Algorithmica
Research AB

Quantlab 3.1

User’s manual

Last update: 2023-09-21 (version 3.1.4044)

1 Introduction

1.1 What is the Quantlab system?

Production and distribution of financial analysis in a broad sense. Quantlab is a comprehensive software environment where financial analysts and traders can build, simulate and visualize different analyses and trading scenarios. Complex calculations involving real time data a multitude of time-series data are quickly performed using the built-in expression language and the powerful real-time evaluation engine. The resulting data may be presented in a view, such as a graph or a table, and can also be exported into an external application (e.g. Microsoft Excel) using a COM interface, and using API:s to REST, Python and C#.

Developer and User Edition. Quantlab is available in two versions, a Developer edition and a User edition. The Developer edition is the more complex application, which the analyst uses for constructing trading strategies and other types of analysis. Traders, sales staff and brokers use the Quantlab User edition to review and examine the analysis available within a workspace. They may for example change the financial yield curves or instruments involved in the analysis or choose to look at a specific historical time segment of interest.

Powerful expression language. The built-in expression language is highly versatile, and the end-user is free to extend it using the innovative library functionality. The syntax resembles C++ with the addition of high-level treatment of vectors and matrices: There is the possibility to express normal algebraic expressions using vectors and matrices and you can expand scalar functions over vectors and matrices. A powerful visual expression editor aids in the creation of complex expressions and instant help is available for all functions and objects. Expressions are easily attached to different views using the workspace manager, and a view may contain an unlimited number of expressions. For the advanced user there is a C++ API available, exposing the full capability of Quantlab, thus creating a truly open and extensible environment.

Real-time calculations. Connected to one of the supported real-time feeds, Quantlab is able to display and manipulate expressions involving real-time term structures as well as historical time-series data. The evaluation engine efficiently handles the re-calculation.

Historical data. An important purpose of Quantlab is to facilitate the analysis of time-series data, for example the development of the TSIR over time. Any expressions created may thus easily be evaluated over a certain time period, with the evaluation engine automatically taking care of multiple sets of holidays, instruments going on and off the curves involved, different real-time links used for the same instrument over time, to mention but a few of the complexities that Quantlab automatically handles.

SQL Database. The evaluation engine is connected to an industry standard SQL database, where the necessary instrument and curve definitions resides, along with available time series data. The data may come from an external source or may be automatically collected from a real-time feed such as Reuter Triarch by using the Algorithmica History Server. More details on this solution, and the History Server in particular, are available upon request.

External API. Quantlab comes with a C++ API, exposing the full capability of the application, and presenting the C++ programmer with an abundant financial class environment to build upon. Any imaginable additional functionality may thus be added by the inclusion of one or several external plug-in modules. Third party libraries are also expected to be available for a wide range of financial calculations.

COM library. The Quantlab function library is available through a COM interface in Visual Basic. This means that the advanced user can build own applications involving Visual Basic, MS Excel with help of the Quantlab financial calculations and database communication. Moreover, all functions available in

Quantlab are also possible to export to the COM interface, including your own Qlang library files and C++ dll:s.

Python API. As with the com-library, all built-in Qlang functions as well as user-built Qlang extensions are exported to Python using a dynamic link. Every time the ql-extension is loaded to Python, all Qlang functions are automatically loaded, with no need to create function stubs as a separate step.

1.2 Quantlab on-line function browser

Quantlab has an extensive function help browser that describes all functions, object types and keywords, together with numerous code examples. The function browser can be found by pressing F1 when in the Editor. (Editor opened by Ctrl-L). By default, the Function browser connects to Algorithmica Research' on-line help library. This ensures that you always use the latest help documentation. By right-clicking on the Function browser you may select to use the local copy of the browser instead.

1.3 Contents of this document

The pdf help document is organized into five main chapters.

1st chapter – Introduction

2nd chapter – Guides through the workspace environment

3rd chapter – Talks about the programming basics of Quantlab

4th chapter – Describes how to build the user interface with graphs and tables

5th chapter – Presents a collection of case studies for typical uses of Quantlab

2 Development environment

The development environment is designed with the specific requirements of the financial analyst in mind. Traditionally the quantitative analyst has used a combination of development tools including mathematical software packages, spreadsheets, C++ or Visual Basic to complete the work. Quantlab integrates a high level programming language with the possibility to add user-defined functions or classes from C++ or any COM compliant programming language. Quantlab handles all database interaction and real time connections.

2.1 Starting and quitting Quantlab

2.1.1 Starting Quantlab

Start Quantlab by double-clicking on the Quantlab icon on your desktop. An empty workspace is automatically opened. To open an existing workspace file use File|Open. There are example workspaces in the folder Algorithmica Research\Quantlab\Examples\ Workspaces.

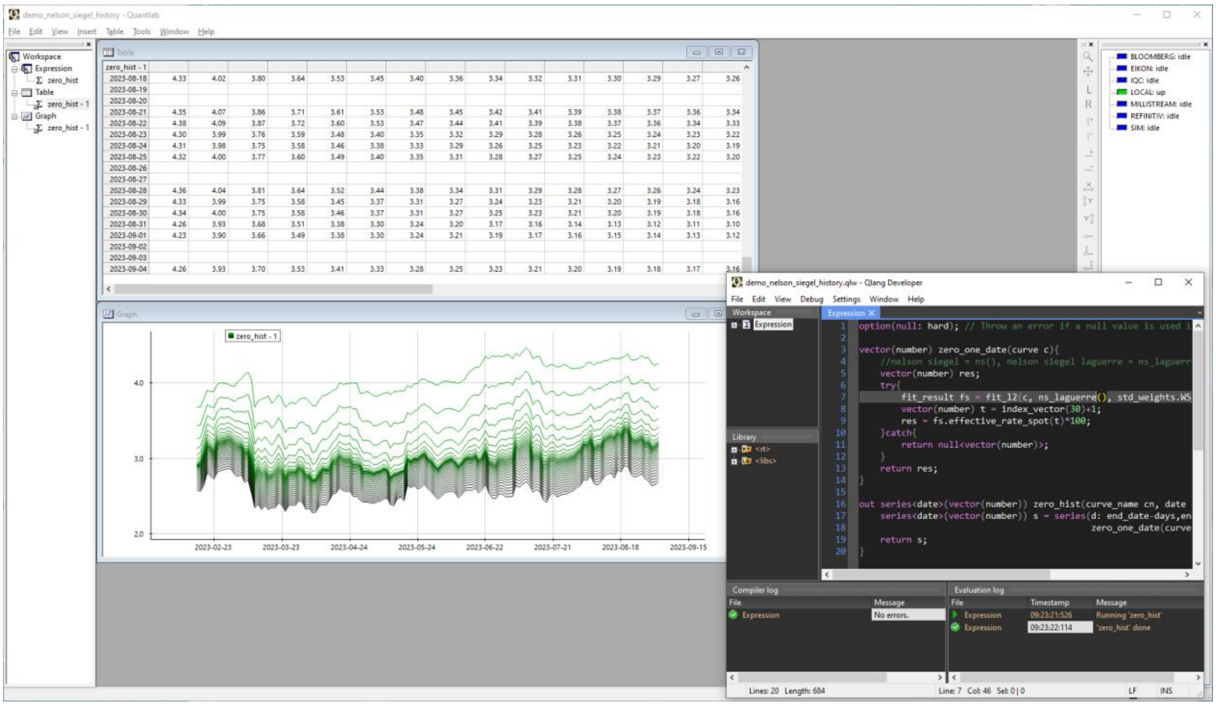
Valid license! Quantlab requires a valid license in order to work. The license is connected either to the computer on which the software initially was installed on or to the user who initially signed for the product. Please contact your IT department to check which type of license you have.

2.1.2 Quitting Quantlab

You quit your Quantlab session by clicking on File | Exit in the main menu.

2.2 Quantlab workspace

The Quantlab system has two faces, a developer environment and a user environment. The developer version work with a workspace and in parallel an Editor opened by Ctrl-L, designed for easy programming and testing. The Function Browser is found within the Editor window by pressing F1.



zero_hist - 1															
2023-08-18	4.33	4.02	3.80	3.64	3.53	3.45	3.40	3.36	3.34	3.32	3.31	3.30	3.29	3.27	3.26
2023-08-19															
2023-08-20															
2023-08-21	4.35	4.07	3.86	3.71	3.61	3.53	3.48	3.45	3.42	3.41	3.39	3.38	3.37	3.36	3.34
2023-08-22	4.38	4.09	3.87	3.72	3.60	3.53	3.47	3.44	3.41	3.39	3.38	3.37	3.36	3.34	3.33
2023-08-23	4.30	3.99	3.76	3.59	3.48	3.40	3.35	3.32	3.29	3.28	3.26	3.25	3.24	3.23	3.22
2023-08-24	4.31	3.98	3.75	3.58	3.46	3.38	3.33	3.29	3.26	3.25	3.23	3.22	3.21	3.20	3.19
2023-08-25	4.32	4.00	3.77	3.60	3.49	3.40	3.35	3.31	3.28	3.27	3.25	3.24	3.23	3.22	3.20
2023-08-26															
2023-08-27															
2023-08-28	4.36	4.04	3.81	3.64	3.52	3.44	3.38	3.34	3.31	3.29	3.28	3.27	3.26	3.24	3.23
2023-08-29	4.33	3.99	3.75	3.58	3.45	3.37	3.31	3.27	3.24	3.23	3.21	3.20	3.19	3.18	3.16
2023-08-30	4.34	4.00	3.75	3.58	3.46	3.37	3.31	3.27	3.25	3.23	3.21	3.20	3.19	3.18	3.16
2023-08-31	4.26	3.93	3.68	3.51	3.38	3.30	3.24	3.20	3.17	3.16	3.14	3.13	3.12	3.11	3.10
2023-09-01	4.23	3.90	3.66	3.49	3.38	3.30	3.24	3.21	3.19	3.17	3.16	3.15	3.14	3.13	3.12
2023-09-02															
2023-09-03															
2023-09-04	4.26	3.93	3.70	3.53	3.41	3.33	3.28	3.25	3.23	3.21	3.20	3.19	3.18	3.17	3.16

```
option(null: hard); // Throw an error if a null value is used i

vector(number) zero_one_date(curve c){
    //nelson siegel = ns(), nelson siegel laguerre = ns_laguerr
    vector(number) res;
    try{
        fit_result fs = fit_l2(c, ns_laguerre(), std_weights.WS
        vector(number) t = index_vector(30)+1;
        res = fs.effective_rate_spot(t)*100;
    }catch{
        return null<vector(number)>;
    }
    return res;
}

out series<date>(vector(number)) zero_hist(curve_name cn, date
    series<date>(vector(number)) s = series(d: end_date-days,en
                                        zero_one_date(curve
    return s;
}
```

```
Compiler log
File            Message
Expression      No errors.
```

```
Evaluation log
File          Timestamp       Message
Expression    09:23:21:526    Running 'zero_hist'
Expression    09:23:22:114    'zero_hist' done
```

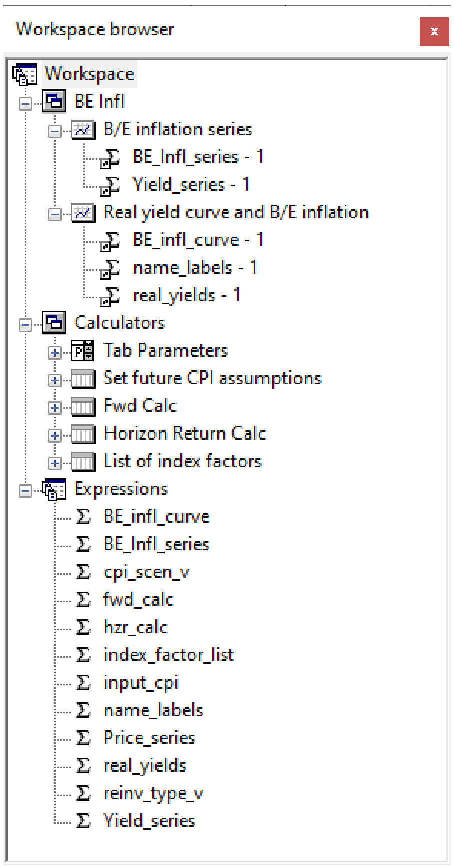
Workspace showing the main features of the developer edition

Windows within the workspace can be moved around freely. Windows with double bars on one edge dock to the side you move it to. To un-dock a window simply double click on the window side with double bars. If you do not want the window to dock, move it having the Ctrl button pressed down.

2.3 Workspace tools

2.3.1 Workspace browser

In the workspace browser you can organize your expressions, tables, and graphs into folders and tabs. On opening Quantlab an empty workspace will be created.



Workspace browser ×

- Workspace
  - BE Infl
    - B/E inflation series
      - BE_Infl_series - 1
      - Yield_series - 1
    - Real yield curve and B/E inflation
      - BE_infl_curve - 1
      - name_labels - 1
      - real_yields - 1
  - Calculators
    - Tab Parameters
    - Set future CPI assumptions
    - Fwd Calc
    - Horizon Return Calc
    - List of index factors
  - Expressions
    - BE_infl_curve
    - BE_Infl_series
    - cpi_scen_v
    - fwd_calc
    - hzr_calc
    - index_factor_list
    - input_cpi
    - name_labels
    - Price_series
    - real_yields
    - reinv_type_v
    - Yield_series

A workspace example

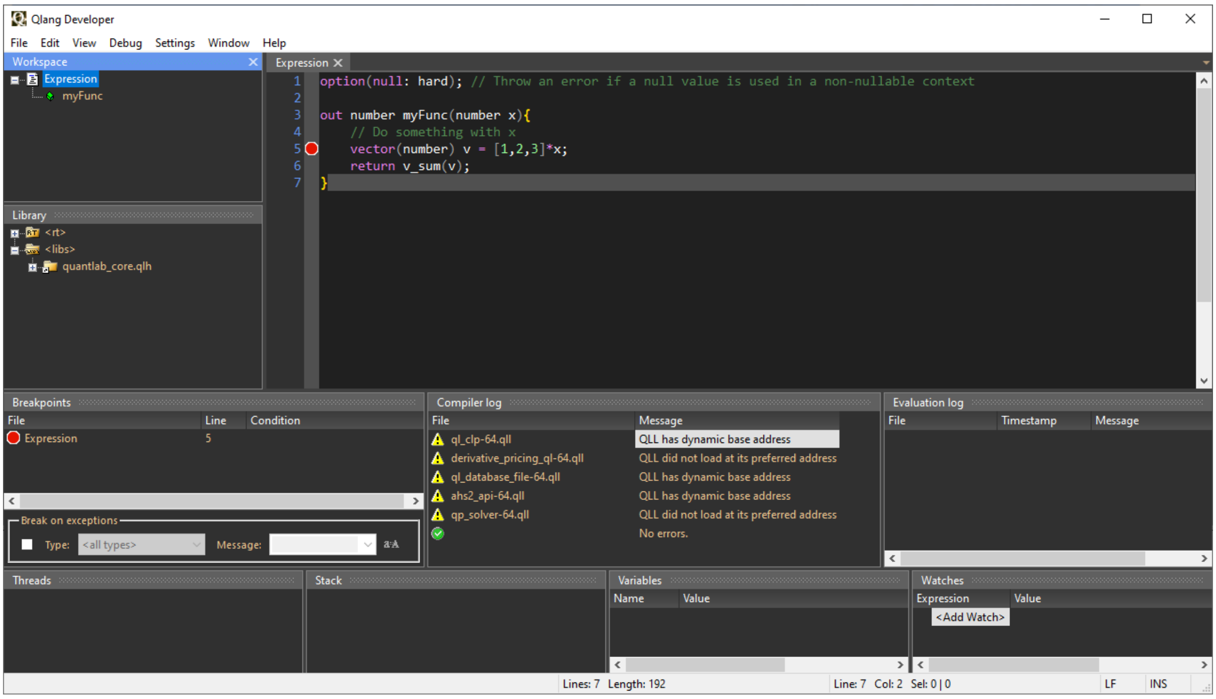
By right clicking with the mouse on the workspace icon you can insert folders to organize your project. To visually organize your tables and graphs in the workspace, you can insert tabs using Insert | Tab on the menu. Graphics and tables can then be dragged-and-dropped under a relevant tab.

Rename a folder, tab, expression, graph or table by right clicking on the object you want to edit. Right clicking also displays the most common properties of each object.

You can change the order among tabs by dragging and dropping: Use the left mouse button to drag a tab to the workspace symbol in the top of the workspace browser. This will put the selected tab last in the list.

2.3.2 Qlang Developer Editor

In the Editor you write the code to be executed and viewed in tables or graphs. It opens in a separate window using Ctrl+L, or Tools | Qlang Developer. Code written in the editor is saved in plain text so you can import and export this code to any text editor. To get help on using the built-in functions please see the Function browser (opens with F1).



```
option(null: hard); // Throw an error if a null value is used in a non-nullable context

out number myFunc(number x){
    // Do something with x
    vector(number) v = [1,2,3]*x;
    return v_sum(v);
}
```

Breakpoints

File	Line	Condition
Expression	5	

Compiler log

File	Message
ql_clp-64.qll	QLL has dynamic base address
derivative_pricing_ql-64.qll	QLL did not load at its preferred address
ql_database_file-64.qll	QLL has dynamic base address
ahs2_api-64.qll	QLL has dynamic base address
qp_solver-64.qll	QLL did not load at its preferred address
	No errors.

Evaluation log

File	Timestamp	Message

Sample code in the Expression editor

Coding help is available as colour-coded text, parenthesis checks, function parameter lookups, and object member lists.

Default colour codes:

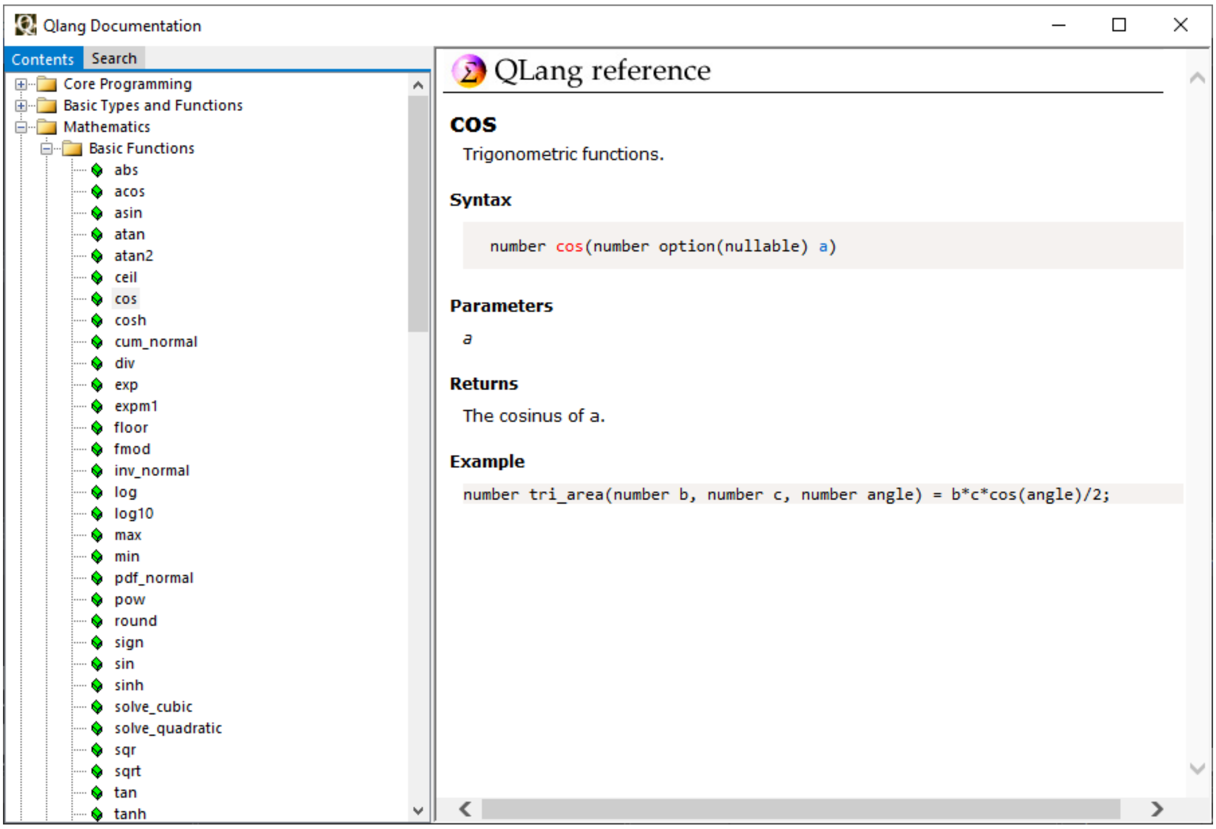
Colour code	Type of code
Green	Comments
Blue	Key words in Qlang
Beige	String
Orange	Unfinished string
Green-blue	Date
Fluorescent green	Unfinished date
Violet	Number

User defined settings for the Developer Editor can be found in the top menu Settings, and Settings | Appearance, where also background colour and indentation can be changed. Note that there are detailed descriptions in the separate document Quantlab Editor 3.1.x.pdf document.

Useful tool-tip! To assist in entering built-in or user functions, a tool tip will be displayed when you enter the first left hand parenthesis of the function. All objects will show its member list when entering the dot after the name.

2.3.3 Function browser

An important part of the help that comes with Quantlab is the function browser. Press F1 or Help | Qlang documentation to open. All types, functions, objects, and members are displayed. If you write library files or program your own C++ functions using the Quantlab API, these functions will also show up in the function browser.



cos

Trigonometric functions.

Syntax

```
number cos(number option(nullable) a)
```

Parameters

a

Returns

The cosinus of a.

Example

```
number tri_area(number b, number c, number angle) = b*c*cos(angle)/2;
```

The Function browser

The left window will display all available objects, members and functions. The right-hand window will display the active functions' overloaded variants including the type and name of the parameters.

Some functions have example code for ease of understanding. The help window is by default read directly from Algorithmica Research' web servers. A locally stored help can be accessed by right-clicking in any top window and de-selecting "online help" in the menu. (If initially installed with the program.)

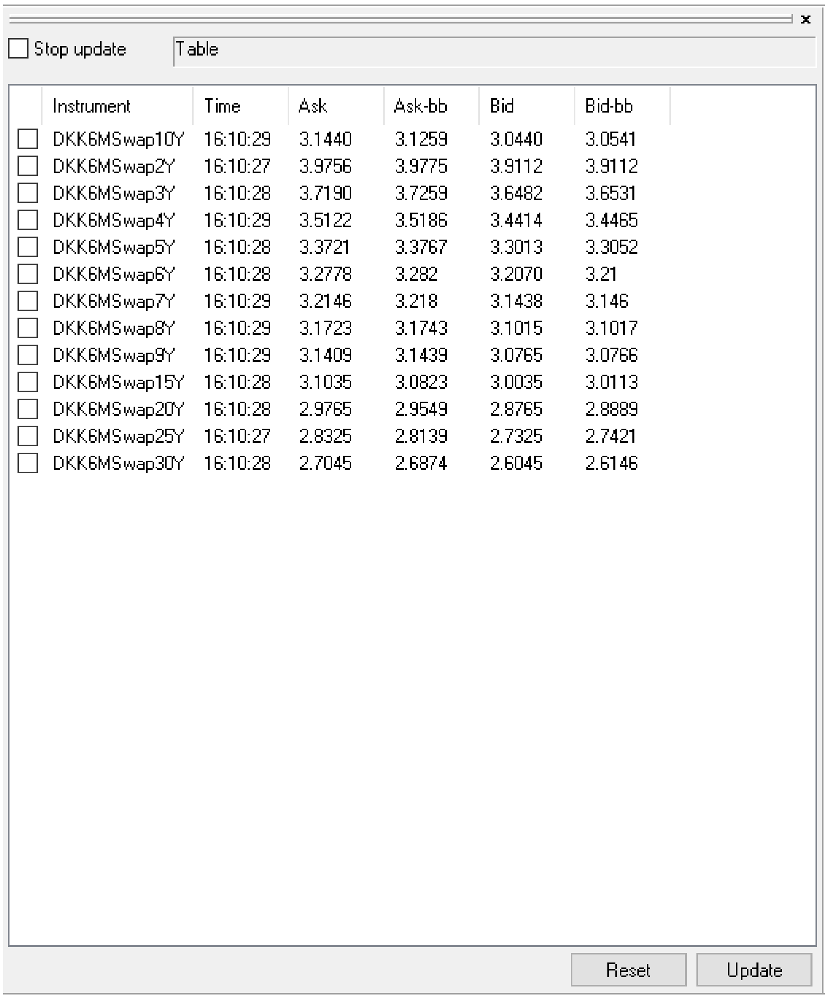
Speed tip! Using search and then clicking on a given found function, you will get to the place in the tree where that function is placed.

2.3.4 Real time quotes browser

When viewing any graph or table that includes today's date, the real time browser will display real time data. By default, today's date is used whenever you want to view real time data.

Only instruments relevant to the *active* graph or table will be displayed in the real time browser. In order to view real time quotes for another graph or table, simply click on it to make it the active window.

The order of the columns can be changed and switched on/off by right-clicking in the window and selecting “column properties”.



□ Stop update   Table

	Instrument	Time	Ask	Ask-bb	Bid	Bid-bb
□	DKK6MSwap10Y	16:10:29	3.1440	3.1259	3.0440	3.0541
□	DKK6MSwap2Y	16:10:27	3.9756	3.9775	3.9112	3.9112
□	DKK6MSwap3Y	16:10:28	3.7190	3.7259	3.6482	3.6531
□	DKK6MSwap4Y	16:10:29	3.5122	3.5186	3.4414	3.4465
□	DKK6MSwap5Y	16:10:28	3.3721	3.3767	3.3013	3.3052
□	DKK6MSwap6Y	16:10:28	3.2778	3.282	3.2070	3.21
□	DKK6MSwap7Y	16:10:29	3.2146	3.218	3.1438	3.146
□	DKK6MSwap8Y	16:10:29	3.1723	3.1743	3.1015	3.1017
□	DKK6MSwap9Y	16:10:29	3.1409	3.1439	3.0765	3.0766
□	DKK6MSwap15Y	16:10:28	3.1035	3.0823	3.0035	3.0113
□	DKK6MSwap20Y	16:10:28	2.9765	2.9549	2.8765	2.8889
□	DKK6MSwap25Y	16:10:27	2.8325	2.8139	2.7325	2.7421
□	DKK6MSwap30Y	16:10:28	2.7045	2.6874	2.6045	2.6146

Example of the real time browser with instruments and quotes

The Stop check box will turn of all pass through of quotes to the analysis and code. The individual check boxes at the side of the instrument will stop only that instrument to update.

