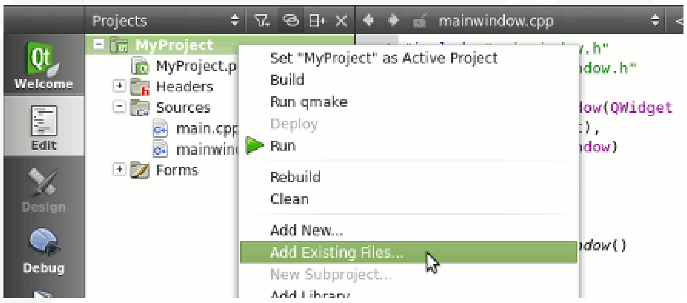
Setting up QCustomPlot

Getting QCustomPlot to work with your application is very easy:

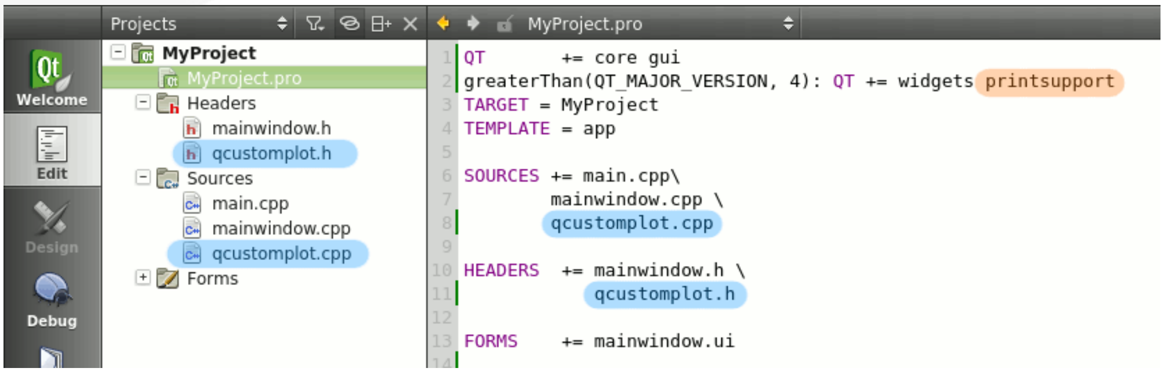
- Get the latest version of QCustomPlot from the download section.
- Use the *qcustomplot.h* and *qcustomplot.cpp* file like any other ordinary class file

For QtCreator users

Right click on the root entry of your project in the left sidebar and choose *Add Existing Files...*



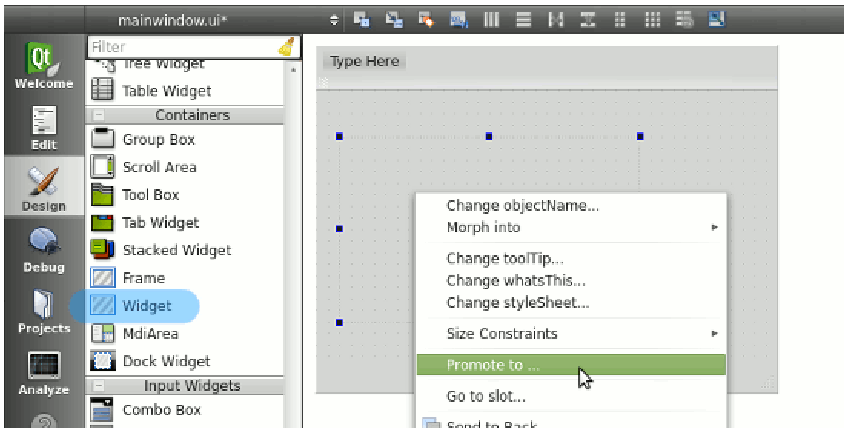
In the appearing file dialog, select the *qcustomplot.h* and *qcustomplot.cpp* files, to add them to your project. If this is done, your project structure and *.pro* file should look something like this:



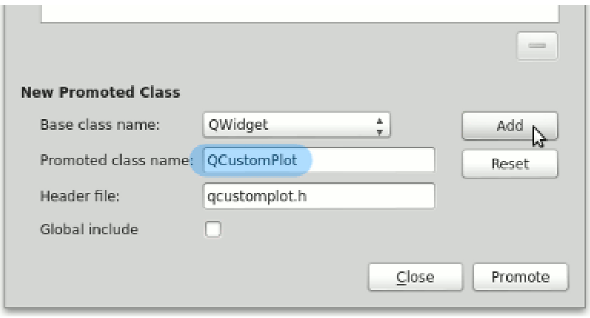
```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets printsupport
TARGET = MyProject
TEMPLATE = app

SOURCES += main.cpp\
        mainwindow.cpp \
        qcustomplot.cpp

HEADERS  += mainwindow.h \
        qcustomplot.h

FORMS    += mainwindow.ui
```

If you are using Qt version 5.0 upwards, you need to add `printsupport` to the `QT` variable in your *.pro* file. In the case shown above, this is done after a `greaterThan(QT_MAJOR_VERSION, 4)` conditional. This makes sure the `printsupport` (and `widgets`) is not added when using older Qt versions.

The project is now ready to use QCustomPlot. Place a regular QWidget on your form in the desired location. Right click on it and hit Promote to...



In the appearing dialog, enter `QCustomPlot` in the input field next to *Promoted class name*. The input next to *Header file* should automatically fill with the correct `qcustomplot.h` value. Hit *Add* to add QCustomPlot to the promoted classes list and finally hit *Promote* to turn the QWidget on your form into a QCustomPlot.



New Promoted Class

Base class name: QWidget
Promoted class name: QCustomPlot
Header file: qcustomplot.h
Global include ☐

You won't see any immediate visual changes in QtCreator (or QtDesigner), but while running the application, you will see an empty plot with axes and grid lines.

Using QCustomPlot as shared library .so/.dll

Using a shared library means to not include the .h/.cpp file into your project, but linking with an external *qcustomplot.so* (GNU/Linux) or *qcustomplot.dll* (MSWindows) file. QCustomPlot is ready to be built as a shared library by setting the compiler define `QCUSTOMPLOT_COMPILE_LIBRARY`. To use the shared library in your application, set the define `QCUSTOMPLOT_USE_LIBRARY` before including the QCustomPlot header.

The *sharedlib* package in the download section provides two projects that demonstrate this: one compiles the shared QCustomPlot library and the other uses the shared library. This should quickly get you started using QCustomPlot as a shared library.

Running the examples

The *QCustomPlot.tar.gz* package in the download section contains the example projects ready to be compiled. Just extract the whole package to a new directory, navigate inside the example directories and run `qmake; make`. Alternatively you can open the *.pro* files in QtCreator and work with the examples from there.

