# Das Qwt Entwicklerhandbuch

Andreas Nicolai andreas.nicolai@gmx.de

Version 2.0, Mai 2025

# Inhaltsverzeichnis


**Über dieses Handbuch** . . . **1**
**1. Überblick über die Qwt Bibliothek** . . . **2**
1.1. Entwicklungsgeschichte . . . 2
1.1.1. Download der Bibliothek . . . 2
1.2. Widget-Konzept und Erscheinungsbild . . . 2
1.3. Besitzer/Eigentümer-Konzept des QwtPlot-Widgets . . . 3
1.4. Zeichenobjekte und deren Achsenabhängigkeit . . . 3
1.5. Vererbungskonzept . . . 3
1.6. Qwt Designer Plugins . . . 3
**2. Erste Schritte und ein interaktives Diagramm** . . . **5**
2.1. Programmrohbau . . . 5
2.1.1. QMake Projektdatei . . . 5
2.1.2. Minimalistisches Hauptprogramm . . . 5
2.2. Diagrammelemente hinzufügen . . . 6
2.2.1. Linie hinzufügen . . . 6
2.2.2. Legende hinzufügen . . . 8
2.2.3. Diagrammtitel hinzufügen . . . 8
2.2.4. Diagrammraster hinzufügen . . . 8
2.2.5. Achsenkonfiguration . . . 9
2.2.6. Logarithmische Achsen . . . 10
2.2.7. Markierungslinien . . . 10
2.3. Interaktion mit dem Diagramm . . . 11
2.3.1. Zoomfunktionalität mit QwtPlotZoomer . . . 11
2.3.2. Plotausschnitt verschieben mit QwtPlotPanner . . . 12
**3. QWT Widgets und Eingabekomponenten** . . . **13**
3.1. Schieberegler (Slider) . . . 13
3.2. Drehräder/Einstellräder und . . . 13
3.3. Drehknöpfe . . . 14
3.4. Analoge Zeiger-Anzeigen . . . 14
**4. Allgemeine Grundlagen des QwtPlot** . . . **15**
4.1. Aufbau und Elemente der Diagrammkomponente . . . 15
4.1.1. Achsen und Koordinatensystem . . . 15
4.1.2. Diagramm-/Zeichenelemente . . . 15
4.1.3. Hinzufügen/Entfernen von Zeichenelementen . . . 16
4.2. Datenhaltung im QwtPlot / QwtSeriesStore . . . 17
4.3. Automatisches Zeichnen oder Zeichnen bei Bedarf . . . 18
**5. Kurvendiagramme** . . . **19**
5.1. Datenübergabe . . . 19
5.2. Kurvenarten . . . 20
5.2.1. Liniendiagramm . . . 20
5.2.2. Stäbchen (Sticks) . . . 20
5.2.3. Stufendiagramme . . . 21
5.2.4. Punkte (Dots) . . . 22
5.2.5. Keine Linie . . . 24
5.3. Symbole/Punkte . . . 24
5.3.1. Symbolstile/Eingebauten Symbolformen . . . 25
5.3.2. Nutzerdefinierte Formen via QPainterPath . . . 26
5.3.3. SVG-Symbole . . . 26
5.3.4. Bild-Symbole (Pixmaps) . . . 28
5.4. Ausgefüllte Kurven . . . 28
5.5. Legendeneinträge . . . 29
5.6. Zeichenattribute, Zeichengeschwindigkeit und Performanceoptimierung bei großen Datenreihen . . . 31
5.6.1. Zeichenattribute (PaintAttribute / PaintAttributes) . . . 31
5.6.2. Punktfilter . . . 32
5.6.3. Aggressives Punktefiltern . . . 33
5.6.4. Polygon-Clipping . . . 36
5.7. Kurvenfilter/Kurvenanpasser . . . 37
5.7.1. Kurvenglättung/Spline-Interpolation . . . 37