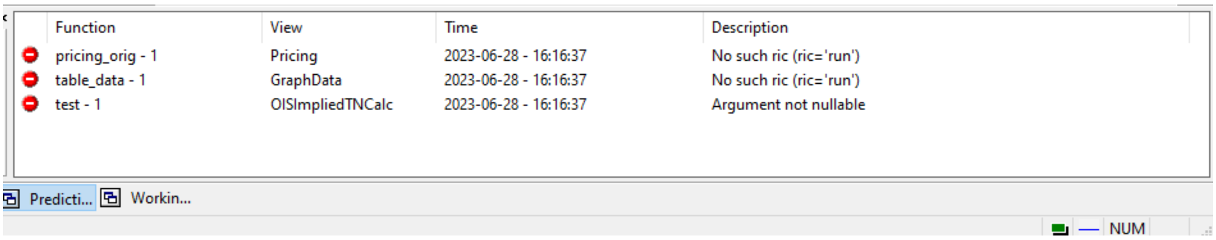
Note! It is no longer possible to override the quotes with user defined values directly in this window. In order to override quotes, code needs to be written using the “.set_quote()” function for the instrument.

2.3.5 Messages - warnings and run-time errors

There are two phases to programming and testing financial expressions in Quantlab. First the written expression must be syntactically correct. This is taken care of within the Qlang Developer Editor and compiler. Secondly, the financial expression must make sense when used on real world data.

General settings for when and why the messages should display can be found in Tools | Options | Messages. Read further under 2.4.4.

When running an expression window any run-time messages will show in the View | Messages tab.



Function	View	Time	Description
pricing_orig - 1	Pricing	2023-06-28 - 16:16:37	No such ric (ric='run')
table_data - 1	GraphData	2023-06-28 - 16:16:37	No such ric (ric='run')
test - 1	OISImpliedTNCalc	2023-06-28 - 16:16:37	Argument not nullable

Example of warnings in the run time environment

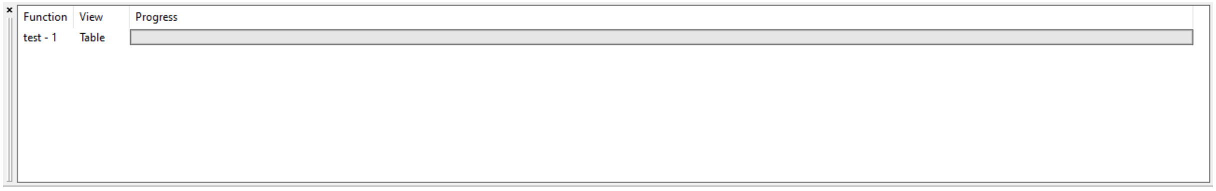
Typical warnings occur when:

- historical data is missing in the database
- calculations fail due to missing data
- required static data is missing for any instrument

By right clicking on the warnings, copy, clear and remove functions appear.

2.3.6 Progress list

Workspaces can eventually contain hundreds of expression that will be evaluated each time any parameters or quotes change. The progress list will display which expressions that currently are being re-calculated.



Function	View	Progress
test - 1	Table	

Example of the progress list

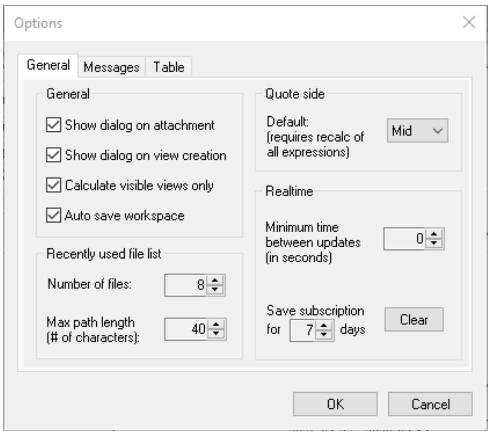
For each graph or table a star (*) in the window header is shown as long as there is at least one attachment that is still calculating.

To manually stop the execution of an expression, right click on the expression attached to a specific graph or table in the workspace browser and press Stop. The evaluation cannot be stopped in the progress window.

Speed tip! If the computer starts to slow down due to extensive re-calculations, you can manually set a re-calculation frequency. Under the Tools | Options menu, the real time re-calculation frequency can be set to a higher number. For example, entering 10 seconds will drastically reduce the load.

2.4 *Setting preferences (Tools)*

Under the Tools Option menu, you will find some useful possibilities to set preferences for the general appearance of Quantlab, for the expression window and for paths when loading various library files.



General

☑ Show dialog on attachment

☑ Show dialog on view creation

☑ Calculate visible views only

☑ Auto save workspace

Recently used file list

Number of files: 8

Max path length (# of characters): 40

Quote side

Default: (requires recalc of all expressions) Mid

Realtime

Minimum time between updates (in seconds) 0

Save subscription for 7 days

2.4.1 Database options

The name of the ODBC data source that is used for retrieving all instrument and time series data. You can change the ODBC source and then press Reload in order to switch the database. This is equivalent to closing down Quantlab and re-open it with the current workspace.

2.4.2 In tab General:

- **Option to show a dialog** when attaching functions to graphs or tables
- **Option to show a name dialog** when creating a view (table or graph)
- **Option to calculate visible views only.** Normally, this should be checked as it speeds up performance. If all views must be calculated at each relevant real time update, it should be unchecked.
- **Number of files in the recently used files** list and the maximum number of characters used for the name and the paths in the menu.
- **Default quote side**. When constructing an instrument or a curve you have several possibilities to set the quote side. However, often the quote side is an optional parameter and if it is not set, the default value defined in this dialog will be used.
- **Minimum time between calculations**. Useful for reducing the number of recalculations when real time data is updated frequently. This is only used for the calculations, the Realtime Quotes window (see 2.3.4) will not be affected.
- **Save favourite real time instrument names** (real time identification codes) for faster workspace start-up times. You set the number of days that Quantlab will save what instruments you have used. When the options is set at 1 or more days, on start-up Quantlab will request a subscription to these real time items even before you open a workspace. Often, this will drastically reduce the time it takes to open workspaces having many real time instruments. The accumulated history of real time items can be cleared by pressing the Clear button.

2.4.3 In tab Extended:

- **Maximum number of warnings**. Put a limit on how many warnings will be written in the warnings window each time an attachment is evaluated. If the number of errors in your expressions or database is very large, the mere update of this window may be time consuming. In such a case it may be useful to limit the number of warnings until the errors are taken care of.
- **Calculation thread priority**. This can be changed in order to let Quantlab get larger or smaller part of the CPU time of the computer.

2.4.4 In tab Messages:

- **Warnings,** displaying run-time messages such as real-time data missing or non-evaluating functions.
- **Compiler messages and errors,** displaying information from the compiling session.

The display level can be individually set to:

- **Always show**
- **Show if message window visible**
- **Don’t show**

2.4.5 In tab Table:

- **Show errors messages**. In some cases, errors may be specific for a cell in a table. To show all upcoming errors within the tables, click this check box.
- **Enable direct cell editing.** If this is checked you can select a cell in an input column in a table and edit directly. Otherwise you have to double-click or use F2.

2.4.6 In tab Edit (for the expression editor):

- **Choice of indentation**. Automatic indentation and tab length.
- **Debug windows.** Number of decimals when showing numerical values of variables.
- **Colour display setting**. Either use default setting or choose your own colours for different types of text in the expression windows.

3 Programming in Quantlab using Qlang

3.1 *Introduction*

The Quantlab language Qlang makes financial programming easy! Qlang is designed to work with time series data vectors and matrices. Included in Qlang is also an extensive function library with many tailor-made fixed income functions that makes programming fast and efficient.

Programming in Quantlab follows these basic steps:

1. Write code in an expression editor
2. Check and compile the code [by pressing F7, or choosing Exp, Compile]
3. If code compiled correctly, the "out functions" appear in the Workspace browser
4. Drag and drop the functions from the Workspace browser to graphs and tables

When programming is finished and workspaces have been created, they are ready to be distributed to others. Everyone using Quantlab, either Developer or User Edition, can now run all analytics of the workspace presented in graphs and tables.

3.2 *A simple example*

Let's look at a simple function that adds two numbers:

```
number my_add(number x, number y) = x + y;
```

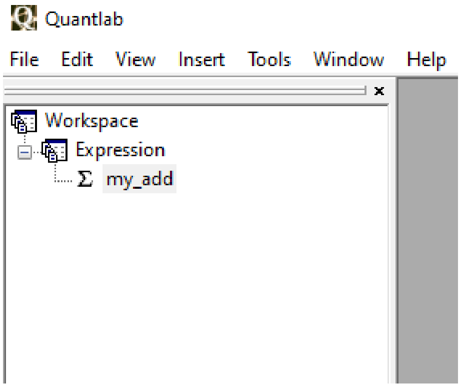
In order to make this function available for presentation in tables or graphs the keyword out is used, written at the start of the function definition:

```
out number my_add(number x, number y) =  x + y;
```

If you write this function in an expression window (Insert – Expression) you will be able to compile the code by pressing F7. As the code is correct you will not get any warnings. Try to change the code to something incorrect, for example:

```
out number my_add(number x, number y) =  x + y +
```

If you then press F7 again, you will get an error message (if messages have not been switched off in under options | messages).



Change back to the original version of the function, recompile the code and look for the Workspace window. In the Workspace window (View – Workspace) you will find a + sign. If you click the plus sign, you will find the function my_add. This function is available to attach to a table. To do so, insert a new table (Insert – Table) and give it a name. Then drag the function my_add (using the left key of the mouse) to the table window and drop it there. Two Number Edit Boxes will appear next to the table, one for each parameter of the function, and these are used for choosing the values of x and y. Enter two numbers in the boxes and press Recalc to make Quantlab evaluate your function and show the result in the table.

Chapter 6 explains more about how to format tables and graphs.

3.3 Keyword list

This section is a summary of the keywords that form the language base. Keywords are marked with blue when written in the expression editor. (In default appearance mode. See Settings | Appearance for more options.) Note that a description on each keyword can be found in the function browser.

The keywords Module, Public and Import are used for the management of expressions.

module	The module keyword creates a namespace of functions.
public	The public keyword assigns a function to be available outside a module.
import	The import keyword loads the public functions of a module so that they become local functions.

The keyword Return is used when defining functions in the standard form.

return	The return keyword terminates the current function call and returns the value or object following it. See 3.4.1.

The keyword void is for defining procedures.

void	The void keyword replaces the return value type for a function that does not return any value. See 3.4.1.

The keyword out is special for Qlang and used for making functions available in the user interface.

out	The out keyword assigns a function to become available for attachments to graphs or tables after compilation. See 3.4.4 for the use on parameters.

The keyword option is special for Qlang and used for certain options for variables and functions.

option	option(nullable) is used in a function definition before a parameter name to make the function possible to call with a null value in that parameter. option. option (category: <string>) is used immediately after the function header in a library file to indicate what category the function shall appear in within the Function browser. The string contains the name of the category, either an existing category or a new one. option (com_name: <string>) is used immediately after the function header in a library file to publish the function to the COM interface (for use in excel or C#) using a user defined name. This is necessary if functions in library are overloaded as COM does not allow for overloaded functions.

The keywords if, switch, else, do, while, for, break and continue are used for flow control.

if else	The if and else keywords are used for conditional expression evaluation. See 3.8.1.

switch	The switch keyword is used when there are several cases in a comparison situation. See 3.8.3
while	The while keyword is used for conditional loops with initial condition test. See 3.8.1.
do while	The do while keywords are used for conditional loops with final condition test. See 3.8.1.
for	The for keyword is used for unconditional loops. See 3.8.2.
break	The break keyword terminates the smallest enclosing loop statement (do, for, or while) in which it appears.
continue	The continue keyword terminates the current iteration of the smallest enclosing loop statement (do, for, or while) in which it appears, and the execution continues with the next iteration.

The keywords Try and Catch are used for error handling.

try	The try keyword introduces a code section in which errors are expected to occur. See 3.8.4.
catch	The catch keyword introduces a code section that takes care of the errors in the preceding try section. See 3.8.4.

The keywords String, Matrix and Vector are used for creating specific types.

string	The string keyword assigns a variable, function or a function parameter to be a string. See 3.7.1.
matrix	The matrix keyword assigns a variable, function or a function parameter to be a matrix. See 3.7.7.
vector	The vector keyword assigns a variable, function or a function parameter to be a vector. See 3.7.7.

These keywords are always used in connection with an object type. For example, to create variable which is a vector of numbers, you write

```
vector(number) my_variable;
```

The keyword Series is a special Qlang feature used in particular for time-series calculations.

series	The series keyword creates a series of elements by evaluating an expression over a range. The dimension of the series is equal to the number of ranges. See 3.7.10.

The keywords Object and New are used when defining objects.

class	The class keyword is used when defining an object class. See 3.7.3.
object	The object keyword is used when defining an object class. See 3.7.3.
new	The new keyword is used when creating an instance of an object. See 3.7.3.

The keywords typedef and enum are used for defining types.

typedef	The typedef keyword is used when giving new names to types. See 3.7.6.
enum	The enum keyword is used when creating enum lists. See 3.7.8.

The keyword Function is used for function reference in function definitions.

function	The function keyword refers to a function parameter in a function definition. See 3.4.6.

The keyword operator is used when defining operators, typically for objects.

operator	The operator keyword is used when defining operators, see 3.7.4.

3.4 Functions

Functions can be defined in two ways, in the standard way and the compact way. In the standard function definition, the syntax resembles very much the syntax of C++ or similar languages, in the compact form the function definition is written by using only one expression. Both forms can be used in the same expression window.

3.4.1 Standard function definition

The standard way of defining a function looks like this:

```
return_value_type function_name (parameter_type1 parameter_name1, …)
{
  < function body >
  return < expression >;
}
```

The function is defined by declaring the return value type, the function name followed by a parenthesis, and then a function body. Inside the parenthesis all the function parameters are defined by writing their types and names. Note that each parameter requires its own type definition, and that these types are specific for Qlang. Within the function body the usual variable declarations and operating statements are written, each ending with a semi-colon. The function returns the value of the expression that follows immediately after the keyword return.

For example, a function adding two numbers:

```
out number my_add(number x, number y){

       return x+y;

}
```

The return value type must be declared as any of the Qlang types. If the return value type is set to void and the return statement is omitted, the function returns no value:

```
void function_name (parameter_type1 parameter_name1, …)
{
  < function body >
}
```

The keyword out is used to make a function available for attaching to graphs or tables. For example, the following function could be attached to a table in order to display a multiplication table:

```
out vector(number) mult(number x){
  vector(number) y = [1, 2, 3, 4];
  return x*y;
}
```

3.4.2 Compact function definition

This is an alternative way of defining functions, which is useful for simple expressions. It is written starting with a return value type and a function name, followed by a parenthesis containing the parameters, and then an equality sign. After the equality sign must be a complete function expression. The function takes the following form:

```
return_value_type function_name (parameter_type1 parameter_name1, …) = <
function_expression >;
```

Note that the expression ends with a semi-colon. The my_add example used earlier would look like this:

```
out number my_add(number x, number y) = x + y;
```

These functions are often written in one row, but for the sake of readiness, they can be written in several rows:

```
out number my_add(number my_x_parameter, number my_y_parameter) =
my_x_parameter + my_y_parameter;
```

New since 3.1.2048: the generic return type "auto" can be used. It is used in the same way as in C++.

```
out auto mult(number x){
  vector(number) y = [1, 2, 3, 4];
  return x*y;
}
```

3.4.3 Instances of functions

When a function is attached to a table or a graph an *instance* of the function is created. You may have several instances of the same function in the same graph or table. In the calculations and user interface, Quantlab will treat them as separate functions that may or may not use the same input parameters. Only when the code, i.e. the definition of the function, is changed this will affect all function instances.

3.4.4 Function parameters

Function parameters are defined by the type followed by the parameter name, as described in 3.4.1. There is also a possibility to define optional parameters with a default value by setting these parameters equal to an expression, for instance:

```
number my_add(number x, number y = 1){
      return x+y;
}
out number test_my_add(number x){
      return my_add(x);
}
```

The first function, my_add, has one optional parameter y which is defaulted to 1 if not set, as in the second function test_my_add. The default value may also be something more complex, for example a function call:

```
number default(number x){
        return x*2;
}
number my_add(number x, number y = default(2)){
        return x+y;
}
out number test_my_add(number x){
        return my_add(x);
}
```

In this example we have defined a separate function called default which calculates the default value for the function.

When attaching functions to graphs or tables, all parameters, including optional ones are displayed, without default values.

When calling functions, the function parameters are by default copies (for number, date and logical) or copies of references (for string and object) of the original variable.

In the following example, the function f2 returns 0 as function f1 only changes a copy of the original variable.

```
void f1(number c)
{
  c = 1;
}
out number f2(){
  number b = 0;
  f1(b);
  return b;
}
```

Using the keyword out before a parameter definition will cause that parameter to be referring to the original variable (for number, date and logical) or original reference (for string and object), allowing change of the external environment. This is similar to the Pascal VAR parameter and the C++ way of declaring a parameter as a reference (using &).

In the following example, the function f2 returns 1 as function f1 changes the value of the original variable.

```
void f1(out number c){
  c = 1;
}
out number f2() {
  number b = 0;
  f1(b);
  return b;
}
```

3.4.5 Calling functions

There are a large number of pre-defined functions in Qlang for general and financial purposes. These are divided into groups according to their purpose and can be found in the Quantlab function browser. The function browser is made visible from the Quantlab menu bar (View – Function Browser).

Both user-defined functions and pre-defined functions are called similarly to common programming languages. For example the user-defined function my_add in 3.2 could be called like this:

```
result = my_add(2, 3);
```

if we have defined a variable called result, of the type number.

Important news! From version 3.0 and onwards, functions without any parameters must be called using an empty parentheses, for example my_func(). This is a common case for many object member functions.

3.4.6 Function pointers

In Quantlab 3.0 the concept of function pointers is introduced. It is useful in various cases, for example when performing optimisation calculations. Here is an example which uses the minimisation function zero_bisect. See also 3.7.3 for information on object classes.

```
class param_object{
// This object contains all parameters that are used for
// calculating the function f,except for the variable x.
public:
       number a_param;
};

number f(param_object p, number x){
// The object function
       return -p.a_param*x + 1;
}

out number test(){
// Find x that makes f = 0.
       param_object p = new param_object;
       p.a_param = 4;
       number x1 = -3;
       number x2 = 4;
       number tol = 0.001;
       return zero_bisect(p, &f, x1, x2, tol);
}
```

On the last line we call zero_bisect with a reference to the function f using the &-sign. The function zero_bisect requires that the function that is referenced to (in our case, f) has two parameters: An object and an x-parameter. In this way you can construct a function f that is arbitrarily complicated as long as it is a function of a single variable x.

The following example shows how to define your own function pointers. The function calc below performs any calculation using a function f that takes two parameters. We have defined two such functions: plus and minus. The last function below uses these functions depending on the user input.

```
number plus (number a, number b) = a + b;
number minus (number a, number b) = a - b;

number calc(number a, number b, number function (number a, number b) f){
       return f(a, b);
}

out number test(number a, number b, string method){
       if(method == 'plus')
              return calc(a, b, &plus);
       if(method == 'minus')
              return calc(a, b, &minus);
}
```

The syntax for the declaration of the function pointer is thus:

```
<return_type> function (<type>param_1, <type> param_2…) function_param_name
```

More examples of function pointers are found in the case study in 8.12.

3.5 Local and global variables

In a function, local variables may be declared as in other common programming languages, with or without initialisation, for example:

```
<function declaration>{
number x;
number y = 5;
vector(instrument) i;
…
}
```

For global variables the treatment is somewhat more special. Global variables are only common to all calls from functions within the same expression window. An example:

```
number x;
out number my_function(number y){
	if (null(x)){
		x = 0;
	}
	else{
		x = x+1;
	}
	return x*y;
}
```

Each time my_function is called, the variable x will be updated. An extensive example of how to use global variables is discussed in 8.9.

Important! It is important to note that multiple instances of functions attached to the same or different tables or graphs will all refer to the same instance of the global variable.

3.6 Operators

Common mathematical and logical operators are available in Qlang. Operators work like ordinary functions, for example allowing vector and matrix expansion, see 3.7.7.

Arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
^	power (can also be written using the function pow)

Note! The type of multiplication is determined by the operands. A multiplication of two vectors will result in a scalar (so called scalar product). A multiplication of matrices or a matrix and a vector is treated as a matrix multiplication. See 3.7.7.

Logical operators:

!	logical negation
!=	not equal to
&&	logical AND
||	logical OR
<,>	logical relation operators
==	logical equality operator

There is also a simple conditional operator available using the following syntax:

```
<conditional expression> ? <expression1> : <expression2>
```

If the conditional expression is evaluated as true then expression1 will be returned, otherwise expression2. For example, the following function will return x if it is positive, otherwise 0.

```
number my_function(number x) = x>0 ? x : 0;
```

Quantlab will perform a short-circuit evaluation of logical and conditional expressions, only executing those that are necessary.

Note! As in many languages, assignment operators are allowed together with the assignment sign ("="). Such that <variable> /= 10; means divide value in variable with 10.
Also:
+= for add

-= for subtract

*= for multiplication

3.7 Data types

Qlang carries a rich family of types, much like any modern programming language.

3.7.1 Basic types

The number of basic types in Qlang is limited for ease of use. Qlang has the following seven basic types.

Type	Description	Literals
Number	The basic float numeric type	Numbers are simply entered as they are. Very small or large numbers can be written with mantissa, then d, D, e or E, then the exponent. 1.2e-4 is interpreted as 0.00012, or 1.2 basis points.

Integer	Integer type	Same as in other languages. Any indexation for vectors, loops etc starts with 0.
Date	The basic date type	Dates are written using # then ISO standard dates. #2000-12-29 is interpreted as the 29th of December 2000.
Timestamp	Basic long date including time	Timestamps are written using # with iso date and time down to optional thousands of a second. ex. timestamp tss = #2021-01-01 12:30:22.222;
Logical	The basic Boolean type	True or false.
String	The basic character string type	Strings are encapsulated by simple or double quotation marks. Both 'text' and "text" are interpreted as strings containing the word text.
Month	A month basic type	It is a date without any day reference. e.g. #2023-03.
Object	The basic object reference type	No literal.

Of the above basic types, only the object basic type cannot be used directly in variable or function parameter declaration. Instead of the object basic type, the object types described below are used.

This is an example of using some basic types to get different user controls when attached to a table.

```
out number myFunc(string myStr, number myNum, date myDate)
{
  …
}
```

Quantlab recognizes the types and automatically creates appropriate controls.

3.7.2 Object types and member functions

There are many object types available in Qlang. Many of them are financial, such as instrument and curve, but there are a others used for presentation of data, for mathematical purposes and so on.

Objects are generally created as a result of Qlang function calls. They can then be stored in variables or used directly for further function calls. Objects also have member functions, which can be called using a standard dot notation.

Below is an example of a function that returns the yield of an instrument on a specified trade date. An instrument object is created from the information in the database, and then the member function `yield()` is called to extract the sought yield.

```
number my_yield(instrument_name i, date tradeD){
  return instrument(i, tradeD).yield();
}
```

Some objects methods have a corresponding function in another object. As an example, both rows below will return the dirty price of an instrument priced from a zero coupon curve model fit of the market rates.

```
fit_result.dirty_price (instrument);

instrument.dirty_price (fit_result);
```

3.7.3 Creating new classes

An introduction to working with user-defined classes

In Quantlab 3.0 it is possible to create new classes with member variables and member functions. The syntax is very much in line with C++. Below is an example of a definition of an object class that stores two numbers, called my_pair. The object class has a member function that adds the two numbers and a creator with the same name as the object class. The last function can be attached to a table in order to test the object class.

```
class my_pair
{
       public:
       number add() ;

       number x ;
       number y ;
};


//Note that you need to declare all member functions inside the class definition – as
is done with the function add() above.

my_pair my_pair(number x, number y){
       my_pair t_n = new my_pair ;
       t_n.x = x ;
       t_n.y = y ;
       return t_n ;
}

number my_pair.add(){
       return x + y ;
}

out number test_my_pair(number x, number y){
       my_pair p = my_pair(x, y);
       return p.add();
}
```

The class definition syntax

A user-defined class is defined using the following syntax:

```
class <class name> [ : <class to inherit from> ]
{
       [public:]|[private:]
       ...
       [<name of constructor (i.e. class name)>(<params>) ;]
       ...
       [virtual <return type> <name of virtual member function>(<params>) ;]
       ...
       [<name of member function>(<params>) ;]
       ...
       [<type of member variable> <name of member variable> ;]
       ...
} ;

<class name>.<name of constructor>(<params>)
[: <name of member variable>(<params>),...]
{ <body> ]
...

<return type> <class name>.<name of member function>(<params>)
{ <body> }
```

```
...
```

A class may inherit from another class – called a super-class – by using the familiar `: <class to inherit from>` notation above.

In a `class` – as opposed to an `object` – all members and member functions (including constructors) are declared `private:` by default. This means that they cannot be accessed by any code outside the class. In order to make them accessible from the outside they need to be within the scope of a preceding `public:` declaration:

```
class myClass
{
        public:
        number get_secret_number() ;

        private:
        number secret_number ;
} ;
```

Now, the secret_number above can only by accessed through the get_secret_number().

A special type of member function called a **constructor** is used to initialize the member variables of a class. It is possible to do this using the familiar [: <name of member variable to initialize>(<params>)] notation, as in the following example:

```
class myClass
{
        public:
        myClass(number n) ;

        number get_secret_number() ;

        private:
        number secret_number ;
} ;

myClass.myClass(number n)
: secret_number(n)
{}

number myClass.get_secret_number()
{
        return secret_number ;
}
```

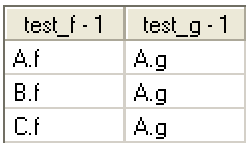
Note that a constructor cannot have an explicit return type, since it implicitly returns the newly created class object.

