pythonocc-generator Documentation

Release review/rtd

Mar 05, 2020

Contents

CHAPTER 1

Overview of the pythonOCC SWIG files generation process

The pythonocc-generator creates automatically a python wrapper for the OCE development library. The wrapper is a set of SWIG files that need to be compiled using cmake/gcc or any other C++ compiler.

The underlying process regarding this generation is:

- parse all OCE header files (there are about 14.000) that define the OCE API
- get all class names, method, C++ enums and typedef that can be exported to python
- apply some transformations required so that those classes/methods can be accessed using python
- generate a SWIG interface that define the python API to access OCE features.

This interface is known as pythonocc-core, basically a python packagenamed OCC:

> $ python >>> import OCC

As a result, the pythonocc-core API is:

- huge: (almost) all OCE features can be used. Thousands of classes/methods are available
- very close to the OCE API itself: all transformations that come with the wrapper are only the ones necessary to get

a compileable/usable python package. No other class/method naming transformation is made. No higher level API is provided through the generation process. The result is a "raw" OCE access through python. Class names may seem ugly, for instance 'BRepPrimAPI_MakeBox', 'TopTools_MapIteratorOfMapOfShape' etc., the same for method names. Moreover, class/method naming does not follow python convention or PEP8 specification. Despite all these arguments, the decision was taken to remain as close as possible to the original library semantics to: easily port C++ OCE code to pythonocc, benefit from the original OCC documentation.

So the deal is just: provide you with an access to the OCE world using python. Nothing more.

CHAPTER 2

How does it work

## 2.1. Wrapper structure

Section 2.1.1. presents OCE library structure. Then section 2.1.2. introduces the wrapper structure, and 2.1.3. the way to go from one to the other.

### 2.1.1. OCE structure

OCE adopts the following structure

–>framework

—-> Toolkit

——> Modules

In other words, a framework is collection of toolkits, which are collections of Modules. There are about 5 frameworks, 40 toolkits, 200 Modules (I should count precisely, it's only an order of magnitude). Each Toolkit name starts with 'TK'. The list of all available toolkits can be found here: https://github.com/tpaviot/oce/tree/master/adm/cmake.

For instance, the **Foundation** framework is composed of 3 Toolkits: TKernel, TKMath and TKAdvTools. The TKernel toolkit is composed of the following modules (see https://github.com/tpaviot/oce/blob/master/adm/cmake/TKernel/CMakeLists.txt): FSD, MMgt, OSD, Plugin, Quantity, Resource, SortTools, Standard, StdFail, Storage, TColStd, TCollection, TShort, Units, UnitsAPI, IncludeLibrary, Dico, NCollection, Message. That is to say: 19 different modules for the TKernel toolkit.

After OCE is compiled, each of this toolkit results in a dynamic library: on Windows TKernel.dll, on Linux libTKernel.so, on OSX libTKernel.dylib.

Each module is composed of several classes. Each class name starts with the module name. For instance, the BRepPrimAPI module (part of the TKPrim toolkit) coes with the following classes (non exhaustive): BRepPrimAPI_MakeBox, BRepPrimAPI_MakeTorus, BRepPrimAPI_MakeSweep etc.

### 2.1.2 pythonocc-core structure

The python wrapper for OCE is a python package composed of several python modules. The base name is OCC, and the directory structure is:

–> $YOUR PYTHON$/site-packages

—-> OCC/

——> __init__.py

——> Core/

————> __init__.py

————> Module1.py

————> Module2.py

————> etc.

Each OCE module is wrapped to a python module. As a consequence, the python wrapper for OCE is a python package including about 200 different modules. Each module can be imported using its name. For instance, to use the BRepPrimAPI module, just:

All OCE classes of the BRepPrimAPI OCE module can be accessed using python:

In your OCC package path, you can see that each python module (eg Adaptor2d.py) comes with a dynamic library (*_Adaptor2d.so*).