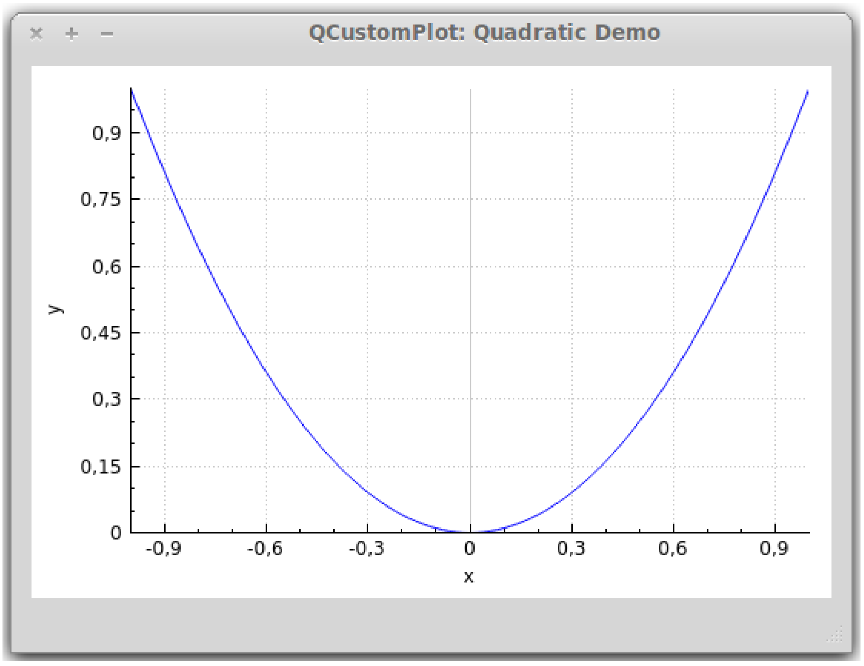
Basics of plotting with QCustomPlot

First, you create a graph with `QCustomPlot::addGraph`. Then you assign the graph some data points (e.g. as a pair of `QVector<double>`s for x and y values) and define the look of the graph (line style, scatter symbol, color, line pen, scatter size, filling...). finally, call `QCustomPlot::replot`. Note that `replot` will be called automatically when the widget is resized and when the built-in user interactions are triggered (e.g. dragging axis ranges with mouse and zooming with mouse wheel). By default, `QCustomPlot` has four axes: `xAxis`, `yAxis`, `xAxis2`, `yAxis2` of type `QCPAxis`, corresponding to the bottom, left, top and right axis. Their range (`QCPAxis::setRange`) defines which portion of the plot is currently visible.

Here's a minimal example, `customPlot` is a pointer to the `QCustomPlot` widget:

```
// generate some data:
QVector<double> x(101), y(101); // initialize with entries 0..100
for (int i=0; i<101; ++i)
{
  x[i] = i/50.0 - 1; // x goes from -1 to 1
  y[i] = x[i]*x[i]; // let's plot a quadratic function
}
// create graph and assign data to it:
customPlot->addGraph();
customPlot->graph(0)->setData(x, y);
// give the axes some labels:
customPlot->xAxis->setLabel("x");
customPlot->yAxis->setLabel("y");
// set axes ranges, so we see all data:
customPlot->xAxis->setRange(-1, 1);
customPlot->yAxis->setRange(0, 1);
customPlot->replot();
```

The output should look something like shown below. Note that the positions of the tick marks are chosen automatically. However, you can take full control over the tick-step and even the single tick mark positions by calling `setTickStep` or `setTickVector` on the respective axis. For disabling or enabling the automation, call `setAutoTickStep` or `setAutoTicks`. Disabling the automation will be necessary, if you want to provide your own tick step or tick vector. If you just want to change the approximate number of ticks in the visible range, while leaving the details to QCustomPlot, use `setAutoTickCount`.



You'll see that the tick labels (the numbers) of the axes are not clipped even when they get wider. This is due to the automatic margin calculation, which is turned on by default. If you don't wish that the axis margin (the distance between the widget border and the axis base line) is determined automatically, switch it off by calling `customPlot->axisRect()->setAutoMargins(QCP::msNone)`. Then you can adjust the margin manually via `QCPAxisRect::setMargins`.

Changing the look

The **look of the graph** is characterized by many factors, all of which can be modified. Here are the most important ones:

- **Line style**: Call `QCPGraph::setLineStyle`. For all possible line styles, see the LineStyle documentation or the line style demo screenshot on the introduction page.
- **Line pen**: All pens the QPainter-framework provides are available, e.g. solid, dashed, dotted, different widths, colors, transparency, etc. Set the configured pen via `QCPGraph::setPen`.
- **Scatter symbol**: Call `QCPGraph::setScatterStyle` to change the look of the scatter point symbols. For all possible scatter styles, see the QCPScatterStyle documentation or the scatter style demo screenshot shown on the introduction page. If you don't want any scatter symbols to show at each data point, use `QCPScatterStyle::ssNone`.
- **Fills under graph or between two graphs**: All brushes the QPainter-framework provides can be used in graph fills: solid, various patterns, textures, gradients, colors, transparency, etc. Set the configured brush via `QCPGraph::setBrush`.

The **look of the axis** can be modified by changing the pens they are painted with and the fonts their labels use. A look at the documentation of QCPAxis should be self-explanatory. Here's a quick summary of the most important properties: `setBasePen`, `setTickPen`, `setTickLength`, `setSubTickLength`, `setSubTickPen`, `setTickLabelFont`, `setLabelFont`, `setTickLabelPadding`, `setLabelPadding`. You can reverse an axis (e.g. make the values decrease instead of increase from left to right) with `setRangeReversed`.

The **look of the grid lines** is modified by accessing the respective QCPGrid instance. Each axis has its own grid instance which is accessible via `QCPAxis::grid()`. So changing the look of the horizontal grid lines (which are tied to the left axis) could be achieved by accessing `customPlot->yAxis->grid()`. The look of the grid lines is basically the pen they are drawn with, which can be set via `QCPGrid::setPen`. The grid line at tick 0 can be drawn with a different pen, it can be configured with `QCPGrid::setZeroLinePen`. If you do not wish to draw the zero line with a special pen, just set it to `Qt::NoPen`, and the grid line at tick 0 will be drawn with the normal grid pen.
Sub-grid lines are set to be invisible by default. They can be activated with `QCPGrid::setSubGridVisible`.

