Algorithmica Research AB

The Quantlab API

for Quantlab version 3.1.1606 and later

Table of Contents

Introduction to building Quantlab DLLs in C++

The standard function library of Quantlab may be enriched by objects and functions written by the user in a C++ environment. The objects and functions are written in a dll that communicates with Quantlab through the Quantlab API.

When Quantlab is started, it loads all dll-files located in the ql_libs directory. These dll-files may add objects and functions into QLang (the Quantlab language), which then can be used from within Quantlab just like any of the original Quantlab objects and functions.

Any examples given in this document refer to Microsoft Visual C++ version 6.0 or later. If you are using any other development platform, some changes may be needed.

The C++ Project

Start by creating a C++ project that produces a dll as output (In Microsoft Visual C++: File – New – Project – Win32 Dynamic-Link Library). The project name is not important as long as the name of the output dll is unique.

Add the “qlab31.lib” file in the project to enable linking your code to the Quantlab types and functions.

Include the “ql.h” header file (located in the Quantlab\ql_api\include directory) in every source file, and make sure the preprocessor has access to the directory where it is located (In Microsoft Visual C++: Project – Settings – C/C++ – Category: Preprocessor – Additional include directories).

The file “ql.h” contains all header files used by Quantlab:

"ql_object.h"

"ql_func.h"

"ql_base.h"

"ql_date.h"

"ql_error.h"

"ql_string.h"

"ql_calendar.h"

"ql_curve.h"

"ql_fit.h"

"ql_fit_result.h"

"ql_instrument.h"

"ql_model.h"

"ql_realtime.h"

"ql_rng.h"

"ql_support.h"

"ql_tag.h"

The compiled dll file has to be saved in the folder that you specify in the second path in Tools Options in Quantlab (DLLs used by Quantlab).

Programming

Your dll should export the entry point ql_init, which Quantlab will call when loading. The only argument to ql_init is a context, the Quantlab container for types and functions. Your code should therefore contain something looking like:

```
__declspec(dllexport) void ql_init(QL::symtab &ctxt)
{
//Code for adding types and functions here
}
```

It is to the symtab object that your types and functions are added. Any number of types and functions can be added to QLang from one ql_init function.

1.1 Adding a function to Quantlab

A function is added to QLang by specifying the arguments it takes, the return value type and finally adding the function to the context. The syntax of add_func is:

```
      void   add_func(QL::symtab        &ctxt,
                     function_t         *f,
                     const char         *name,
                     const char         *category,
                     const arg_spec            &return_type,
                     int                n_args,
                     const arg_spec            *args,
                     const char         *help_url    = 0,
                     const char         *com_name    = 0);
```

The next to last parameter indicates where the help text is located. However, currently the function browser in Quantlab does not support the user defined help_url pages. The last parameter is for overloaded functions: If you define two functions (taking different parameters) with the same name they have to have different alias names in order to be exposed in Visual Basic.

The return value and each element of the argument vector should be an initialized object of type arg_spec.

The following example shows how to define a function that increases a number by one or by a number specified as second parameter. The example also shows how to use arg_spec and the pointer to a function_t object.

```
#include "ql.h"
static double my_increase_function(const double x, const double y){
      return x + y; //A very simple function indeed!
}
__declspec(dllexport) void ql_init(QL::symtab &ctxt)
{
    QL::arg_spec    r;
    QL::arg_spec    a[2];
    //The return value is of type number
    r    = QL::arg_spec("number");
    //The two argument are numbers called x and y.
    //The optional argument y has a default value of 1.
    a[0] = QL::arg_spec("number", "x");
    a[1] = QL::arg_spec("number", "y", "1");
    //The function my_function is added to the category "My category" in
    //Quantlab and is called "increase".
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(my_increase_function),
               "increase", "My category", r, 2, a);
}
```

Tip! The default value in the arg_spec initializing (the 3rd argument) is treated as a Quantlab expression, except that it cannot contain any conditionality or iterations (like series). QLang function calls are allowed, however.

Save the compiled dll file in the folder that you specify in the second path in Tools Options in Quantlab (DLLs used by Quantlab), restart Quantlab and try your function.

1.1.1 Function return types

Depending on the return value type, your functions must have a special syntax. All types are listed in Appendix 1.

- Functions having as return value type any type that is represented by a double should return a double.
- Functions having as return value type any type that is represented by a string should return void and have a reference to a Quantlab handle as first argument. The return string itself must be created in the function, using the function create_string.
- Functions having as return value type any type that is represented by a Quantlab handle should return void and have a reference to a Quantlab handle as first argument.
- Functions having as return value type any type that is represented by a Quantlab handle_t (e.g. one of your own object types) should return void and have a reference to a Quantlab handle_t as first argument.

1.1.2 The symtab argument

Some API functions need a Quantlab symtab as argument, in general because they need to add instruments to the context. To receive a symtab object to your function, you must make the argument vector one element longer and initialize the last element using the ctxt_arg function. You must also increase the number of arguments (n_arg) parameter by one when calling add_func to add your function to QLang. From within Quantlab your function will look identical, but in your code your function now takes a context as the last argument. See Appendix 3 for a complete example where the context is used.

1.2 Adding an object type or a type name to Quantlab

1.2.1 Adding an object type

An object type is added to QLang simply by adding the object type name to the Quantlab context. The syntax of add_object_type is:

```
void   add_object_type(context    &ctxt,              //The context (argument to
ql_init)
                    const char    *object_type,       //Object type name
                    const char    *superclass = 0,    //Reserved
                    const char    *help_url = 0);           //Url to help file
```

To use your new object type, you should use the Quantlab typed reference handler called handle_t, see further down.

1.2.2 Adding member functions to an object type

Adding a member function to the object type you have created is very similar to adding a normal function to QLang. The syntax of add_mem_func is:

```
      void   add_mem_func(context             &ctxt,
                          function_t          *f,
                          const char          *object_type,
                          const char          *name,
                          const arg_spec &return_type,
                          int                 n_args,
                          const arg_spec *args,
                          const char          *help_url     = 0,
                          const char          *com_name     = 0);
```

The function pointed at should take the object itself as the first argument, or second if the function returns a handle.

In Appendix 3 there is an example showing how to add the object type “pair” to QLang with a creation function and member functions.

1.2.3 Adding a type name

A type name, being a special name for an already existing object type, is added to QLang using add_type_name. The syntax of add_type_name is:

```
void   add_type_name(context      &ctxt,       //The context (argument to ql_init)
                    const char    *type_name,  //Type name
                    const char    *type,       //Object type name
                    const char    *help_url = 0);     //Url to help file
```

You may treat your new type name as an object type when adding functions to QLang (i.e. arg_spec will recognize it), but for programming purposes the type name is ignored and only the object type is used.

1.3 Accessing Quantlab functions

The Quantlab API function declarations can be found in the header files mentioned earlier. They contain most functions in QLang and some additional functions that Quantlab uses internally. The names of API functions are mostly self-explanatory and very similar to the names in QLang. To ensure unique function names in the API some explanatory prefix or suffix might be added to the function name. Some examples of this are:

QLang function name	**API function name**
curve	curve_create_db
ns	model_create_ns
label (for date graphs)	label_date_create
label (for period graphs)	label_number_create

For functions that are part of QLang it is probably easiest to look in Quantlab for the syntax and usage notes. The API syntax is the same as the QLang syntax. The exceptions to this are the member functions, described below.

The functions that are not available within Quantlab are typically dealing with the compound types (series, vectors and matrices), and are used for example for indexing. Functions for compound types are found in “ql_base.h”.

There are a few QLang functions that are not available through the API. These are in general quite advanced such as horizon analysis.

In Appendices 3 and 4, there are a number of examples of how to access QLang functions.

1.3.1 Object member functions

The dot notation to access member functions is currently not available in the API. There are, however, API functions that can be used for the same purpose. The name of the function starts with the object type name, then underscore, then the name of the member function. Some examples:

QLang member function name	**API function name**
instrument.yield	instr_yield
fit_result.simple_rate (using dates)	fit_result_simple_rate_dc
label_date.text	label_date_text

The argument list of the API function starts with the object itself, followed by the usual member function parameters.

In Appendix 3, there are a number of examples of how to access Quantlab object member functions.

1.4 Referring to Quantlab object types and member functions

1.4.1 Using the handle object

The handle is used as reference to most original Quantlab object types. The object itself cannot be accessed in any way, but the handle can be used as argument to Quantlab functions that extracts the sought information. For example, a function taking a series of numbers and returning a number would look something like:

```
double a_series_function(const QL::handle &s){}
```

If you are adding your own object type to Quantlab, you should use the typed reference object template handle_t. A function taking a my_type and returning a number looks like:

```
double my_function(const QL::handle_t<const my_type> &m){}
```

Typedef is recommended in order to avoid long handle expressions:

```
typedef QL::handle_t<const my_type> my_type_pointer;
double my_function(const my_type_pointer &m){}
```

1.4.2 Accessing object member functions

As stated above, the dot notation is not implemented yet and therefore there are no object member functions. Instead, the corresponding functions are accessed using the object as a parameter, see 1.3.1 above.

1.4.3 Using strings

A string, and any type name that is represented by a string, is referred to using a character pointer (const char *) when it is argument to a function. Within the function the string is then accessed and handled normally.

Strings as return values use the Quantlab handle, and must be created using the create_string function.

1.4.4 Using doubles and integers

Types represented by a double do not need references. Note, though, that there are no integers in QLang. In case your want an integer as parameter for your function, you must receive a double as argument and then test if it is an integer.

1.5 Null handling for compounded types

When using the compound types (series, vectors and matrices) it is possible for elements to be invalid. At any time that you extract an element from a compound type you should therefore test if it is valid or not before continuing. The testing functions are presented in Appendix 1.

Warning! Quantlab functions do not handle invalid arguments, so they will cause Quantlab to crash. Therefore, validation tests are essential.

1.6 Error handling

You may call the throw_error function to generate an error. This throws an exception that is caught by Quantlab, which then takes over, sends your error message to the Quantlab Warnings window and sets the result of the current branch of calculations to Null.