The `virtual` keyword declares a virtual member function that can be overridden by any inheriting class as seen in the following example. Note that all virtual member functions need to be defined in all classes – also in the super-class:

```
class A
{
        public:
        virtual string f() ;
        string g() ;
} ;
```

```
string A.f() { return “A.f” ; }
string A.g() { return “A.g” ; }

class B : public A
{
       public:
       virtual string f() ;
       string g() ;
} ;

string B.f() { return “B.f” ; }
string B.g() { return “B.g” ; }

class C : public A
{
       public:
       virtual string f() ;
       string g() ;
} ;

string C.f() { return “C.f” ; }
string C.g() { return “C.g” ; }

out vector(string) test_f()
{
       vector(A) v = [ new A, new B, new C ] ;

       return v.f() ;
}

out vector(string) test_g()
{
       vector(A) v = [ new A, new B, new C ] ;

       return v.g() ;
}
```

The output table below shows the difference between the `virtual` member function f() and the normal member function g():



test_f - 1	test_g - 1
A.f	A.g
B.f	A.g
C.f	A.g

Scope of class members and the use of this

The familiar dot notation is used to access member variables and calling member functions in a class:

```
A a = new A ;

a.my_number = 42 ; // Accessing one of A:s members

a.f() ; // Calling one of A:s member functions
```

By explicitly naming the class name after the dot it is possible to access member functions or variables that are normally hidden by definitions in inheriting classes. This can be done even if the function is not declared as `virtual`:

```
B b = new B ;

b.f() ;   // Calling the f() member function defined in B

b.A.f() ; // Calling the f() member function defined in A
```

It is possible for a member function of a class to obtain a handle to itself by using the `this` keyword. This can for example used as a parameter in function calls as in the following example:

```
string func(A a)
{
       return a.f() ;
}

logical A.func()
{
       string s1 = this.f() ;
       string s2 = func(this) ;

       return s1 == s2 ;  // always returns true
}
```

Please note though, that a member function can always access all of its own member variables directly without using `this`.

Casting classes

When working with class hierarchies it is often useful to convert a handle to a super class object into a handle of the actual base class it belongs to (or any other class in between). This is called a dynamic cast and is performed by using the `dynamic_cast` operator:

```
A a = new B ;        // This is possible since class B inherits from class A

B b = dynamic_cast<B>(a) ; // Convert into a handle to a B
```

If the cast is not possible due to the classes not being members of the same class hierarchy it will fail and an error will be thrown.

When writing library files, it is also possible to add new member functions to built-in object classes, see 4.3. More examples of creating object classes are found in the case study in 8.12.

3.7.4 Example of classes and operators

Here is a simple example where complex numbers and the + operator is defined in a library file:

```
class complex {
private:
   number r, i;
public:
   complex( number r, number i );
   number re();
       number im();
};


complex.complex( number r, number i ) : r(r), i(i) {;}
```

```
number complex.re(){
       return r;
}

number complex.im(){
       return i;
}

// Operator overloaded using a member function
complex operator + (complex c, complex d)

{
   return  new complex(c.re() + d.re(), c.im() + d.im());
}
```

It can be tested by calling a function like this:

```
out vector(number) test_complex() {
   complex a = new complex( 1.2, 3.4 );
   complex b = new complex( 5.6, 7.8 );

   complex c = a + b;
   return [c.re(), c.im()];
}
```

3.7.5 Type names

In Qlang several type names are defined. The type name is simply a different name for one of the already existing basic or object types, similar to using typedef in C/C++. An example of a type name is instrument_name, which really is a string used for finding an instrument in the database.

There are two purposes of type names: The first is to clarify the programming code; the second is that the graphical interface of Quantlab might recognize them and create control boxes appropriate for the input. Taking instrument_name again as an example, a control box for an instrument_name presents a list of all instruments in the database, thus making instrument selection easier.

3.7.6 Definition of types

In Quantlab 3.0 you can use the keyword typedef to rename existing types, as in C++. For example the number type can be called my_n:

```
typedef number my_n;

my_n j = 1;

out my_n test(){
       return j;
}
```

The user interface recognises your types as they are only other names for existing types.

3.7.7 Enum types

There are a number of enum types defined in Qlang. In earlier versions, they where only strings, now they are distinct types written with capital letters. A couple of examples are:

error_type: E_UNSPECIFIC, E_CONSTRAINT, E_NULL, E_RANGE, etc.

rate_type: RT_CONT, RT_SIMPLE, RT_EFFECTIVE, etc.

day_count_method: DC_ACT_365, DC_ACT_360, DC_30_360, etc.

bd_convention: BD_NONE, BD_FOLLOWING, BD_MOD_FOLLOWING, etc.

See the Function browser for more information on types.

Important news! In Quantlab workspaces, or lib files, created in version 2.4 or earlier, you must change the string enum type names to the new type names in order to compile the files.

3.7.8 Declaring your own enum types

In Quantlab there is a possibility to create own enum types. If used as an enum class, the prefix for the enum must always be used, otherwise there will be a hard compiler error. It is possible to create an enum without being a class as well.

```
class enum weather

{

        SUNNY option(str:"sunny weather"),

        CLOUDY option(str:"cloudy weather"),

        WINDY option(str: "windy weather")

};
```

To extract a string equivalent for each enum in a list you can use the expand_enum() function.

```
out vector(string) weather_types(){

        vector(weather) w;

        expand_enum(w);

        return string(w);

}
```

It is also possible to use a string as input and find its corresponding enum.

```
out string test_reverse(string myenum)
{
        weather w;
        str_to_enum(myenum,w);

        if (w == weather.SUNNY)
                return "Weather will be sunny!";
        else
                return "Sorry, no outdoors today.";
}
```

If “option(<string>)” is not given, the enum itself will also be used as a string equivalent.

3.7.9 Vectors and matrices

Creating vectors and matrices

Objects can be aggregated into vectors and matrices. The basic way of creating vectors or matrices is by using brackets:

Vectors are created using brackets and comma, *[element1, element2, ...]*.

Matrices are created from row vectors with brackets and comma, *[[element11, element12, ...], [element21, element22, ...], ...]*. All vectors must have the same size.

All elements in a matrix or vector must be of the same type. The type is declared within parentheses after the keyword vector or matrix. Here is an example of how to create a vector of three elements:

```
vector(number) v = [1, 2, 3];
```

It is possible to specify the dimension of the vector or matrix without assigning it:

```
matrix(number) m[3,7];
```

The matrix m will have three rows and seven columns. It is also possible to omit the dimension when declaring the matrix or vector:

```
matrix(number) m;
```

The matrix m will initially be null and have zero rows and columns and but this can be changed in runtime. Here is an example of how a vector can be declared and assigned:

```
out vector(number) vtest(){
       vector(number) v;
       v = [2,3];
       return v;
}
```

A common way of producing vectors or matrices is however by the use of *vector (or matrix) expansion*. This means that if you for example call a function with a vector rather than a scalar, Quantlab calculates a function value for each element in the vector. This also works for matrices. In the following example a function taking scalars is called with one scalar and one vector.

```
number my_add(number x, number y)
{
  return x + y;
}
out vector(number) f1(number x)
{
  vector(number) v = [1, 2, 3];
  return my_add(x, v);
}
```

When calling the function my_add with a vector in the second argument, the function will expand over the vector v. This means that the number x is added to each element in the vector, and the result is a vector that the function f1 returns.

Note! When using vector expansion, you must be sure that you call the function with a vector where the elements are of the same type as the argument type in the function you are calling. For example, if the argument is of the type date, then you must have a vector of dates as input.

Some functions naturally return a vector, for example the curve object member function instruments() that returns a vector of instruments.

Copying vectors and matrices

A direct assignment of one vector to another gives only an assignment of the reference, i.e. not the content of the vector. Therefore, the following example returns [1, 45, 3]:

```
out vector(number) utest(){
       vector(number) u, v;
       v = [1,2,3];
       u = v;
       v[1] = 45;
       return u;
}
```

To copy the content of the vector you could for example use a help function:

```
v_c(number v) = v
```

which uses the vector expansion to create a copy. There is also a built-in function clone_vector that gives a true copy of the vector.

Accessing and assigning elements in vectors and matrices

Elements in vectors and matrices can be accessed via indexation, using brackets. Indexations start at 0. For example, the following function returns the value 6.

```
out number my_vector_function(){
vector(number) x = [3, 5, 6];
return x[2];
}
```

For matrices, elements are accessed via row and column number, separated by comma. The following function takes out the value 5 from the matrix.

```
out number my_matrix_function() {
matrix(number) y = [[1, 3], [5, 6]];
return y[1,0];
}
```

(Remember that indexation starts at 0.) Assigning values to matrices and vectors is done in the same way. For example

```
y[1,0] = 77;
```

will set the element in the second row and first column of the matrix y to the value of 77.

Note! Since Quantlab 3.1 vectors can be called using the colon operator, such that x[2:4] will give the range from index 2 to 4 element, and x[2:] will give element 2 to end of vector. The same notation can be used for matrices.

It is also possible to replace the function index_vector() and one_vector() with a constructor directly on the vector type, see example below;

```
// ***********************************************************************
// A construct that replaces: one_vector(), index_vector(), range_vector()
// ***********************************************************************

// An index vector, from 1 to 10.
// replace "index_vector(10) + 1"
out vector(integer) v_ix() = vector(i:10 ; i+1);

// replace "one_vector(10) "
out vector(integer) v_ix2() = vector(i:10 ; 1);

// A 10 size string vector with empty string
out vector(string) empty_string_v() = vector(i:10 ; '');

// A 10 size date vector with only business dates
out vector(date) mydates() = vector(i:10 ;
calendar("SWEDEN").move_bus_days(today(), i));
```

Multiplication of matrices and vectors

When multiplying two vectors, the inner product is always used. When multiplying a matrix with a vector the number of columns or rows must be the same as the number of elements in the vector. If v is a vector and m is a matrix, then

```
v*m
```

will produce vector if the number of elements in v is the same as the number of rows in m, otherwise an error message will be shown. In the same manner,

```
m*v
```

will produce a vector only if the number of columns in m is the same as the number of elements in v.

Note that a vector in Qlang does not have a "direction"; there are no explicit column or row vectors. If you want to be explicit when handling column and row vectors, they must be declared as matrices. For example, the following two functions produce the same scalar result:

```
out number v_mult() {
  vector(number) v = [1, 3, 5];
  matrix(number) m = [[4, 5, 6], [2, 1, 7], [3, 5, 2]];
  return v*m*v;
}
out matrix(number) m_mult() {
  matrix(number) v_row = [[1, 3, 5]];
  matrix(number) m = [[4, 5, 6], [2, 1, 7], [3, 5, 2]];
  return v_row*m*transpose(v_row);
}
```

Note that in the first case the Qlang compiler will know that a scalar always will be returned, if the code could be run. In the second case the dimension of the result is dependent of the dimensions of the matrices, if they are changed. So the function has to be declared as a matrix.

For multiplication of two vectors elementwise, dot-notation can be used. Same dot-notation can be used for division but a “regular” division will return the same result, i.e. elementwise division.

```
vector(number) v = [1, 3, 5].*[2, 3, 5];
```

3.7.10 Series

The series is a special form of aggregate, based on one or more range objects. The ranges describe a multidimensional space, and to each point corresponds one element that can contain any object, vector or matrix. Each element must, however, contain the same type of object.

As opposed to a vector or a matrix, the series contains information about the range that has been used to produce the aggregate object. Therefore, series are very useful when dealing with historical time series data, or when producing graphs with equidistant values on the x-axis. For example, a series can contain a date range together with prices on an instrument for the date range. A series can be converted to a vector, but then the information about the range is lost.

In practice, a series is a compact way of making the familiar 'for-loop' construction, and keeping the information about the range. To construct a series you can either call a function with a series return type or use the keyword series. When defining a variable of any type of series or using a series as a return type you have to specify the series. it is done with the following syntax:

```
series<loop_type>(calculation_type)
```

where `loop_type` is the type of the loop variable, for example a number or a date, and `calculation_type` is the type of the values calculated, for example a number, an instrument, a vector(number) etc.

This is an example of a one-dimensional series:

```
number myHelpFunction(number x)
{
  return x * x;
}

out series<number>(number) myOutExpr()
{
  return series( i : 1, 10, 1; myHelpFunction(i) );
}
```

The first function takes one number argument and multiplies the number with itself. The second function calculates the content of a series. In the return type we have specified that the loop goes over numbers and the resulting values will be numbers as well. The first “arguments” in the series definition (before the semi-colon) are defining the range, as it states that the loop variable `i` will start at one and go to ten with step one. The last argument is the expression that is evaluated for each value of each loop variable. The return object will be a series of ten numbers: 1*1, 2*2, and so on. When the function myOutExpr is attached to a table you will see the range in the first column and the result from myHelpFunction in the second column.

The series function is often used to loop over time series data having dates as input or over numeric values for curve creation, in particular when creating graphs.

Important news! In Quantlab workspaces, or lib files, created in version 2.4 or earlier, you must change the series definitions. The type of the loop variable has to be specified and the range function has to be removed. Note also the use of semi-colon in the series definition. It is no longer possible to create multi-dimensional series of the type series<date><number> …

It is also possible to create a series where each element is a vector or matrix. One common way is to use the vector expansion, as in the following example:

```
series<number>(number) my_series(number x){
       return series(t: 1,10; x*t^2);
}
out series<number>(vector(number)) vector_series() {
       vector(number) v = [1, 4, 6];
       return my_series(v);
}
```

There only exists series of vectors, not vectors of series. But series of vectors can for example be efficiently applied when calculating time series dependent statistics for several financial instruments, stored in a vector.

It is possible to do vector algebra manipulations on a series of vectors, for example taking a scalar product:

```
out series<number>(number) series_prod(){
       series<number>(vector(number)) y = series(t: 1,10; [1, t, t^2]);
       vector(number) x = [1, 2, 3];
       return y*x;
}
```

A financial application of this could be to calculate the value of a portfolio; then y would contain daily prices for a number of assets and x the corresponding asset holdings (constant over time). Then the series_prod would give the daily value of the portfolio.

In general, vector manipulation, such as inner product or concatenation, can be done on two series of vectors, affecting each vector separately, for example:

```
out series<number>(number) series_prod2(){
       series<number>(vector(number)) a = series(t:0, 10; [t, t*t]);
       series<number>(vector(number)) b = series(t:0, 10; [2*t, 2*t*t]);
       return a*b;
}
```

where a vector of number is returned, or:

```
out series<number>(vector(number)) series_concat(){
       series<number>(vector(number)) a = series(t:0, 10; [t, t*t]);
       series<number>(vector(number)) b = series(t:0, 10; [2*t, 2*t*t]);
       return concat(a,b);
}
```

where a series of vector with four elements is returned.

It is also possible to retrieve particular elements from a series using brackets []. The index value within the brackets starts at zero for the first element and then increases by one for each element, independently of the range type. For example,

```
out number test(){
       series<number>(number) x = series(t: 5, 15; t^2);
       return x[0];
}
```

This function will return 25.

There is a possibility to expand over a series similar to vector or matrix expansion (3.7.7). For example you may write a function `f` that takes to scalars and call it by two series<number>(number):

```
number f(number x, number y) = x*y;

out series<number>(number) s(){

       series<number>(number) s1 = series(t: 1, 10; t^2);

       series<number>(number) s2 = series(t: 1, 10; t);

       return f(s1, s2);

}
```

The function `s` will return a series of number with the index range going from 1 to 10. This possibility can also be useful when you want to plot a scatter graph using two series of number. Then you can create a series of points:

```
out series<date>(point_number) scatter(instrument_name i_n1, instrument_name i_n2,
date from, date to){
  series<date>(number) s1 = series(t: from, to; instrument(i_n1, t).quote());
  series<date>(number) s2 = series(t: from, to; instrument(i_n2, t).quote());
  return point(s1, s2);
}
```

This function can be attached to a graph.

Some other examples of how to use series objects are found in 8.1, 8.8 and 8.10.

3.8 Flow control

Qlang supports common flow control structures: if/else, while, do/while, for and try/catch.

3.8.1 If, else, do and while

These structures work like in C/C++ (and many other languages), using a logical expression, called condition in the example below. The if statement has the following syntax:

```
if (condition){
  true_statements;
}
else{
  false_statements;
}
```

Alternatively, the else part can be conditional as well:

```
if (condition){
  statement;
}
else if(condition){
  statements;
}
```

The while statement is used for iterated calculations, depending on a condition:

```
while (condition){
  loop_if_true_statements;
}
```

The statement can also be executed before the conditional test:

```
do{
  loop_until_false_statements;
}
while (condition);
```

3.8.2 The for loop

The for loop has been changed in version 3.0 in order to be in line with C++. Thus the loop variable has to be explicitly defined (in or before the for-loop), and the start and stop criteria as well as the step can be defined more elaborately. Here is one example using a range from 0 to 10 with a step size of 2, and another example using a date range from 1 of March 2002 to 31 of March 2002.

```
for (number t = 0;t <= 10; t = t+2){
  loop_statements;
}
for (date t = #2002-03-01; t <= #2002-03-31; t++){
  loop_statements;
}
```

This means that if you use 0 as start and < as end condition the for loop will correspond naturally to the vector indices. Note that for loops in workspaces created in version 2.4 include the end point which corresponds to a <= end condition. Assume you have the following code in Quantlab 2.4:

```
out vector(number) for_24(){
        vector(number) x[10];
        for(i:0, v_size(x)-1)
                x[i] = i;
        return x;
}
```

This should in Quantlab 3.0 be changed to:

```
out vector(number) for_30(){
        vector(number) x[10];
        for(number i = 0; i<v_size(x); i++)
                x[i] = i;
        return x;
}
```

For your convenience, it is not necessary to change the for loop in Quantlab 3.0 as the old syntax is still valid. However, you are advised to make the change as the old style for loop may be obsolete in later versions.

Note! Since ver. 3.1.4029 range based for-loops are introduced.

```
out integer t(integer n){
        vector(integer) v = vector(i:n;i);
        integer s = 0;
        for(i:v) // reads "for each (index)value in vector v"
                s += i;

        return s;
}

// loop variable can be any type such as a date vector
out vector(string) t2(){
        calendar cal = calendar("SWEDEN");
        vector(date) mydates = vector(i:5;cal.move_bus_days(today(), i));
        vector(string) tmp;
        for(d:mydates){ //to be read "for each date in mydates"
                push_back(tmp, weekday_s(d));
        }
        return tmp;
}
// it is possible to loop over several ranges of equal length at the same time
out vector(string) t3(){
        vector(string) v1 = ["hello", "and","goodbye", "are", "we", "good"];
        vector(string) v2 = [":", "", "!", "%", "@", "?"];
        vector(string) ret1[v_size(v1)];

        //example with first element vector used by reference to original vector
        //setting the values of that as output
        //for each element in v1 and v2 fill element in ret i.e. in ret1

        for(out ret:ret1, i:v1, j:v2)
```

```
                ret = strcat(i,j);

        return ret1;
}
```

3.8.3 The switch statement

The switch statement is a substitute for nested if/then/else statements that compare a variable to several "integral" values (such as a number or an enum). The basic syntax is outlined below:

```
switch(<variable>)
{
        case first_value:
                < statement to execute when variable equals first_value> ;
                break ;
        case second_value:
                < statement to execute when variable equals second_value > ;
                break ;
        default:
                < statement to execute when variable does not equal any of the cases >
;
                break ;
}
```

Here is an example of how to use the switch statement.

```
out string spell_number(number n)
{
        switch(n)
        {
                case 1:
                        return("One") ;
                        break ;

                case 2:
                        return("Two") ;
                        break ;

                case 42:
                        return("Fortytwo") ;
                        break ;

                default:
                        return("Dunno") ;
                        break ;
        }
}
```

3.8.4 Error handling: Try and catch

Try and catch allow handling of errors that may occur. The syntax of the try-catch statement is the following:

```
try{
                < statement >;
        }
        catch(error_type1){
                < statement >;
        }
        catch(error_type2){
                < statement >;
        }
        <...>
```

```
catch{
              < statement >;
       }
```

The catch statement takes an error_type as input. To catch all types of errors, the catch statement can be written without parentheses and argument. The following error types are available in version 3.0:

TYPE NAME	Old name	Description
E_ABORTED	N/A	Aborted calculation.
E_CALC	'calc'	A calculation was unsuccessful, for example "Fit failed" in a zero-coupon estimation.
E_CONSTRAINT	'constraint'	Element-wise call using for example different sizes of vectors.
E_DATABASE	'database'	A database communication failure, for example an attempt to retrieve a quote in a quote field that is not defined for an instrument.
E_ENUM	'enum'	Invalid enum string, for example Invalid rate_type.
E_INIT	N/A	Not initialized object.
E_INVALID_ARG	'invalid_arg'	Invalid argument to a function.
E_IO	N/A	I/O error.
E_NAME_LOOKUP	'name_lookup'	Error in external name lookup, for example Unknown instrument.
E_NO_DATA	'no_data'	Data is missing, for example in a price quote.
E_NULL	'null'	An attempt to use a null value, for example as a condition in an if-statement.
E_PARSE	N/A	Parse error.
E_RANGE	'range'	Index out of range, for example in a vector.
E_REALTIME	N/A	Realtime feed error.
E_TIMEOUT	N/A	Time out error.
E_UNSPECIFIC	'unspecific'	All other errors.

The categories E_CALC and E_NO_DATA are "soft" errors, i.e., those that Quantlab handles and converts to null output values if the user does not handle them. The others are "hard" errors: if the user wants to ignore them, they must be taken care of in a try-catch statement.

Here is an example of how to use the try-catch statement.

```
out number f(number n) {
vector(number) a = [1, 2, 3];
       try{
              return a[n];
       }
       catch(E_RANGE){
              return 4711;
       }
       catch(E_INVALID_ARG){
```

```
			return 17;
		}
		catch{
			return 42;
		}
}
```

If n is between 0 and 2 the function will return the corresponding element of a. Depending of the type of error that may occur because of the argument n, the function returns other numbers instead (4711, 17 or 42).

Here is another example of how to use try and catch in combination with the throw() function:

```
out number f(number x){
	try{
		if (x == 1) throw(E_UNSPECIFIC, 'hello');
		if (x == 2) throw(E_RANGE, 'hi');
		else return x*10;
	}
	catch(E_RANGE) {
		return x*7;
	}
}
```

In this case we produce errors and throw them, depending of the value of x. If x is equal to two, the range error is caught and 14 is returned, but if x is equal to one, the unspecific error will appear in the warnings window with the text 'hello' and the function evaluation is terminated.

Sometimes it is useful to get hold of the error message. This can be done using a variable called "err" which is of the object type error. This variable is created by Quantlab when an error is produced and it is available within the catch statement. Here is an example of how it can be used:

```
out string test(number n){
	try{
		vector(string) x = ['Hello', 'Ciao', 'Salut'];
		return x[n];
	}
	catch(E_RANGE){
		return err.message();
	}
}
```

3.9 Comments

Comments are created with // at the beginning of the row or by using /* and */. Here are examples of the two possibilities.

```
// A one-line comment
/* A comment using
two lines */
```

Important news! The old style comment using % is no longer valid. Please use // instead.

3.10 Debugging

The Qlang Developer Editor enables the developer of Qlang code to debug one or several functions attached to graphs or tables. It is important to notice that what you actually debug is a selected attached expressions with the input parameters given by the user interface. Follow the procedure below to debug the code:

1. In the workspace window, go to Tools | Qlang Developer or press Ctrl + L. An independent Developer Environment will open.
2. To debug and step the code, press Ctrl + Shift + D or use menu Debug | Toggle Debugging in the Editor window.
3. From here, you can set breakpoints by putting the cursor at any row and pressing F9 or by clicking the left mouse-button in the left margin of the edit window. A red circle will appear to indicate a breakpoint.
4. In the workspace window, run the function (instance) that you want to debug. This will activate the code for the particular the graph or table.
5. Step in the code by using the function keys:
   Press F5 to continue to next breakpoint,
   Press F11 to step into each row of code,
   Press F10 to step over,
   Ctrl-F5 to finish debugging

When debugging you may inspect the call stack and the values of local and global variables by selecting View | Debug |Call stack or View | Debug | Variables. Vectors may be expanded by clicking the + sign in the list.

For more detailed information about the Editor its commands and features, please see separate document Quantlab Editor 3.1.xxxx.pdf.

4 Writing library files

All functions written in an expression window can only be accessed within the workspace. To create functions accessible to all workspaces you can write library files.

4.1 *Creating library functions*

In the Editor, select View | Library. This will open a sub window with corresponding to the paths in the ini-file which loads the library files. Right click in the library tree and use menu to add, select, delete and change any library files. A new folder can be created for library files at this point. This folder will be added to the path if it is not already there.

Write the following code in the expression window corresponding to the library file:

```
number my_lib_function(number x)
option (category: 'Test')
{
	return x*x;
}
```

The keyword option is in this case used for defining the category for the function. This means that a folder called Test will appear in the Function browser, containing the function my_lib_function.

Choose File | and any compile option, which will compile your library file together with all other library files. This can be done with or without saving the file to disk. Now you can open the Function browser and inspect your function.

TIP! If you do not wish to compile but rather just check the code for syntax errors, Ctrl + F7 can be used instead. This would not result in the loss of attached functions in the Workspace in case the compile fails, which is otherwise the case.

4.2 *Writing overloaded functions*

It is possible to define overloaded functions, i.e., functions with the same name as another function but having other parameter definitions. For example, you can define several functions with the same name and return type but with different types of a parameter. Or you can define several functions with different number of parameters. Of course, all overloaded functions must have the same return type.

4.3 *Adding member functions to object classes*

In library files you can write functions that are treated as member functions to existing Quantlab objects. This means that you can for example hook on your own valuation methods or you can add methods giving real-time or database data. We illustrate this with two examples.

4.3.1 Adding a valuation method to an instrument

In this example we will show how to add a function giving a number output to an instrument object. A member function is created by using the dot-notation that also is used when calling the function:

```
number instrument.my_price(instrument i, number param){
	number answer;
	// Some clever calculations…
```

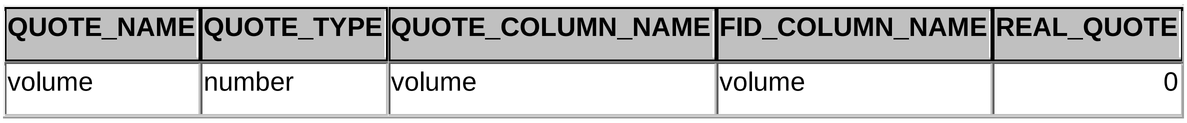
```
        return answer;
}
```

The first argument to the member function must be the object itself. After you have compiled the library file this function will appear in the member function list of the instrument object.

4.3.2 Adding a quote field method to an instrument

In this example we will show how to add a member function to an instrument that returns a turnover volume for that particular instrument.

