lcallenbach lcallenbach@web.de

20170301

LibreOffice Calc addin based on python (version 0.91)

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 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 variable has what meaning in the constructor. And another question is the hidden type of the QuantLib object of the parameter (there is no obvious link). All these mappings are quite complex and with predefined values (not 'known' by the data in the interface description). Looking at these points it is complicated to find out the data flow from the interface to 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 XML files, QuantLib addin code and in the QuantLib code – so you must traverse some c++ source code and XML files to solve this problem).

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 for variable types – that's 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 code necessary to generate the addin will be generated automatically. Calling an addin function in Calc you will see in the description of the parameter its QuantLib type (defined in lines 3-4). So you know what type of object you should provide for each parameter of the addin function and – if you know the QuantLib code – it is much easier to interpret the parameters. 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 invocation of the discount() member function).

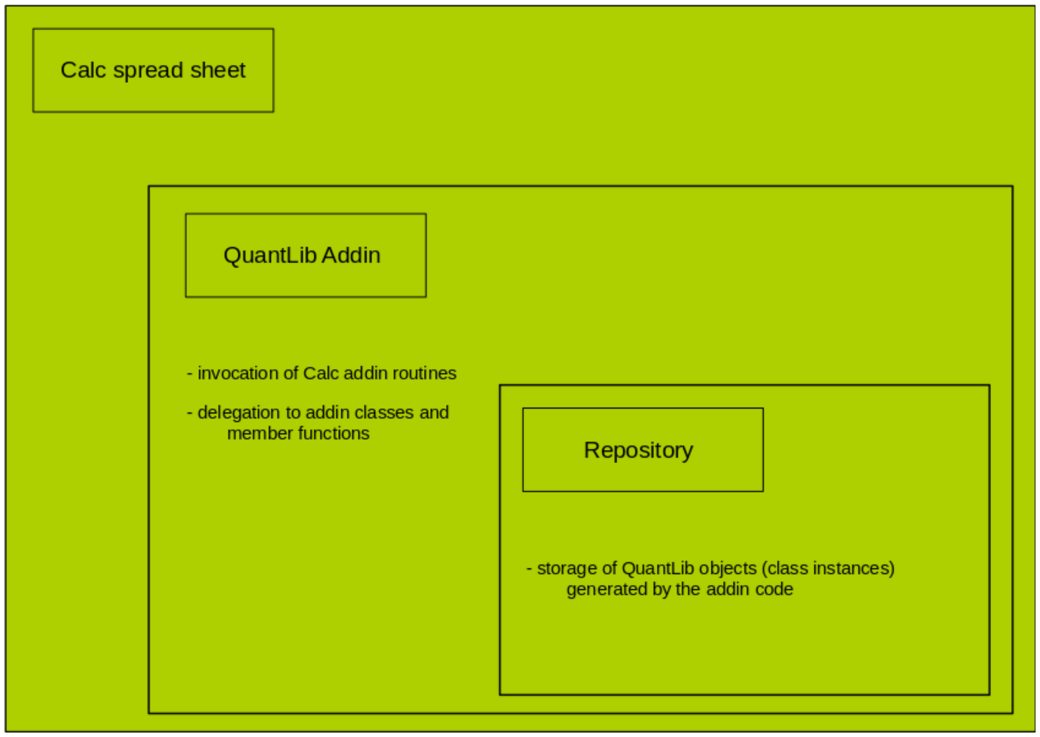
A side remark: assigning the member function to the right class enables the code to get rid of the coerce handling of the old code. Since the ZeroCurve is a derived class of YieldTermStructure you can invoke the discount member function for a ZeroCurve – in the generated code this will be done by (down-)casting.

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 different types of QuantLib addin objects for source code generation and how objects should be defined in the metatdata files. This part will be relevant for you if you want to extend the addin functions and add / modify meta data for c++ code generation.

LibreOffice Calc addin

The following picture shows in a simple and abstract manner the interdependence of different parts of the addin.