Using the COM interface

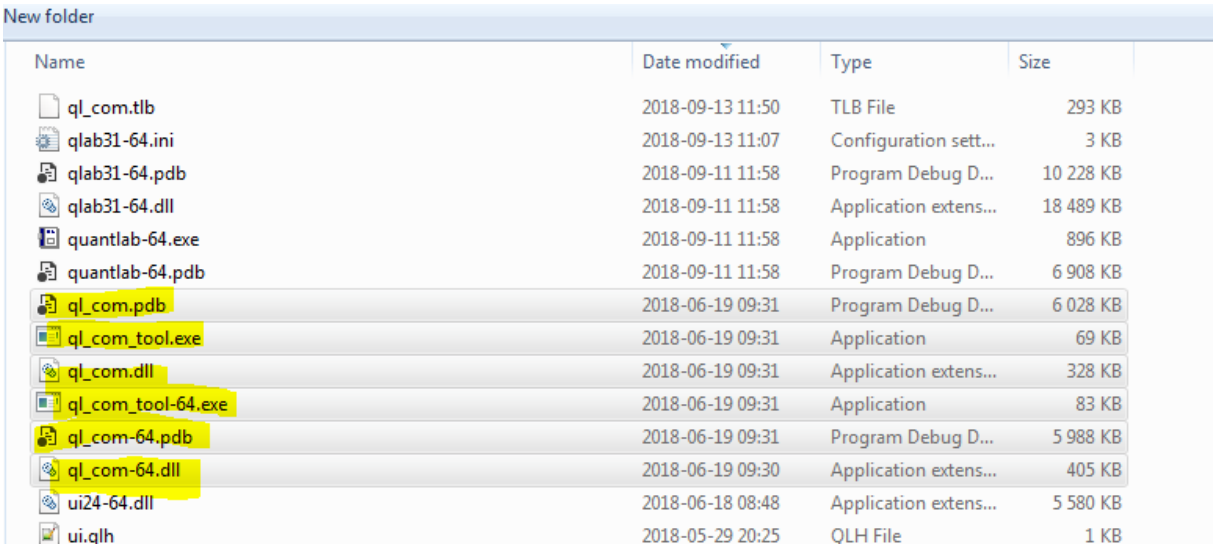
1.7 Using the COM interface in VBA

All functions available within Quantlab may be called from any COM-aware program such as Microsoft Excel or Microsoft Visual Basic. This includes not only the built-in library of financial functions but also user-written functions implemented in Qlang (and saved as a library file), as well as compiled functions implemented as a Dynamic Link Library (DLL).

1.7.1 Generating the tlb file

Before you try to access your own library functions and DLL:s you need to generate a .tlb-file (type library file). Using this file Excel (or Visual Basic) knows enough about your functions to include them in the Object Browser. To produce such a file from Quantlab, proceed as follows:

Install the x86 and/or the x64 version(s) of the com dll and the exe plus a suitable tlb-file (ql_com.tlb) next to quantlab-64.exe in a quantlab installation.

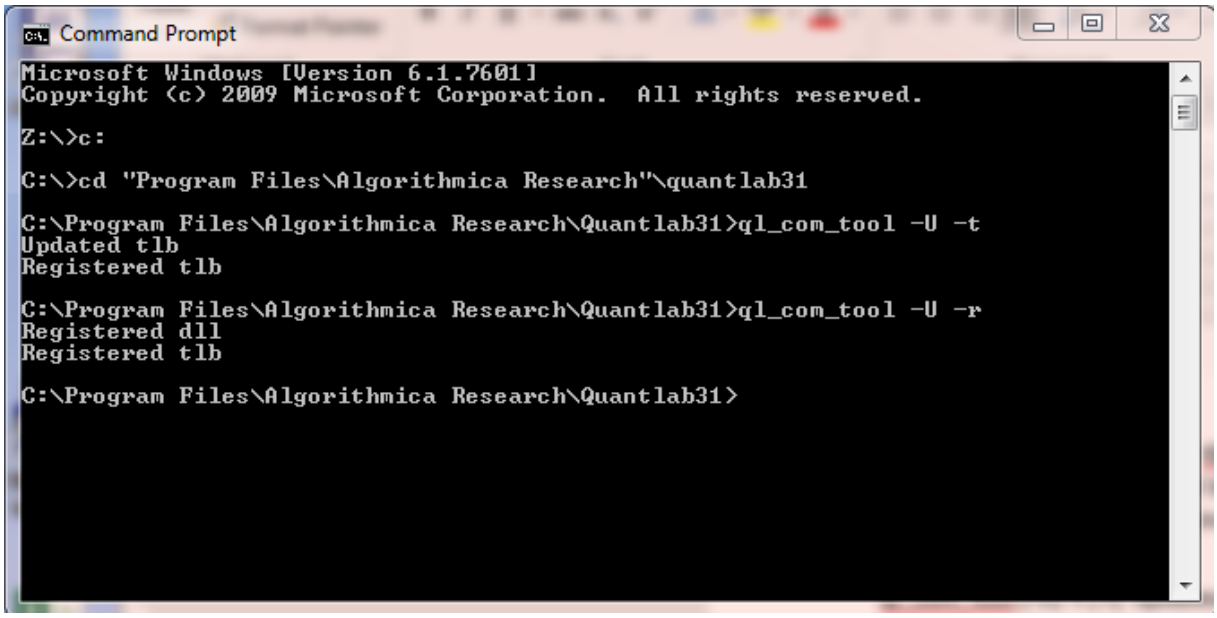


Generate and administer the tlb file using a command promt set to the current Quantlab working directory. In general, it should be enough to run the command window as user (using the -U option below.) If that does not work you might need to run the command promt with administrator privileges in order to register to tlb file.

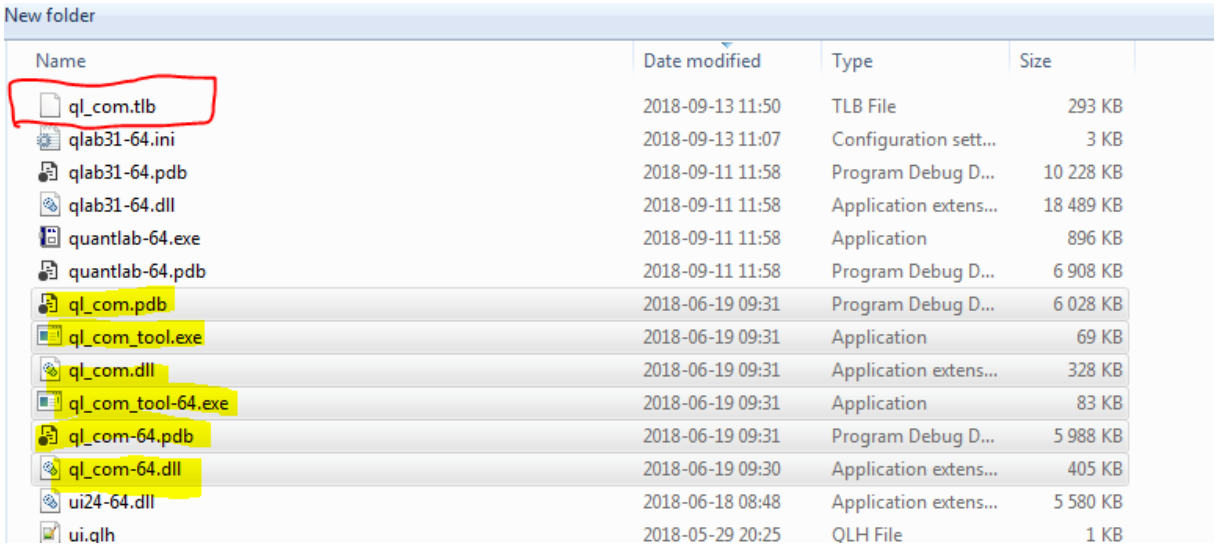
'ql_com_tool [-U] -t [-f]' updates/generates a tlb-file and registers it. '-U' means registration per-user, and '-f' forces the creation of a tlb file if one is not present or is invalid.

'ql_com_tool [-U] -r' registers the com library dll and tlb.

'ql_com_tool [-U] -u' unregisters the com library and tlb.



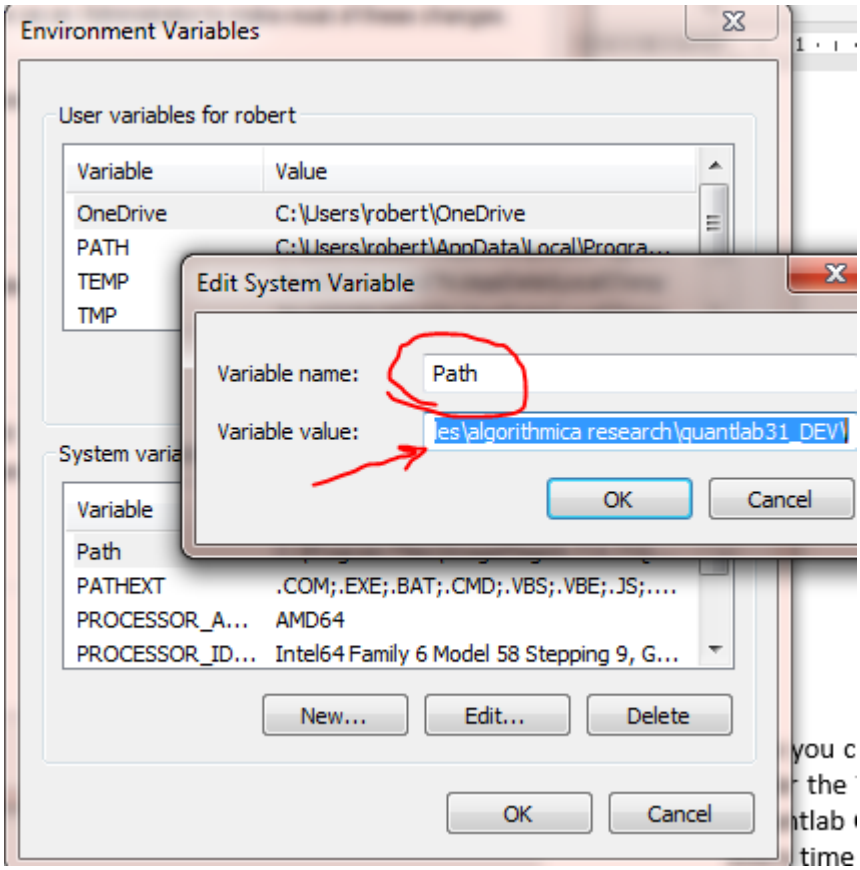
This results in a ql_com.tlb file being generated in your Quantlab folder. The file includes information about all functions currently available in your library files and DLL:s. Note that you need to generate a new .tlb file if you update your library files (or DLL:s) and want the new functions to be available via the COM-library.