First, you have to define the appropriate quote field in the database. See the manual for DatabaseTool for further information on this subject. Start by defining a new quote field by entering a new row in the table QuoteDef.



QUOTE_NAME	QUOTE_TYPE	QUOTE_COLUMN_NAME	FID_COLUMN_NAME	REAL_QUOTE
volume	number	volume	volume	0

The volume is of the basic Qlang type number and refers to a column in the Quote table that has to be defined, and which in this case is called volume. If you want to have real time data, you also have to define a new column in the table RealtimeLink called volume. There you write the FID number for the volume for each instrument. The last column is set to zero which means that this quote is not a "real quote", i.e., it cannot be used as a quote_side when pricing instruments.

Now you have to define a member function that retrieves this data. This is simple as you only have to call the member function get_quote_num which gives any numeric quote, given the name in the QuoteDef table:

```
number instrument.volume(instrument i){
        return i.get_quote_num('volume');
}
```

After you have compiled the library file this function will appear in the member function list of the instrument object.

The string parameter of the get_quote functions is not limited to the type quote_side.

If you try to access a quote_side that is not defined for an instrument, this will give a runtime error of the type 'database'.

5 Using the COM interface

This is a condensed version of the chapter with the same heading in the API manual "Quantlab API 3.1.xxxx.pdf".

All Qlang functions, including your own functions in library files and dll:s, can be exposed via COM to Visual Basic. The function definitions are generated by producing a tlb-file. To produce such a file from Quantlab, proceed as follows.

Save your library files in the path that you set in Tools Options, and restart Quantlab.

Choose Tools, Advanced, Generate Type Library.

To access the Qlang functions from VBA in MS Excel, start VBA and verify under Tools, References, that the Quantlab COM Library is in the list and is active. If it does not appear in the list you have to browse to the COM library file qlab31.tlb in your Quantlab folder.

In the object browser in VBA all Quantlab functions appear in the ql object with their function definitions. To get more help on each function, use the function browser in Quantlab.

See the manual for the Quantlab API for more information about the COM interface.

6 Using the Inter-Quantlab Communication Server - IQC

Some applications have the need of sharing information between them. There are many ways of solving such user interaction depending on the available it-infrastructure. A common method is by using a common database where users can read-and-write information. For some applications where data is of a streaming type with very frequent updates a more direct communication might be better.

Having an IQC server in place will create such a direct communication bridge between users of the Quantlab clients, regardless if they are using Quantlab through an Excel sheet or direct.

The IQC server will mimic a real-time source feed such as Reuters or Bloomberg. The difference is, of course, that you have to provide the IQC with your own streaming data coming from a Quantlab user within the community.

To get some feel for what the IQC can do we will look at two different examples. First we will create a chat room where Quantlab users can send and receive messages to and from a bulletin board. Secondly we will create a market data feed where a market maker can internally distribute some spreads for an illiquid bond pricer.

First we will look at how to install the IQC server

6.1 *Step-by-step installation of the IQC on the server*

The IQC server is only needed on one server/pc. All Quantlab clients can then communicate using the same server node.

Using the command line - go the folder containing the iqcs.exe programme.

Install the service using the following syntax:

```
C:\> iqcs –S service_name description [-p=port] [-f=state_file]
```

Go to the services window in the control panel and start the service.

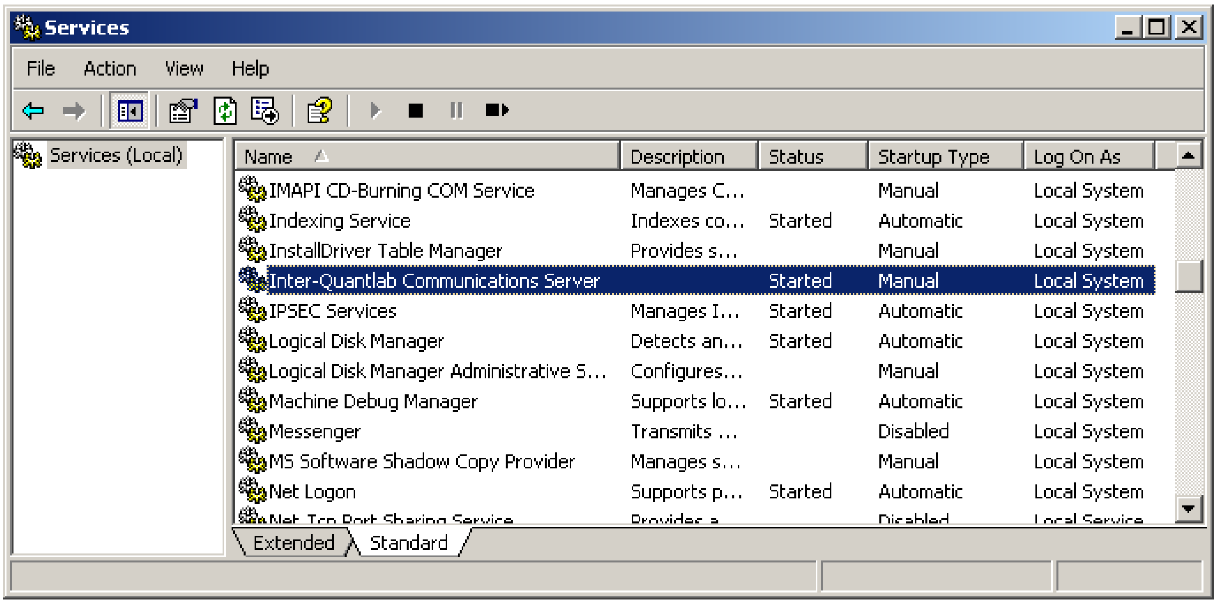
It is also possible to run the IQC server in non-service mode – in this fashion:

```
C:\> iqcs –s [-p=port] [-f=state_file]
```

It will then service requests until the process is terminated.

The IQC service can be un-installed with the –U command:

```
C:\> iqcs –U service_name
```



Name	Description	Status	Startup Type	Log On As
IMAPI CD-Burning COM Service	Manages C...		Manual	Local System
Indexing Service	Indexes co...	Started	Automatic	Local System
InstallDriver Table Manager	Provides s...		Manual	Local System
Inter-Quantlab Communications Server		Started	Manual	Local System
IPSEC Services	Manages I...	Started	Automatic	Local System
Logical Disk Manager	Detects an...	Started	Automatic	Local System
Logical Disk Manager Administrative S...	Configures...		Manual	Local System
Machine Debug Manager	Supports lo...	Started	Automatic	Local System
Messenger	Transmits ...		Disabled	Local System
MS Software Shadow Copy Provider	Manages s...		Manual	Local System
Net Logon	Supports p...	Started	Automatic	Local System
Net Tcp Port Sharing Service	Provides a...		Disabled	Local Service

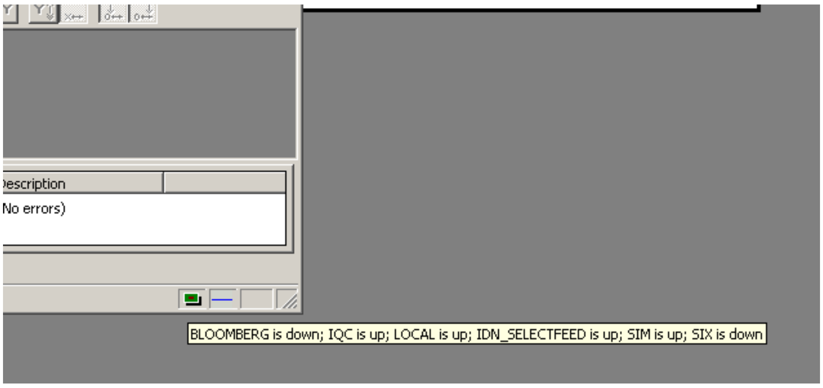
6.2 *Creating a connection to the IQC server from the Quantlab client*

In the same folder as the Quantlab.exe there should be a file called iqc24.qrt or iqc30.qrt depending on the version of Quantlab. This file is the local communication program that will give the user/programmer the function library used for reading and publishing information to the central IQC node.

In the qlab30.ini file the following tag will tell Quantlab that there is an additional real-time source available. In this example the IQC service was installed on a server called “qlbhill”. If the installation of the IQC service was on your local pc, this would be the name of your pc.

```
iqc {
                dll = 'iqc30.qrt'
                feed = 'IQC'
                server = 'qlbhill'
                port = ‘4711’
        }
```

Now we are ready to start Quantlab and see in the lower right hand corner (green icon) that the IQC is connected as a real-time source.


Description
(No errors)
BLOOMBERG is down; IQC is up; LOCAL is up; IDN_SELECTFEED is up; SIM is up; SIX is down

6.3 Example of creating a chat room using IQC

Let’s start with writing some code to publish rows to the chat.

```
out result send(string user, out string message)
{
   string result = message;
   if (!null(message) || message != "") {
       iqc_publish("IQC", "chat",
              [ "user", "time", "message" ],
              [ user, sub_string(str(now()), 11, 8), message ]);
       message = "";
   }
   return result;
}
```

We create a function that takes the name of the user and a message as input. The message we declare as an “out” parameter which means that it will be called by reference. In the user interface we can then clear the message box as we reference the message variable and set it to an empty string.

The iqc_publish function takes four input arguments;

the name of the feed (here “IQC”)

the name of the iqc identifier that will hold our information (here “chat”)

a vector or field identifiers for the different bits of information in the iqc identifier

a corresponding vector with data for each field in the identifier.

As information we send three strings to the “chat” iqc-identifier each time the function is called. This information will replace the old information that was last updated in the same way as the last price of a stock coming in the market data feed.

After we have published the user name, timestamp, and message we clear the message.

In order to keep track of the history of the chat we can now create a function that will subscribe to the iqc identifier and its fields and then store all incoming messages in a local vector.

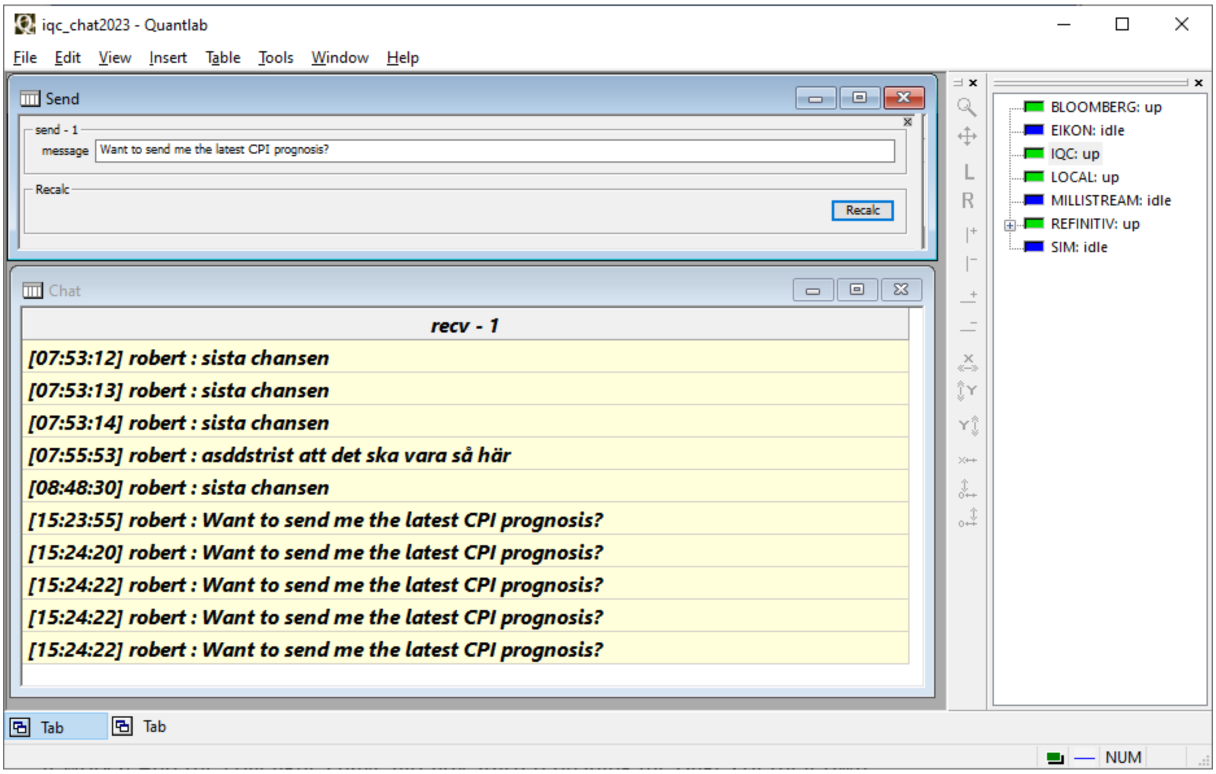
We do not need to ask the iqc server if there is any new information. The iqc server will push any new messages out to all clients that are connected and listening on a particular identifier. Again, in the same way as Quantlab would be triggered by a tick from a quote in the real-time feed from Reuters or Bloomberg.

```
vector(string) v_chat;
out vector(string) recv()
{
   push_back(v_chat,   strcat([   "[",   realtime_str("chat",   "time",   "IQC"),   "]
",rt.get("chat", "user", "IQC"), ": ", realtime_str("chat", "message", "IQC") ]));
   return v_chat;
}
```

First we have created a global variable (locally in the workspace) called v_chat. This vector of strings will hold all our received messages from the chat.

Second we create a function “recv()” that will execute at any time when the chat identifier has an updated data in it. The push_back function will just add another concatenated string into the v_chat variable. The function realtime_str() is a generic function that can be used to listen to realtime information streaming into Quantlab. It takes three arguments; the iqc identifier, the field identifier, and the name of the feed.

We can now attach both the send and receive functions to two tables in the user interface and we can start chatting.



send - 1

message: Want to send me the latest CPI prognosis?

Recalc

recv - 1
[07:53:12] robert : sista chansen
[07:53:13] robert : sista chansen
[07:53:14] robert : sista chansen
[07:55:53] robert : asddstrist att det ska vara så här
[08:48:30] robert : sista chansen
[15:23:55] robert : Want to send me the latest CPI prognosis?
[15:24:20] robert : Want to send me the latest CPI prognosis?
[15:24:22] robert : Want to send me the latest CPI prognosis?
[15:24:22] robert : Want to send me the latest CPI prognosis?
[15:24:22] robert : Want to send me the latest CPI prognosis?

BLOOMBERG: up
EIKON: idle
IQC: up
LOCAL: up
MILLISTREAM: idle
REFINITIV: up
SIM: idle

It works! And the colleagues have already started pushing messages of their own …

6.4 Example of feeding some market-maker corp spreads

We will create a mini-workspace with a table where the market maker can do manual input for three corporate bond spreads. Then we will create a user workspace that will price these bonds in terms of a base curve plus the spread published by the market maker.

```
vector(string) v_instr_name = ['CORP_BBB_1Y','CORP_BBB_5Y','CORP_BBB_10Y'];
out void publish_spread(out vector(number) v_spread) {
   for(i:0,v_size(v_spread)-1)
       iqc.publish('IQC', v_instr_name[i], ['mid'], str([v_spread[i]]));


}
```

Above is the code for the publishing part of the exercise. We place our three instrument names in a global variable. Then we create a function with an “out” vector as input argument. When the function is attached to a table the v_spread vector will be available for input by the user.

The loop will for each spread in the vector publish an iqc identifier and for each of these identifiers a mid quote.

In the second part we create some subscription code that will use the published spreads.

```
out vector(point_number) yields(curve_name base_c, date d, quote_side q){
    vector(number) v_maturity = [1,5,10];
    disc_func f = bootstrap(curve(base_c, d, q));
    vector(number) zero_yields = f.zero_rate(0,v_maturity,RT_CONT);
    vector(number)s = str_to_number(rt.get(v_instr_name, 'mid', 'IQC' ))/10000;
    return point(v_maturity,(zero_yields + s)*100) ;
}
```

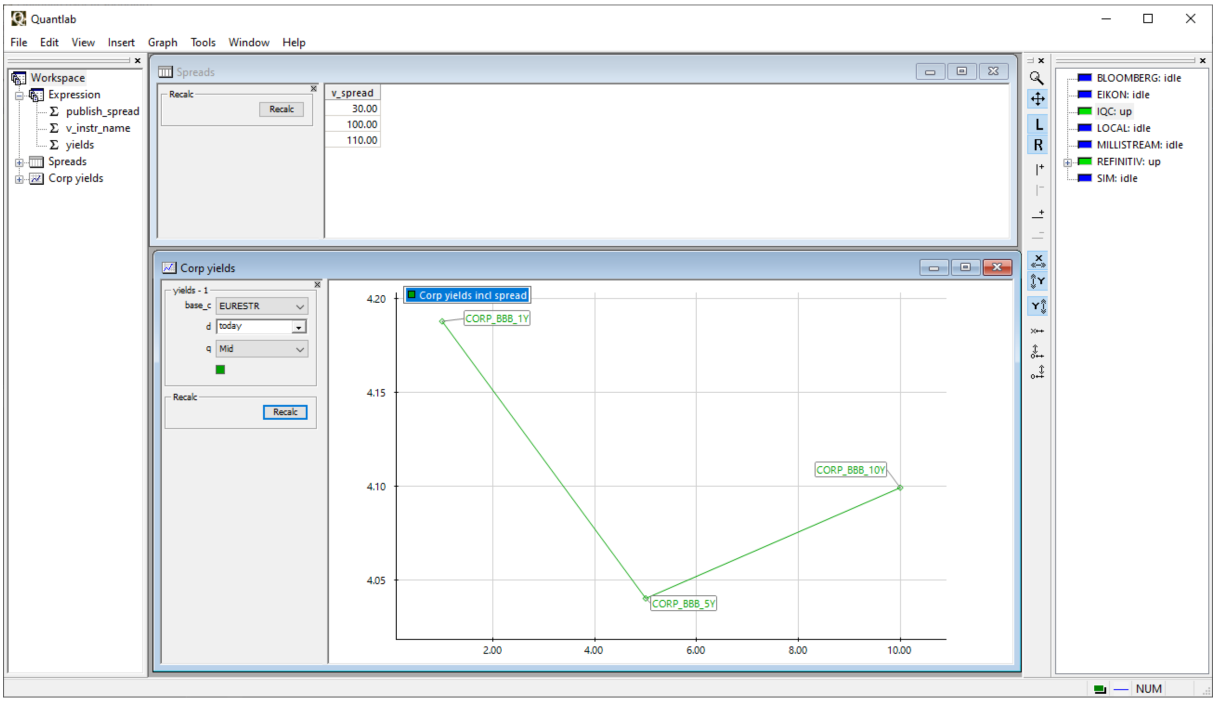
We have a function that will return a vector of points that we can attach to a graph. As input to our function we will allow the user to choose the base curve to price the bonds from. We will also allow the user to choose for which date to take the market quotes for the base curve as well as the quote side.

The base curve will be stripped from coupons to a zero coupon curve before we use it for pricing. We have chosen the bootstrap method. From the fitted curve we extract the zero yields for the maturities of our corporate bonds.

We then subscribe to the published corporate bond spreads using the realtime_str() function and divide the basis points with 10000.

It is now easy to return the three bonds zero yields as the sum of the base curve and the spreads.

Let’s look at the workspace when we have attached the functions to a table and a graph.



v_spread
30.00
100.00
110.00

base_c EURESTR
d today
q Mid

For every time the market maker updates any of the spreads in the spread vector in his Quantlab workspace, the pricing will immediately be pushed to all other users listening to these IQC identifiers.

7 Output - tables and graphics

Quantlab is designed for both the developer and the end-user of the analytics. To display analytics in a pedagogic yet comprehensive way, Quantlab have three different "display objects" to choose from. The expressions can be displayed in a graphical window and/or in a table. There are two table types, a general-purpose table where any scalar, vector or matrix can be displayed and a special-purpose table for instrument display.

7.1 *General purpose table*

A general-purpose table can be created on the menu Insert | Table or by pressing Ctrl T. This table type will display any scalar, vector or matrix expression. Multiple expressions can be attached to the table.

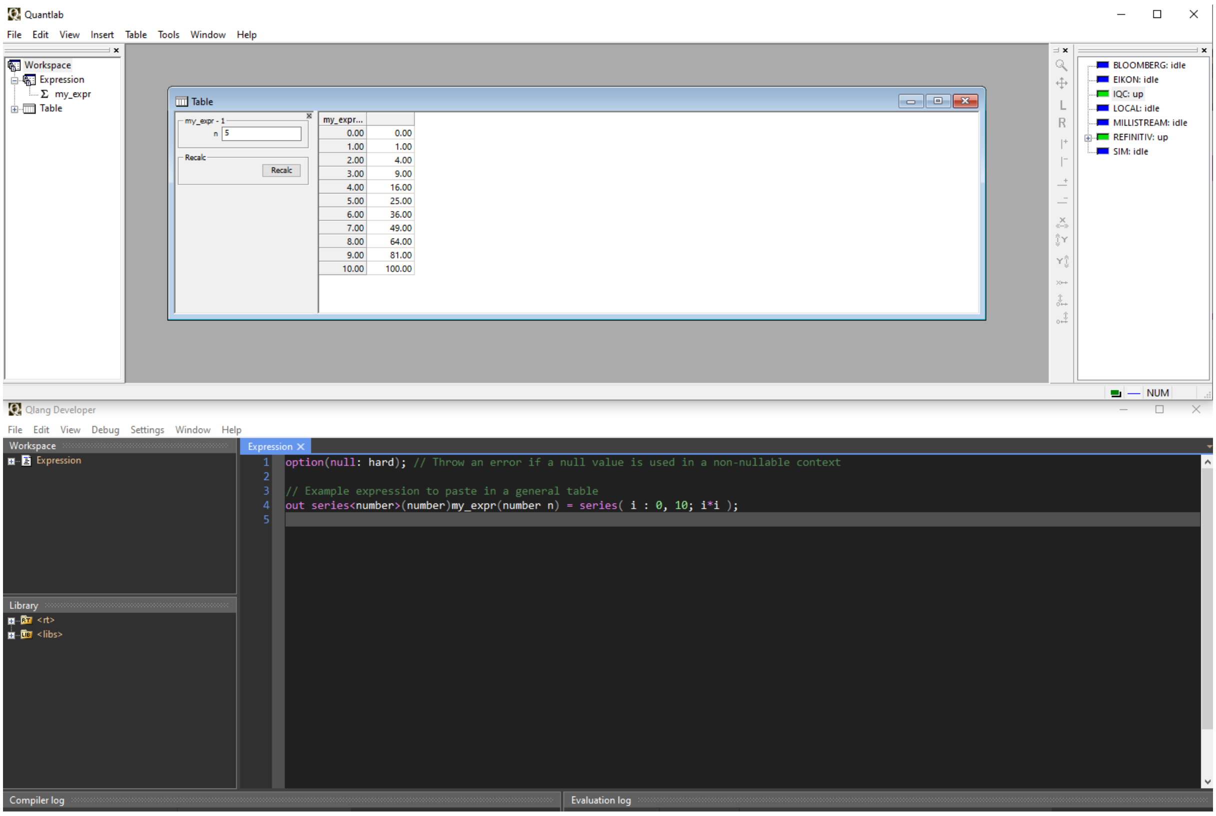
7.1.1 Attaching an expression to a table

Let's look at a simple example. Open a new workspace by the menu File | New workspace and insert an expression window by the menu Insert | Expression. Then type the following:

```
// Example expression to paste in a general table
out series<number>(number)my_expr(number n) = series( i : 0, 10; i*i );
```

Compile this expression in the Editor using the menu File | Compile, or press F7. If you have the workspace browser open (View | Workspace browser) then you will see a + sign to the left of the Expression window. Click the + sign and you will see the symbol for the function my_expr. Now, insert a table using the menu Insert | Table. Drag the expression my_expr from the workspace browser (pressing the left mouse button) and drop it on the table.

The result should now look something like this.



my_expr...	
0.00	0.00
1.00	1.00
2.00	4.00
3.00	9.00
4.00	16.00
5.00	25.00
6.00	36.00
7.00	49.00
8.00	64.00
9.00	81.00
10.00	100.00

BLOOMBERG: idle
EIKON: idle
IQC: up
LOCAL: idle
MILLISTREAM: idle
REFINITIV: up
SIM: idle

```
option(null: hard); // Throw an error if a null value is used in a non-nullable context

// Example expression to paste in a general table
out series<number>(number)my_expr(number n) = series( i : 0, 10; i*i );

```

Example workspace with the my_expr function

You can now see the attached expression "my_expr – 1" in the workspace browser by clicking the + sign to the left of the table symbol. The number behind the expression name will help in keeping track of which instance of the expression you are working with.

The table will now display an input box for the user to input a value for the parameter n. A table will recalculate when the return key is pressed, or by pressing the "Recalc" button. If the table includes any real time data in the expression the table will update on any changed data.

When attaching expressions to tables or graphs Quantlab always creates auto-generated controls which correspond to the types of the parameters. If you want several parameters to be determined by the same control you can use the Parameters Options dialog, as described in 7.4.4.

7.1.2 Table options and formatting

Parameter canvas

Clicking on the close x, in the upper right hand corner, hides the parameter canvas displayed in the table. The same function will show by right clicking on the canvas and using the menu choice Hide Parameters.

Formatting the table

By right clicking on the header a number of formatting options are available:

Format Attachment | Color and border	Change colour settings and border style
Format Attachment | Font	Change font size and style
Format Attachment | Number	Change the number formatting of the selected column
Format Attachment | Text alignment	Change the horizontal alignment of any text in the cells
Auto format	Let the font be dependent on the value in the cell (only for numerical attachments). See 7.1.4.
Column order	Change order of presentation when multiple functions are used
Minimal frames	Will display table with minimal frame
Display name	Change the header name of the column. Dynamic header variables reflecting the current parameter setting can be inserted by double-clicking on the parameter.
Rename table	Change the name of the table
Holiday	Set a holiday calendar for the table. For expressions having date ranges the relevant holidays will be suppressed in the table.
Hide/show parameters	Switch the parameter view on and off
Duplicate	Will make a copy of the table.

By right clicking on a specific cell, or multiple selections of cells, the same formatting options are available by choosing Format Cell. Cell(s) are available for formatting when the selection turns black.

Changing attachment order

The attachment order can also be changed using drag and drop of columns. Put the curser on a column header, press Ctrl and use the left mouse-button to drag the column to the desired location.