Examples

Simple plot of two graphs

Here's an example which creates the image of the decaying cosine function with its exponential envelope, as shown in the Demo Screenshots.

```
// add two new graphs and set their look:
customPlot->addGraph();
customPlot->graph(0)->setPen(QPen(Qt::blue)); // line color blue for first graph
customPlot->graph(0)->setBrush(QBrush(QColor(0, 0, 255, 20))); // first graph will be filled with translucent blue
customPlot->addGraph();
customPlot->graph(1)->setPen(QPen(Qt::red)); // line color red for second graph
// generate some points of data (y0 for first, y1 for second graph):
QVector<double> x(250), y0(250), y1(250);
for (int i=0; i<250; ++i)
{
x[i] = i;
y0[i] = exp(-i/150.0)*cos(i/10.0); // exponentially decaying cosine
y1[i] = exp(-i/150.0); // exponential envelope
}
// configure right and top axis to show ticks but no labels:
// (see QCPAxisRect::setupFullAxesBox for a quicker method to do this)
customPlot->xAxis2->setVisible(true);
customPlot->xAxis2->setTickLabels(false);
customPlot->yAxis2->setVisible(true);
customPlot->yAxis2->setTickLabels(false);
// make left and bottom axes always transfer their ranges to right and top axes:
connect(customPlot->xAxis, SIGNAL(rangeChanged(QCPRange)), customPlot->xAxis2, SLOT(setRange(QCPRange)));
connect(customPlot->yAxis, SIGNAL(rangeChanged(QCPRange)), customPlot->yAxis2, SLOT(setRange(QCPRange)));
// pass data points to graphs:
customPlot->graph(0)->setData(x, y0);
customPlot->graph(1)->setData(x, y1);
// let the ranges scale themselves so graph 0 fits perfectly in the visible area:
customPlot->graph(0)->rescaleAxes();
// same thing for graph 1, but only enlarge ranges (in case graph 1 is smaller than graph 0):
customPlot->graph(1)->rescaleAxes(true);
// Note: we could have also just called customPlot->rescaleAxes(); instead
// Allow user to drag axis ranges with mouse, zoom with mouse wheel and select graphs by clicking:
customPlot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom | QCP::iSelectPlottables);
```

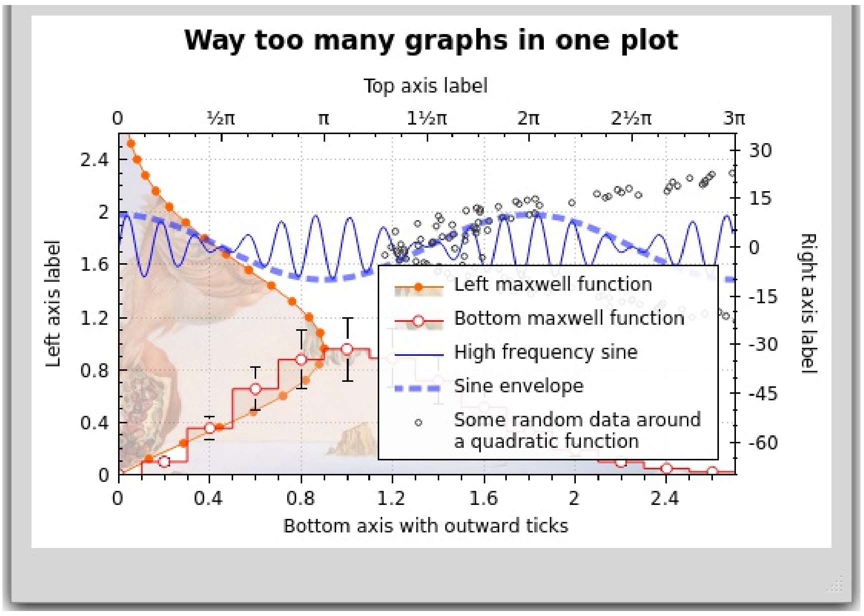
As you can see, applying a fill to a graph is as easy as setting a brush that is not `Qt::NoBrush`. The fill will go from the graph (here graph 0) to the zero-value-line parallel to the key (here x) axis. If we wanted a channel fill between this and another graph, we would additionally call `QCPGraph::setChannelFillGraph(otherGraph)`. To remove the channel fill, just pass `0` as other graph, and the fill will reach all the way to the zero-value-line as before. To remove the fill completely, call `QCPGraph::setBrush(Qt::NoBrush)`.

Plotting with multiple axes and more advanced styling

Now, let's look at a more complex example for creating the demo screenshot which contains the five graphs on four axes, textured filling, vertical error bars, a legend, dots as decimal separators etc.

```
customPlot->setLocale(QLocale(QLocale::English, QLocale::UnitedKingdom)); // period as decimal separator and comma as thousand separator
customPlot->legend->setVisible(true);
QFont legendFont = font();  // start out with MainWindow's font..
legendFont.setPointSize(9); // and make a bit smaller for legend
customPlot->legend->setFont(legendFont);
customPlot->legend->setBrush(QBrush(QColor(255,255,255,230)));
// by default, the legend is in the inset layout of the main axis rect. So this is how we access it to change legend placement:
customPlot->axisRect()->insetLayout()->setInsetAlignment(0, Qt::AlignBottom|Qt::AlignRight);

// setup for graph 0: key axis left, value axis bottom
// will contain left maxwell-like function
customPlot->addGraph(customPlot->yAxis, customPlot->xAxis);
customPlot->graph(0)->setPen(QPen(QColor(255, 100, 0)));
customPlot->graph(0)->setBrush(QBrush(QPixmap("./dali.png"))); // fill with texture of specified png-image
customPlot->graph(0)->setLineStyle(QCPGraph::lsLine);
customPlot->graph(0)->setScatterStyle(QCPScatterStyle(QCPScatterStyle::ssDisc, 5));
customPlot->graph(0)->setName("Left maxwell function");

// setup for graph 1: key axis bottom, value axis left (those are the default axes)
// will contain bottom maxwell-like function
customPlot->addGraph();
customPlot->graph(1)->setPen(QPen(Qt::red));
customPlot->graph(1)->setBrush(QBrush(QPixmap("./dali.png"))); // same fill as we used for graph 0
customPlot->graph(1)->setLineStyle(QCPGraph::lsStepCenter);
customPlot->graph(1)->setScatterStyle(QCPScatterStyle(QCPScatterStyle::ssCircle, Qt::red, Qt::white, 7));
customPlot->graph(1)->setErrorType(QCPGraph::etValue);
customPlot->graph(1)->setName("Bottom maxwell function");

// setup for graph 2: key axis top, value axis right
// will contain high frequency sine with low frequency beating:
customPlot->addGraph(customPlot->xAxis2, customPlot->yAxis2);
customPlot->graph(2)->setPen(QPen(Qt::blue));
customPlot->graph(2)->setName("High frequency sine");

// setup for graph 3: same axes as graph 2
// will contain low frequency beating envelope of graph 2
customPlot->addGraph(customPlot->xAxis2, customPlot->yAxis2);
QPen blueDotPen;
blueDotPen.setColor(QColor(30, 40, 255, 150));
blueDotPen.setStyle(Qt::DotLine);
blueDotPen.setWidthF(4);
customPlot->graph(3)->setPen(blueDotPen);
customPlot->graph(3)->setName("Sine envelope");

```

As you can see, you can define freely which axis should play which role for a graph. Graph with index 0 for example uses the left axis (`yAxis`) as its key and the bottom axis (`xAxis`) as its value. Consequently the graph is *standing upward* against the left axis:



In order to apply error bars for graph 1, we need to enable them via `QCPGraph::setErrorType`. Its argument is used to specify whether the error bars are for the value, the key, both or none dimensions. Then we call one of the many `QCPGraph::setData` functions which take the arguments we want. Here they are keys (`x1`), values (`y1`) and value errors (`y1err`). For further explanation of the used methods, have a look at the extensive documentation.