1.7.2 Setting Environment PATH

Before starting Excel to run the com library, the environment path must point to where you have placed the ql_com.dll file. This in order for excel to find it in the list of available libraries in Tools | References in the VBA editor.

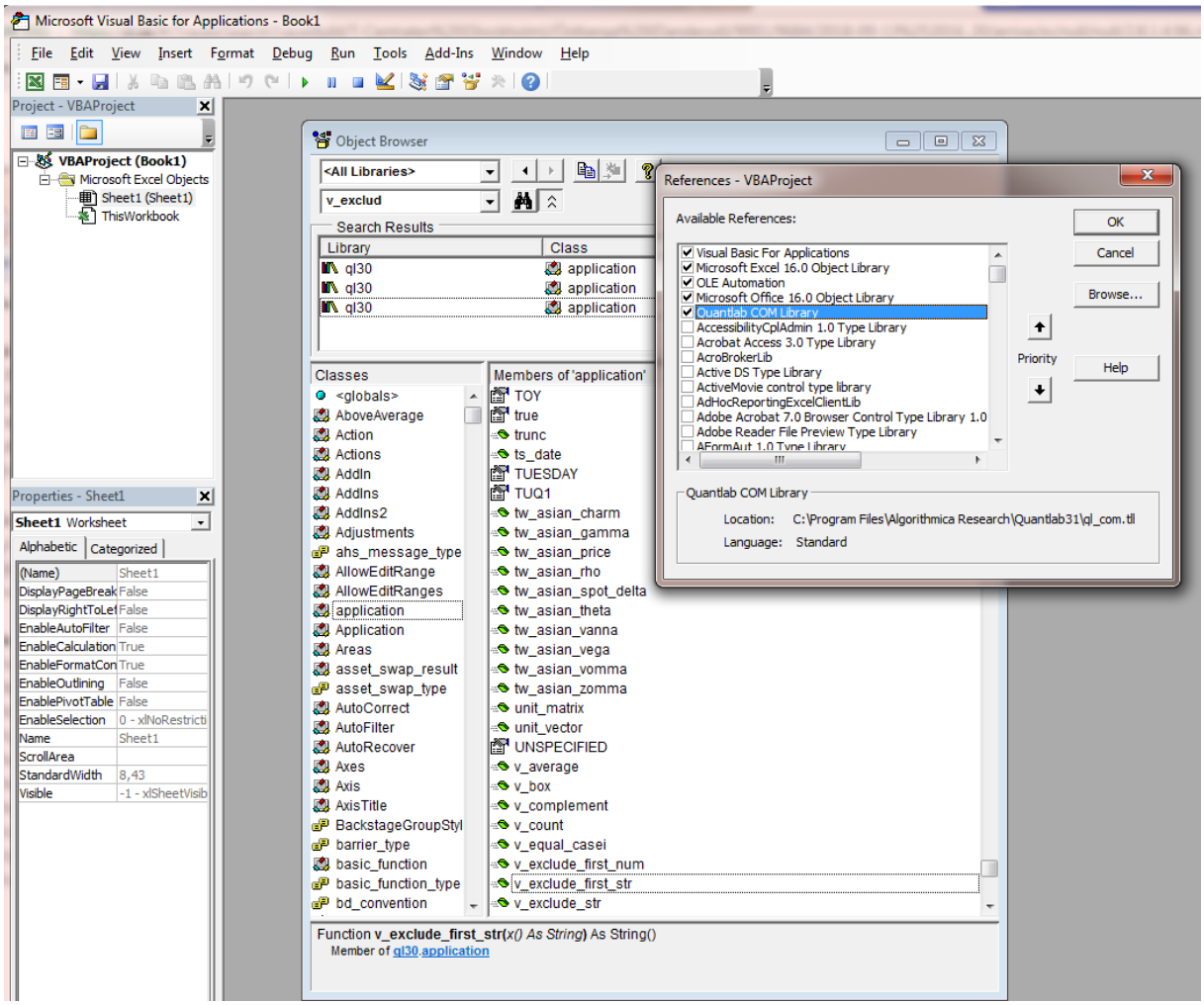
To set the path, open the environment variables on your PC and extend the list of paths to include your local path. Either user or system path must be set.



1.7.3 Running Excel and setting references in VBA Editor

Now you can start Microsoft Excel or Visual Basic, start the Visual Basic Editor and open References under the Tools menu. Use the Browse button to find the generated ql_com.tlb file and verify that 'Quantlab COM Library' is included in the list and marked as active. You only have to do this once – every time you update the .tlb file it is enough to close and reopen Excel. Make sure that Excel or any program that uses the .tlb file is closed when you generate a new file - otherwise Quantlab will be unable to overwrite the existing file.

Use the Object Browser and browse the 'ql30' library to examine what functions are available. To get more help on each function, use the function browser in Quantlab.



1.7.4 COM function overloading

Note that COM does not allow for overloaded functions to be exported. To get around this, it is possible to publish a user-defined name to the COM interface.

By using the key-word: option(com_name: <string>) directly after the function name, an alternate name of the function will be published to the COM interface. Example:

```
number my_new_func(number x, number y)
        option(com_name: 'my_new_func2')
{
        return x*y;
}
```

1.8 Example of VBA project (qltest_com.xls)

Here is an example of how to calculate so called tail yields using QLang functions in VBA. The example file is found in Quantlab\examples\COM. The code corresponds to the case study Calculating tail rates (Chapter 5.5 in Quantlab User Manual).

```
Function mat(c As String, d As Date) As Variant
     On Error GoTo err:
        Dim out() As Date
        Dim iv() As ql.instrument
        Dim cv As ql.curve
```

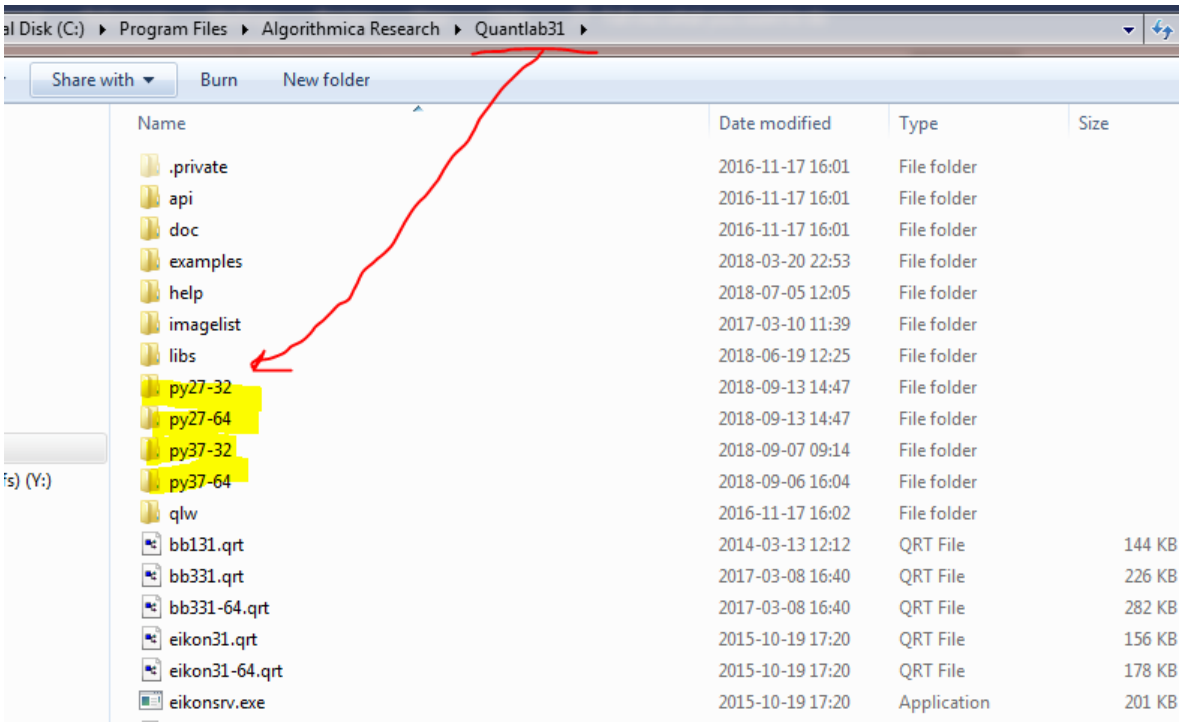
```
        Set app = CreateObject("QLang.Application")

        Set cv = app.db_curve(c, d)
        iv = cv.instruments()


        ReDim out(UBound(iv))
        For i = 0 To UBound(iv)
            out(i) = iv(i).maturity
        Next

        mat = out

        Exit Function

err:
    MsgBox "Error" & Str(err.Number) & ": " & err.Description
End Function
Function tail_graph(curvename As String, tradedate As Date) As Variant
     On Error GoTo err:
        Dim fr As ql.fit_result

        Set app = CreateObject("QLang.Application")

        Set fr = app.bootstrap(app.db_curve(curvename, tradedate))

        Dim m() As Date
        m = mat(curvename, tradedate)

        Dim out() As Double
        ReDim out(UBound(m) - 1)

        For i = 0 To UBound(out)
            out(i) = fr.zero_rate_dc(tradedate, m(i), m(i + 1), "simple", "ACT360")
* 100
        Next
        tail_graph = out

        Exit Function
err:
    MsgBox "Error" & Str(err.Number) & ": " & err.Description

End Function
```

Using the Quantlab Python interface

All functions available in Quantlab may be called from any Python 2.7 and/or Python 3.7 environment. This includes not only the built-in library of financial and mathematical functions, but also user-written functions and classes implemented in Qlang (and saved as a library file). Internally compiled c/c++ binarys implemented as a Dynamic Link Library (DLLs) will also be exported to Python.

