lcallenbach lcallenbach@web.de

20170125

LibreOffice Calc addin based on python (version 0.1)

License

This repository consists of two different source code types: python code (for c++ code generation) and c++ code (to compile the addin for LibreOffice/Calc).

The python code for generating the addin c++ source code is under GPLv3 (see <http://www.gnu.org/licenses>), the generated c++ source code is under the QuantLib license (see <http://quantlib.org/license.shtml>) [for the total view: one Makefile for addin code compilation is a modified copy of a makefile used in the examples of the LibreOffice SDK and under the BSD license].

So what you can do (and I think most of you will do): generate the c++ code (or use the generated code) in the Calc directory which is under the QuantLib license (and if you provide source code look at the BSD license of the Makefile) and compile and install the addin. The license of the c++ code is just the license used for QuantLib in the community (see QuantLib license for details).

But: I have spent some time to write the python code for the c++ source code generation. I am interested in CopyLeft in the license and so the relevant part of the repository (the folders *code/* and *metadata/*) are under the GPLv3 so that anybody can use and extent it – but has to be aware of the CopyLeft.

Please be aware of my intended distinction between python code generation (GPLv3) and c++ code for the addin compilation (QuantLib license).

Introduction

The OpenOffice / LibreOffice Calc addin exists for a few years. Currently you can download an alternative implementation of addin code generation under the official QuantLib download path.

So: why another implementation?

I have used QuantLib at my employers for clients in the financial area for many years (audit and independent validation). It is c++ based and (if you know the necessary structures) easy to extend and to customize to your needs. But (c++ and boost etc.) not everyone understands it – due to missing c++ know how or due to missing knowledge in financial modelling.

The other/old implementation for LibreOffice (and OpenOffice) is based on XML files describing the addin functions. E.g. if you would like to have a class represented in memory in Calc you have to define a constructor in one XML file with all the parameters for the class invocation, one XML file for the class interface handling (how to convert Calc objects to addin objects) and some fixed c++ code for the mapping of the interface code to the QuantLib class constructor (e.g. if you have predefined variables which are not part of the interface). So there are a lot of mappings in XML files and hand-written c++ code for the addin. Unfortunately there is another difficulty (for me): since you have all

freedoms for order and appearance (think of default values) of the variable for the addin it is not clear which variabel has what meaning in the constructor. And another question is the hidden type of the QuantLib type of the parameter (there is no obvious link). All these mappings are quite complex and or with a lot of predefined values (not known in the interface description). Looking at these points it is complicated to look at the underlying QuantLib object if you have a question concerning the parameters in the addin function (and you need a c++ aware programmer to find the relevant code in the QuantLib addin code and in the QuantLib code – you need some open source code and XML files to find your way).

In the current implementation I have used another approach which is related to the QuantLib c++ code and the QuantLib types. It is similar to SWIG – but you need not redefine all of the classes and reproduce their interdependence (inheritance and so forth). The intention is to define a python tuple like

```
1   (“Constructor",
2   "ql/termstructures/yield/zerocurve.hpp",
3   "QuantLib::ZeroCurve(std::vector<QuantLib::Date> Dates, std::vector<QuantLib::Real> ZeroRates,
4           QuantLib::DayCounter DayCounter=QuantLib::Actual365Fixed())", "",
5   "qlZeroCurve")
```

If you lokk at lines 3-4 you can see the constructor of the ZeroCurve class which is defined in zerocurve.hpp. This is just a copy of the constructor enriched with namespace information – thats it. These five lines describe the type of the addin function (line 1), its c++ header file location (line 2), the constructor invocation (lines 3-4) and the name of the addin function (line 5).

In short: the parameters for constructors are identical to the parameters in QuantLib header files for classes and all interface code will be generated automatically. And in the addin interface description in Calc the generated addin code will present the QuantLib variable type for each parameter listed in the addin function (so you know what type of object you should fill in – if you know the QuantLib code). So e.g. in the help of the variable “Date” the type will be displayed as text, namely “QuantLib::Date”.

I have chosen a similar approach for member function of QuantLib classes, e.g.

```
1   ("MemberFunction",
2   "ql/termstructures/yieldtermstructure.hpp",
3   "QuantLib::YieldTermStructure()",
4   "QuantLib::DiscountFactor discount(QuantLib::Date Date)",
5   "qlYieldTSDiscount")
```

This is the definition of the addin function “qlYieldTSDiscount” calling the QuantLib function “discount” of the class “YieldTermStructure”.

For constructors and member functions a further argument will be added in the first position of the parameter list of the addin function for the name of the object in the repository which is referenced (e.g. the yield curve name for the discount() member function).

A side remark: assigning the member function to the right class enables the code to get rid of the coerce handling of the old code. Since the ZeroCurve is a derived class of YieldTermStructure you can invoke the discount member function for a ZeroCurve – in the generated code this will be done by down-casting.

If you know the structure and interdependence of QuantLib classes and their functions/procedures it is simple to understand the creation of objects in the addin and the meaning of variables

I have structured the document in the following chapters: in the next chapter I give a short overview of how the addin works. The next chapter describes the code generation with python code. In the last chapter I describe the current types of QuantLib addin objects that can be defined.

TODO

LibreOffice Calc addin

Overview of c++ addin code generation

The following picture presents a quick overview:

1. *read configuration* — *class GensrcConfiguration*
2. *read metadata* — *class ParseMetaData*
   - *read type conversions*
   - *read constructors, memberfunctions, functions, enumerated classes and enumerated types*
3. *generate LibreOffice IDL file and addin code (storage of code in files) in addin.calc.py*
   - *IDL and addin specific code* — *addin.calc.py*
   - *code for enumerated types* — *codegeneration.enumeratedobjects.py*
   - *code for the base classes* — *codegeneration.enumeratedclasses.py*

Types are analysed for each variable by the class Parameter (parser.parameter.py). The information for constructors is analysed by the class Constructor (parser.constructor.py), for functions by the class MemberFunction (parser.memberfunction.py).

In the addin you have to decide whether a variable is an object of the repository, a native type (e.g. float, string) or should be created “on the fly” (e.g. enumerated class). The idea is the following: if a type is not listed in the conversions then it is a type of an object (and is stored with a name as key in the repository). Native types and enumerated types are defined in the conversions.

QuantLib Addin types

lcallenbach lcallenbach@web.de