Plotting date and time data

Next, we'll look at how to plot date and/or time related data. It basically comes down to two additional function calls to tell an axis, it should output the labels as dates/times in lines 28 and 29 :

```
1   // set locale to english, so we get english month names:
2   customPlot->setLocale(QLocale(QLocale::English, QLocale::UnitedKingdom));
3   // seconds of current time, we'll use it as starting point in time for data:
4   double now = QDateTime::currentDateTime().toTime_t();
5   srand(8); // set the random seed, so we always get the same random data
6   // create multiple graphs:
7   for (int gi=0; gi<5; ++gi)
8   {
9     customPlot->addGraph();
10    QPen pen;
11    pen.setColor(QColor(0, 0, 255, 200));
12    customPlot->graph()->setLineStyle(QCPGraph::lsLine);
13    customPlot->graph()->setPen(pen);
14    customPlot->graph()->setBrush(QBrush(QColor(255/4.0*gi,160,50,150)));
15    // generate random walk data:
16    QVector<double> time(250), value(250);
17    for (int i=0; i<250; ++i)
18    {
19      time[i] = now + 24*3600*i;
20      if (i == 0)
21        value[i] = (i/50.0+1)*(rand()/(double)RAND_MAX-0.5);
22      else
23        value[i] = fabs(value[i-1])*(1+0.02/4.0*(4-gi)) + (i/50.0+1)*(rand()/(double)RAND_MAX-0.5);
24    }
25    customPlot->graph()->setData(time, value);
26  }
27  // configure bottom axis to show date and time instead of number:
28  customPlot->xAxis->setTickLabelType(QCPAxis::ltDateTime);
29  customPlot->xAxis->setDateTimeFormat("MMMM\nyyyy");
30  // set a more compact font size for bottom and left axis tick labels:
31  customPlot->xAxis->setTickLabelFont(QFont(QFont().family(), 8));
32  customPlot->yAxis->setTickLabelFont(QFont(QFont().family(), 8));
33  // set a fixed tick-step to one tick per month:
34  customPlot->xAxis->setAutoTickStep(false);
35  customPlot->xAxis->setTickStep(2628000); // one month in seconds
36  customPlot->xAxis->setSubTickCount(3);
37  // apply manual tick and tick label for left axis:
38  customPlot->yAxis->setAutoTicks(false);
39  customPlot->yAxis->setAutoTickLabels(false);
40  customPlot->yAxis->setTickVector(QVector<double>() << 5 << 55);
41  customPlot->yAxis->setTickVectorLabels(QVector<QString>() << "Not so\nhigh" << "Very\nhigh");
42  // set axis labels:
43  customPlot->xAxis->setLabel("Date");
44  customPlot->yAxis->setLabel("Random wobbly lines value");
45  // make top and right axes visible but without ticks and labels:
```

The string you pass to `QCPAxis::setDateTimeFormat()` has the same date formatting options as the string passed to `QDateTime::toString`, see Qt docs. All date/times are handled as seconds since midnight 1. January 1970, UTC. This is the format you use, when calling `QDateTime::toTime_t` or `setTime_t` on the Qt date/time classes. For sub-second accuracy, you can use `QDateTime::toMSecsSinceEpoch()/1000.0`, which results in a double value representing the same timespan as `toTime_t` returns, but with millisecond accuracy. For more of those demo codes, see the examples in the downloadable package.

Beyond Graphs: Curves, Bar Charts, Statistical Box Plot,...

Up to now we've only looked at graphs. Since they are such a dominant use case, QCustomPlot offers a specialized interface for them. We've been using it all the time: `QCustomPlot::addGraph`, `QCustomPlot::graph` etc. But that's not the the whole story. QCustomPlot has a more general interface for *classes that draw data inside the plot*, I call them *Plottables*. This interface is built around the abstract base class *QCPAbstractPlottable*. All Plottables derive from this class, also the familiar QCPGraph class. QCustomPlot offers many other plottable classes:

- **QCPGraph**: That's the plottable class we've been using. Displays a series of data points as a graph with different line styles, filling, scatters and error bars.
- **QCPCurve**: Similar to QCPGraph with the difference that it's made for displaying parametric curves. Unlike function

graphs, they may have *loops*.

- **QCPBars**: A Bar Chart. Takes a series of data points and represents them with bars. If there are multiple QCPBars plottables in the plot, they can be stacked on top of each other, as shown in the screenshot on the introduction page.
- **QCPStatisticalBox**: A Statistical Box Plot. Takes a five-number-summary (minimum, lower quartile, median, upper quartile, maximum) and represents it as a statistical box. Outliers can also be displayed.
- **QCPColorMap**: A 2D map which visualizes a third data dimension by using a color gradient. The class QCPColorScale accompanies this plottable to visualize the data scale in the plot.
- **QCPFinancial**: A plottable which can be used to visualize for example stock price open, high, low, close information by either using Candlesticks or OHLC bars.

Unlike graphs, other plottables need to be created with `new` outside of QCustomPlot and then added with `QCustomPlot::addPlottable`. This means that there is no *addCurve* or *addBars* function in the way there is an *addGraph* function. QCustomPlot takes ownership of the passed plottable. Existing plottables can be accessed with `QCustomPlot::plottable(int index)`, the total number of plottables in the plot (including graphs) can be retrieved with `QCustomPlot::plottableCount`. Here's a quick example that creates a bar chart with three bars:

```
QCPBars *myBars = new QCPBars(customPlot->xAxis, customPlot->yAxis);
customPlot->addPlottable(myBars);
// now we can modify properties of myBars:
myBars->setName("Bars Series 1");
QVector<double> keyData;
QVector<double> valueData;
keyData << 1 << 2 << 3;
valueData << 2 << 4 << 8;
myBars->setData(keyData, valueData);
customPlot->rescaleAxes();
customPlot->replot();
```

More details about the other plottables can be found in the example project and the other tutorials. Further, each plottable type has a detailed description on the documentation page of the respective class.

Of course, it's absolutely possible to write your own plottable to make any data look *exactly* the way you need it. You should look at the QCPAbstractPlottable documentation for a guide how to start subclassing it. You can also look at the existing plottables to see how they work. For that purpose, I recommend QCPBars or QCPCurve for a start. QCPGraph is quite feature rich and thus might not be perfectly suited as a starting point.