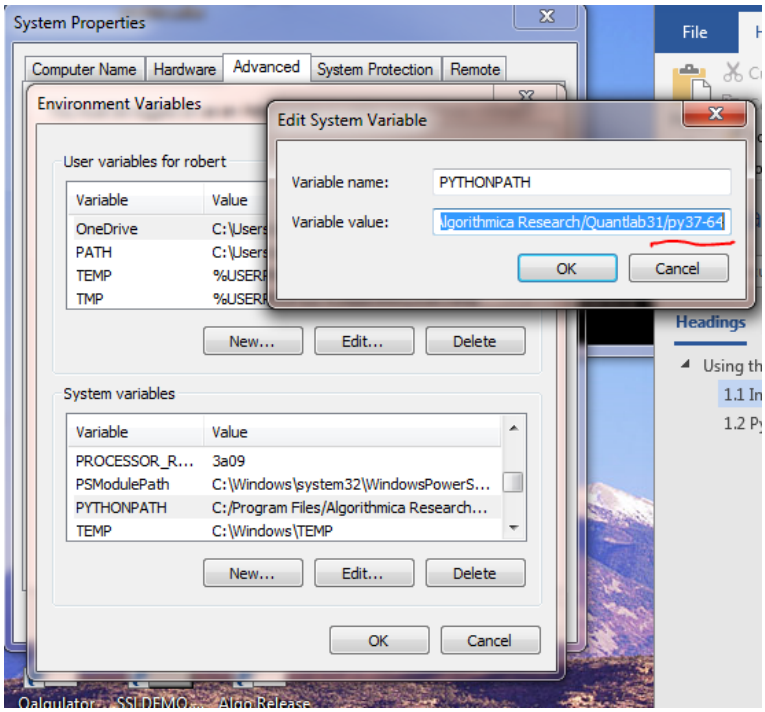
1.9 Installing the Python Quantlab API

Before getting started you need to ensure that you have the proper Quantlab python library in your local Quantlab setup. The relevant version of python support should be placed directly under the Quantlab root. Both 32-bit and 64-bit Quantlab is currently supported.

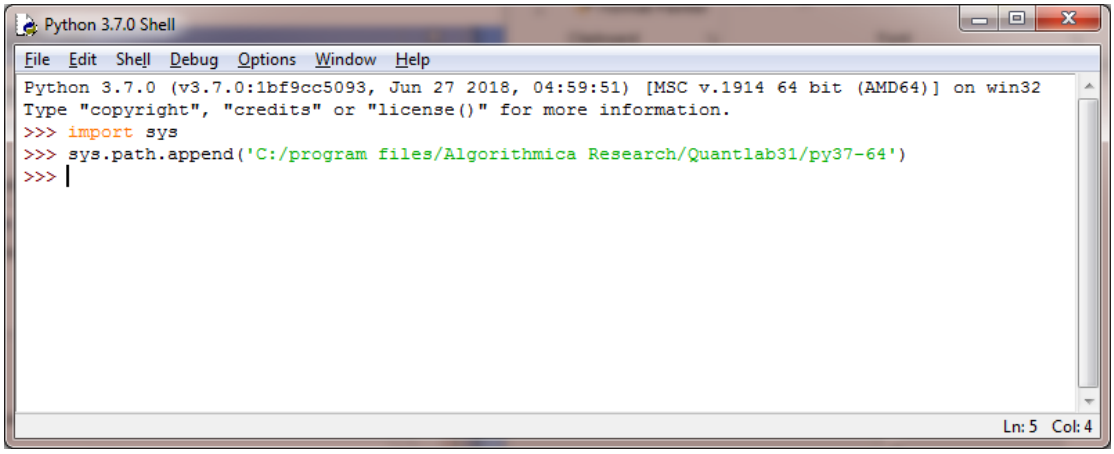


There are two options to load the Quantlab application into Python.

1) To permanently add a system variable PYTHONPATH to the systems environment variables. It should point to the corresponding Python version folder using either 32-bit or 64-bit Quantlab. This means that any time python is started it will recognize the possibility to load the Quantlab library.



2) To dynamically add the Python path using the sys library. Then the Python code will work for any user having a correct Quantlab and Python file-setup regardless of system environment.



To use Quantlab functions in Python it is enough to use the “import ql” command to have Quantlab available. The session will load all library functions set in the ini file of the current Quantlab. If the instr31.dll is available and a valid ODBC source is pointing to an instrument and curve database, the instrument definitions will be loaded in the same way as starting a Quantlab session.

Remember that also real-time connections to Bloomberg and Reuters will be available to the Python interface.

NOTE: Loading of the Quantlab function set including your internal ql/qll library files is automatically done every time the “import ql” command is run in Python. There is no need to export and register tlb files as with the COM connection.

1.10 Quantlab to Python – handling of function overloading

Note that Python (as well as COM) **does not allow** for overloaded functions and classes to be exported. If any function or class constructor have overloaded variants in Quantlab, none of them will be loaded. To get around this, it is possible to publish a user-defined name to the Python/COM interface. Python and COM use the same user-defined code extension.

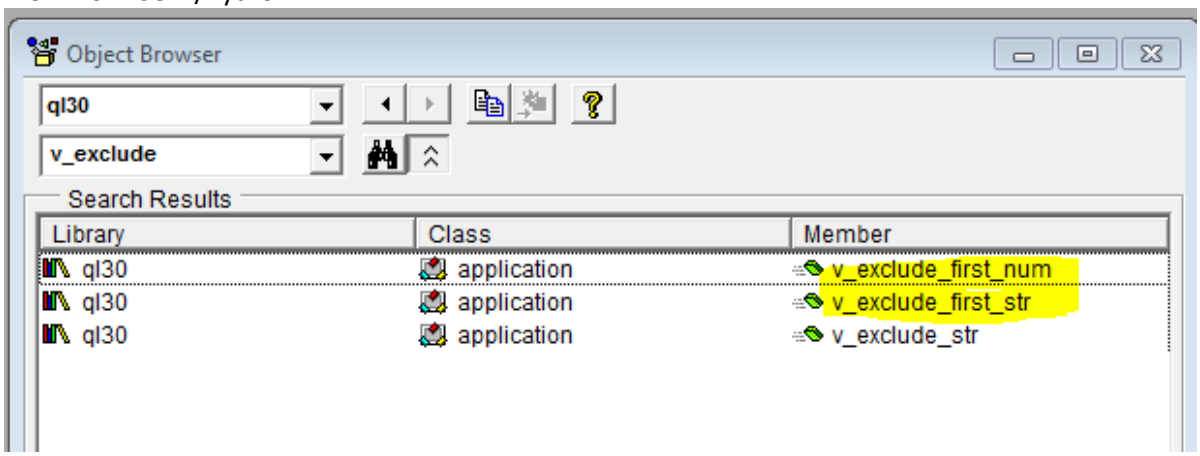
By using the key-word: option (com_name: <string>) directly after the function name, an alternate name of the function will be published to the Python/COM interface.

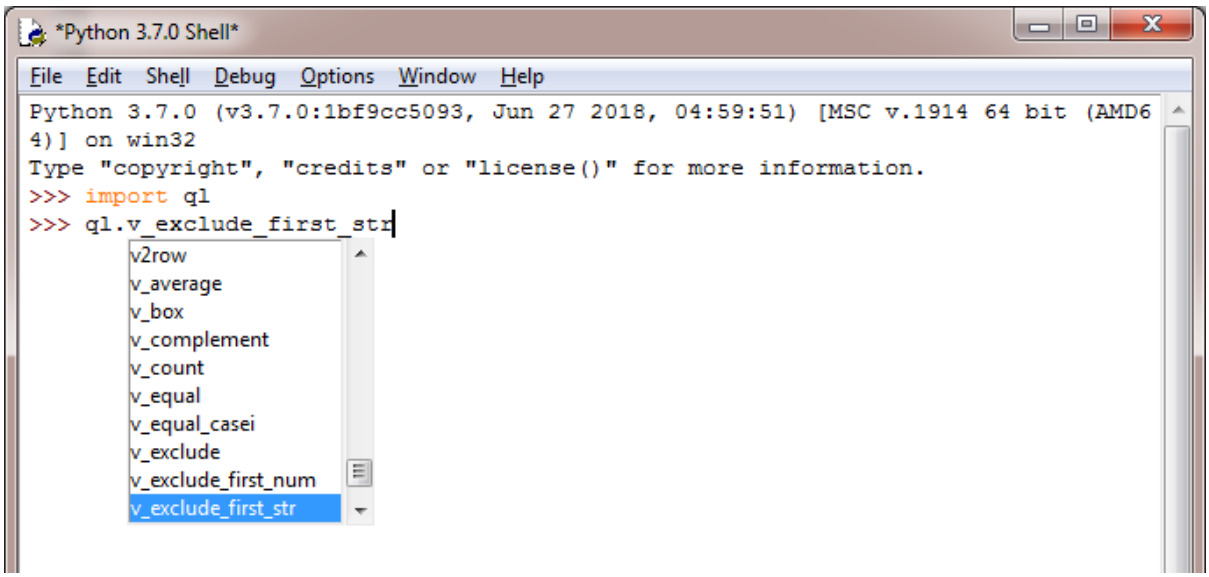
Example:

libs/extras/Python_playground.ql

```
vector(number) v_exclude_first(vector(number) x)
    option(com_name:'v_exclude_first_num')
{
    return x[1:];
}
vector(string) v_exclude_first(vector(string) x)
    option(com_name:'v_exclude_first_str')
{
    return x[1:];
}
```