Merging parameters

The parameters that are needed in order to evaluate functions attached to tables can be merged to common controls. This is done in the same way as for graphs, see 7.3.2.

Transposing the table

The table can be transposed so that columns and rows change places. Right click on the table and choose Transpose.

7.1.3 Vector parameters in general tables

When several input values of the same type are needed for a calculation it is convenient to use a vector parameter. When attaching an expression containing one or several vector parameters they will show up within the table. By clicking the right mouse-button on the column head it is possible to set or change the number of rows in the vector, by choosing "Input parameters|Set rows". If you know that all input vectors will be of the same length, you may use the choice "Set entire row". Note that sometimes it may be necessary to check the size of the input vectors in the code.

Vector parameters can be merged within a table as other parameters. They appear in the Parameters options dialog in a separate attachment symbol.

For some examples of how to use input parameters, see the case studies in 8.6 and 8.7.

7.1.4 Automatic text formatting

In financial applications it is often useful to format the text (or numbers) in a table depending on the values shown, or other parameters. By right-clicking the column header of an attachment you can choose Autoformat. This gives the possibility to set up simple rules that changes the font and/or background colour depending on the number in each cell.

For more complicated situations there is a possibility to set the font and background colour from the code. The background colour can also be transient in order to flag for a change. For this purpose we have created the object text_rgb that can be attached to a table. This object is created by calling the function with the same name:

```
        text_rgb(text, [fg], [bg], [transient_bg])
```

where text is the text as a string, fg is the font (foreground) colour, bg is the backround colour and transient_bg is a logical parameter that makes the background colour transient if true. The colours are given as RGB-triplets and can easily be defined using the function rgb(), for example:

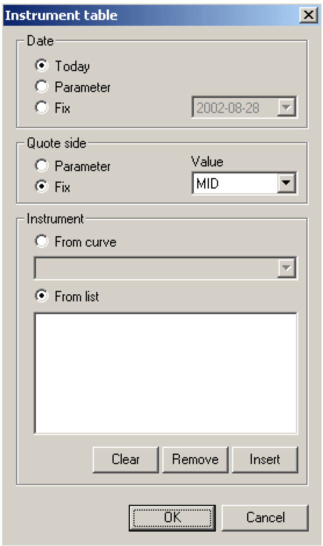
rgb(255,0,0)	(Red)
rgb(0,255,0)	(Green)
rgb(0,0,255)	(Blue)
rgb(0,0,0)	(Black)

For an extensive example of the use of auto-formatting, see 8.13.

7.2 The instrument table

Instrument tables are possible to use in Quantlab 3.0 but are not recommended as the effect of an instrument table can be obtained through an ordinary table.

The instrument table is specially created for displaying lists of instruments and other input/output that refers to these instruments. The instrument table is itself defined by a vector of instruments and a date for which any evaluation should be done. Any function or method that can be used on an instrument can then be displayed in the table. The user can for certain settings change the instrument vector and evaluation date. There is also the possibility to define the instrument vector through a function, see below.


Instrument table

Date
● Today
○ Parameter
○ Fix 2002-08-28

Quote side
○ Parameter
● Fix
Value MID

Instrument
○ From curve
● From list

7.2.1 A standard instrument table

We will describe the standard instrument table using an example: We want to display a list of bonds and their current prices and durations.

Insert a new instrument table from the menu Insert | Instrument table.A "wizard" will ask for which date and which instruments to initially display. Here is shown the left side of this dialog.

In the date option, only the parameter choice will give the user of the table a possibility to change the date again. The other two options will fix the date for the table permanently.

Either you can let the quote side be a parameter for the user to change, or you can use the fix value (which is defaulted to the value under Tools Options.

If your Quantlab already is prepared with curves, choosing a complete curve is obviously convenient. If not, instruments can be chosen from the list of available instruments.

Tip! If you have no curves defined in your Quantlab database, it is easily done from the Database tool that was shipped with your Quantlab installation. See separate manual for further instructions.

The table is now created with your initial choice of date and instrument selection. User controls have also been added to the table. Next step will be to add our needed information *about* the instruments chosen.

Now to some simple programming[1]: We will need to create an expression having three instrument methods in order to get the name, price, and duration. For more functions see the function index.

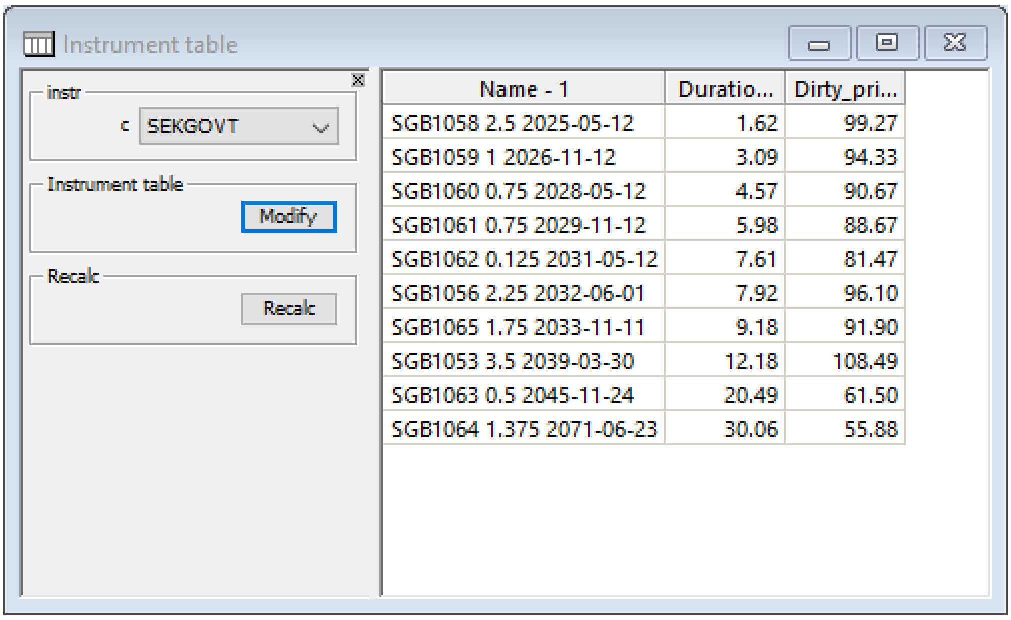
```
// Example of some functions that apply for instruments
out vector(instrument_name) Name (vector (instrument) i) = i.name();
out vector(number) Dirty_price (vector (instrument) i) = i.dirty_price();
out vector(number) Duration (vector (instrument) i) = i.mac_dur();
```

Note! Any code written to display functions in an instrument table must have the vector of instrument as the last argument.

---

[1] See chapter 3 for more information about programming details. In this example we have created functions that take a vector of instruments as input argument. Since the instrument table at creation uses an instrument vector as base, the methods will be evaluated for all instruments in the vector.

Compile (press F7) and confirm that the expressions appear in the workspace browser. Now it is simple to drag-and-drop the instrument information to the instrument table. The result should look something like in the following picture.



c SEKGOVT

Name - 1	Duratio...	Dirty_pri...
SGB1058 2.5 2025-05-12	1.62	99.27
SGB1059 1 2026-11-12	3.09	94.33
SGB1060 0.75 2028-05-12	4.57	90.67
SGB1061 0.75 2029-11-12	5.98	88.67
SGB1062 0.125 2031-05-12	7.61	81.47
SGB1056 2.25 2032-06-01	7.92	96.10
SGB1065 1.75 2033-11-11	9.18	91.90
SGB1053 3.5 2039-03-30	12.18	108.49
SGB1063 0.5 2045-11-24	20.49	61.50
SGB1064 1.375 2071-06-23	30.06	55.88

Example of an instrument table with three columns (showing Swedish government bonds)

One of the special features of the instrument table is that it will automatically evaluate any expression over all the instruments in the chosen vector. A general-purpose table cannot do this.

7.2.2 Formatting the instrument table

By right clicking on the header a number of formatting options are available:

Sort	Will sort the table according to the chosen column
Format Attachment | Color and border	Change colour settings and border style
Format Attachment | Font	Change font size and style
Format Attachment | Number	Change the number formatting of the selected column
Format Attachment | Text alignment	Change the horizontal alignment of any text in the cells
Auto format	Let the font be dependent on the value in the cell (only for numerical attachments)
Column order	Change order of presentation when multiple functions are used
Minimal frames	Will display table with minimal frame
Display name	Change the header name of the column. Dynamic header variables reflecting the current parameter setting can be inserted by double-clicking on the parameter.

Rename table	Change the name of the table
Holiday	Set a holiday calendar for the table. For expressions having date ranges the relevant holidays will be suppressed in the table.
Hide/show parameters	Switch the parameter view on and off
Duplicate	Will make a copy of the table.

By right clicking on a specific cell, or multiple selections of cells, the corresponding formatting options are available by choosing Format Cell. You can select multiple columns or cells by using the left mouse button. Cells or column headers are available for formatting when the selection turns black.

The attachment order can also be changed using drag and drop of columns. Put the curser on a column header, press Ctrl and use the left mouse-button to drag the column to the desired location.

Note! The parameters that are needed in order to evaluate functions attached to tables can be merged to common controls. This is done in the same way as for graphs, see 7.3.2.

7.2.3 Creating instrument tables using an instrument vector function

It is possible to create instrument tables based on a vector of instrument that is defined through a function. If you write a function returning a vector of instruments, for example,

```
out vector(instrument) my_v_i(curve_name c_n, date d){
	return curve(c_n, d).instruments();
}
```

this function will be possible to select in the right hand side of the Modify-dialog of the instrument table. Then the instrument table will iterate over the vector that this function delivers.

The instrument vector function will always be calculated before any other function is evaluated in the table, see 7.5.3. And the evaluation of an instance of this function causes the evaluation of all other functions in the instrument table, as they are dependant on the instrument vector. This is a typical case when dealing with real-time data. It the date d in the function above is chosen to be today’s date, then by default the instrument vector will be updated with real-time updates on the quotes of the instruments. Each time any update comes to any instrument the whole vector will be calculated and then all other functions in the instrument table. Thus the whole table is connected to real time updates only by the instrument vector. No other explicit real-time update is necessary.

This type of instrument table can of course be used when you want to do a more sophisticated filtering of the instruments than just using curves and dates.

7.3 The graph window

For many users the graph window is the most popular way to analyse financial data. In order for a graph to reveal as much information as possible in a limited space a number of special features have been added to Quantlab's graphing capabilities.

To create a new graph use the menu Insert | Graph or use the Ctrl-G command.

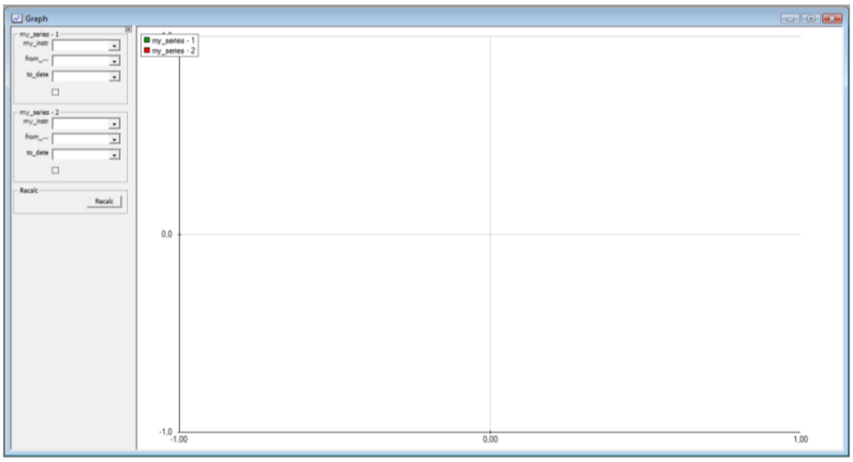
7.3.1 An example of a time series graph

In order to list features of the graph component let's go through a simple example first. We want to plot two time series on the right and left hand y-axis.

First create the expression having three user parameters instrument name, from date and to date:

```
out series<date>(number) my_series(instrument_name my_instr, date from_date, date
to_date){
       return series(d : from_date, to_date; instrument(my_instr,d).yield());
}
```

Since we use parameters in the my_series function, we can re-use this expression for many instances of the same expression. Compile the expression and drag two instances of the same expression to the same graph. The graph window should look like this.



Example of graph with two instances of the mySeries expression attached

As we have not yet given the controls any parameter values the graph is still empty. Before we start to use the graph we will use some of the more common graph formatting features available.

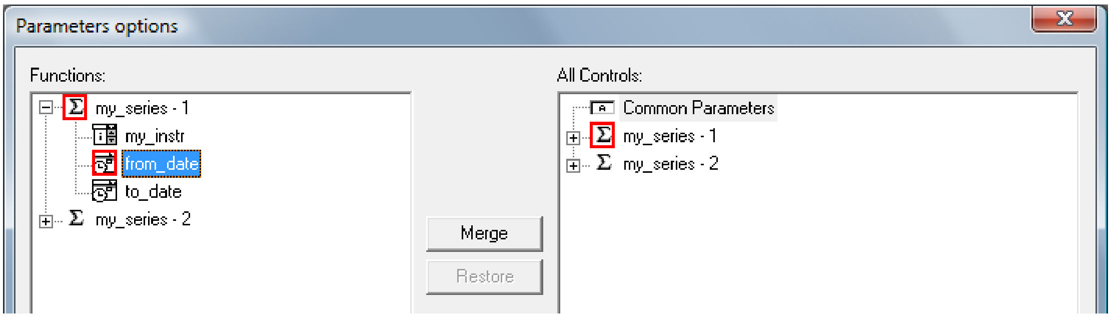
7.3.2 An example of how to merge parameters to common controls

In this example we will always want to have the same from and to date for both instruments, so next step will be to use the merge control function.

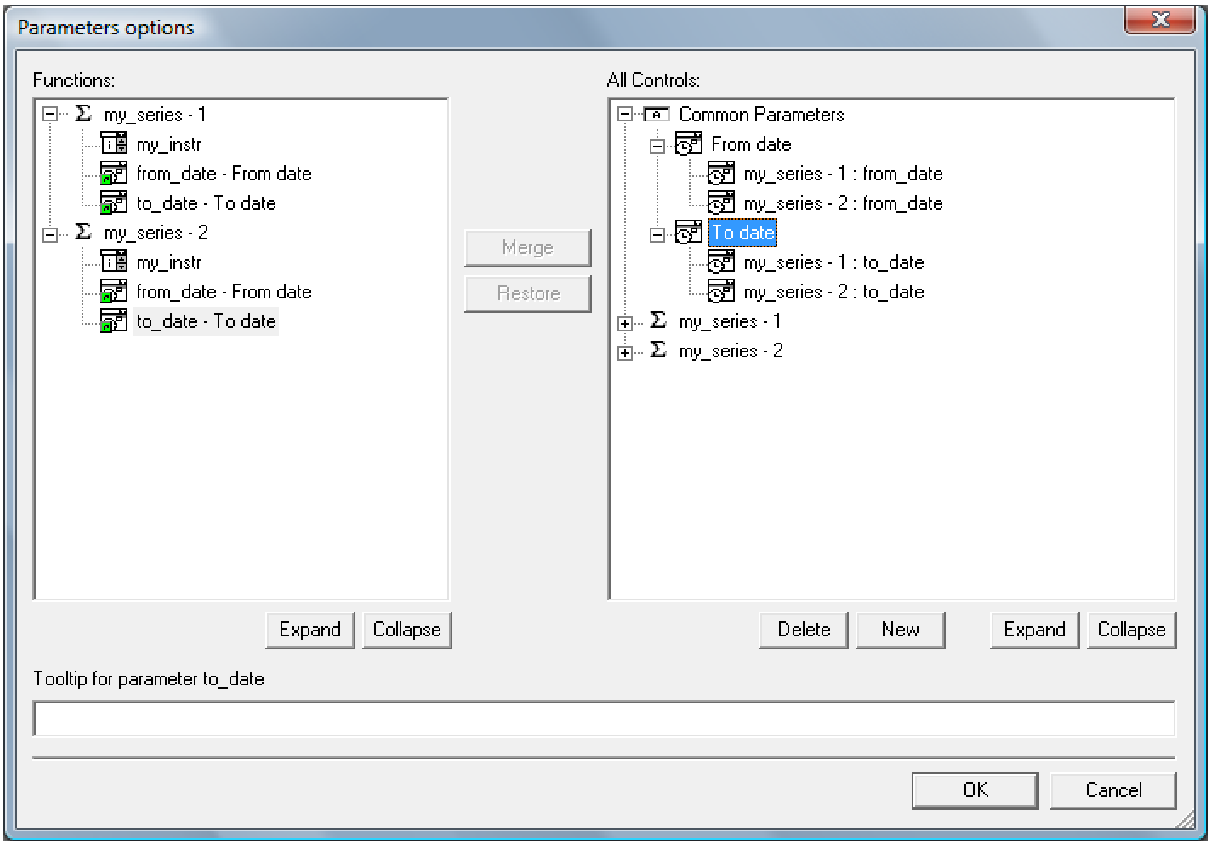
You find the dialog by right clicking on the canvas and choosing "Parameter options" or through the menu Graph | Parameter options.

On the left part of the dialog we find our functions with their parameters, on the right hand side all the auto-generated controls are shown. In the top of the right panel there is an empty group for common controls.

1. Start by choosing the first instance of the my_series function. The available parameters appear in the left list, if you click on the + sign to the left of the expression symbol Σ. Here we want to merge the date parameters from the two functions.

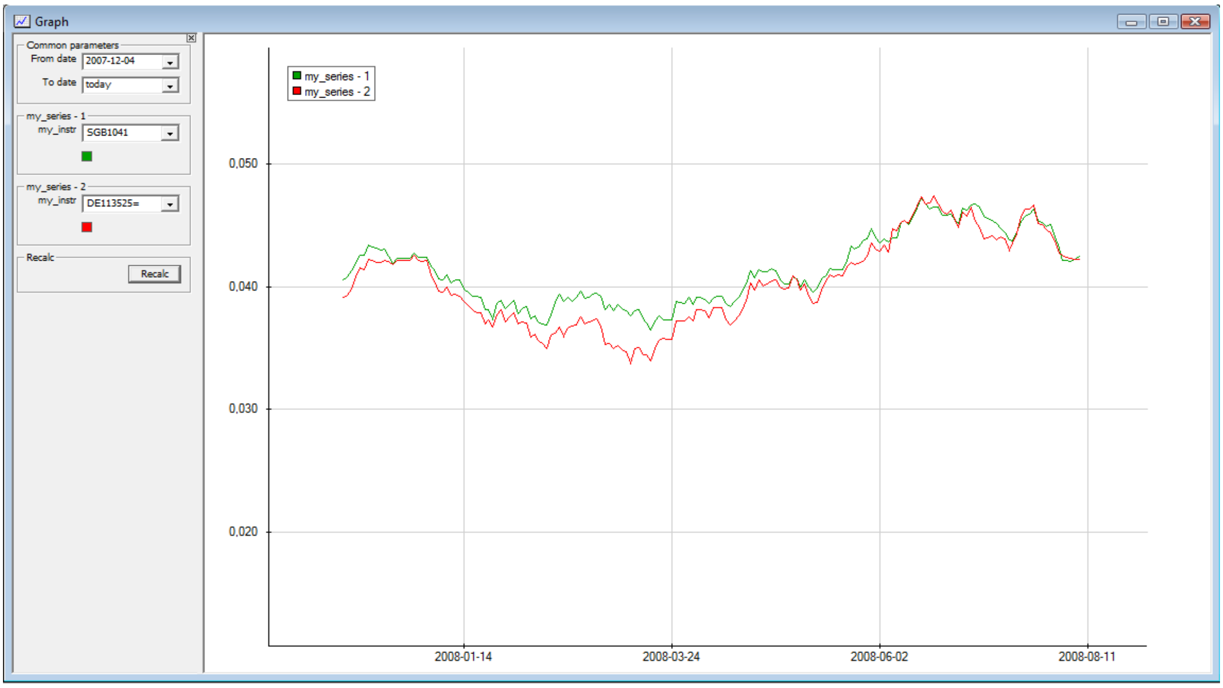


2. Grab the from_date parameter and drop it in the right hand list panel on the Common Parameters group. This creates a new common Date control and removes the corresponding auto-generated control further below. Give the common control a descriptive name, for example "From date". In the same fashion, drag the to_date to the right and rename it.
3. In the function list on the left click the + sign to the left of the second instance of the function mySeries. This function's parameters are now displayed below.
4. Drag-and-drop the second function's fromDate and toDate on the corresponding common controls on the right.
5. You have now merged the date controls for this graph. If correctly done, it should look something like in the picture below.



Example of merging date controls together

So how does the graph look like now, having chosen some instruments and dates to display?



From date: 2007-12-04
To date: today

my_series - 1
my_instr: SGB1041 🟩

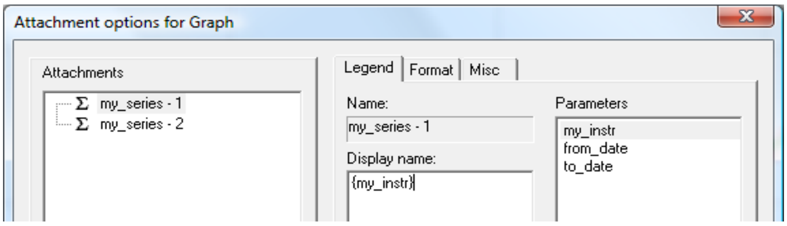
my_series - 2
my_instr: DE113525= 🟥

Example of graph having common date controls

The common date controls now drive both my_series functions. "Today" is the key word used in a date control to always getting today's date after you save and re-open a workspace. Setting today's date also implies getting quotes in real time if connected to a real time source.

Tip! You can use the merge functionality in order to give the parameter control a user-defined label text, other than the function name that is the default. In this case you only merge one parameter to each control.

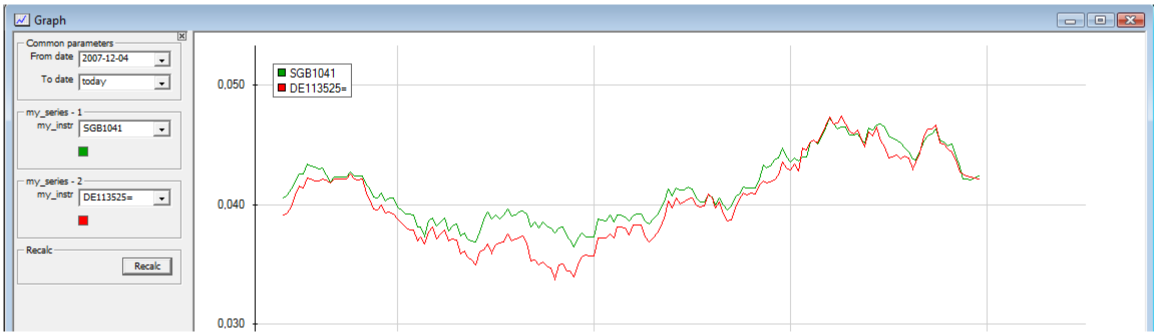
Now, we would like to have a more descriptive legend text than “my_series - 1”. Double-click at the graph to get the dialogue Attachment Options for Graph. Choose the Legend tab and delete the default legend text my_series - 1. Then double-click at the my_instr parameter to the right. It appears within {} signs.



Name: my_series - 1
Display name: {my_instr}

Editing the legend text.

Then do the same with my_series - 2. The effect is that the legend text will be dependant of the choice of instrument.



Common parameters
From date 2007-12-04
To date today

my_series - 1
my_instr SGB1041
🟩

my_series - 2
my_instr DE113525=
🟥

Recalc
Recalc

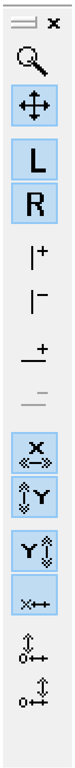
The graph with edited legend text.

Of course, you can also type a constant string into the same dialogue, or combine strings with parameters.

Tip! It is possible to merge parameters on a separate canvas/window that will contain all graphs and tables' parameters attached to a specific tab. This option can be found under View | Show tab parameters or by pressing Alt +5. For more on controlling an entire tab's parameter space see section 7.4.5 about "merging parameter in a tab".

7.3.3 Using the graph mode toolbar

When a graph window is active the graph mode toolbar can be found under menu View | Graph mode or by pressing Alt + 5.



The buttons guide in which way the mouse interacts with the graph. By default, holding down the left mouse button over the graph will move the centre left and right.

Zoom in the graph by switching to the magnifying glass and creating an area to zoom in on by holding down the left mouse button.

Enable and disable zooming functions for the left and right y-axis by depressing the L and R button.

To display a value cursor in the graph, enable the line button |$^{+}$. This will show y- and x-axis values in the legend box while you move the value cursor left and right in the graph. You can insert several value cursors by clicking this button repeatedly. To remove the value cursors, use the button marked with |$^{-}$.

To insert a horizontal line use the button with the symbol —$^{+}$ and to remove those lines use the button with the symbol —$^{-}$.

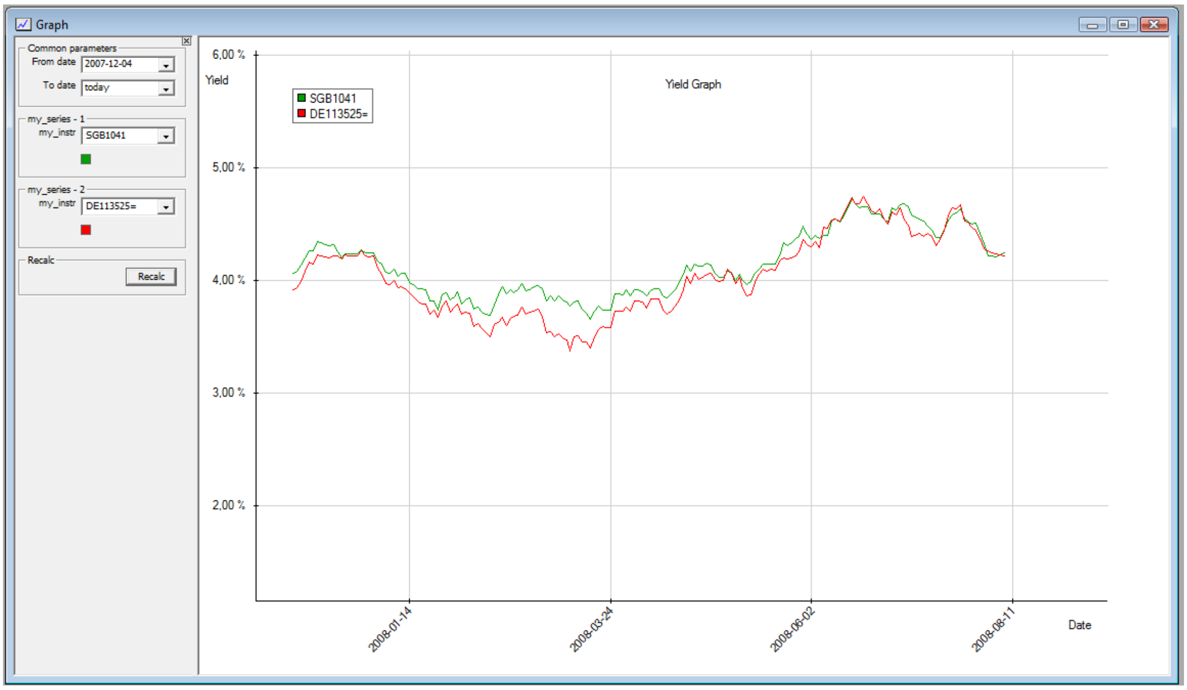
When changing any parameters used by the graph or when updates come from the real time feed an auto-zoom function is available. With the auto-zoom turned on it will refocus the graph on every update that changes position or size of the displayed graphics. It is possible to turn on the auto-zoom for each axis separately by pressing down the relevant axis button with double arrows.