Calc spread sheet
QuantLib Addin
- invocation of Calc addin routines
- delegation to addin classes and member functions
Repository
- storage of QuantLib objects (class instances) generated by the addin code

After loading the addin you can access Quantlib addin functions. These functions interact with the internal repository which e.g. stores the QuantLib classes defined as constructors (calling a constructor addin function simply means creating an instance of a QuantLib class).

I will give you an example for interest yield curves: if you define a QuantLib::ZeroCurve with the addin function qlZeroCurve you have to define a name for this curve which can be used in the spreadsheet (which is a key in the repository for the QuantLib object which will be set up) and to provide the remaining parameters (QuantLib::Date array, QuantLib::Rate array, …). The QuantLib addin code (generated in qladdin.* in the Calc folder) will delegate the analysis of the parameters to the addin code (generated in addin.*). In the addin code the interaction with the repository is defined and the QuantLib functions/constructors are called.

The interface of the addin code is of the type CSS::uno::any. Under LibreOffice/Calc it is possible to analyse the content of the variable using this type (looking whether a value is provided by the user or a default value should be used). This analysis is not possible for native types like double or int.

The return type of all addin functions is a matrix of type CSS:uno::any. The return types of QuantLib functions and constructor calls will be converted to this type. Using the return type CSS:uno::any you can return a string, a double or a float – so it is possible to return the double value of a discount factor or e.g. the string of an exception if the discount factor can not be calculated. If you choose a wrong parameter for the addin function you can 'see' the error message in a spread sheet in the cell with the addin function call.

The addin for Calc consists of two logical components: one is specific for the Calc interface (the code generated in qladdin.* in the Calc folder) and one which defines the handling of the data provided by the interface (the code generated in addin.*). The data structures in addin.* are 'generic' in the sense that with minor modifications in the meta data you can reuse the logic to generate the code. If you have time and ambitions it is up to you to write an interface handling for Excel or other software – just by using the addin code logic and writing new interface code. If you choose e.g. Excel you have to define new mappings of QuantLib data types to interface data types.

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, member functions, functions, enumerated classes and enumerated types	
3.	*generate LibreOffice IDL file and addin code (storage of code in files)*	
	IDL and addin interface code	*code.addin.calc*
	addin code for enumerated types	*code.codegeneration.enumeratedobjects*
	addin code for the remaining classes	*code.codegeneration.addinclasses*

After reading the configuration in the file 'configuration.txt' (1.) the metadata for c++ code generation are read (2.). The metadata files define the handling of data types (ImplicitConversions) and objects of the addin (QuantLibTypes). If you analyse objects you have to parse parameter lists and return types. Parameter types/classes are analysed for each variable by the class Parameter (code.parser.parameter). The information for constructors is analysed by the class Constructor (code.parser.constructor), for functions by the class MemberFunction (code.parser.memberfunction).

In the addin you have to decide whether a parameter of a function call is an object of the repository, a native type (e.g. float, string) or should be created "on the fly" (an enumerated object). The idea for parameter type conversions is the following: if a type is not listed in the conversions then it is a type of an object (and is stored with its name as key in the repository). Native types and enumerated types must be defined using type conversions.

lcallenbach lcallenbach@web.de

QuantLib Addin types and functions

In the first part I give you an overview of different types of objects used by the addin code generation. This part is related to the ImplicitConversions in the meta data files. The next section describes objects that can be created using the current implemented QuantLibTypes.

ImplicitConversions

Before I proceed: always keep in mind that data types not defined in conversions will be treated as strings and then have the meaning that these data types are data types of objects in the repository.

