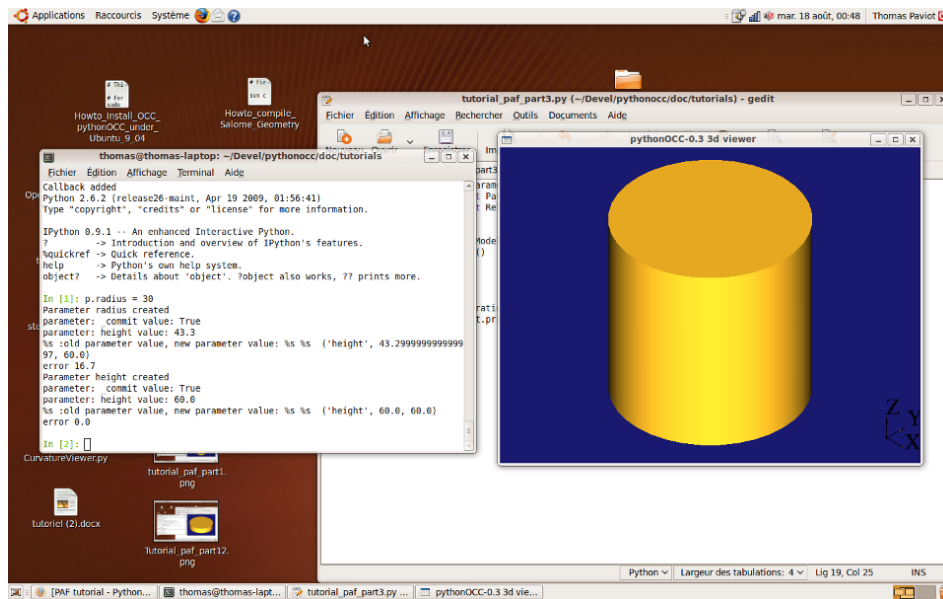


Figure 3: Relation between the *radius* and *height* parameters

### 3.4 Adding a rule (the Rule class)

From the previous sample, if you set the radius to a negative value, for instance `p.radius = -2` you will get the following error:

```

/usr/local/lib/python2.6/dist-packages/OCC/SGEOM.pyc in Update(self,
*args)
 280 def Update(self, *args):
 281     """Update(self, int arg0, TDF_LabelSequence theSeq) -> bool""" ->
 282     return _SGEOM.GEOM_Solver_Update(self, *args)
 283
 284 def UpdateObject(self, *args):
RuntimeError: Standard_Failure

```

It's because the `MakeCylindreRH` method can not build a cylinder with a negative radius. To prevent such a behaviour, let's set a rule saying that the PAF must check that the radius is positive.

A Rule is a python function/method that returns either True or False and tells the user if the rule was broken or not.

#### 3.4.1 Step 1 : import the Rule class

```
from OCC.PAF.Parametric import Rule
```

### 3.4.2 Step 2 : define the python function

In order to check that the *radius* is positive, the `IsPositive` function is defined as:

```
def IsPositive(x):  
    return x>0.0
```

Note that this basic python function can be used as usual. For instance:

```
print IsPositive(1)  
>>> True  
print IsPositive(-2)  
>>> False
```

### 3.4.3 Step 3 : creation of the Rule to connect a parameter to the python function

The rule is created with the following code:

```
Rule(p, "radius", IsPositive)
```

NOTE: each time a parameter change, all the Relation are updated and all the Rule are checked again. If one rule is not checked (that is, the python function returns False), then an `BrokeRule` python exception is raised. Let's now try the complete code within a `ipython` session.

The complete code now looks like:

```
from OCC.PAF.Context import ParametricModelingContext  
from OCC.PAF.Parametric import Parameters  
from OCC.PAF.Parametric import Relation  
from OCC.PAF.Parametric import Rule  
  
p = Parameters()  
tc = ParametricModelingContext(p)  
tc.init_display()  
# init parameters values  
p.height = 43.3  
p.radius = 12.9  
  
# create geometry  
tc.register_operations(tutorial_context.prim_operations)  
my_cylinder = tc.prim_operations.MakeCylinderRH(p.radius, \  
p.height, name="cylinder1", show=True)  
  
# define a relation  
radius = p.radius.symbol  
height = p.height.symbol  
my_relation = 2*radius  
Relation(p, "height", my_relation)
```

```
# create a rule and connect it to the parameter  
def IsPositive(x): return x>0  
    Rule(p, "radius", IsPositive)
```

Launch your ipython session:

```
ipython -wthread tutorial_paf_art4.py
```

And check that changing the radius parameter still works. For instance:

```
p.radius = 30
```

```
p.radius = 30
```

Now, enter a negative value for radius:

```
p.radius = -4
```

The BrokeRule exception is raised, since radius is not positive anymore:

```
BrokeRule: the rule with function: <function IsPositive at 0xb515dbc> broke  
with argument(s):-4
```

Now the geometry will not be updated anymore until you change the radius to a value compliant with the rule. Get back to a positive value: `p.radius = 64.8` and you'll see that it works again.