Further auto-zoom features admit the user to always show the x-axis at the bottom of the graph rather than at zero, and always show zero level on the left and right y-axis when re-zooming. (The last three buttons on the lower row of the control these features.)

Continuing our example from 7.3.1 and using the following formatting options …

1. Right click on the left axis to get the dialogue Graph properties and choose Multiply by 100 to the right in the Character pane. Also, choose 2 digits for decimal places and % as symbol. Press the Home button on the keyboard to get auto-scale.
2. Right click on the left axis to get the dialogue Graph properties and tilt the dates by choosing a 30 degrees slant.
3. Right click on the background of the graph window and choose Graph properties | Titles. Write appropriate titles for the graph and the axis. After pressing OK, the titles can be dragged and dropped at the ends of the axis.
4. Right click on the background of the graph window and choose Graph properties | Holiday and check that Hide weekends are clicked. Then you can also click at Sweden and Germany (for this example where we have a Swedish and a German bond) in order to hide all days that are holidays in any of the two countries.

…you will get a graph similar to this one:



Common parameters
From date 2007-12-04
To date today
my_series - 1
my_instr SGB1041
my_series - 2
my_instr DE113525=

Note! Any residual holes remaining in the graph, after proper holiday calendar(s) are chosen, are due to missing data in the historical database. Use your data cleansing tools to repair this missing data. See manual for the Database tool for help on finding missing data.

7.3.4 Graph formatting options

Right click on data series line (the graph) to:

Change the Order - by selecting a choice in the sub-menu you can change the order for how the graphs are displayed if you have attached several expressions to the same graph window. The legend text that corresponds to the last painted graph (in the front) is the last one in the legend text box.

Snap labels – see special chapter about creating labels (not valid for time series graphs)

Linear regression – to display a linear regression line for the time series

Copy – copy the underlying data from the selected graph making it available, for example, for an Excel spreadsheet.

Copy legend – copy the legend text

Properties – see table below

Properties detail

	Functionality	**Description**
Legend	Automatic legend showing current value of parameters	In the display name text box free legend text can be written. Any parameters used in the graph can be attached to the legend by double clicking on the parameter in parameters list. A parameter is inserted using {} brackets. Example: MyGraph showing {myInstr} from date {fromDate} to {toDate}

Format	Graph type	Allows for line, column or points.
	Point type	Options include plus (+), diamond (◇), circle (o), square (□) filled or not filled.
	Color, font, width	For vector of lines, a colour scheme can be set.
Misc	Right/left axis	Choose to place the graph on the right or left y-axis
	Regression	Change the colour and width of the regression line
	Optimise	Will give a smoother appearance when zooming in and out.

Right clicking in the graph space reveals:

Attach/Detach		Attach and detach any expression from the graph
Show/hide curves		If many curves are attached to one graph it is possible to hide one or more curves temporarily without detaching them from the graph window.
Graph properties	Holiday	To choose which holiday calendars that the graph should handle. If a market is chosen, the dates set as holidays will not show in the graph. Multiple choices are valid. Also weekends can be turned on/off.
	Titles	To edit the main graph title, the y-axis and x-axis title. Font, size, and colour can also be set.
	Scale	As default, the scaling is automatic and will follow the zooming. It is also possible to manually set the min and max scaling for each of the axis and also lock the scale.
	Misc	To change the column width when displaying bar chart style. Changing font, size, and colour of the legend. Formatting the Value cursor(s) settings.
	Axis	Change the date format, character display, line format for the chosen axis. Same dialog will show when double clicking directly on any axis (see description below).
Parameter options		Display the merge function dialog (see separate description)
Minimal frames		Will minimize the window frame of the graph (or table).
Rename		To rename the current graph
Show/Hide parameters		To show and hide the parameter canvas
Duplicate		Will create a copy of the whole graph including parameters and format settings. A reference to the copy will also appear in the workspace browser.

Right- or double clicking on the right or left y-axis and x-axis:

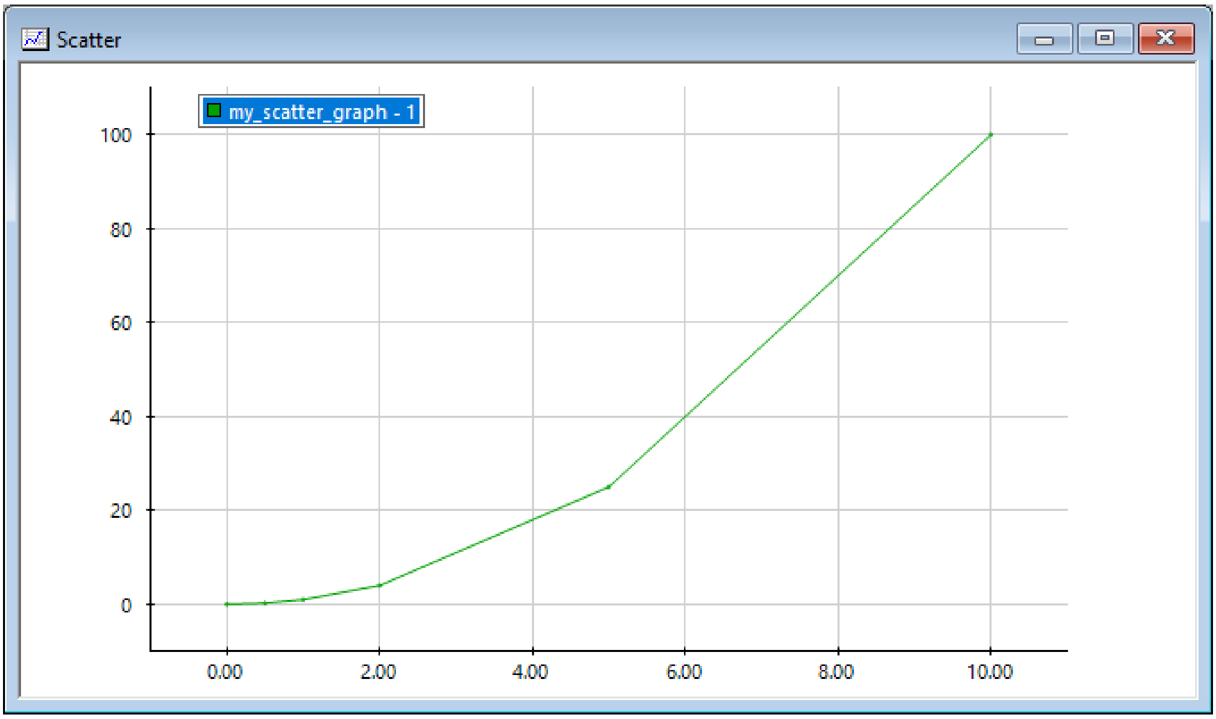
Text angle	Edit the slant of the text
Font, size, and colour	Change font, size, and colour of the x or y-axis
Line width	Change the line thickness of the x or y-axis
Symbol	Place a symbol or other text behind the numbers (ex. '5.0 %' or '5 Kr')
Date format	Use default setting or format display using an interactive wizard

Tip! By pressing the ***home button*** on your keyboard the graph will automatically fit and centre the graph. This feature can be set on automatic by pressing the buttons in the graph toolbar. Holding down the ***shift button and the left mouse button*** will zoom the graph when the mouse is moved.

7.3.5 Scatter graphs

For graphs where data don't come in the form of a series the point function is useful. This function returns a point object, which consists of the x- and y-coordinate for a point in a graph. A vector of such object can be used for producing a scatter graph. For example, to plot a square function on some non-equidistant x-values you may use the following code:

```
out vector(point_number) my_scatter_graph(){
        vector(number) x = [0, 0.5, 1, 2, 5, 10];
        vector(number) y = x^2;
        return point(x, y);
}
```



Result of the scatter graph example above

This function can be attached to a graph window and can then be formatted to show just the points or with lines in between, as described in 7.3.4.

7.3.6 Plots using matrices or series(vector(number))

It is possible to plot a matrix of numbers or a matrix of points. Quantlab will interpret the matrix as a collection of column vectors that will be plotted as usual vectors.

Likewise, a series of vector of numbers will be interpreted as a collection of series which each will be plotted.

In order to distinguish between the columns in the matrix or the different series you can use the start and end colouring in the Properties dialog of the attached expression.

Similar plots can be created using a series with two range variables, see 3.7.10.

7.3.7 Column graphs

To produce graphs consisting of columns, mark the data series (the graph) and click the right mouse button. Select Properties and select the tab Format. Here you can select the graph type column and set the width of the columns.

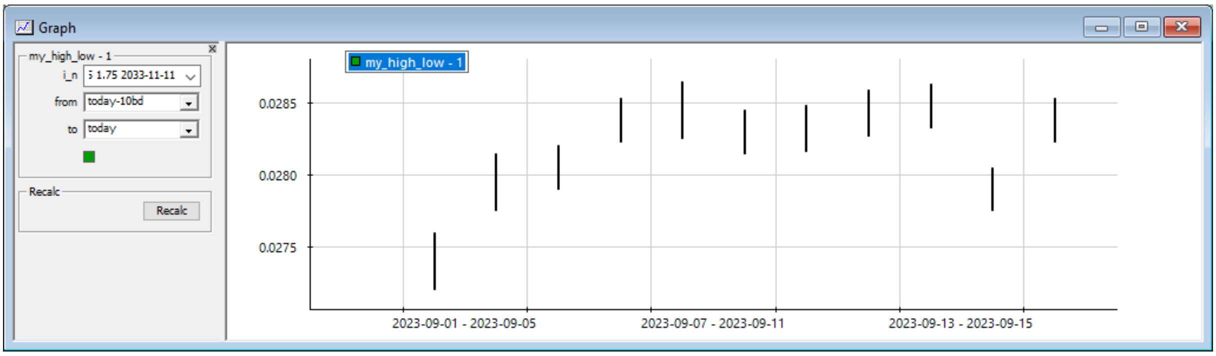
7.3.8 Bar charts (hi-lo etc)

Often, financial data are displayed in the form of bars showing for example high-low or open-close prices. This can be done in Quantlab by using the *pair* object. It is simply a vector of two numbers that, when used in graphs, it is displayed as a bar starting at the first number and ending at the second. For instance, the following expressions can be used for creating a bar chart with bid and ask yields.

```
out number my_yield(instrument_name i_n, date d, quote_side q) =
                         instrument(i_n, d, q).yield();

out series<date>(pair) my_high_low(instrument_name i_n, date from, date to){
      return series(d: from, to;
pair(my_yield(i_n, d, 'bid'), my_yield(i_n, d, 'ask')));
}
```

If the second function is attached to a graph, it will show a typical bar chart which can be formatted with the desired bar width etc.

In the formatting dialog you can choose between line and column. In the first case the width will be constant, in the second case it will change when zooming in the graph window.



Graph

my_high_low - 1

i_n 5 1.75 2033-11-11

from today-10bd

to today

🟩

Recalc

In order to show the dates on the grid it can be necessary to right click on the background of the graph and choose Graph Properties | Axis and un-click “When applicable display interval” in the X-axis format pane. Also, choose the tab Misc and click On grid.

To construct a chart with several values for each date, for instance open-close and high-low, you can simply attach several functions using pair objects. Of course, it can also be combined with a normal graph if the number of values is odd.

Pairs can also be combined with point objects. Then the second value in the point object will be a pair object.

7.3.9 Creating labels

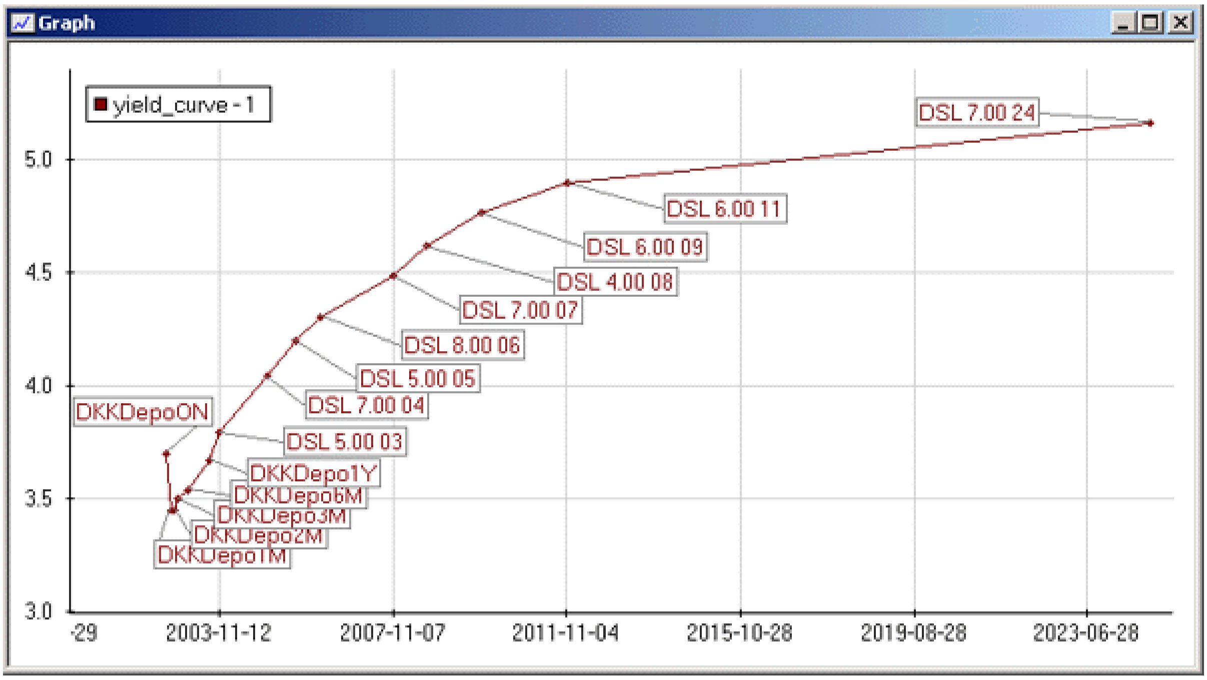
A common case where the point function is used is when producing various kinds of yield curves. Then it is useful to show labels telling the names of the bonds. The following is an example of how to produce a yield curve graph with the instrument names.

```
out vector(point_date) yield_curve(curve_name c_n, date d){
      curve c = curve(c_n, d);
      vector(date) maturities = c.instruments().maturity();
      vector(number) yields = c.instruments().yield()*100;
      return point(maturities, yields);
}
out vector(label_date) yield_curve_labels(curve_name c_n, date d){
      curve c = curve(c_n, d);
      return label(c.instruments().maturity(), c.instruments().name());
}
```

First, attach the yield_curve function to a graph window and then the yield_curve_labels function. Then you get a question which function to associate this labels to. If you choose to attach it to the first function you will get a yield curve with labels connected to each point.

If the labels cover the graph you can drag and drop them where you want. To get them in the original position you can select the graph, right-click on the mouse and choose Snap labels.

Attention! In this example the curve is constructed twice for the sake of clarity. If there are frequent real time updates and many curves it may be necessary to store calculated data in global variables, see 3.5 and 7.5.2.



Example of a Danish yield curve with labels attached to each point.

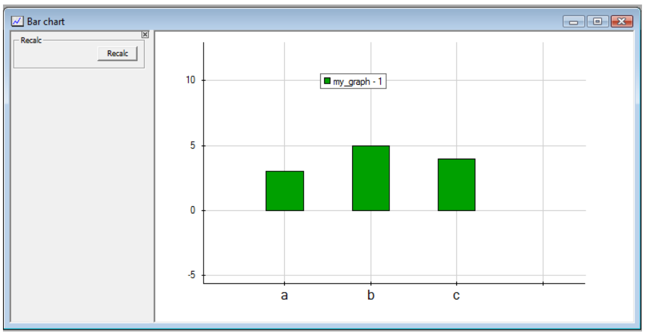
Tip! If you only want the labels to appear when holding the curser over a point, you can select the graph, right-click on the mouse and choose Properties. Then go to the format tab and un-click Show in the Label box. All labels disappear but each label text will be shown in the yellow box that appears when holding the cursor on a point.

In the example above, the labels where attached to the graph function, another possibility that could be useful in some cases, for example bar charts, is to attach the labels to the x-axis. For example, given the following code

```
out my_graph() = [3, 5, 4]
out my_labels() = ['a','b','c']
```

you can produce labels on the x-axis by choosing that option when attaching the label function to the graph. If you want bar charts you select the graph and click the right mouse-button to get the Properties dialog. There you select Show attachment as Column. Note that in order to associate the labels to the x-axis, the graph function must consist of only a vector of number rather than a vector of points.

It might be necessary to zoom in or out in order to view the labels correctly.



A simple bar chart.

7.4 Handling parameters

7.4.1 Simple parameter controls

Below follows a list of control choices that can be used for common controls where several parameters are merged into one control.

Control	Used for common instances of:	Example of Qlang code
String edit	String parameters (ex. 'string')	MyFunc(string myX)
Number edit	Number parameters (ex. 12.4)	MyFunc(number myX)
Instrument control	Instruments (ex. SGB1044)	MyFunc(instrument_name myX)
Curve list	Curves (ex. EURGOVT)	MyFunc(curve_name myX)
Date control	Dates (ex. 2002-02-02)	MyFunc(date myX)

Day count list	Day count conventions (ex. ACT/360)	MyFunc(day_count_method myX)
Rate type list	Rate type basis (ex. effective)	MyFunc(rate_type myX)
Quote side list	Quote side choices (ex. Bid)	MyFunc(quote_side myX)
Asset swap list	Asset swap calc types (ex. par_value)	MyFunc(asset_swap_type myX)

7.4.2 The instrument control

The instrument control leads to an extensive dialog identical to the one in DatabaseTool. The first element in the drop down list box is always the entry “Select instrument…” which gives access to the instrument dialog. In this dialog you have several possibilities for searching the instrument. You can also view more extended information about a particular instrument by clicking the Info button in the top right corner.

Tip 1! To find an instrument, write the beginning of the name in the index tab. The search function immediately goes to the first instrument that matches what you have written. To select the desired instrument, you can use the up and down arrows, and the press Enter.

All instruments that have been chosen are saved in the drop down list box.

Tip 2! After a while the number of entries in the drop down list box can be quite long. You can decrease it by pressing the delete button repeatedly, after having chosen an instrument in the list.

7.4.3 The curve control

The curve control resembles the instrument control as it is a list with one special entry ("Select curve type…") giving the possibility to limit the number of curves in the list. Each curve in the database is of a user-defined curve type and in the dialog you can select a curve type that will be used in the list.

7.4.4 Creating common controls using Parameters Options

Common controls in graphs and tables can be used for input to several parameters by the use of *merging* in the Parameters Option dialog.

The dialog has to list panes, each showing the same information but in two different ways:

To the left there is a tree showing each attachment (instance of function) with its parameters as leaves.

To the right, there is a tree with groups of controls (common controls and auto-generated controls) with the controls as branches and all parameters associated to the controls as leaves.

See the example in section 7.3.2 where the concept of merging is explained.

In the Parameters dialog it is also possible to set the order of common controls by using the right mouse button in the list to the right. The group of common controls is always above the attachment controls, however.

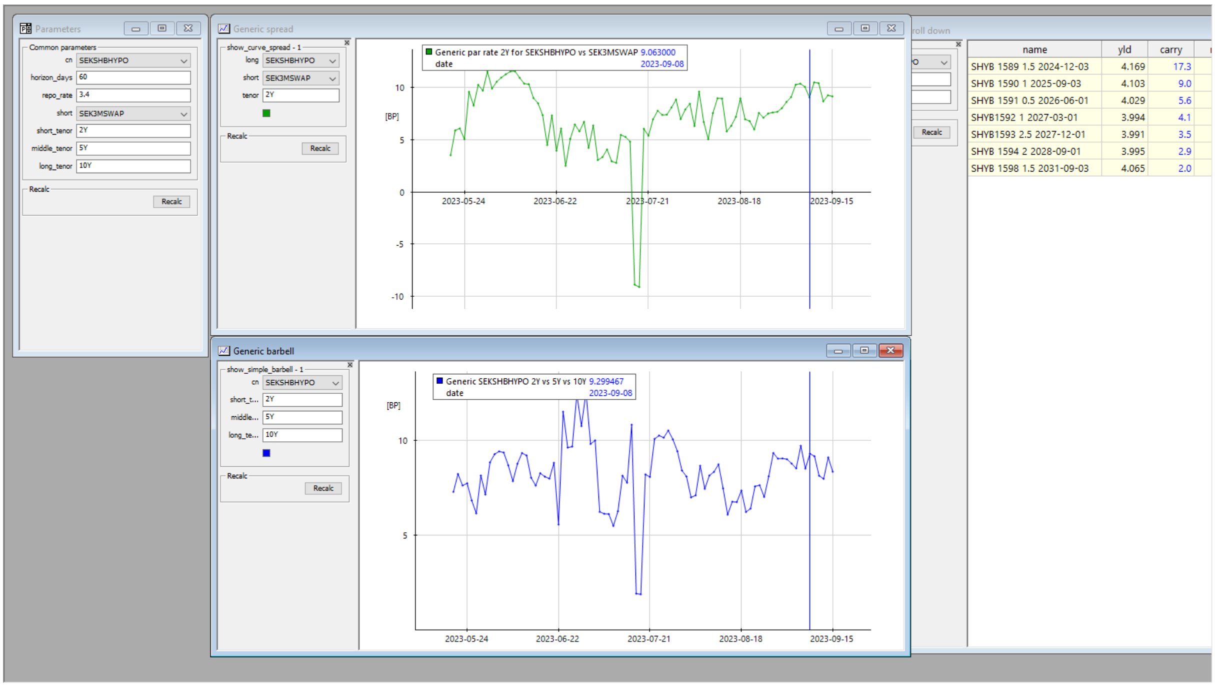
7.4.5 Tab parameters

It is common to organize multiple graphs and tables analysing similar things in the same Tab. Many times it is convenient to have common controls that guide all graphs and tables within the same tab. This can be achieved in the specific tab parameter window, which can be moved around and docked independently. The format of this window is specific to each tab.

To active the tab parameter window use View | Tab parameters, or Alt+5.

At first all parameter controls for every graph and table belonging to the tab will be listed in the window. It is now possible to merge desired controls into common ones. By right clicking on the tab parameter window and selecting parameter options, the merge dialog appears. (This is the same merge functionality available for a single graph or table, as explained in section 7.4.4.)

In an example we wish to merge all curve controls and date controls into two common ones for the entire tab. Adding two common controls and dragging and dropping the individual function parameters into the common ones will give a workspace having overriding parameters in a separate window.


Parameters

Common parameters

cn: SEKSHBHYPO
horizon_days: 60
repo_rate: 3.4
short: SEK3MSWAP
short_tenor: 2Y
middle_tenor: 5Y
long_tenor: 10Y

Generic spread

show_curve_spread - 1

long: SEKSHBHYPO
short: SEK3MSWAP
tenor: 2Y

Generic par rate 2Y for SEKSHBHYPO vs SEK3MSWAP 9.063000
date 2023-09-08

Generic barbell

show_simple_barbell - 1

cn: SEKSHBHYPO
short_t...: 2Y
middle...: 5Y
long_te...: 10Y

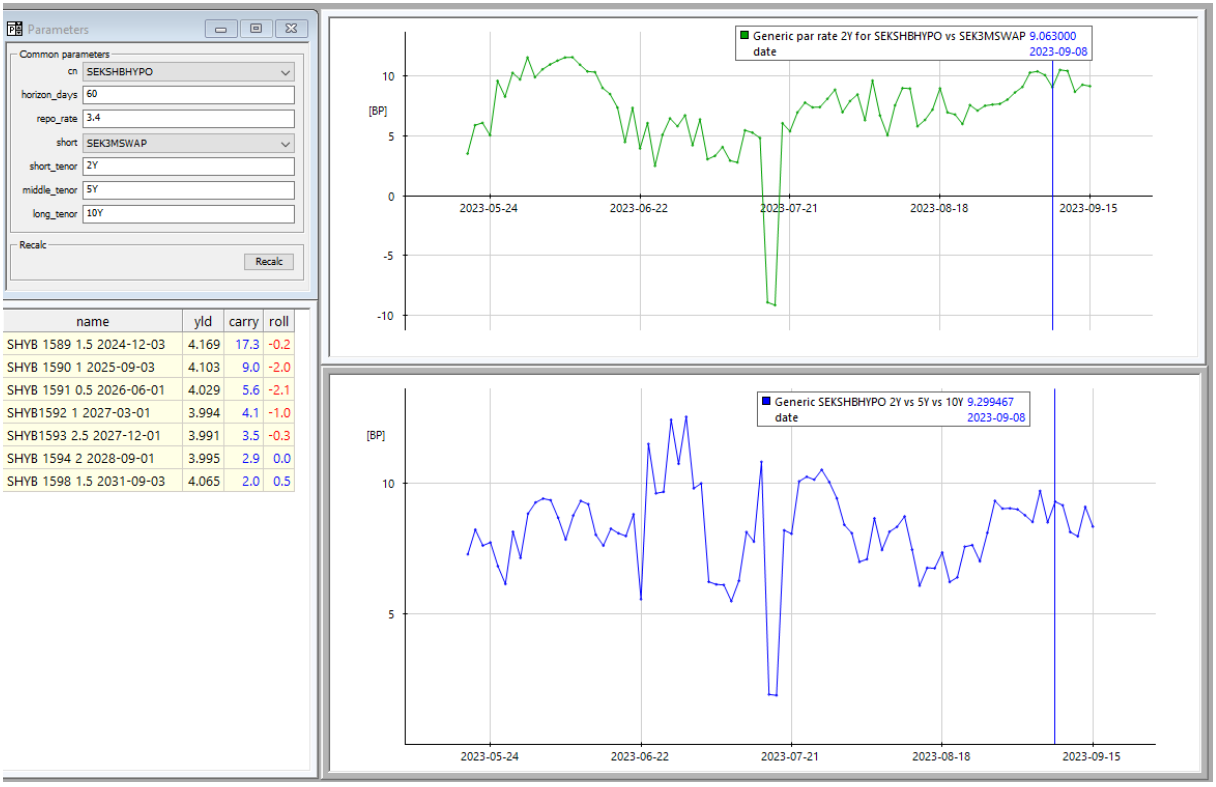
Generic SEKSHBHYPO 2Y vs 5Y vs 10Y 9.299467
date 2023-09-08

name	yld	carry
SHYB 1589 1.5 2024-12-03	4.169	17.3
SHYB 1590 1 2025-09-03	4.103	9.0
SHYB 1591 0.5 2026-06-01	4.029	5.6
SHYB1592 1 2027-03-01	3.994	4.1
SHYB1593 2.5 2027-12-01	3.991	3.5
SHYB 1594 2 2028-09-01	3.995	2.9
SHYB 1598 1.5 2031-09-03	4.065	2.0

Example – upper left window – of common tab parameters overriding each individual window's parameters.