An overview of different applications of conversions is given in metadata.types, metadata.time.date and metadata.termstructures.yield.bootsraphelper. The following listing is from these files:

```
1    ("QuantLib::Matrix",     "double",    "double",    "interfaceToMatrix", "interfaceFromMatrix", 2)
2    ("QuantLib::BigInteger",    "long",      "long",      "Default", "Default")
3    ("QuantLib::Integer",      "long",      "long",      "Default", "Default")
4    ("QuantLib::Date",  "QuantLib::Date", "long", "interfaceToDate", "interfaceFromDate")
5    ("QuantLib::Pillar::Choice",  "std::string", "string", "EnumeratedType", "", "<<")
6    ("QuantLib::Calendar",  "std::string", "string", "EnumeratedClass", "", "name()")
```

In code.configuration.parsemetadata these tuples are parsed in their order and mapped to the named tuple '*Conversion*' with the components *'typ cpp idl fromidl toidl flag'*. The meaning of these names is the following:

- typ: relevant data type which has to be defined (e.g. QuantLib::Matrix)
- cpp: c++ code data type of the addin code after conversion from the interface (meaning with dimension=0, e.g. QuantLib::Matrix stores double values)
- idl: interface data type corresponding to the data type
- fromidl: conversion function from the interface to the c++ addin code type (this means to the type defined in 'cpp')
- toidl: conversion function from the addin code type to the interface code type (this means to the type defined in 'idl')
- flag: separate flag (interpreted as a QuantLib conversion function defined for the QuantLib type/class if 'fromidl' is an enumerated object or as the dimension of 'typ' (either scalar with dimension 0, vector like with dimension 1 or matrix like with dimension 2)

In line 1 of the example above the QuantLib class Matrix is of dimension 2 with predefined conversion functions. The type QuantLib::BigInteger will be handled as a native long type in the addin code. For native types with 'cpp'=='idl' the conversion setting 'Default' can be chosen (no conversion function is required). QuantLib::Date in line 4 is similar. But the handling in the interface is of type 'native long' and should be converted to the type QuantLib::Date using the conversion function 'interfaceToDate'. In line 5 an example for the enumerated type QuantLib::Pillar::Choice is given. Since 'fromidl'=='EnumeratedType' and 'flag'=='<<' the conversion of this type is defined in the QuantLib code where '<<' has been defined for QuantLib::Pillar::Choice. Enumerated objects require more definitions in QuantLibTypes (strings representing their value). Line 6 declares calendars to be of enumerated class type. The QuantLib::Calendar() class function name() provides its name in the interface code.

Conversion to higher dimension will result in 'cpp' types of type *std::vector<...>*. This is an implicit assumption, i.e. *std::set<>* or *std::map<>* are currently not supported (and not required up to know).

QuantLibTypes

There are some types which are defined:

1. constructors and member functions (you have seen examples in the introduction)
2. enumerated objects (of ‘typ’==’EnumeratedType’ or ‘EnumeratedClass’)
3. template arguments

We will look at them in the order above. The relevant named typles in python are defined in code.configuration.parsemetadata.

Constructors and member functions are stored in the named tuple ‘Type’ with the components '*typ incfile class_name memfunc addin_name'* and the following meaning:

• typ:	‘Constructor’, ‘MemberFunction’ or ‘Function’
• incfile:	c++ include file
• class_name:	constructor call or name of the class of the member function
• memfunc:	name of the function or member function
• addin_name:	name of the addin in interface function (name in the spread sheet)

Class_name and/or memfunc will be parsed (analysing the input and return parameters). Each parameter has a dimension (dimension==1 for 1D array objects or dimension==2 for 2D array objects) and some further parameters like a default value (if provided in the meta data file). Constructor and member functions are the basis for the interface code generation in the files Calc/qladd.* and for the addin code genearation in the files Calc/addin.*. For member functions you should provide the class to which it belongs in the original c++ code. The addin code will (down-)cast objects to the the type of ‘class_name’ provided in the member function declaration.