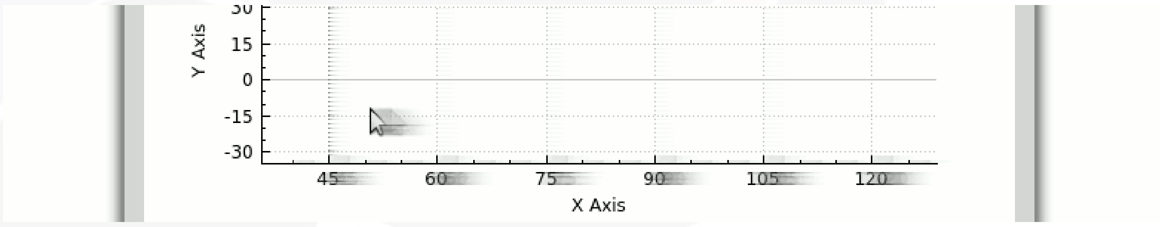
User interactions

QCustomPlot offers multiple built-in user interactions. They can be roughly categorized as

- Range manipulation by dragging with the mouse and scrolling the mouse wheel
- Selection of plot entities by clicking
- Signals emitted upon clicks of the user on plot entities

Range Manipulation

The default method for the user to manipulate the axis range is by performing a drag operation on the respective QCPAxisRect.

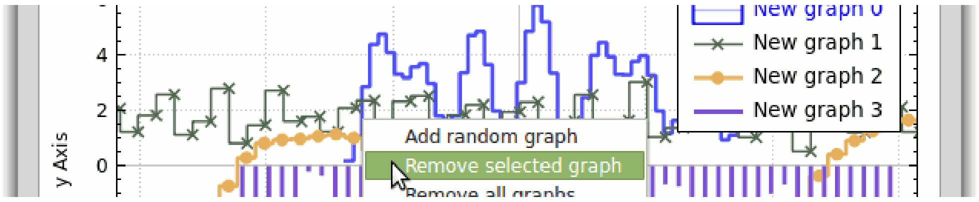


To enable range dragging in a QCustomPlot widget, the flag `QCP::iRangeDrag` needs to be added to the currently allowed interactions. This can be done with `customPlot->setInteraction(QCP::iRangeDrag, true)`. To only allow dragging in one orientation, use `QCPAxisRect::setRangeDrag` and specify `Qt::Vertical` or `Qt::Horizontal`. The default ist to allow both directions with `Qt::Vertical | Qt::Horizontal`.

During the drag operation, the axes configured via `QCPAxisRect::setRangeDragAxes` update their ranges in realtime, automatically causing replots. This gives the user the impression of moving the plot coordinate plane by grabbing it with the mouse. Initially, the range drag axes are configured to be the rect's bottom and left axes. For the default axis rect of the QCustomPlot widget, those are `QCustomPlot::xAxis` and `QCustomPlot::yAxis`.

To change the size of the range, i.e. zoom in or out of the plot, the user may use the mouse wheel. This behaviour is controlled with the interaction flag `QCP::iRangeZoom` which also needs to be activated with `QCustomPlot::setInteraction`. Just like the range drag, the zoom may also be selective with respect to the affected axes and orientations, see the functions `QCPAxisRect::setRangeZoomAxes` and `QCPAxisRect::setRangeZoom`. Additionally the scaling strength can be controlled with `QCPAxisRect::setRangeZoomFactor`. On common mouse hardware, one mouse wheel step corresponds to this factor applied to the axis range. If the factor is greater than one, scrolling the mouse wheel forwards decreases the range (zooms in) and scrolling backwards increases it (zooms out). To invert this behaviour, set mouse wheel zoom factors smaller than one (but greater zero). The scaling is always centered around the current mouse cursor position in the plot. This means pointing the cursor on a feature of interest and scrolling the mouse wheel allows zooming into that feature.

The selection mechanism



QCustomPlot offers a selection mechanism that allows the user to select potentially every component in the plot, like axes and graphs. Whether a certain category of entities is generally selectable in the plot can be controlled with the interaction flags that start with `QCP::iSelect(...)`. For example, setting `customPlot->setInteraction(QCP::iSelectPlottables, true)` will allow the user to select plottables (e.g. graphs) by clicking on them. Have a look at the QCP::Interaction documentation for all interaction flags.

To allow multiple objects to be selected simultaneously, set the `QCP::iMultiSelect` interaction flag. The user may then select multiple objects in succession by holding the multi-select-modifier (see `QCustomPlot::setMultiSelectModifier`), which is *Ctrl* by default.

Controlling individual selectability and selection state

The selectability can further be fine-tuned with `setSelectable` functions on individual objects. For example, if a specific graph in the plot shall not be selectable by the user, call `thatGraph->setSelectable(false)`. The selected state can be modified programmatically via `setSelected` functions. Changing the selection state programmatically is possible even if the selectability for the user is disabled.

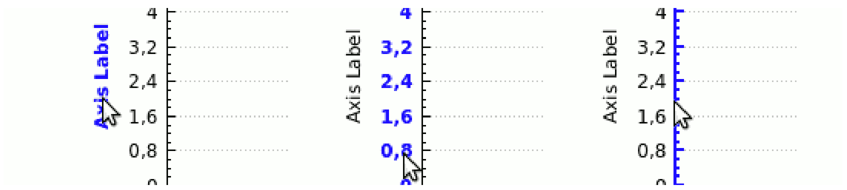
To deselect all objects in the plot, call `QCustomPlot::deselectAll`.

Appearance of selected objects

A selected object is usually displayed with a different pen, brush or font. This can be configured with methods like `QCPGraph::setSelectedPen`, `QCPGraph::setSelectedBrush`, `QCPAxis::setSelectedLabelFont`, `QCPAxis::setSelectedBasePen`, `QCPItemText::setSelectedColor`, just to name a few. As can be seen, they are named like the original (non-selected) property, but with the prefix "Selected".

Multi-Part objects

Some objects such as axes and legends have a more complex appearance such that a single boolean for selection isn't sufficient. In those cases, both selectability and selection state is an or-combination of `SelectablePart` flags (the respective QFlags type is called `SelectableParts`). Each multi-part object defines its own `SelectablePart` type.



For example, QCPAxis is conceptually made of three parts: the axis backbone with tick marks, the tick labels (numbers), and the axis label. And since those three parts shall be selectable individually, `QCPAxis::SelectablePart` defines `QCPAxis::spNone`, `QCPAxis::spAxis`, `QCPAxis::spTickLabels`, and `QCPAxis::spAxisLabel`. To make the axis backbone and the tick labels selectable, but not the axis label, call `theAxis->setSelectableParts(QCPAxis::spAxis|QCP::spTickLabels)`. To control the current selection state of a multi-part object, use the `QCPAxis::setSelectedParts` method.