Using the common tab parameter window it is possible to minimize the unnecessary space used by each window's parameter canvas. This will also enable the user to use a single control to change settings for all analysis contained in a tab.

In the Parameters dialog it is also possible to set the order of controls and attachments. In the list of attachments to the left, click the right mouse button on an attachment and select move up or move down. To change the order of controls within an attachment, select the corresponding parameter and use the right mouse button. You can also change the order of common controls by using the right mouse button in the list to the right. The group of common controls is always above the attachment controls, however.



cn: SEKSHBHYPO
horizon_days: 60
repo_rate: 3.4
short: SEK3MSWAP
short_tenor: 2Y
middle_tenor: 5Y
long_tenor: 10Y

name	yld	carry	roll
SHYB 1589 1.5 2024-12-03	4.169	17.3	-0.2
SHYB 1590 1 2025-09-03	4.103	9.0	-2.0
SHYB 1591 0.5 2026-06-01	4.029	5.6	-2.1
SHYB1592 1 2027-03-01	3.994	4.1	-1.0
SHYB1593 2.5 2027-12-01	3.991	3.5	-0.3
SHYB 1594 2 2028-09-01	3.995	2.9	0.0
SHYB 1598 1.5 2031-09-03	4.065	2.0	0.5

🟩 Generic par rate 2Y for SEKSHBHYPO vs SEK3MSWAP 9.063000
date 2023-09-08

■ Generic SEKSHBHYPO 2Y vs 5Y vs 10Y 9.299467
date 2023-09-08

Note! It is advisable not to show tab parameters and specific parameters for the views at the same time, as the tab parameters override the specific parameters but not the other way round.

7.4.6 Writing tool tips for parameters

For each control it is possible to write a short help text, a tool tip, that will show up when the cursor is above the corresponding parameter. Open the Parameters Option dialog and select any parameter you want to describe, write the tool tip in the text box below, and click OK.

There are some exceptions to the tooltip possibility described above: Currently tooltips cannot be written for vector parameters that appear within a table. For some controls, such as a rate type list, Quantlab has its built-in tooltips that cannot be overridden.

7.4.7 User defined lists (fill functions)

In many cases it is useful to be able to create a user defined list. For example, you could define a list that gives the user various options for the calculations of a yield curve, or you could put limitations on how many instruments that should be shown in an instrument list.

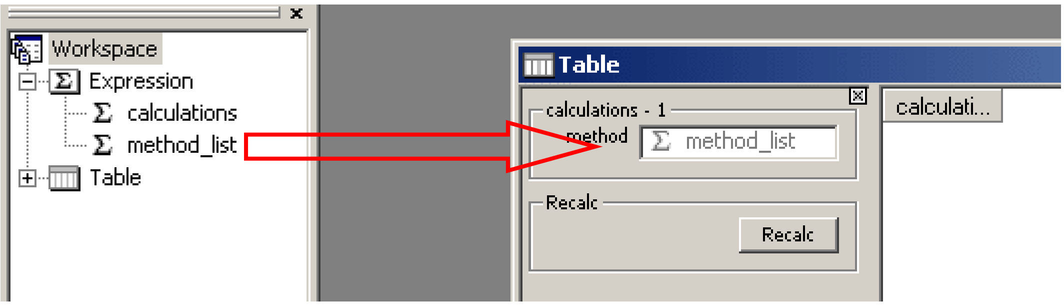
We will describe this feature using two examples. In the first example, we will produce a completely new list. In this case you create a function that takes a string as input parameter which shows up as an edit box in the user interface. By attaching a function returning a vector of strings to this edit box you will create a list containing the elements in the vector. Here is the code:

```
out number calculations(string method){
        number answer;
        if(method == 'bootstrap'){
                // use bootstrap
                // answer = something;
        }
        else if(method == 'tanggaard'){
                // use Tanggaard's model
                // answer = something;
        }
        else {
                // use bootstrap
                // answer = something;
        }
        return answer;
}

out vector(string) method_list() = ['bootstrap', 'tanggaard'];
```

Now, proceed as follows:

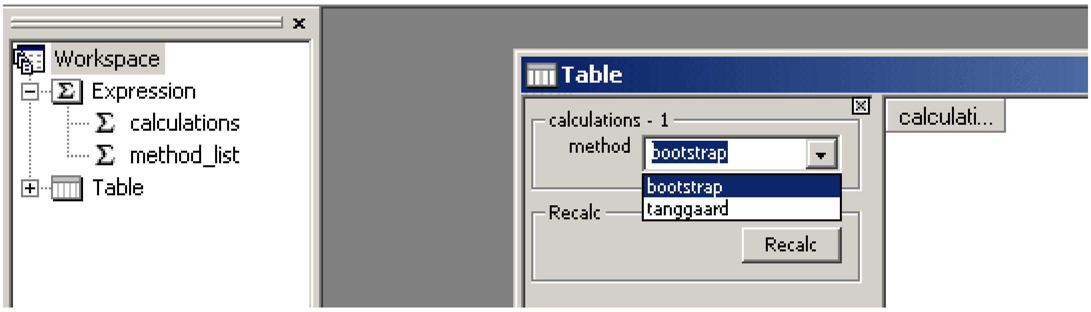
- First, attach the first function, called calculations, to a table.

- Then drag the second function and drop it on the edit box that corresponds to the method parameter of the first function, as illustrated by the red arrow below.



Drag the second function and drop it on the edit box as the red arrow indicates.

Having done this, and pressed Recalc, the text box is transformed to a list containing the two entries given by the vector, see the illustration below.



The text box is transformed to a list.

Our second example shows how to construct a list of instruments given a curve name. The first function calculates the yield of an instrument, given a curve fit.

```
out number test(instrument_name i_n, date d, curve_name c_n){
        curve c = curve(c_n, d);
        fit_result f_r = bootstrap(c);
        return instrument(i_n, d).yield();
}
```

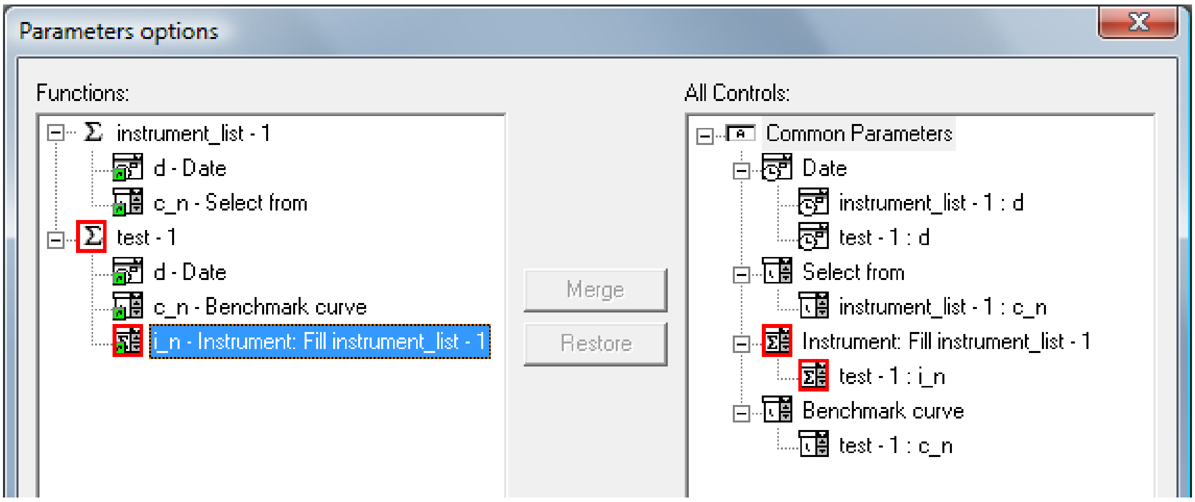
The second function gives a list of instrument names. (Note that we have set the quote side to an empty string, which will enforce the system to not look for a quote in the database or in the real-time source. We have done this, as we are only interested in the names of the instruments.)

```
out vector(instrument_name) instrument_list(curve_name c_n, date d){
      return curve(c_n, d, '').instruments().name();
}
```

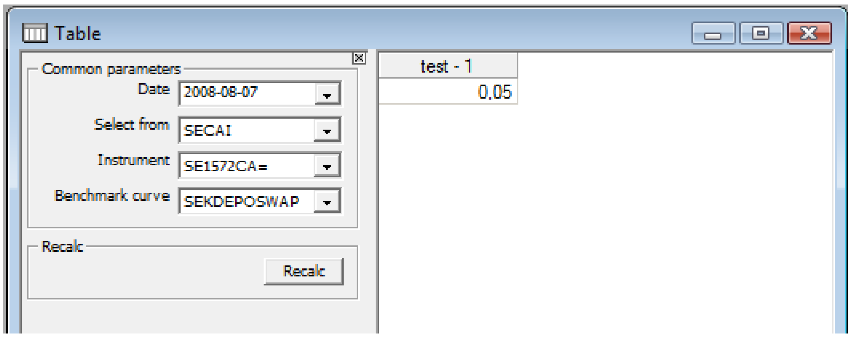
Now, proceed as follows:

- First, attach the first function, called test, to a table.
- Then drag the second function and drop it on the instrument list that corresponds to the instrument _name parameter of the first function.

This will make the instrument list control of the first function dependent on the second function, i.e., the chosen curve. It may be necessary to use the Parameters option dialogue to put the controls in a natural order:



The Parameters options dialogue.



Date 2008-08-07
Select from SECAI
Instrument SE1572CA=
Benchmark curve SEKDEPOSWAP

test - 1
0,05

The table after editing the controls.

Fill functions cannot be removed using the Attach/Detach dialog. Instead, open the Parameters Options dialog and click the right mouse button on a control that has a fill attachment. A drop-down menu appears where you can select Remove fill attachment. Note that fill expressions can be attached both to auto-generated controls and common controls.

See also 8.2 for further examples of using fill attachments.

7.4.8 Out-parameters in attached functions

If an attached function has parameters marked as out the treatment in the user interface corresponds to the treatment within the code. This means that you can set such a parameter in the function and the value will appear in the corresponding control in the user interface. This is particularly useful when initialising controls or when correcting erroneous input values. For example you could have a instrument table where the user is supposed to input a yield. To get an appropriate starting value you could use the market yield of the instruments. For example you could write the following code:

```
logical initiated = false;

out vector(instrument_name) names(vector(instrument) i) = i.name();

out vector(number) price_calc(out vector(number) yield_v, vector(instrument) i){

        if(!initiated){
                yield_v = i.yield()*100;
                initiated = true;
        }
        return i.set_yield(yield_v/100).clean_price();
}
```

If you attach these two functions to an instrument table the yields will always be taken from the market when the workspace is opened but then determined by the user input in the table.

Note that, contrary to standard parameters, the values of out-parameters are not stored in the workspace when it is closed. The reason for this is that, typically, the purpose of the out-parameters is that you want to initiate the parameters by taking values from a distinct source, such as the real time data or the database.

See also 8.6 for further examples using out parameters.

7.5 Handling calculation order

7.5.1 General rules for calculation order of attachments

In some cases it is important to understand in which order Quantlab evaluates functions attached to graphs or tables to properly get correct results. This is especially true when using global variables and ensuring that they have been updated before proceeding with other calculations dependent on the global variable.

Ordinary function calls do not have a pre-defined calculation order. However, void functions receive special treatment in the evaluation engine. All void functions in a tab are evaluated first by the engine. This is also true for multiple expression windows (i.e. all void functions in all expression windows are evaluated before any other function is evaluated) in the same tab. However, there is no particular order among void functions, if several void functions are attached to the same graph, for example. Therefore, it is often best to use one void function and call the others.

Knowing that the void function evaluates first comes in handy when you need control over any global variables that need to be pre-processed. When this control should be extended to the user, in graphs and tables, the void function can simply be attached to the graph or table as any ordinary 'out' function.

An example;

```
number c; // the global variable availabe to all functions in the expression window
out void f1() // the 'out' keyword exposes the void function to the interface
{
  c = rng.gauss();
}
out number f2(number b)
{
  return b + c;
}
out number f3(number x)
{
  return x + c;
}
```

In the example above we assume that all three functions are attached to the same table. This will ensure that the global variable c always will be refreshed with a new random number before f2 and f3 are evaluated.

When there comes new input data to an attachment that currently is being evaluated (from the user interface or from the real time source), all this input data will be used in the next call of the function. This means that there is no queue of function calls of the same function attachment, so each attachment can only have three states:

- Evaluation completed
- Evaluation in progress
- Waiting for evaluation, due to all new input data.

The two second cases can occur at the same time.

Often it can be useful to do some initialisations before all other calculations, i.e., on opening the workspace. This can be done using a global variable that is set by a function that does all initialisations:

```
logical init_all(){
        // Initialization code
        return true;
}
```

```
logical g_init = init_all;
```

A special case in the evaluation order is that in an instrument table, the function calculating the instrument vector has to be evaluated first. This is because it is impossible to do any calculations at all before the vector is well-defined. See also 7.2.3 and 7.5.3.

7.5.2 Performance optimisation

An important application of the calculation order in combination with global variables is the case where you have a time-consuming calculation, for example involving time series data, and some faster calculations, for example involving real-time data, in the same graph or table. In such a case you could separate your calculations so the time series calculations do not involve any real-time data and put them in a void function that puts the result in one or several global variables. Then you can use ordinary functions to display the values of the global variables and combine them with real-time data. This will reduce the number of recalculations of the time consuming part to only the cases when it is necessary, i.e., when the user has changed any input variable and not each time there is a real-time update.

Note! In some cases it may be natural to attach a function that, given an instrument name as a parameter, retrieves data from a global variable. Then it should be noted that this function will not be updated in real time if it doesn't also create an instrument. The real time engine is only triggered whenever an instrument (or a curve) is created using today's date.

7.5.3 Calculation order in the instrument table

The Instrument table has a built-in initiation of the vector of instruments prior to the evaluation of both the void and ordinary functions. Any change in curve, quote side, or date will trigger a re-initiation of the vector, then evaluate any void functions, and last evaluate all ordinary functions.

This is also true for the case when the instrument vector is defined through a user-defined function as described in 7.2.3.

7.5.4 Using buttons to trigger calculations

Normally the calculation of an attached function is triggered by a change in a real-time quote or by pressing the Recalc button (explicitly, or implicitly when pressing Enter after having changed a value in a control or in a table). It is also possible to letting an attached function be evaluated only when a button is pressed.

To get a function controlled by a button, first attach the function to a view (a graph or a table), then, in the workspace browser, right-click on the attachment and choose Add Recalc button. A button appears for which you can set a caption text.

Whenever this button is pressed the attached function will be evaluated using the latest real-time quotes. This is the only way that the calculation of this particular attachment will be initiated. Hence, it will not be automatically updated by changes in the real time source.

8 Case studies

The following case studies are aimed to illustrate common financial calculation subjects. Most of the examples are programmed using the short form for functions, i.e., one-row functions. Each example corresponds to a workspace file in the folder \Quantlab\examples\workspaces\.

8.1 Producing a zero coupon curve: zero_curve.qlw

In this example we will take a set of instruments – a yield curve – and calculate zero coupon rates using the bootstrap method. The zero coupon rates are then plotted against time to maturity in order to produce a zero coupon curve.

Here is a function that solves the problem:

```
out series<number>(number) zero_curve(curve_name c_n, date trade_d){
      disc_func f_r = bootstrap(curve(c_n, trade_d));
      return series(t: 0.1, 10, 0.1; f_r.zero_rate(0, t, RT_EFFECTIVE));
}
```

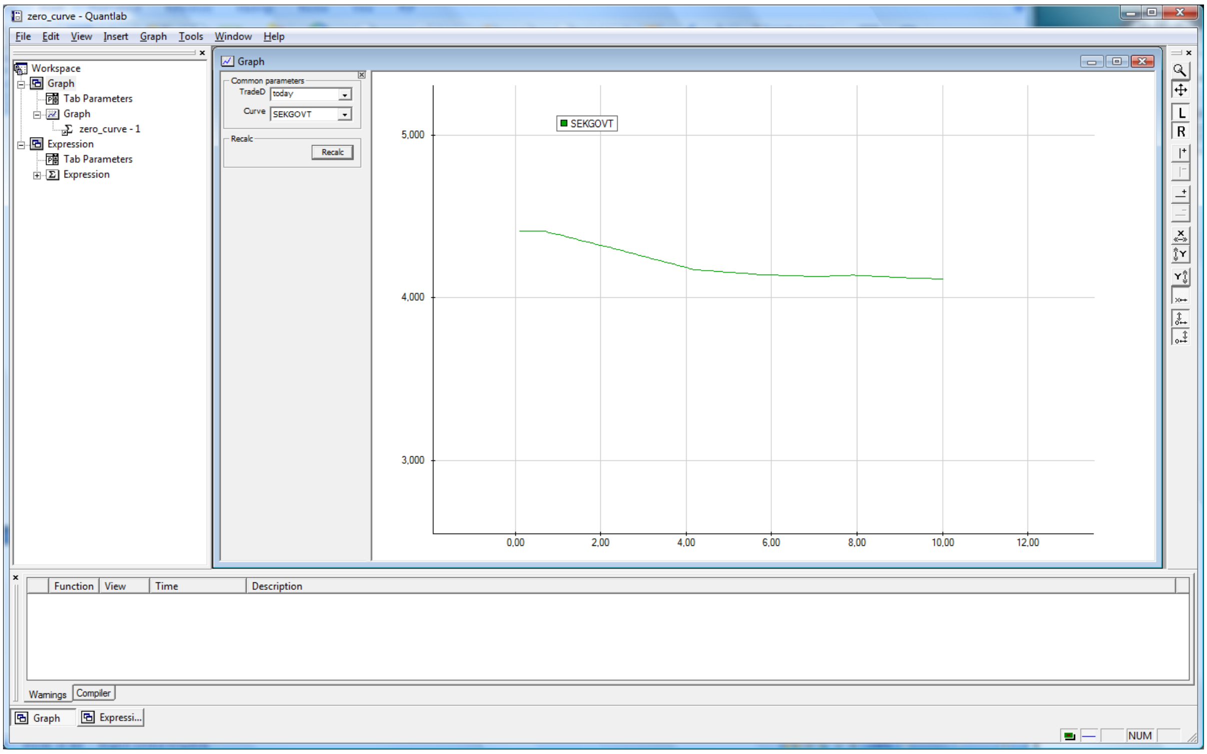
In the first line of code of this function, we create a curve using a curve name and a trade date. Then we apply the bootstrap function which gives us a fit_result object z_c which contains all information on the zero coupon rates. What Quantlab does when it performs this row, is that it searches in the database for a curve with the curve name stored in the parameter c_n for the date trade_d. It then collects all static data for the instruments on the curve on the specified trade date and performs a zero coupon calculation using the bootstrap method.

In order to plot a graph, we have to produce a series of zero coupon rates. Here we take a maturity range from 0.1 years to 10 years with a step size of 0.1, and calculate the zero coupon rate for each maturity using the method zero_rate of the fit_result object.

We have chosen to plot the effective zero coupon rate. The zero coupon rate starts at the trade date and matures at t years later. As there is no forward start the first argument of zero_rate is set to 0.

The function can be attached to a graph. Although there are no common parameters, you can rename the parameters by clicking the right mouse button, choosing parameters options and then create two controls; one for the trade date, one for the curve name. For more information about merging parameters, see 7.3.2.

Depending on what curves are defined in your database, you can choose a curve and get a zero coupon curve based on that collection of instruments.



TradeD today

Curve SEKGOVT

Correctly applying the example should give a workspace with yield curve and date controls.

8.2 Zero coupon curve with blending and choice of methods: zero_curve2.qlw

The previous case can easily be extended with the option to choose the zero coupon method. Let's say you will give the end-user the possibility to choose between the bootstrap, Nelson-Siegel, and Maximum Smoothness methods. Then the zero coupon function in the previous case can be extended like this:

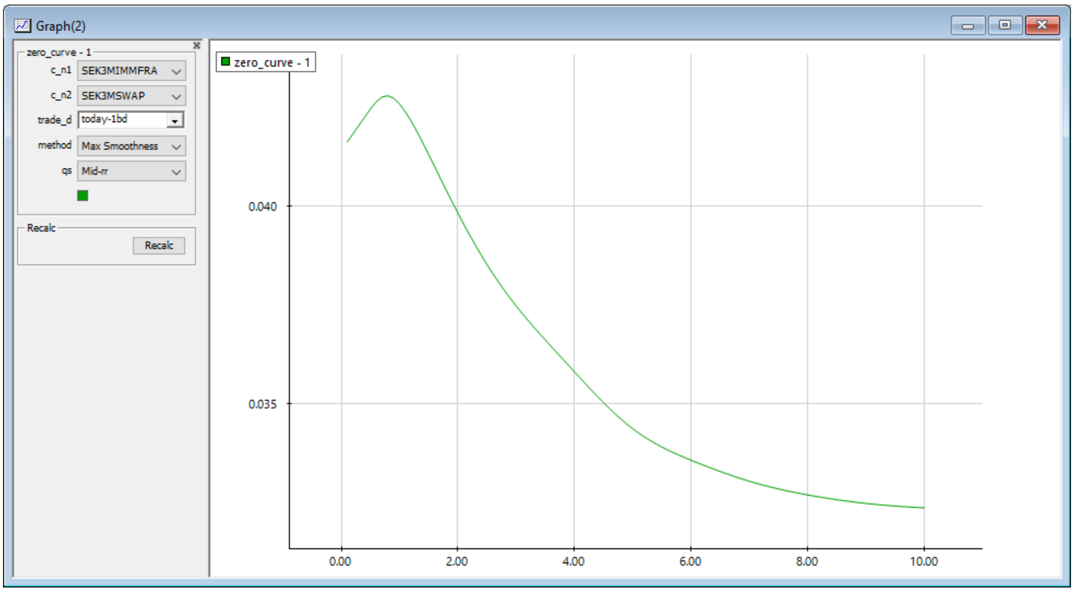
```
out series<number>(number) zero_curve(curve_name c_n1, curve_name c_n2, date
trade_d, string method, quote_side qs){
        curve c = blend_curve(curve(c_n1, trade_d, qs), curve(c_n2, trade_d, qs));
        disc_func f_r;
        if(method=='Bootstrap')
                f_r = bootstrap(c);
        else if(method =='Nelson-Siegel')
                f_r = fit(c, ns(), WS_PVBP, 2);
        else if(method == 'Max Smoothness')
                f_r = max_smooth(c, SMOOTH_C2);
        else
                throw(E_INVALID_ARG, 'Unknown zero coupon method');

        return series(t: 0.1, 10, 0.1; f_r.zero_rate(0, t, RT_EFFECTIVE));
}
```

We have also taken the opportunity to extend the curve creation with a blending function: This function will take a curve with short maturities and a curve with long maturities and merge them. If there are overlapping instruments, they will be removed from the short curve. For other blending options, see the Function browser. The parameter `qs` gives the possibility to choose among pre-defined quote sides (bid, ask or mid).

Instead of letting the user manually type the strings for the zero coupon methods we can create a list from which it is possible to make a selection:

```
out vector(string) methods() = ['Bootstrap', 'Nelson-Siegel', 'Max Smoothness'];
```

This vector function can then be attached to the string control in the user interface that corresponds to the parameter `method`.



zero_curve - 1

c_n1 SEK3MIMMFRA

c_n2 SEK3MSWAP

trade_d today-1bd

method Max Smoothness

qs Mid-rr

The Maximum smoothness method applied to a blending of two curves.

8.3 A zero coupon studio: zero_studio.qlw

This example is a more elaborate version of the preceding zero coupon workspaces. We will not go through the code row by row but give some general comments.

The most important calculation is done in the void function calc_zero which sets the global fit_result variable g_f_r to the result of a zero coupon estimation of the chosen type. Then there are a number of functions that use the global variable to produce the zero coupon curve, the forward curve or zero coupon implied yields for the bonds. As the void function is evaluated first, all other functions will always use the global variable when it is updated with the most recent real time quotes and user input.

However, if a function only presents data that is based on a global variable, it will not be triggered by real time updates, therefore the first row of these functions creates a curve of the relevant instruments. If today's date is chosen this will make these functions triggered by real time updates in any of the instruments on the curve.

In the user interface, we have merged parameters for all functions in the tab parameter pane. For the zero coupon models and the weighting methods we have used fill-attachments on the merged controls.

8.4 Pricing a bond relative to a benchmark curve: bond_pricing.qlw

Often fixed-income instruments are priced relative to a benchmark. Either this can be a single instrument where you simply calculate the yield spread between the two instruments, or a whole curve. In the latter case you have to calculate the corresponding zero coupon benchmark curve and then price all cash-flows of the selected instruments using the zero coupon rates. This gives a fair value of the bond, if it were an instrument on the benchmark curve. The spread between the corresponding zero-curve implied yield and the market yield is therefore an accurate measure of the spread to the benchmark curve.