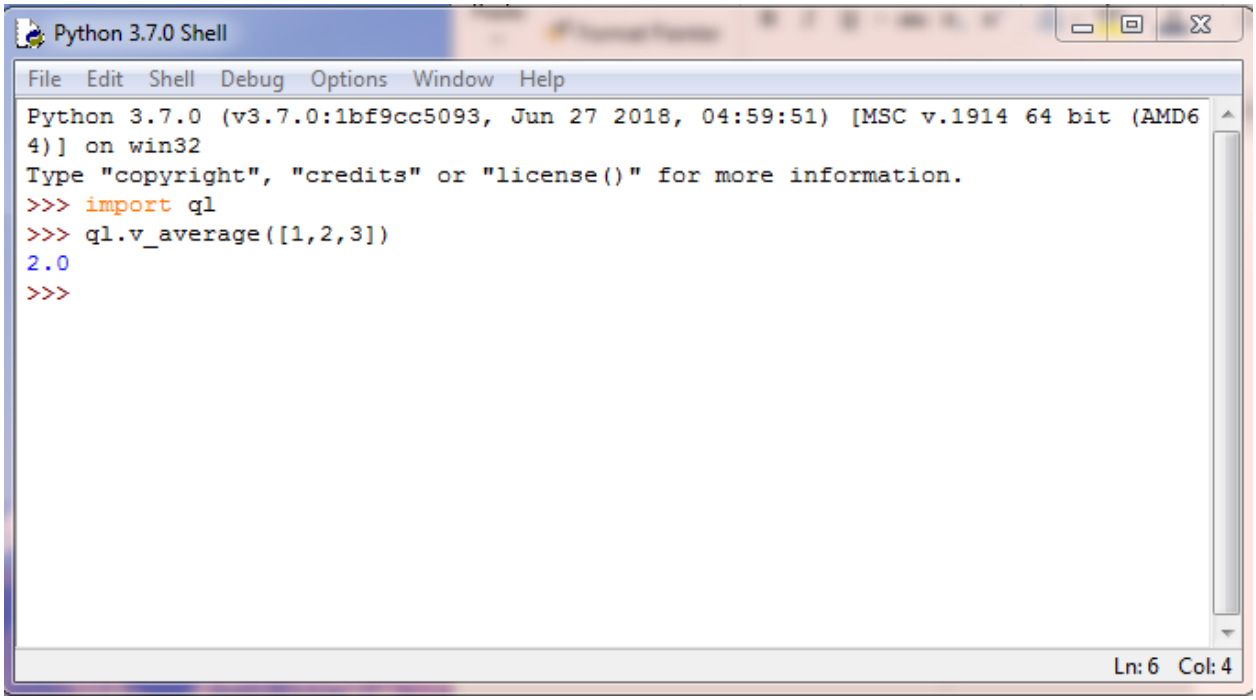
View from COM/Python:



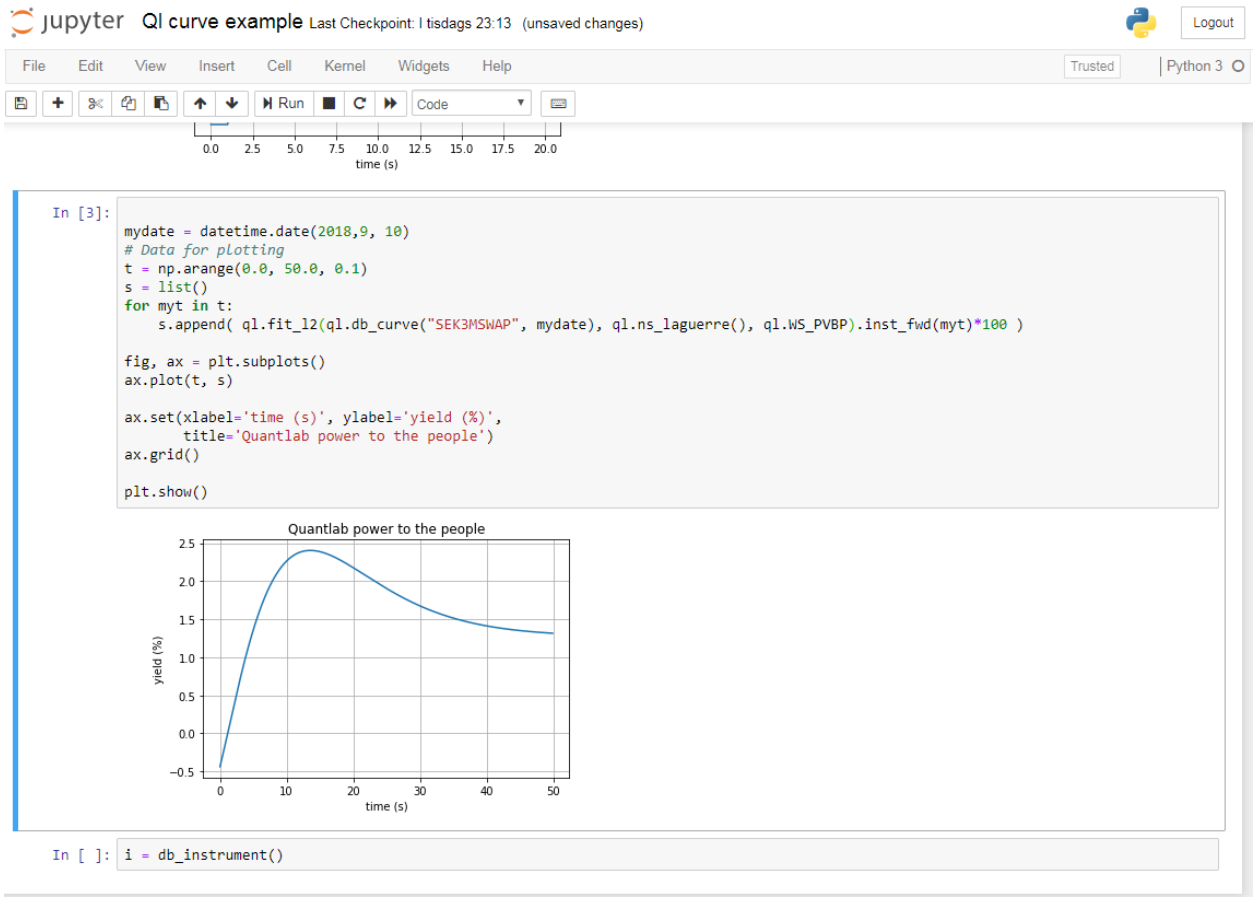


1.11 Python examples

There are many ways to write and run Python code. The Quantlab Python API works equally well from both the IDLE command shell as well as from a Jupyter notebook.



Using the Python Shell

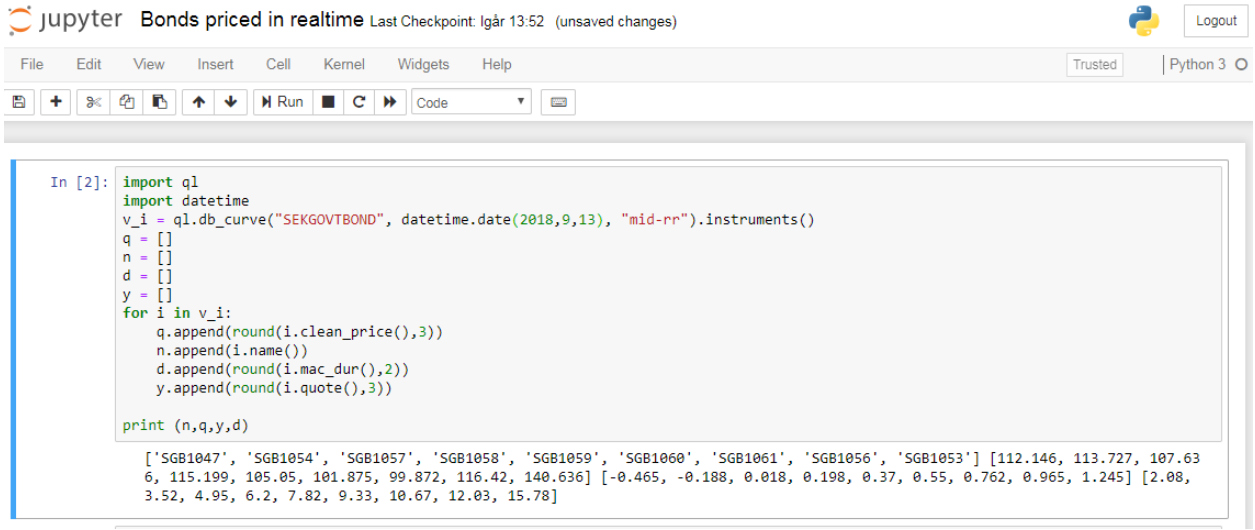


Using Jupyter Notebook

A few things to note when using Qlang / Quantlab from Python:

- As mentioned above, no overloading in Python
- No vector and matrix function expansion in Python. Must loop in Python.
- Function variables passed by reference are not allowed in Python, as they are in Qlang and C++.
- Date type in Python has no overloaded operator +/- which means that “today() – 1” does not work.