Reacting to a selection change

Upon a selection change, each object emits a signal called `selectionChanged`. It does not matter whether the change was caused by the user or programmatically by a call of `setSelected`/`setSelectedParts`.

If a selection in the plot is changed by user interaction, the QCustomPlot-wide signal `QCustomPlot::selectionChangedByUser` is emitted. In slots connected to this signal, you may check the selection state of certain objects and react accordingly. The methods `QCustomPlot::selectedPlottables`, `selectedItems`, `selectedAxes`, and `selectedLegends` may be useful here to retrieve the selected objects of a certain kind.

User interaction signals

Independent of the selection mechanism, QCustomPlot emits various signals upon user interaction. The most low-level ones are the `QCustomPlot::mouseDoubleClick`, `mousePress`, `mouseMove`, `mouseRelease`, and `mouseWheel` signals. They are emitted when the corresponding event of the QCustomPlot widget fires. Note that the cleanest way would be to subclass QCustomPlot and reimplement the event methods (inherited from QWidget) with the same names. However, these signals allow easier access to user interactions for simple tasks, if you don't want to subclass QCustomPlot.

There also are higher level signals that report clicks and doubleclicks of certain objects in the plot: `QCustomPlot::plottableClick`, `plottableDoubleClick`, `itemClick`, `itemDoubleClick`, `axisClick`, `axisDoubleClick`, `legendClick`, `legendDoubleClick`, `titleClick`, and `titleDoubleClick`. All those signals report which object was clicked (and which part, if it's a multi-part object), as well as the associated QMouseEvent.

The examples in the full package include one project which makes extensive use of various aspects of the interaction system. It also demonstrates how to fine-tune the behaviour to fit ones needs.

Items: Supplementary graphical elements

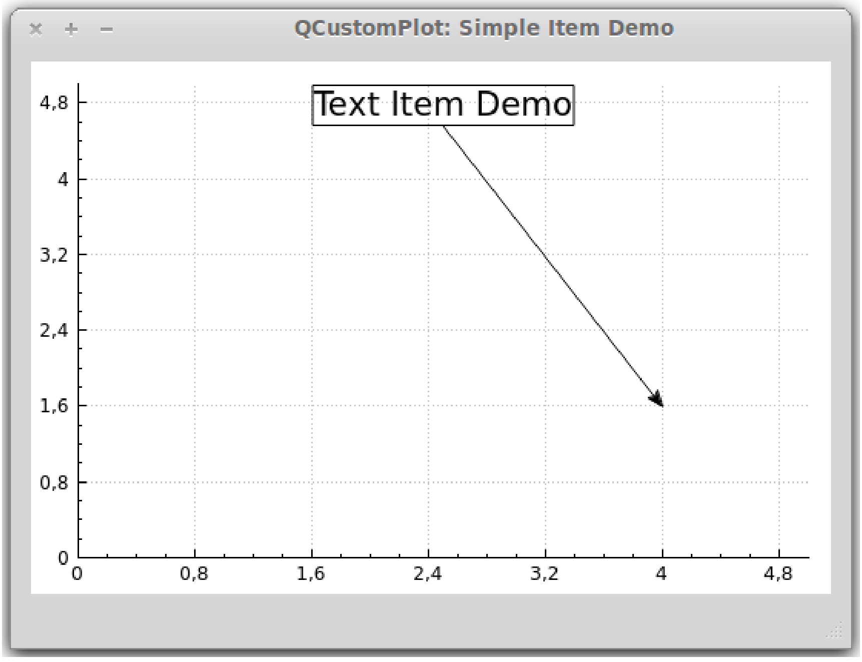
QCustomPlot allows placing and anchoring of graphical elements such as text, arrows, lines, rectangles, arbitrary pixmaps etc. on the plot. They are based on the abstract base class QCPAbstractItem. A detailed description of the item mechanism and what built-in items are currently available can be found in the documentation of QCPAbstractItem.

Basic example using arrow and text

This example shows how to create a text label that is always positioned at the top of the axis rect and an arrow that connects a point in plot coordinates with that label.

```
// add the text label at the top:
QCPItemText *textLabel = new QCPItemText(customPlot);
customPlot->addItem(textLabel);
textLabel->setPositionAlignment(Qt::AlignTop|Qt::AlignHCenter);
textLabel->position->setType(QCPItemPosition::ptAxisRectRatio);
textLabel->position->setCoords(0.5, 0); // place position at center/top of axis rect
textLabel->setText("Text Item Demo");
textLabel->setFont(QFont(font().family(), 16)); // make font a bit larger
textLabel->setPen(QPen(Qt::black)); // show black border around text

// add the arrow:
QCPItemLine *arrow = new QCPItemLine(customPlot);
customPlot->addItem(arrow);
arrow->start->setParentAnchor(textLabel->bottom);
arrow->end->setCoords(4, 1.6); // point to (4, 1.6) in x-y-plot coordinates
arrow->setHead(QCPLineEnding::esSpikeArrow);
```

Notice that even when the plot range is dragged, the arrow head stays attached to the plot coordinate (4, 1.6) and rotates/stretches accordingly. This is achieved by the flexibility of QCustomPlot's item positioning. Items may be positioned in plot coordinates, in absolute pixel coordinates and in fractional units of the axis rect size. The documentation of `QCPAbstractItem` and `QCPItemPosition` goes into more detail about how to use these different possibilities.



As with plottables, it is easy to create own items, too. This can be done by making your own subclass of QCPAbstractItem. See the subclassing section in the documentation of QCPAbstractItem.