In this example we will produce a graph of the daily spread between a bond and a benchmark curve during a chosen time period.

As in example 8.1, we must first create a curve using a curve name and a trade date. Then we apply the bootstrap function which gives us a fit_result object which contains all information on the zero coupon rates:

```
disc_func zero_rate_structure(curve_name c_n, date trade_d)
{
	return bootstrap(curve(c_n, trade_d));
}
```

When this function is calculated, Quantlab searches in the database for a curve with the curve name stored in the parameter c_n for the date trade_d. It then collects all static data for the instruments on the curve on the specified trade date and performs a zero coupon calculation using the bootstrap method.

Now, we want to calculate the spread between the bond and the benchmark. The following line of code solves that problem:

```
return i.yield() - i.yield(zero_rate_structure(c_n, trade_d));
```

The function first retrieves the market yield of the bond and then subtracts the yield implied from the zero coupon function. Note that this yield is calculated from the sum of the present values of all cash-flows of the bond, valued using the zero coupon curve.

Finally, we want to plot this spread for each day during a chosen time period:

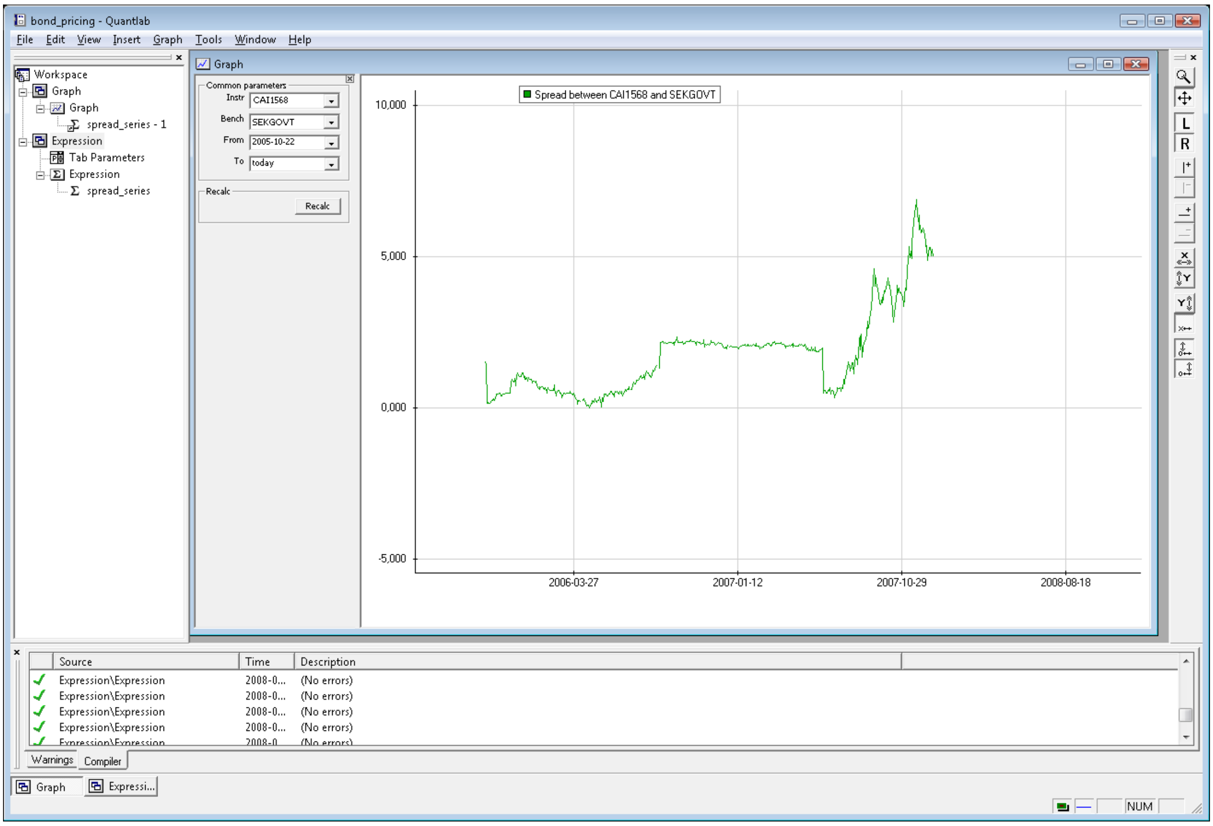
```
out series<date>(number) spread_series(curve_name c_n, instrument_name i_n, date
from_d, date to_d)
{
	return series(d : from_d, to_d ; yield_spread(c_n, instrument(i_n, d), d)) ;
}
```

Here, we construct a series from the date from_d to the date to_d and call our spread function for each day in the date range.

On each day in the date range the following steps are performed:

- Retrieve the instrument data from the database.
- Retrieve the curve data from the database (what instruments are on the curve on that specific date).
- Retrieve the instrument data for each instrument on the curve.
- Retrieve market prices for all instruments above.
- Calculate a zero coupon curve (a disc_func or fit_result).
- Calculate the present value of all cash-flows of the bond.

- Convert the present value to an equivalent zero-implied yield, using the calculation method of the bond.
- Calculate the spread between the market yield and the zero-implied yield.



Instr CAI1568
Bench SEKGOVT
From 2005-10-22
To today

Source	Time	Description
✓ Expression\Expression	2008-0...	(No errors)
✓ Expression\Expression	2008-0...	(No errors)
✓ Expression\Expression	2008-0...	(No errors)
✓ Expression\Expression	2008-0...	(No errors)
✓ Expression\Expression	2008-0...	(No errors)

The example – showing a Swedish mortgage bond spread to the SEKGOVT curve.

8.5 An instrument table with spreads to a benchmark curve: bench_spreads.qlw

In this example we will make use of an instrument table. This is a special-purpose table where each row corresponds to a particular instrument and each column corresponds to a function that operates on the instruments. Actually, the function takes as input parameter a vector of instruments which corresponds to all rows in the table. See 7.2 for general information about the instrument table.

First, create an instrument table by choosing Insert and then Instrument Table in the menu bar. In the Instrument Table dialog you will be able to decide which yield curve or which instruments that should form the instrument vector for the table. You can also choose trade date and quote side for the instruments. These settings can be changed in the parameter pane if you press the button called Modify.

The first column in an instrument table typically contains the instrument names. This can be done by attaching the following expression to the instrument table:

```
out vector(instrument_name) names(vector(instrument) i) = i.name();
```

The input parameter i is set by Quantlab according to your choice in the Instrument Table dialog. In the same manner you can create functions that will give the maturity dates and the yields of the instruments:

```
out vector(date) mats(vector (instrument) i) = i.maturity();

out vector(number) yields(vector (instrument) i) = i.yield();
```

For the calculation of spreads, we will proceed similar to the example in 8.2:

```
out vector(number) yield_spread(curve_name c_n, date trade_d, vector(instrument)
i){
       disc_func f_r = bootstrap(curve(c_n, trade_d));
       return i.yield() - i.yield(f_r);
}
```

Note that in the yield spread function we have changed the instrument parameter to a vector of instruments. The vector of instruments has to be the last parameter of the function. In order to only do the bootstrap calculation once, we save the fit_result object in the local variable f_r.

If you attach all these functions to the instrument table you will get a table of spreads to a given benchmark curve.

Probably, you want to see the yields in percentage points and the spreads in basis points. This can be done by right-clicking the appropriate column header and choosing Number format. In the Number format dialog you can choose to multiply all numbers in the column by a factor, for example 100 or 10,000.

You can also right-click to reach menus for formatting the cells in the table and the column header text. For example, in the Column display dialog you can set the display name of the spread column to “Spr to ” and then double click on the c_n parameter in the list box to the right in order to get the current curve name in the header. The column header will then be “Spr to SWAP” if you have chosen a curve called SWAP.

8.6 *Extending spread calculations with user input: bench_spreads2.qlw*

The previous example can easily be extended with a possibility to manually input the yields. In order to get relevant yields to start with the workspace will present the market yields in the input column. To be able to differentiate between the two cases (1) the first time the calculation is done, or (2) when a manual input is done, we have introduced a global variable g_initiated:

```
logical g_initiated = false;

out vector(number) yield_spread(curve_name c_n, date trade_d, out vector(number)
yields, vector(instrument) i){
       if(!g_initiated){
              yields = i.yield()*100;
              g_initiated = true;
       }
       disc_func f_r = bootstrap(curve(c_n, trade_d));
       return yields/100 - i.yield(f_r);
}
```

When the workspace is opened and the function is evaluated for the first time, `g_initiated` is false, but each time the user makes an input in the yields column, `g_initiated` will be true. Thus, the workspace will use the market rates for the first time and then the user input.

As the function is attached to an instrument table, the input vector yield_spread will automatically have the same length as the instrument vector i.

8.7 Calculating covariances: covariance_matrix.qlw

This simple example shows how to use vector expansion and how you can use vector input in tables. We will calculate a covariance matrix and a correlation matrix using price quotes for some instruments.

We start by creating a time series with logarithmic changes of quotations on one instrument:

```
out series<date>(number) log_series(instrument_name i_n, date from_d, date to_d) =
            change(series(t : from_d, to_d; log(instrument(i_n, t).quote())));
```

The function change operates on the series and takes the difference between each value and the previous one. If one of the values is Null, then the value of the change also is Null.

This function can be called using vector expansion in order to calculate the covariance matrix:

```
out matrix(number) covar_matrix(vector(instrument_name) i_n, date from_d,
date to_d) =
              covariance(log_series(i_n, from_d, to_d));
```

This function can be attached to a table where you will have to set the number of rows in the input vector of instrument names. This is cone by right-clicking on the corresponding column header and selecting Input parameters.

If we instead want to have the correlation matrix we write:

```
out matrix(number) corr_matrix(vector(instrument_name) i_n, date from_d,
date to_d) =
              correlation(log_series(i_n, from_d, to_d));
```

8.8 *Creating a simple portfolio Value-at-Risk function: Portfolio_VaR.qlw*

This example will use the matrix calculation possibilities in order to calculate a Value-at-Risk measure for a small portfolio. Actually we will test three different VaR models.

In order to get easy access simulation possibilities, we will expose the most important parameters to graphic interface. The function definition nmb looks like this

```
out vector(number) myRisk(vector(instrument_name) i_n, vector(number) w, number
conf, number horizon, date startdate, date enddate)
```

The parameters are instrument names and their weights and a confidence level and horizon number of days. The dates startdate and enddate are used to choose the date-range for the covariance matrix calculation.

We will use the fact that a series expression can take a vector of instruments as input and return a series of vectors of numbers. The objective is to extract the time series (the clean price) for the chosen instruments and the period specified between startdate and enddate. We also take the logarithmic changes:

```
series<date>(vector(number)) price_series =
series(d:startdate,enddate;instrument(i_n,d).clean_price());
series<date>(vector(number)) log_series = change(log(price_series));
```

We can now create the covariance matrix from the logarithmic changes. To analyse the difference between VaR methods using different time weighting, we create two covariance sets. The first row produces a standard set and the second a RiskMetrics set.

```
matrix(number) Cov = covariance(log_series);
matrix(number) CovExp = covariance_exp(log_series,0.94);
```

As a third method, we compute the daily portfolio returns as a percentage. Multiplying the time series of vectors with the weights will return the daily portfolio value. For each day there will be an inner product of the returns and weights:

```
series(number) dPortf = change(log(price_series*w));
```

We are now ready to conclude and return the three equivalent Value-at-Risk figures together with the sum of weights:

```
number  A = sqrt(w*Cov*w*horizon)*inv_normal(conf);
number  B = sqrt(w*CovExp*w*horizon)*inv_normal(conf);
number  C = std_dev(dPortf)*sqrt(horizon)*inv_normal(conf);
number  D = v_sum(w);
// return a vector with the results
return [A, B, C, D];
```

We will not give a lesson on Value-at-Risk formulas here but concludes that we again use Quantlab's series and vector capabilities to write the formulae in short-form.

8.9 *Calculating tail rates: tail.qlw*

This is another example that illustrates some vector functions. We will calculate tail yields for some chosen yield curves. A tail yield is a zero-coupon forward rate between the maturity dates of two consecutive bonds. In order to perform the calculations we have to have two vectors with the settlement dates and the maturity dates for the forward rates, respectively.

First we define two simple help functions:

```
disc_func boot(curve_name c, date d) = bootstrap(curve(c, d));
vector(date) mat(curve_name c, date d) = curve(c, d).instruments().maturity();
```

With the second function we can take out the settlement dates and the maturity dates for the forward rates. The question is now how to extract the correct settlement and maturity dates for the tail rates. The first settlement date is the maturity date of the first bond and the last settlement date is the next-to-last maturity date. For the maturity dates of the tail rates it's the opposite: The first is the next-to-first maturity date among the bonds and the last is the last maturity date of the bonds. This can be solved by using the sub_vector function:

```
vector(date) settle(curve_name c, date d) = sub_vector(mat(c,d), 0,
v_size(mat(c,d))-1);

vector(date) matur(curve_name c, date d) = sub_vector(mat(c,d), 1,
v_size(mat(c,d))-1);
```

Remember that vectors are indexed starting at 0. In order to calculate the tail rates we use bootstrap and extract the relevant zero coupon yields:

```
vector(number) tail(curve_name c, date d) = boot(c,d).zero_rate(d, settle(c,d),
matur(c,d), RT_SIMPLE, DC_ACT_360)*100;
```

In order to plot the tail yields versus the maturity dates we use the following function:

```
out vector(point_date) tail_graph(curve_name c, date d) = point(matur(c,d),
tail(c,d));
```

To get labels with instrument names the function names_matur is attached to the tail_graph functions in the graph.

If we want to look at a particular tail rate development over time we can write:

```
out series<date>(number) tail_series(curve_name c, instrument_name i1,
instrument_name i2, date from, date to) =
       series(d: from, to; boot(c,d).zero_rate(d, instrument(i1, d).maturity(),
       instrument(i2, d).maturity(), RT_SIMPLE, DC_ACT_360)*100);
```

Obviously, the maturities of two bonds are required to calculate one tail rate.

8.10 Has the market been wrong or right?: expectations.qlw

This is an example of how you can create quite interesting graphs with a very limited amount of work. We are going to plot a graph of the development of the 90 days rate, together with some graphs showing the market's expectations of the same rate, i.e., forward rates.

First we define our zero coupon function, for example the following:

```
fit_result my_fit(curve c){
       fit_result f_r;
       try
              return fit(c, ns_svensson(), str_to_std_weights('pvbp'), 2);
       catch
              return f_r;
}
```

By using the norm euqal to 2 in the fitting algorithm, price discrepancies are measured using the sum of squares.

This function will be used in two other functions. The first one deal with the historical time series of the fixed short rate:

```
out series <date>(number) z_series(curve_name cn, date from, date to, number
tenor){
       return series(t:from, to;
              my_fit(curve(cn, t)).zero_rate(t, t, t + tenor, RT_SIMPLE,
              DC_ACT_365));
}
```

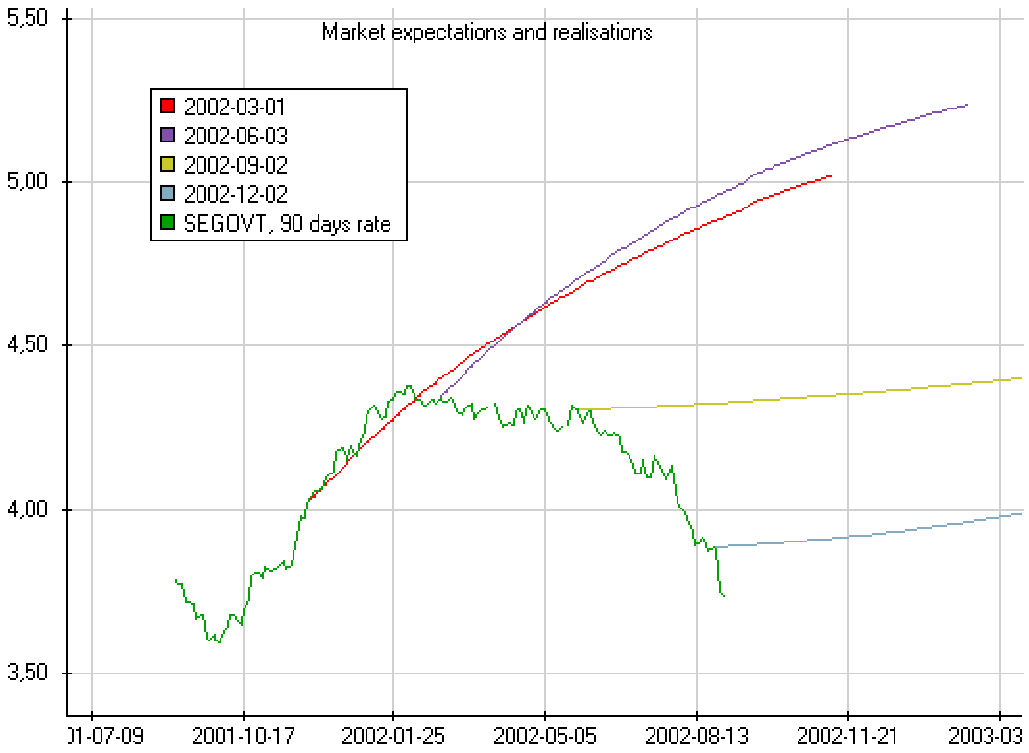
When dropped in a graph window this function will plot the simple Act/360 rate with a maturity time given by tenor, in days, for the time period from the from date to the to date.

Then we want to add the market expectations at some given dates. This can be done by plotting the forward curves, starting at different dates. We write a function for forward curves:

```
out series<date>(number) z_c(curve_name cn, date d, number tenor){
       curve c = curve(cn, d);
       fit_result f_r = my_fit(c);
       return series(t:d, d+360; f_r.zero_rate(d, t, t + tenor, RT_SIMPLE,
       DC_ACT_365));
}
```

Given a date d and a tenor, this function plots a curve with one year's length (the range is 365 days), showing at date d the rate from time t to time t plus the tenor.

Drag several instances of this function to the same graph window as the function above. Then click the right mouse-button and choose Parameters options. Merge the appropriate parameters (for example the curve name and the tenor), see 7.3.2. If you then choose dates for the forward curves during the period for the first function you will get a graph looking like this:



In this case the market was right about the rising rates during the end of 2001 but then it has continuously over-estimated the future short rates.

If the forward curves look rough it may be because of omitted weekends in the time scale. Click the right mouse button on the graph background and choose Graph Properties and the tab Holiday. Unclick the option Hide holidays.

As we have defined one single function that does all zero coupon calculations we can change this to bootstrap or any other type of model and all calculations will remain consistent in the workspace.

Another change you might want to do is to let the user choose between rate types and day count conventions, which is done by introducing parameters for those choices.

8.11 Creating an intra-day chart: intraday_graph.qlw

This is an extensive example using global variables for the storage of intraday data. The purpose of the workspace is to draw a graph of the quoted price of an instrument using each intra-day tick.

The kernel of the workspace consists of the following global variables and function:

```
// This variable is used to see when the user changes instrument
instrument_name i_name;

// This global variable stores the vector of quotes
vector(point_timestamp)  v;


// Shows an instrument's quotes in a graph in real time
out vector(point_timestamp) quote_vector(instrument_name i)
{
        number n;
        number quote = instrument (i, today()).quote();

        // If the vector is empty we must create it
        if (null(v) || i_name != i) {
                vector(point_timestamp) tmp[1];
                v = tmp;
                n = 0;
                i_name = i;
        }
        else {
                n = v_size(v);
        }

        // If the vector has less than two points
        // we add the new point
        if (n < 2) {
                if (n == 0 || v[0].y() == quote) {
                        push_back(v, point (now(), quote));
                }
                else { // n == 1 && v[0].y != quote
                        resize(v, 3);
                        v[1] = point (now(), v[0].y());
                        v[2] = point (now(), quote);
                }
        }
        // Otherwise points are added only to
        // reflect any changes
        else {
                if (v[n - 1].y() == quote) {
                        if (v[n - 1].y() == v[n - 2].y()) {
                                v[n - 1] = point(now(), quote);
                        }
                        else {
                                push_back(v, point (now(), quote));
                        }
                }
                else {
                        resize(v, n + 2);
                        v[n] = point(now(), v[n - 1].y());
                        v[n + 1] = point (now(), quote);
                }
        }

        return v;
}
```

If attached to a graph this function will extend the tick-graph each time there comes a real-time update. If an update has the same value as the preceding one, the x-value will be changed to the new time-stamp without adding a new point. The result will be a typical graph consisting of horizontal and vertical lines.

The global vector v holds the graph data in the form of the Qlang type point_timestamp. It is updated each time the function is called, either by the real-time input or by the user.

The global instrument name i_name is used to check whether the user has changed the instrument which should trigger a complete reset of the graph.

The workspace also draws horizontal lines for closing price and updated maximum and minimum.

Note! Attaching multiple instances of the function created above to a graph will not work properly since they will both share the same instance of the global variable.

8.12Using function pointers and classes: fp_test.qlw

In this example we give some ideas of how to use a couple of new features in Quantlab 3.0: Function pointers and object classes.

We begin by implementing a generic delta function which calculates a numeric delta by calling the pv-function of an imagined class:

```
number calc_delta(object<1> pos, number function(object<1>, number epsilon) pv){
        number epsilon = 0.0001;
        return (pv(pos, epsilon) - pv(pos, 0))/epsilon;
}
```

This function assumes that there exist a member function pv that returns the present value, given a small disturbance epsilon. The syntax object<1> pos means that this argument is a class object of any type – which in this case must implement the pv-function in a meaningful way.

Let's try this out by implementing a simplified derivative class and it's present value function. Note that we can now implement a delta member function by calling the generic delta function above and refer to our member function for the present value by the use of a function pointer. We begin with the derivative class:

```
class derivative
{
        // Member functions
        number pv(number epsilon) ;
        number delta() ;

        // Members
        number price ;
} ;

derivative derivative(number price)
{
        derivative d = new derivative;
        d.price = price ;
        return d ;
}

number derivative.pv(number epsilon)
{
        return log(price+epsilon) ;        // A very strange pv indeed
}

number derivative.delta()
{
       return calc_delta(this, &derivative.pv) ;
}
```

Please note that last line where we use this to refer to the class object and supplies the calc_delta function with a reference to our present value function.

We can now create a derivative object and return the delta value:

```
out number test_derivative(number price)
{
        derivative d = derivative(price);

        return d.delta();
}
```

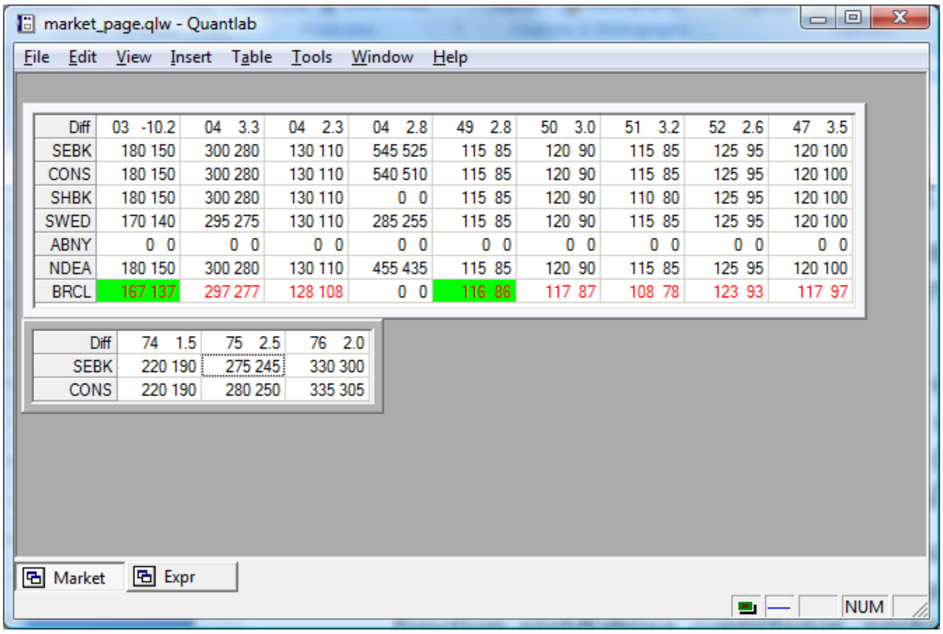
8.13 A condensed market page: market_page.qlw

This is an example of how to build a market page with unlimited number of tables showing quotes for financial instruments with several price contributors. It uses the test_rgb object to highlight quote changes.

The function instr(curve_name c_n, date d) can be used repeatedly for creating new instrument tables, given different curve names. Typically d is today(). The function stores the RICs of the instruments in a global list (a map object). Each time there is a new instrument it is added to the list. So, all instrument tables share the same list of RICs.

In the function format_yield(string ric) a string_rgb object is produced with formatting depending on the recent history of changes in the quote. This function is called by the out-function yield(string contributor, vector(instrument) i) which appends a contributor to the RIC.

When formatting the instrument tables it is convenient to transpose the table, to hide the parameter list and to use minimal frames. Then you can get something like the tab below:


market_page.qlw - Quantlab
File Edit View Insert Table Tools Window Help
Market
Expr
NUM

Diff	03 -10.2	04 3.3	04 2.3	04 2.8	49 2.8	50 3.0	51 3.2	52 2.6	47 3.5
SEBK	180 150	300 280	130 110	545 525	115 85	120 90	115 85	125 95	120 100
CONS	180 150	300 280	130 110	540 510	115 85	120 90	115 85	125 95	120 100
SHBK	180 150	300 280	130 110	0 0	115 85	120 90	110 80	125 95	120 100
SWED	170 140	295 275	130 110	285 255	115 85	120 90	115 85	125 95	120 100
ABNY	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
NDEA	180 150	300 280	130 110	455 435	115 85	120 90	115 85	125 95	120 100
BRCL	167 137	297 277	128 108	0 0	116 86	117 87	108 78	123 93	117 97

Diff	74 1.5	75 2.5	76 2.0
SEBK	220 190	275 245	330 300
CONS	220 190	280 250	335 305

An example of a market page showing Swedish bonds.

The header of the table (which actually is the attachment name(vector(instrument) i)) consists of the last two characters in the instrument name and the quote difference from yesterday's closing. This special header may of course have to be changed for other markets in order to be meaningful. Also the database name (in our case QLDemo) which gives the RICs has to be changed to your database name (i.e., ODBC source).