Not all QuantLib classes need to be stored in the repository. Objects which require an amount of calculation time are ‘expensive’ and each on the fly generation costs performance. So these objects are candidates for the repository. But performance is not an issue for simple c++ classes like calendars or type definitions (e.g. types to indicate a payer swap / receiver swap or an option of type call / put). The idea of enumerated objects is to define rules for translating interface input data of type ‘string’ to QuantLib objects on the fly (inline in the addin code).
Enumerated object (‘EnumeratedType’ and ‘EnumeratedClass’) definitions are stored in the named tuple ‘Enumeration’ in the python code with the components '*typ incfile class_name cpp arg value*'. Examples for enumerated objects are provided in the files metadata.time.calendars and metadata.option:

```
1    ("EnumeratedType", "ql/option.hpp", "QuantLib::Option::Type", "std::string", "Call",
            "QuantLib::Option::Call")
2    ("EnumeratedClass", "ql/time/calendars/target.hpp", "QuantLib::Calendar()", "std::string",
            "TARGET", "QuantLib::TARGET()")
```

The components have the following meaning:

• typ:	either ‘EnumeratedClass’ (object with constructur) or ‘EnumeratedType’ (object without constructor like an named enumeration
• incfile:	c++ include file with declaration
• class_name:	c++ class name
• cpp:	interface code representation type (e.g. std::string for definitions as text parameters)
• arg:	the interface parameters

- value: the c++ code object which should be generated by invocation with parameter ‘arg’

Interpreting line 2 you see that the parameter “TARGET” of type std::string should represent the object QuantLib::TARGET(). So if the addin code requires a calendar you can provide the string ‘TARGET’ and the calendar QuantLib::TARGET will be used in the addin. For each enumerated object type you should have a declaration in ImplictConversions and mapping rules in QuantLibTypes.

Some QuantLib classes require template arguments (e.g. for bootstrapping of yield curves in QuantLib::PieceWiseYieldCurve). The constructor definition in metadata.termstructures.yield.piecewiseyieldcurve looks like

("Constructor", "ql/termstructures/yield/piecewiseyieldcurve.hpp",
"<Traits, Interpolator>QuantLib::PiecewiseYieldCurve(QuantLib::Date, ..., QuantLib::Real Accuracy)",
"", "qlPiecewiseYieldCurve")

This metadata definition will be parsed with two template arguments with the name ‘Traits’ and ‘Interpolator’. You have to define a conversion from a parameter (which is of interface type string) to a QuantLib type using the type ‘TemplateArgument’ with components ‘*typ incfile name cpp arg value addin_name*’. Examples are given in the following lines

1 ("TemplateArgument", "ql/termstructures/yield/bootstraptraits.hpp", "Traits", "std::string", "ZeroYield", "QuantLib::ZeroYield", "qlPiecewiseYieldCurve"),
2 ("TemplateArgument", "ql/termstructures/yield/bootstraptraits.hpp", "Traits", "std::string", "Discount", "QuantLib::Discount", "qlPiecewiseYieldCurve"),
3 ("TemplateArgument", "ql/termstructures/yield/bootstraptraits.hpp", "Traits", "std::string", "ForwardRate", "QuantLib::ForwardRate", "qlPiecewiseYieldCurve"),
4 ("TemplateArgument", "ql/math/interpolations/linearinterpolation.hpp", "Interpolator", "std::string", "Linear", "QuantLib::Linear", "qlPiecewiseYieldCurve"),
5 ("TemplateArgument", "ql/math/interpolations/cubicinterpolation.hpp", "Interpolator", "std::string", "KrugerCubic", "QuantLib::Cubic", "qlPiecewiseYieldCurve"),

Each template argument is a further argument of type ‘string’ for the addin function. If you provide ‘KrugerCubic’ it will be replaced by ‘QuantLib::Cubic’ in the addin code.

The component ‘addin_name’ determines for which addin function this definition should be used. The meaning of the other components are similar to the meaning for enumerated objects.

In the previous example the relevant template arguments for an addin function will be analysed and each possible template argument combination will be implemented in the addin code (for PieceWiseYieldCurve 2 x 3 = 6 different combinations are generated in the addin code).

If you provide example metadata files please use the following convention:

1. variable names should not be abbreviated – they should be at minimum the (probably) short name in the C++ code. As a guide I either use the name in the C++ code or the QuantLib class type as name.
2. Start the variable name with an upper case letter.

lcallenbach lcallenbach@web.de

Changes

0.90	(20170128)	initial version
0.91	(20170301)	added conventions for variable naming in last part of QuantLibTypes