Item clipping

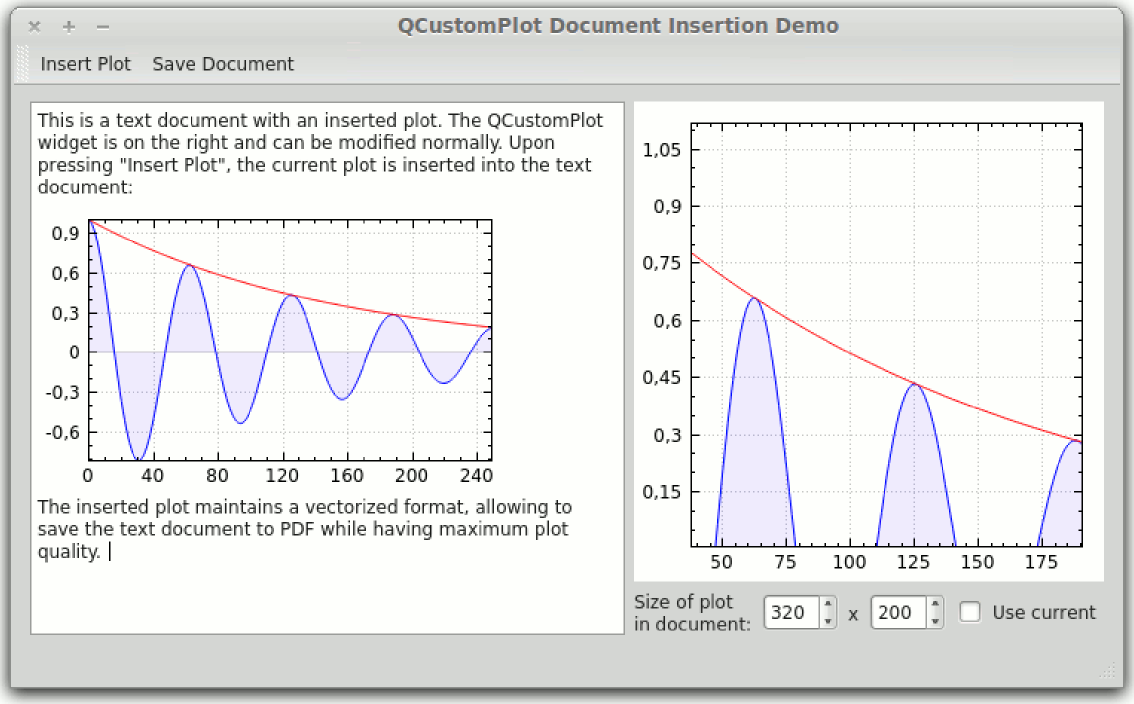
Items are by default clipped to the main axis rect, this means they are only visible inside the axis rect. To make an item visible outside that axis rect, disable clipping by calling `setClipToAxisRect(false)`.

On the other hand if you want the item to be clipped to a *different* axis rect, you can specify it via `setClipAxisRect`. This `clipAxisRect` property of an item is only used for clipping behaviour, and in principle is independent of the coordinate axes the item might be tied to via its position members (see `QCPItemPosition::setAxes`). However, it is common that the axis rect for clipping also contains the axes used for the item positions.

Embedding plots in a QTextDocument

Report generation often requires inserting plots and charts inside a text document. This tutorial demonstrates how QCustomPlot can interact with QTextDocument, to achieve this easily.

The example project accompanying this tutorial is called *text-document-integration* and is part of the full package download.


QCustomPlot Document Insertion Demo
Insert Plot
Save Document
Size of plot in document:
320
x
200
Use current

This is a text document with an inserted plot. The QCustomPlot widget is on the right and can be modified normally. Upon pressing "Insert Plot", the current plot is inserted into the text document:

The inserted plot maintains a vectorized format, allowing to save the text document to PDF while having maximum plot quality.

QCPDocumentObject

The interface between QCustomPlot and QTextDocument is `QCPDocumentObject`. Note that this class is not in the standard *qcustomplot.cpp/.h* files but is defined in *qcpdocumentobject.cpp/.h* in the example project of this tutorial.

It serves two purposes:

- Generation of a text char format from a QCustomPlot. This allows inserting plots into the QTextDocument, e.g. at the position of the cursor.
- Rendering of the static plot inside the QTextDocument, whenever it is redrawn or exported.

In case you are wondering what a text char format is good for: Inserting a `QChar::ObjectReplacementCharacter` with a custom format is the way Qt allows insertion of custom objects into a text document.

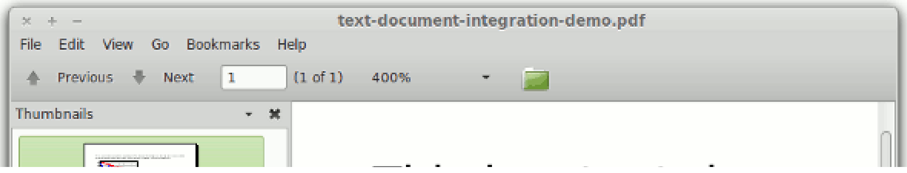
let's assume our QCustomPlot is `ui->plot`, and our QTextEdit with the text document is `ui->textEdit`. The first step is to register the QCPDocumentObject as a handler for plot objects in the text document:

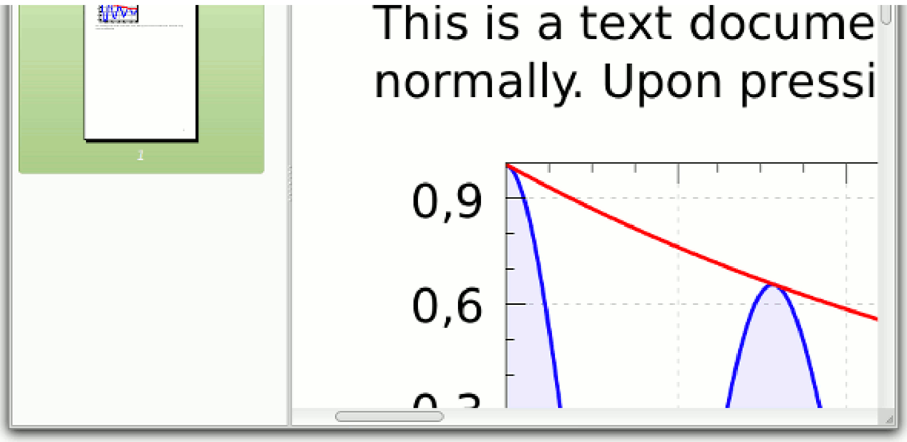
```
// register the plot document object (only needed once, no matter how many plots will be in the QTextDocument):
QCPDocumentObject *interface = new QCPDocumentObject(this);
ui->textEdit->document()->documentLayout()->registerHandler(QCPDocumentObject::PlotTextFormat, interface);
```

After this call, we can start inserting plots into the text document. This is what the static method `QCPDocumentObject::generatePlotFormat(QCustomPlot *plot, int width, int height)` is good for. It takes a vectorized snapshot of the *plot* with the given *width* and *height* (if left to be 0, the current width and height of the plot is used) and attaches it to a QTextCharFormat. The returned QTextCharFormat can subsequently be used to format a `QChar::ObjectReplacementCharacter`, which then appears as the plot object. The insertion of a plot at the current cursor position can thus be done as follows:

```
QTextCursor cursor = ui->textEdit->textCursor();

// insert the current plot at the cursor position. QCPDocumentObject::generatePlotFormat creates a
// vectorized snapshot of the passed plot (with the specified width and height) which gets inserted
// into the text document.
double width = ui->cbUseCurrentSize->isChecked() ? 0 : ui->sbWidth->value();
double height = ui->cbUseCurrentSize->isChecked() ? 0 : ui->sbHeight->value();
cursor.insertText(QString(QChar::ObjectReplacementCharacter), QCPDocumentObject::generatePlotFormat(ui->plot, width, height));

ui->textEdit->setTextCursor(cursor);
```