On linux:

```
` $ cd /usr/lib/python2.7/dist-packages/OCC/Core $ ls -l total 166376
-rw-r--r-- 1 root root 32058 juil. 15 18:47 Adaptor2d.py -rw-r--r-- 1 root
root 390376 juil. 15 18:48 _Adaptor2d.so -rw-r--r-- 1 root root 137162 juil.
15 18:47 Adaptor3d.py -rw-r--r-- 1 root root 923282 juil. 15 18:48 _Adaptor3d.
so -rw-r--r-- 1 root root 162881 juil. 15 18:47 AdvApp2Var.py -rw-r--r-- 1
root root 934525 juil. 15 18:48 _AdvApp2Var.so -rw-r--r-- 1 root root 26207
juil. 15 18:48 AdvApprox.py -rw-r--r-- 1 root root 155056 juil. 15 18:48
_AdvApprox.so -rw-r--r-- 1 root root 646632 juil. 15 18:48 AIS.py -rw-r--r-- 1
root root 3562641 juil. 15 18:48 _AIS.so [...] `
```

The directory structure is the same on windows (.so is replaced with .pyd) and OSX (.so is replaced with .dyld).

Each of this library is dynamically linked to the OCE libraries on which it depends:

On Linux: ``‘ $ ldd _BRepPrimAPI.so

> linux-vdso.so.1 => (0x00007ffded1fd000) libpython2.7.so.1.0 => /usr/lib/x86_64-linux-gnu/libpython2.7.so.1.0 (0x00007f5a054f3000) libTKernel.so.9 => /usr/local/lib/libTKernel.so.9 (0x00007f5a050ac000) libTKMath.so.9 => /usr/local/lib/libTKMath.so.9 (0x00007f5a04d24000) libTKPrim.so.9 => /usr/local/lib/libTKPrim.so.9 (0x00007f5a04aca000) libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f5a047c6000) libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f5a045b0000) libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5a041eb000) libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f5a03fcd000) libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f5a03db4000) libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f5a03bb0000) libutil.so.1 => /lib/x86_64-linux-gnu/libutil.so.1 (0x00007f5a039ad000) libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f5a036a7000) libTKBRep.so.9 => /usr/local/lib/libTKBRep.so.9 (0x00007f5a033cd000) libTKG2d.so.9 => /usr/local/lib/libTKG2d.so.9 (0x00007f5a03159000) libTKGeomBase.so.9 => /usr/local/lib/libTKGeomBase.so.9 (0x00007f5a02a6b000) libTKG3d.so.9 => /usr/local/lib/libTKG3d.so.9 (0x00007f5a0274d000) libTKTopAlgo.so.9 => /usr/local/lib/libTKTopAlgo.so.9 (0x00007f5a022df000) /lib64/ld-linux-x86-64.so.2 (0x00007f5a05c91000) libTKGeomAlgo.so.9 => /usr/local/lib/libTKGeomAlgo.so.9 (0x00007f5a01bb5000)

2.1 $

(use otool on OSX).

### 2.1.3. From OCE to pythonOCC

The 2 files (the python module Module.py and the dynamic library *_Module.so*) are generated from a SWIG file Module.i

For instance, the SWIG file AIS.i, when compiled using the couple SWIG/gcc, results in AIS.py and *_AIS.so* that are copied to the /OCC/Core dist-packages folder.

The .i files are created automaticallyt using 2 python modules and a configuration file :

- Modules.py is used to specify **what** has to be wrapped
- the configuration file tells **where** it is wrapped.
- generate_wrapper.py is used to specify **how** should it be wrapped and **do** it

Each of the following section adress these 3 points in detail.

## 2.2. Modules.py: choose what to be wrapped

### 2.2.1. The OCE_MODULE list

The top of the Modules.py script reproduces the OCE structure. For instance, from lines 19 to 29:

‘‘‘ python TOOLKIT_Foundation = {

> **‘TKernel’: [‘FSD’, ‘MMgt’, ‘OSD’, ‘Plugin’, ‘Quantity’, ‘Resource’,** ‘SortTools’, ‘Standard’, ‘StdFail’, ‘Storage’, ‘TColStd’, ‘TCollection’, ‘TShort’, ‘Units’, ‘UnitsAPI’, ‘IncludeLibrary’, ‘Dico’, ‘NCollection’, ‘Message’],
>
> **‘TKMath’: [‘math’, ‘ElCLib’, ‘ElSLib’, ‘BSplCLib’, ‘BSplSLib’,** ‘PLib’, ‘Precision’, ‘GeomAbs’, ‘Poly’, ‘CSLib’, ‘Convert’, ‘Bnd’, ‘gp’, ‘TColgp’, ‘TopLoc’],
>
> **‘TKAdvTools’: [‘Dynamic’, ‘Materials’, ‘Expr’, ‘ExprIntrp’,** ‘GraphDS’, ‘GraphTools’]}

‘‘‘

The TOOLKIT_Foundation is a dictionary for which keys are toolkits and values are modules.