A code example calling a set of bonds and extracting some static and pricing data:




```
import ql
import datetime
v_i = ql.db_curve("SEKGOVTBOND", datetime.date(2018,9,14), "mid-rr").instruments()
q = []
n = []
d = []
y = []
for i in v_i:
    q.append(round(i.clean_price(),3))
    n.append(i.name())
    d.append(round(i.mac_dur(),2))
    y.append(round(i.quote(),3))
print (n,q,y,d)
```

A code example calling the database for printing historical price quotes:

In [3]:
```
start = datetime.datetime(2018,1,1)
end = datetime.datetime.today()
daterange = [start + datetime.timedelta(days=x) for x in range(0, (end-start).days)]
q = []
for d in daterange:
    tmp = ql.db_instrument("SHB A:XSTO:SEK", d).quote_adj(True,True)
    if tmp != None:
        q.append(round(tmp,2))

print(q)
```

```
[104.93, 103.72, 105.68, 106.47, 107.66, 109.52, 110.46, 109.71, 108.47, 107.68, 107.54, 107.26, 108.1, 107.64, 109.36, 109.8,
109.27, 107.17, 108.01, 108.08, 106.7, 106.98, 108.08, 107.54, 106.4, 103.48, 104.28, 102.92, 101.43, 102.27, 103.55, 106.38,
107.36, 108.36, 108.33, 108.73, 107.64, 107.78, 107.33, 106.35, 106.61, 107.08, 103.81, 104.53, 105.4, 106.89, 108.45, 108.24,
108.52, 108.47, 108.61, 107.57, 107.29, 105.44, 104.21, 103.2, 99.63, 100.62, 99.68, 100.28, 101.38, 104.22, 101.45, 101.62, 1
02.57, 100.72, 100.45, 100.07, 98.69, 98.4, 98.2, 99.06, 100.12, 99.37, 99.97, 99.85, 100.75, 99.18, 95.67, 97.83, 99.2, 98.0
4, 98.54, 98.32, 98.75, 98.6, 99.32, 100.47, 100.05, 99.84, 99.96, 98.28, 99.7, 99.64, 98.8, 98.8, 98.09, 98.47, 97.82, 98.23,
96.92, 97.32, 96.64, 98.81, 97.52, 96.61, 97.34, 97.22, 99.18, 99.56, 96.91, 98.78, 96.92, 96.15, 97.31, 97.52, 96.99, 97.28,
97.45, 98.56, 98.65, 99.69, 98.54, 98.53, 98.17, 98.45, 98.93, 99.4, 99.94, 99.15, 99.05, 98.7, 100.25, 101.25, 98.42, 99.04,
102.55, 105.1, 106.6, 106.53, 107.75, 107.62, 107.55, 108.53, 108.18, 108.22, 108.47, 107.82, 108.35, 108.22, 109.8, 108.7, 10
8.6, 107.78, 109.03, 110.28, 110.55, 111.03, 110.8, 110.93, 111.2, 111.1, 112.2, 111.18, 111.97, 111.72, 110.88, 110.82, 110.
2, 109.72, 109.97, 108.22, 108.47, 107.4, 106.55]
```


```
start = datetime.datetime(2018,1,1)
end = datetime.datetime.today()
daterange = [start + datetime.timedelta(days=x) for x in range(0, (end-start).days)]
q = []
for d in daterange:
    tmp = ql.db_instrument("SHB A:XSTO:SEK", d).quote()
    if tmp != None:
        q.append(round(tmp,2))

print(q)
```

Appendix – Quantlab types

This table shows the types currently in the Quantlab language. The “Type” column presents the type name in Quantlab. The “Identifier” column shows the string used for declaring a parameter or a return value using the arg_spec function. The “Representation” column is how the type is practically represented when used as parameters and return values in your code. The “Test” column presents the testing function to see if a value is valid or not, used when elements are extracted from series, vectors and so on. Strings and handles are pointers that will be Null if they are not valid, so the only specific Quantlab testing functions are valid and valid_date, all others are tested with the common if statement.

Type	Identifier for arg_spec	Representation	Test
number	"number"	double	valid
date	"date"	double	valid_date
logical	"logical"	double	valid
algorithm	"algorithm"	handle	if
asset_swap_result	"asset_swap_result"	handle	if
calendar	"calendar"	handle	if
curve	"curve"	handle	if
date_range	"date_range"	handle	if
fit_result	"fit_result"	handle	if
instr_def	"instr_def"	handle	if
instrument	"instrument"	handle	if
lable_date	"lable_date"	handle	if
lable_number	"lable_number"	handle	if
model	"model"	handle	if
number_range	"number_range"	handle	if
point_date	"point_date"	handle	if
point_number	"point_number"	handle	if
weight_scheme	"weight_scheme"	handle	if
asset_swap_type	"asset_swap_type"	string	if
curve_name	"curve_name"	string	if
day_count_method	"day_count_method"	string	if
instr_class_name	"instr_class_name"	string	if
instrument_name	"instrument_name"	string	if

quote_side	"quote_side"	string	if
rate_type	"rate_type"	string	if
string	"string"	string	if
vector of <type>	"vector(<type>)"	handle	if
matrix of <type>	"matrix(<type>)"	handle	if
series of <type>	"series(<type>)"	handle	if
series of vectors of <type>	"series(vector(<type>))"	handle	if
series of matrices of <type>	"series(matrix(<type>))"	handle	if

Interface

Return types

QL type	C++ type	QL api function	example
void	void	-	-
logical	QL::logical_t	-	-
integer	QL::int_t	-	-
number	QL::number_t	-	-
date	QL::date_t	-	boost::gregorian::date d; return d.is_not_a_date() ? QL::null_date() : QL::date(d.year(), d.month(), d.day());
timestamp	QL::timestamp_t	-	boost::posix_time::ptime t; return t.is_not_a_date_time() ? QL::null_timestamp : QL::timestamp(t->date().year(), t->date().month(), t->date().day(), t->time_of_day().hours(), t->time_of_day().minutes(), t->time_of_day().seconds(), 0);
enum	QL::enum_t	-	-

QL type	C++ type	QL api function	example
string	QL::handle	QL::create_string(const wchar_t *) or QL::create_string(const char *)	return QL::create_string(L"HELLO")
object	QL::handle_t<>		return my_space::my::create()
vector(logical)	QL::handle	QL::create_v_l(size) and QL::set_v_l(object&, ix, logical_t)	return QL::set_v_t(*vec_obj, ix, true)
vector(integer)	QL::handle	QL::create_v_i(size) and QL::set_v_i(object&, ix, int_t)	return QL::set_v_t(*vec_obj, ix, 45)
vector(number)	QL::handle	QL::create_v_n(size) and QL::set_v_n(object&, ix, number_t)	return QL::set_v_t(*vec_obj, ix, 45.345345)
vector(date)	QL::handle	QL::create_v_d(size) and QL::set_v_d(object&, ix, date_t)	return QL::set_v_t(*vec_obj, ix, NR::date_t(y, m, d).nr())
vector(timestamp)	QL::handle	QL::create_v_t(size) and QL::set_v_t(object&, ix, timestamp_t)	return QL::set_v_t(*vec_obj, ix, NR::timestamp_t(NR::date_t(y, m, d), hh, mm, ss, ms).nr())
vector(string)	QL::handle	QL::create_v_s(size) and QL::set_v_s(object&, ix, object *)	return QL::set_v_s(*vec_obj, ix, &(*QL::create_string(L"HELLO")))
vector(object)	QL::handle	QL::create_v_o(size) and QL::set_v_o(object&, ix, object *)	return QL::set_v_o(*vec_obj, ix, my_obj.get())

Argument types

QL type	C++ type (normal argument)	C++ type (output argument)	C++ type (nullable argument) (Also works for normal arguments)	comments
logical	QL::logical_t	QL::logical_t &	QL::logical_t	
integer	QL::int_t	QL::int_t &	QL::int_t	

QL type	C++ type (normal argument)	C++ type (output argument)	C++ type (nullable argument) (Also works for normal arguments)	comments
number	QL::number_t	QL::number_t &	QL::number_t	
date	QL::date_t	QL::date_t &	QL::date_t	
timestamp	QL::timestamp_t	QL::timestamp_t &	QL::timestamp_t	
enum	QL::enum_t	QL::enum_t &	QL::enum_t	
string	const QL::object &	QL::handle &	const QL::object *	"const" can only be used when the object will not be updated or reference counted in your QLL
object	const QL::object &	QL::handle &	const QL::object *	"const" can only be used when the object will not be updated or reference counted in your QLL
vector(...)	const QL::object &	QL::handle &	const QL::object *	"const" can only be used when the object will not be updated or reference counted in your QLL

Converting to C++ string

```
const char* QL::get_string(const QL::object&)      // does a checked_cast to string
const char* QL::get_string_0(const QL::object*)    // returns null pointer if null
```

Converting argument vector to C++ vector

Get the number of elements, to create the corresponding C++ vector:

```
int_t get_v_size(const object&)
```

Extract the elements using one of the following functions, where all functions have arguments `(const object&, int_t); the first argument being the vector and the second the index`.

Function	Return type	QL type
get_v_i	QL::int_t	integer
get_v_n	QL::number_t	number
get_v_l	QL::logical_t	logical
get_v_d	QL::date_t	date
get_v_m	QL::month_t	month
get_v_t	QL::timestamp_t	timestamp
get_v_e	QL::enum_t	enum
get_v_s	QL::handle	string
get_v_o	QL::handle	object
get_v_f	QL::handle	function

Example (vector of strings):

```
int n = QL::get_v_size(val);
vector<string> val_vec(n);
for (int i = 0; i < n; ++i) {
    QL::handle string_h = QL::get_v_s(val, i);
    if (string_h) {
        // Ignore null strings, since std:string cannot be null
        // Had we used char* instead of std:string, we could have called
        // QL::get_string_0 and gotten a null pointer if string was null.
        val_vec[i] = QL::get_string(*string_h);
    }
}
```

Functions and member functions

```
QL::arg_spec ret;
QL::arg_spec args[<size>]; // the Array must be big enough to handle all your
function declarations

void my_func(<arg1>, <arg2>...)

r  = QL::arg_spec("<QLang argument type>");  // e.g QL::arg_spec("void")
args[<index number>] = QL::arg_spec("<QLang argument type>", "<Argument name in
QLang>"); // e.g QL::arg_spec("string", "name")
QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(my_func),  "my_ql_function",
category, ret, <number of arguments>, args);

void my_member_func(myspace::my &m, <arg1>, <arg2>...)

r  = QL::arg_spec("<QLang argument type>");  // e.g QL::arg_spec("void")
args[<index number>] = QL::arg_spec("<QLang argument type>", "<Argument name in
QLang>"); // e.g QL::arg_spec("string", "name")
QL::add_mem_func(ctxt,reinterpret_cast<QL::function_t*>(my_member_func),  "my_ql_ob
j", "my_ql_function", category, ret, <number of arguments>, args);
```

Exceptions

Very important to catch all exceptions generated in your QLL or from libraries that you have used (e.g. C++ Windows Standard Libary) or Quantlab will crash.
You must not though catch std::bad_alloc since Quantlab will handle that