The components `cbUseCurrentSize`, `sbWidth` and `sbHeight` are part of the user interface of the example project. As stated, the plot object inside the text document maintains its vectorized nature. So exporting it to PDF (or other formats capable of vectorized content) results in a maximum quality output. Saving the above document to a PDF file and opening it in a PDF viewer results in the following display


text-document-integration-demo.pdf
File Edit View Go Bookmarks Help
Previous
Next
1
(1 of 1)
400%
Thumbnails

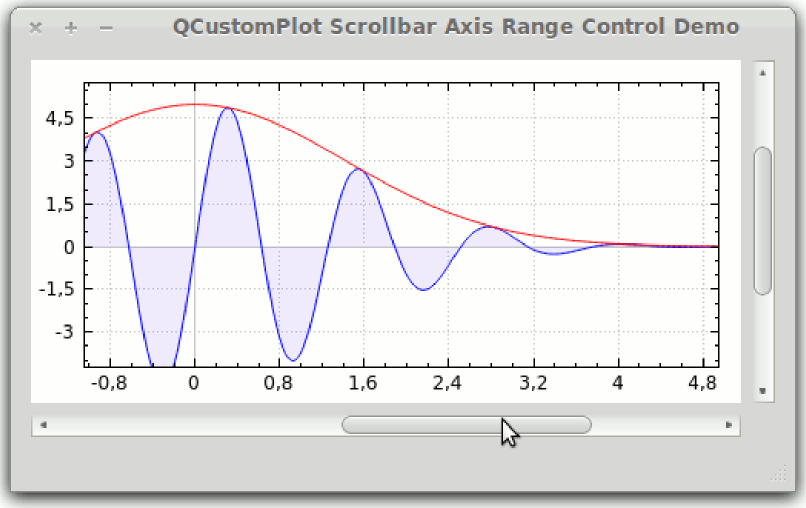


As can be seen, zooming in to the inserted plot reveals smooth lines.

Controlling the axis range with a scrollbar

While the most intuitive way of controlling the axis ranges is the range dragging and zooming mechanism, it might be desirable to also provide scrollbars for this purpose. This can be achieved by connecting an axis with a scrollbar via signals and slots. An intermediate slot is needed to translate between the `QCPRange` of an axis and the integer `value` of the scrollbar.

The example project accompanying this tutorial is called *scrollbar-axis-range-control* and is part of the full package download.



Preliminaries

The signals that will be relevant to propagate changes back and forth between scrollbar and axis are `QScrollBar::valueChanged(int)` and `QCPAxis::rangeChanged(QCPRange)`. Since we want to keep the normal range dragging and zooming, the scrollbar slider position and size must be updated when the axis' `rangeChanged` signal is emitted.

QScrollBar is integer based. For this reason, we need a factor that transforms the integer scrollbar values to axis coordinates. For example, if we want to be able to *smoothly* scroll the axis over the coordinate range -5 to 5, we could set the factor to something like 0.01 (i.e. divide scrollbar values by 100) and thus the range of the scrollbar to -500..500.

```
ui->horizontalScrollBar->setRange(-500, 500);
ui->verticalScrollBar->setRange(-500, 500);
```

If the accessible coordinate range shall change at any point, just change the maximum/minimum values of the scrollbar.

The intermediate slots that will do the coordinate transformations are called `horzScrollBarChanged`, `vertScrollBarChanged`, `xAxisChanged`, and `yAxisChanged`. They are connected to the appropriate signals of the scrollbars and x-/y-Axes:

```
connect(ui->horizontalScrollBar, SIGNAL(valueChanged(int)), this, SLOT(horzScrollBarChanged(int)));
connect(ui->verticalScrollBar, SIGNAL(valueChanged(int)), this, SLOT(vertScrollBarChanged(int)));
connect(ui->plot->xAxis, SIGNAL(rangeChanged(QCPRange)), this, SLOT(xAxisChanged(QCPRange)));
connect(ui->plot->yAxis, SIGNAL(rangeChanged(QCPRange)), this, SLOT(yAxisChanged(QCPRange)));
```

The coordinate transformation slots

Both types of slots (*axis range to scrollbar* and *scrollbar to axis range*) are fairly simple. They take the changed value of the scrollbar or axis, apply the transformation and set the result to the axis or scrollbar, respectively. These are the slots for updating the axis ranges upon moving the scrollbar slider:

```
void MainWindow::horzScrollBarChanged(int value)
{
  if (qAbs(ui->plot->xAxis->range().center()-value/100.0) > 0.01) // if user is dragging plot, we don't want to replot twice
  {
    ui->plot->xAxis->setRange(value/100.0, ui->plot->xAxis->range().size(), Qt::AlignCenter);
    ui->plot->replot();
  }
}

void MainWindow::vertScrollBarChanged(int value)
{
  if (qAbs(ui->plot->yAxis->range().center()+value/100.0) > 0.01) // if user is dragging plot, we don't want to replot twice
  {
    ui->plot->yAxis->setRange(-value/100.0, ui->plot->yAxis->range().size(), Qt::AlignCenter);
    ui->plot->replot();
  }
}
```

There are two things worth mentioning:

First of all, we see here the transformation of the scrollbar `value` to axis coordinates by dividing by `100.0`. Also note that the vertical scrollbar has a low value when the slider is at the top and a high value when it is at the bottom. For plot axes this is the other way around, which is why a minus sign is added to expressions containing `value` of the vertical scrollbar, e.g. when setting the `yAxis` range.

The condition `qAbs(ui->plot->xAxis->range().center()-value/100.0) > 0.01` is necessary such that range dragging doesn't cause double replots, caused by a back-and-forth between change-signals and slots. This could happen because upon range dragging, the QCustomPlot automatically replots itself and emits the `rangeChanged` signals of the dragged axes. In this application the `rangeChanged` signal will call the slot `xAxisChanged` or `yAxisChanged` which, as we will see, updates the scrollbar slider position by calling the scrollbar's `setValue` method. This method in turn emits the scrollbar's `valueChanged` signal which is connected to the slots above. Here, the second replot would happen, if the check wasn't in place. The check makes sure the replot is only performed if the current axis range is actually different from the new (transformed) scrollbar value. This is not the case if the user dragged the axis range, so the redundant replot and axis range update is skipped.

The slots for updating the scrollbars upon axis range changes are simple:

```
void MainWindow::xAxisChanged(QCPRange range)
{
  ui->horizontalScrollBar->setValue(qRound(range.center()*100.0)); // adjust position of scroll bar slider
  ui->horizontalScrollBar->setPageStep(qRound(range.size()*100.0)); // adjust size of scroll bar slider
}

void MainWindow::yAxisChanged(QCPRange range)
{
  ui->verticalScrollBar->setValue(qRound(-range.center()*100.0)); // adjust position of scroll bar slider
  ui->verticalScrollBar->setPageStep(qRound(range.size()*100.0)); // adjust size of scroll bar slider
}
```

They simply transform the range center to a scrollbar value and the the range size to the scrollbar's page step (the scrollbar slider size).