The most important part of Modules.py is the OCE_MODULES list. It is a list of tuples, one tuple for each modules. Each tuple is structured as followed:

(module_name, headers_to_add, classes_to_exlude, member_functions_to_exclude).

- module_name is a string, for instance ‘Dico’, ‘BRepPrimAPI’ etc.
- headers_to_add is a list of strings (OCE modules names). For instance [‘TopTools’, ‘Standard’]. It is used to tell the wrapper to include, at the

top of the generated SWIG file, a list of headers that are required so that the compilation succeeds. If ‘TopTools’ is in the list, then, in the SWIG file generated, all **TopTools_**.hxx headers will be added. Indeed, it is sometimes required to add these headers to prevent a compilation failure

- classes_to_exclude is a list of strings (OCE class names). For instance [‘Message_Msg’]. This means that this class is exluced from the wrapper. As a consequence, it won’t be available from python. This is useful to exclude classes that cause compilation failure. Some classes has to be excluded because they cause compilation failure, some of the on Linux or Windows only, others on both systems. Note that, if the wrapper was perfect, *all* classes should be wrapped, excluding is just a pragmatic way to get the wrapper compile.
- member_function_to_exclude is an optional dictionary which contains member functions of certain classes to be excluded from the wrapper. For

instance, {'Standard_MMgrOpt': 'SetCallBackFunction', 'Standard': 'Free'}). This means that the method SetCallBackFunction of the Standard_MMgrOpt won't be available from the wrapper, but all the other methods will be available. These methods are excluded for the same reason as the previous classes.

Remember that all classes and methods should be wrapped. There's not reason to exlude a priori one class or one member function. The exclusion mechanism has been developped to fix compilation issues.

### 2.2.2. Examples

Let's take the simplest example: the wrapper definition for the 'gp' module. It is defined on line 203 with:

> ('gp', [], []),

This means that all classes, and all member functions, will be made available for the wrapper.

For instance, let's go on with the 'gp' module definition:

> ('gp', [], [], {'gp_Torus': 'Coefficients'}),

Another example is the math module:

> **('math', ['Adaptor3d'], ['math_SingleTab'],**
>
> > **{'math_NewtonMinimum': 'IsConvex',** 'math_NewtonFunctionSetRoot': 'StateNumber', 'math_GlobOptMin': 'isDone'}),

The ['Adaptor3d'] list means that all Adaptor3d_*.hxx headers will be added to the SWIG file. These headers are required in order to compile the generated C++ file from the SWIG file.

The ['math_SingleTab'] list excludes the 'math_SingleTab' class from the wrapper.

The 'IsConvex' method of the math_NewtonMinimum class is excluded from the wrapper. Why ? because if included, it will cause a compilation error. The same for the 'StateNumber' method for 'math_NewtonFunctionSetRoot' and the 'IsDone' method of 'math_GlobOptMin'.

Note 1: try to remove one of the class from the exlusion list. You might see the compilation fail (according to the OS you're running, the compilation may be different).

Note 2: it would obviously be better to *understand* the compilation error, and to fix it, rather than excluding a class. Hopefully, only a few classes need to be exluded and, so far, I guess that no one ever noticed that some of these methods are unavailable. Please report any success in understanding the compilation issues (dont' report the compilation issues themselves).

## 2.3. Configuration file: where to create files

## 2.4. generate_wrapper.py : create .i files

### 2.4.1. Launch

### 2.4.2. Transformation rules

## 2.5. Misc

Some other important tips.

CHAPTER 3

Making your own wrapper

This section describes how to upgrade the wrapper for an oce version change.