Example code

my.h

```
#pragma once

#include <QL/api/ql.h>

namespace myspace {
    class my;
    typedef QL::handle_t<my> my_p;
}

class myspace::my : public QL::object {
private:
    explicit my() {}
public:
    ~my() {}
private:
    my(const my&);
    my &operator=(const my &);
public:
    static my_p create() { return my_p(new my()); }
    void set_name(const char *s) { name.assign(s); }
private:
    std::string name;
};
```

my.cpp

```
#include "my.h"

static const char category[] = "My Utilities";

static myspace::my_p create_my()
{
    try {
        myspace::my_p m = myspace::my::create();
        return m;
    }
    catch (const std::bad_alloc &) {
        throw;
    }
    catch (const std::exception &e) {
        throw QL::error(QL::E_UNSPECIFIC, e.what());
    }
}

QL::handle my_hello(myspace::my &m)
{
    try {
        return QL::create_string("HELLO WORLD");
    }
    catch (const std::bad_alloc &) {
        throw;
    }
    catch (const std::exception &e) {
        throw QL::error(QL::E_UNSPECIFIC, e.what());
    }
}

void my_set_name(myspace::my &m, QL::object &name)
{
    if (strlen(QL::get_string(name)) > 50)
       throw QL::error(QL::E_INVALID_ARG, "Too long name");
```

```
    m.set_name(QL::get_string(name));
}

__declspec(dllexport) void ql_init(QL::symtab &s)
{
    QL::symtab &ctxt = QL::add_module(s, "my_util");

    QL::arg_spec r;
    QL::arg_spec a[12];

    QL::add_object_type(ctxt, "my", 0, category);

    r  = QL::arg_spec("my");
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(create_my), "my",
category, r, 0, a);

    r  = QL::arg_spec("string");
    QL::add_mem_func(ctxt, reinterpret_cast<QL::function_t*>(my_hello), "my",
"hello", r, 0, a);

    r  = QL::arg_spec("void");
    a[0] = QL::arg_spec("string");
    QL::add_mem_func(ctxt, reinterpret_cast<QL::function_t*>(my_set_name), "my",
"set_name", r, 1, a);
}
```

Appendix – API specific functions in ql_base.h

This table presents the functions in the header file ql_base, as these are mostly only available through the Quantlab API and cannot be accessed using QLang.

API function	Header file	Description
valid	ql_base	Tests if a number or logical is valid.
invalid_number	ql_base	Returns an invalid number.
throw_error	ql_base	Throws an error that is caught by Quantlab.
get_v_size	ql_base	Returns the size of a vector.
get_m_size	ql_base	Returns the number of rows and columns of a matrix.
get_s_dims	ql_base	Returns the number of ranges on which a series is based.
get_s_x0	ql_base	Returns the starting number, date or timestamp of one of the ranges of a series.
get_s_dx	ql_base	Returns the step length of one of the ranges of a series.
get_s_size	ql_base	Returns the number of elements of one of the ranges of a series.
get_s_range_number	ql_base	Tests if a range is a number range.
get_s_range_date	ql_base	Tests if a range is a date range.
get_s_range_timestamp	ql_base	Tests if a range is a timestamp range.
get_s_total_size	ql_base	Returns the total number of elements of a series, a multiplication of the sizes of all ranges.
get_s_x	ql_base	Returns the range number, date or timestamp corresponding to one element of a series.
get_sv_v_size	ql_base	Returns the size of each vector in a series of vectors.
get_sm_m_size	ql_base	Returns the number of rows and columns of each matrix in a series of matrices.
get_v_num	ql_base	Returns one element of a vector of numbers.
get_v_obj	ql_base	Returns one element of a vector of objects.
get_m_num	ql_base	Returns one element of a matrix of numbers.
get_m_obj	ql_base	Returns one element of a matrix of objects.
get_s_num	ql_base	Returns one element of a series of numbers.
get_s_obj	ql_base	Returns one element of a series of objects.
get_sv_num	ql_base	Returns one element of a series of vectors of numbers.
get_sv_obj	ql_base	Returns one element of a series of vectors of objects.
get_sm_num	ql_base	Returns one element of a series of matrices of numbers.

get_sm_obj	ql_base	Returns one element of a series of matrices of objects.
create_v_num	ql_base	Creates a vector of numbers.
create_v_obj	ql_base	Creates a vector of objects.
create_m_num	ql_base	Creates a matrix of numbers.
create_m_obj	ql_base	Creates a matrix of objects.
create_num_range	ql_base	Creates a number range.
create_date_range	ql_base	Creates a date range.
create_timestamp_range	ql_base	Creates a timestamp range.
create_s_num	ql_base	Creates a series of numbers.
create_s_obj	ql_base	Creates a series of objects.
create_sv_num	ql_base	Creates a series of vectors of numbers.
create_sv_obj	ql_base	Creates a series of vectors of objects.
create_sm_num	ql_base	Creates a series of matrices of numbers.
create_sm_obj	ql_base	Creates a series of matrices of objects.
set_v_num	ql_base	Sets one element of a vector of numbers.
set_v_obj	ql_base	Sets one element of a vector of objects.
set_m_num	ql_base	Sets one element of a matrix of numbers.
set_m_obj	ql_base	Sets one element of a matrix of objects.
set_s_num	ql_base	Sets one element of a series of numbers.
set_s_obj	ql_base	Sets one element of a series of objects.
set_sv_num	ql_base	Sets one element of a series of vectors of numbers.
set_sv_obj	ql_base	Sets one element of a series of vectors of objects.
set_sm_num	ql_base	Sets one element of a series of matrices of numbers.
set_sm_obj	ql_base	Sets one element of a series of matrices of objects.

Appendix – Example “pair”: Adding an object type and its member functions

This is an example showing how to add the object type “pair” to Quantlab with a creation function and member functions. The “pair” object is simply a container of two numbers, created in Quantlab using the pair function. The two components can be extracted using the member functions first and last on the pair object.

```
#include "ql.h"
namespace
{
    class pair;
    typedef QL::handle_t<const pair>    pair_p;
    // Declaring the pair class from the basic QL object class.
    class pair : public QL::object
    {
    // Declaring a creation function.
    public:
       static pair_p create(double first, double last)
       {
           return pair_p(new pair(first, last));
       }
    // Hiding the object creation to avoid reference problems.
    private:
       pair(double first, double last) : first(first), last(last) {}
    public:
       double first;
       double last;
    };
}
/* Creates a pair object. */
static void create_pair(pair_p &result, double first, double last)
{
    result = pair::create(first, last);
}
/* Extracts the first element of a pair. */
static double pair_first(const pair_p &p)
{
    return p->first;
}
/* Extracts the last element of a pair. */
static double pair_last(const pair_p &p)
{
    return p->last;
}
/*
* Adds the pair object type, a pair creation function and two member functions
* to the Quantlab context.
*/
  __declspec(dllexport) void ql_init(QL::symtab &ctxt)
{
    QL::arg_spec    r;
    QL::arg_spec    a[12];
    QL::add_object_type(ctxt, "pair");
    r    = QL::arg_spec("pair");
    a[0] = QL::arg_spec("number", "first");
    a[1] = QL::arg_spec("number", "last");
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(create_pair),
               "pair", "Pairs of numbers", r, 2, a);
    r    = QL::arg_spec("number");
    QL::add_mem_func(ctxt, reinterpret_cast<QL::function_t*>(pair_first),
                  "pair", "first", r, 0, a);
    QL::add_mem_func(ctxt, reinterpret_cast<QL::function_t*>(pair_last),
                  "pair", "last", r, 0, a);
}
```

Appendix – A series example

This is an example of how to access and use a series. The function my_average will calculate the average of a given series.

```
#include "ql.h"
static const char category[] = "Series example";
static double average(const QL::handle &s)
{
    int             size   = QL::get_s_total_size(s);
    int             count  = 0;
    double   sum    = 0;
    for (int i = 0 ; i < size ; i++) {
       double s_i = QL::get_s_num(s, i);
       if (QL::valid(s_i)) {
           sum += s_i;
           ++count;
       }
    }
    return sum / count;
}
__declspec(dllexport) void ql_init(QL::symtab &ctxt)
{
    QL::arg_spec    r;
    QL::arg_spec    a[12];
    r    = QL::arg_spec("number");
    a[0] = QL::arg_spec("series(number)",      "s");
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(average),
                "my_average", category, r, 1, a);
}
```

Appendix – Blending functions

This is an example of how to blend yield curves. In a typical case you want to combine deposit rates, short-term futures or FRA:s (below, both are called FRA:s) and swap rates to form a complete yield curve. The number of deposit rates will vary depending on the time to the fixing of the first FRA. In the transition from deposit to FRA:s the standard method is to interpolate between the two nearest deposit rates in order to construct a loan that matures on the settlement date of the FRA. This is done by calling the Quantlab function curve_chop_long_interp. In this example, four different functions on this theme are created.

As an alternative, these functions can be created in the Quantlab language directly, which is shown in Appendix 6.

```
#include "ql.h"
static const char category[] = "Curve blending";
static double min_maturity(const QL::handle &curve)
{
    QL::handle      v = QL::curve_instruments(curve);
    int             n = QL::get_v_size(v);
    return QL::instr_maturity(QL::get_v_obj(v, 0));
}
static double max_maturity(const QL::handle &curve)
{
    QL::handle      v = QL::curve_instruments(curve);
    int             n = QL::get_v_size(v);
    return QL::instr_maturity(QL::get_v_obj(v, n - 1));
}
static double min_settle(const QL::handle &curve)
{
    QL::handle      v = QL::curve_instruments(curve);
    int             n = QL::get_v_size(v);
    double   min, tmp;
    min = QL::instr_settle_date(QL::get_v_obj(v, 0));
    for (int i = 1 ; i < n ; i++) {
       tmp = QL::instr_settle_date(QL::get_v_obj(v, i));
       if (min > tmp)
           min = tmp;
    }
    return min;
}
static void blend_depo_swap(QL::handle         &result,
                        const QL::handle       &depo_curve,
                        const QL::handle       &swap_curve,
                        const QL::handle       &ctxt)
{
    if (QL::curve_empty(swap_curve))
       result = depo_curve;
    else {
       double       d      = min_maturity(swap_curve) - 1;
       QL::handle   c_short      = QL::curve_chop_long(depo_curve, d);
       result = QL::curve_merge(c_short, swap_curve);
    }
}
static void blend_depo_fra(QL::handle          &result,
                       const QL::handle &depo_curve,
                       const QL::handle &fra_curve,
                       const QL::handle &ctxt)
{
    if (QL::curve_empty(fra_curve))
       result = depo_curve;
    else {
       double       d      = min_settle(fra_curve);
       QL::handle   c_short      = QL::curve_chop_long_interp(depo_curve,
                                                  d, ctxt);
```

```
       result = QL::curve_merge(c_short, fra_curve);
    }
}
static void blend_fra_prio(QL::handle          &result,
                       const QL::handle &depo_curve,
                       const QL::handle &fra_curve,
                       const QL::handle &swap_curve,
                       const QL::handle &ctxt)
{
    if (QL::curve_empty(fra_curve))
       blend_depo_swap(result, depo_curve, swap_curve, ctxt);
    else {
       double       d;
       QL::handle   c_short, c_long;
       d = min_settle(fra_curve);
       c_short = QL::curve_chop_long_interp(depo_curve, d, ctxt);
       c_short = QL::curve_merge(c_short, fra_curve);
       d = max_maturity(c_short) + 1;
       c_long = QL::curve_chop_short(swap_curve, d);
       result = QL::curve_merge(c_short, c_long);
    }
}
static void blend_swap_prio(QL::handle         &result,
                        const QL::handle       &depo_curve,
                        const QL::handle       &fra_curve,
                        const QL::handle       &swap_curve,
                        const QL::handle       &ctxt)
{
    if (QL::curve_empty(swap_curve))
       blend_depo_fra(result, depo_curve, swap_curve, ctxt);
    else {
       double       d;
       QL::handle   c_mid, c_short;
       d = min_maturity(swap_curve) - 1;
       c_mid = QL::curve_chop_long(fra_curve, d);
       if (QL::curve_empty(c_mid))
           blend_depo_swap(result, depo_curve, swap_curve, ctxt);
       else {
           d = min_settle(c_mid);
           c_short = QL::curve_chop_long_interp(depo_curve, d, ctxt);
           c_short = QL::curve_merge(c_short, c_mid);
           result = QL::curve_merge(c_short, swap_curve);
       }
    }
}
__declspec(dllexport) void ql_init(QL::symtab &ctxt)
{
    QL::arg_spec    r;
    QL::arg_spec    a[12];
    r    = QL::arg_spec("curve");
    a[0] = QL::arg_spec("curve", "depo_curve");
    a[1] = QL::arg_spec("curve", "swap_curve");
    a[2] = QL::ctxt_arg();
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(blend_depo_swap),
              "blend_curves_depo_swap", category, r, 3, a);
    r    = QL::arg_spec("curve");
    a[0] = QL::arg_spec("curve", "depo_curve");
    a[1] = QL::arg_spec("curve", "fra_curve");
    a[2] = QL::ctxt_arg();
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(blend_depo_fra),
              "blend_curves_depo_fra", category, r, 3, a);
    r    = QL::arg_spec("curve");
    a[0] = QL::arg_spec("curve", "depo_curve");
    a[1] = QL::arg_spec("curve", "fra_curve");
    a[2] = QL::arg_spec("curve", "swap_curve");
    a[3] = QL::ctxt_arg();
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(blend_fra_prio),
              "blend_curves_fra_prio", category, r, 4, a);
```

```
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(blend_swap_prio),
                "blend_curves_swap_prio", category, r, 4, a);
}
```

Appendix – Blending functions: Corresponding QLang implementation

This is an example of how to write the blending functions in Appendix 5, using the Quantlab language only. It is stated here for comparison to the code in Appendix 5.

```
curve blend_curves_depo_swap(curve depo_c, curve swap_c)
    option(category: 'Curve blending')
{
    if (swap_c.empty())
       return depo_c;
    else {
       depo_c = depo_c.chop_long(v_min(swap_c.instruments.maturity) - 1);
       return merge(depo_c, swap_c);
    }
}
curve blend_curves_depo_fra(curve depo_c, curve fra_c)
    option(category: 'Curve blending')
{
    if (fra_c.empty())
       return depo_c;
    else {
       depo_c = depo_c.chop_long_interp(v_min(fra_c.instruments.settle_date));
       return merge(depo_c, fra_c);
    }
}
curve blend_curves_fra_prio(curve depo_c,
                        curve fra_c,
                        curve swap_c)
    option(category: 'Curve blending')
{
    if (fra_c.empty())
       return blend_curves_depo_swap(depo_c, swap_c);
    else {
       date   chop_d;
       curve  short_c;
       chop_d  = v_min(fra_c.instruments.settle_date);
       depo_c = depo_c.chop_long_interp(chop_d);
       short_c       = merge(depo_c, fra_c);
       chop_d  = v_max(short_c.instruments.maturity) + 1;
       swap_c = swap_c.chop_short(chop_d);
       return merge(short_c, swap_c);
    }
}
curve blend_curves_swap_prio(curve depo_c,
                         curve fra_c,
                         curve swap_c)
    option(category: 'Curve blending')
{
    if (swap_c.empty())
       return blend_curves_depo_fra(depo_c, fra_c);
    else {
       fra_c = fra_c.chop_long(v_min(swap_c.instruments.maturity) - 1);
       if (fra_c.empty())
           return blend_curves_depo_swap(depo_c, swap_c);
       else {
           date      chop_d;
           chop_d = v_min(fra_c.instruments.settle_date);
           depo_c = depo_c.chop_long_interp(chop_d);
           return merge(depo_c, merge(fra_c, swap_c));
       }
    }
}
```

Appendix – Exponentially weighted series functions

This is an example of how to calculate exponentially weighted versions of the series functions variance, covariance and correlation. It is also an example of how to make the difference between the Quantlab interface functions and the normal C++ functions not using QLang types. The source code can be viewed by opening the exponential_roll project, and it is not printed here due to its length.

The functions added to QLang by the dll all take one or more series plus a decay factor and an optional tolerance level. The first step is to check that the series are one-dimensional and that they are based on the same range. After that, the series are translated into vectors (or matrices) defined for this project in order to move away from the QLang environment. In the translation process the "holes" (null values) in the series are taken care of, making the resulting vectors tested and clean. The weight vector is also filled with exponential weights at this step. The final step is to call the QLang-independent function covariance_weighted, with the translated and tested input data. Covariance_weighted returns the covariance between two series, and using this information the QLang return objects are filled with appropriate data for transport back to the Quantlab environment.

This implementation ignores the index ("day") of a set of series where there is one or more "holes", removing them in the translation process. It will also cut off the calculations when the weight of the remaining indices becomes smaller than the tolerance level. For example, with a tolerance level of 1% (0.01, the default) and a decay factor of 0.97, calculations are based on the last 151 elements of the translated series as the weights of 152 to infinity accounts for less than 1% of the total weight.

The exponential weight vector is constructed from the end. Starting at the ending element with the weight of the decay factor, the element one step towards the beginning of the vector has this weight multiplied by the decay factor. This continues for every element until the start of the weight vector. The main mathematical advantage of this weight scheme is that if one value has been calculated for a series, it is extremely easy to calculate the value of the series with one more element added. This advantage has not been used in this implementation, so there's plenty of room for performance improvement!

Appendix – User-defined yield curve models

By using the `QL::model_create_custom()` it is possible to define your own yield curve model, which can be fitted to a term structure by using the `fit()` function just as you would with any of the built in models.

In the following example a new model `custom_ns()` is created to implement the standard Nelson-Siegel model in the same way as the built in `ns()` model. This is done by inheriting from the class `QL::custom_model` and implementing the four mandatory virtual functions:

```
#include <ql.h>
#include <math.h>
namespace
{
    class custom_ns : public QL::custom_model
    {
    public:
       /* Indices for the param vector. */
       enum { B0, B1, B2, T, N_PARAMS };
       static QL::custom_model_p create()
       {
           return QL::custom_model_p(new custom_ns);
       }
       virtual int n_params() const
       {
           return N_PARAMS;
       }
       virtual double default_param_guess(int p) const
       {
           switch (p) {
           case B0: return  0.04;
           case B1: return  0.01;
           case B2: return -0.03;
           case T:  return  3;
           }
       }
       /* We need a lower bound on beta_0 for stability */
       virtual double default_param_min(int p) const
       {
           switch (p) {
           case B0: return 0.001;
           default: return custom_model::default_param_min(p);
           }
       }
       /* Dito upper bound on T */
       virtual double default_param_max(int p) const
       {
           switch (p) {
           case T:  return 30;
           default: return custom_model::default_param_max(p);
           }
       }
       virtual void disc_fact(const double     *p,
                              int           n,
                              const double *t,
                              double       *df)
       {
           int             i = 0;
           /*
            * t[] is guaranteed to be sorted; we skip possible
            * 0 maturities to avoid division by zero below.
            */
           while (i < n && t[i] == 0)
              df[i++] = 1;
           while (i < n) {
```

```
            double t_i    = t[i];
            double x_t    = t_i / p[T];
            double e      = exp(-x_t);
            double r;
            r = p[B0] + (p[B1] + p[B2]) * (1 - e) / x_t - p[B2] * e;
            df[i++] = exp(-r * t_i);
          }
      }
    private:
      explicit custom_ns()
      {}
      ~custom_ns()
      {}
    };
}
static void create_custom_ns(QL::handle &h)
{
    h = QL::model_create_custom(custom_ns::create());
}
__declspec(dllexport) void ql_init(QL::symtab &ctxt)
{
    QL::arg_spec    r;
    r = QL::arg_spec("model");
    QL::add_func(ctxt, reinterpret_cast<QL::function_t*>(create_custom_ns),
              "custom_ns", "Yield curve models", r, 0, 0);
}
```