5.7.2. PolarCurveFitter . . . 39
5.7.3. Datenreduktionsfilter . . . 39
**6. Intervallkurven . . . 46**
6.1. Gestapelte (Intervall-)Kurven/Flächendiagramme . . . 48
**7. Balkendiagramme . . . 49**
7.1. Grundlegende Eigenschaften der Plots . . . 49
7.2. Bezugslinie . . . 49
7.3. Layout und Abstände . . . 50
7.3.1. AutoAdjustSamples . . . 50
7.3.2. ScaleSamplesToAxes . . . 51
7.3.3. ScaleSampleToCanvas . . . 51
7.3.4. Layout-Empfehlung . . . 52
7.4. Balkenformen und Farben . . . 52
7.5. Balkenbeschriftung auf der X-Achse . . . 53
7.6. Balkenbeschriftungen . . . 54
7.7. Mehrfarbige Balken . . . 55
7.8. Legendeneinträge . . . 56
7.9. Gestapelte Balkendiagramme oder Balkendiagramme mit mehreren Balken pro Gruppe . . . 57
**8. Legende . . . 58**
8.1. Außenseitige Legende . . . 58
8.2. Legenden-Zeichenelement . . . 58
8.3. Eigene Legenden-Icons . . . 58
**9. Markierungslinien . . . 60**
**10. Plotachsen . . . 61**
10.1. Allgemeine Achsenformatierung . . . 61
10.2. Skalen . . . 61
**11. QwtText und Sonderformatierungen . . . 62**
11.1. MathML . . . 62
**12. Interaktiver Zoom und Verschieben von Diagrammausschnitten . . . 63**
**13. Anpassung/Styling der Qwt Komponenten . . . 64**
13.1. Allgemeines zu Farbpaletten . . . 64
13.2. Rahmen und Zeichenfläche des Diagramms . . . 64
13.2.1. Farbe und Rahmen des Plots . . . 64
13.2.2. Zeichenfläche . . . 64
**14. Exportieren und Drucken . . . 66**
14.1. Exportieren des Plots als Pixelgrafik . . . 66
14.1.1. Erstellen einer 1-zu-1 Kopie des Plotwidgets . . . 66
14.1.2. Kopie in die Zwischenablage . . . 66
14.1.3. QwtPlot mit anderer Auflösung abspeichern . . . 67
14.1.4. Drucken . . . 67
14.1.5. PDF Export mittels QPdfWriter . . . 68
14.1.6. SVG Export mittels QSvgDocument . . . 68
14.1.7. EMF Export unter Windows . . . 69
14.1.8. Anpassen des gerenderten Plots . . . 69
14.1.9. Diagrammelemente skalieren (DPI ändern) . . . 71
**15. Fortgeschrittene Themen . . . 74**
15.1. Objekte aus dem QwtPlot loslösen . . . 74
15.2. Splines und Bézier-Kurven . . . 74
**16. Download/Installation/Erstellung der Qwt Bibliothek . . . 78**
16.1. Download fertiger Pakete . . . 78
16.1.1. Windows/Mac . . . 78
16.1.2. Linux . . . 78
16.2. Erstellung aus dem Quelltext . . . 78
16.2.1. Windows . . . 78
16.2.2. Linux/Mac . . . 79
16.3. Qt Designer Plugins . . . 79
16.4. Verwendung des Plots in eigenen Programmen . . . 79
16.4.1. Windows . . . 79
16.4.2. Linux/Mac . . . 79
16.5. Das QwtPlot in eine Designer-Oberfläche/ui-Datei integrieren . . . 79

16.5.1. Definition eines Platzhalterwidgets . . . . . 80
16.5.2. Verwendung der Designer-Plugins . . . . . 81
**17. Über den Autor** . . . . . **81**

# Über dieses Handbuch

Dieses Projekt ergänzt die API-Dokumentation zur Qwt-Bibliothek und bietet sowas wie ein Programmiererhandbuch mit vielen Details zu den Internas der Bibliothek. Der Fokus liegt aber ganz klar auf der `QwtPlot` Diagrammkomponente. Das ist übrigends schon die 2. Auflage (komplett neu überarbeitet), weil es zum Zeitpunkt der ersten Ausgabe noch kein tolles AsciiDoc gab und ich irgendwie beim Textschreiben hängengeblieben bin.

Die englische Version und die PDF-Variante gibt's hier: https://ghorwin.github.io/qwtbook.



Häufig wird das `QwtPlot` ja benutzt, um ein eigenes Postprozessing oder Visualisierungstool zu schreiben. Vielleicht lohnt sich hier der Blick auf das kostenfreie *PostProc 2*, welches wir an der TU Dresden entwickelt haben und das intern ein angepasstes und erweitertes `QwtPlot` verwendet. Das Programm ist spezialisiert auf die Visualisierung von dynamischen Simulationsergebnissen und Messdaten/Zeitreihen und man kann da einen guten Eindruck davon bekommen, was mit `QwtPlot` alles machbar ist. *Postproc 2* wird weiter aktiv gepflegt und kann hier kostenfrei heruntergeladen werden: https://bauklimatik-dresden.de/software/postproc2/.

Viel Spaß bei der Lektüre - und falls noch Inhalte fehlen, einfach Geduld haben und später wiederkommen (oder im Github-Repo ein Issue anlegen). Und schaut Euch vielleicht auch meine anderen Tutorials unter https://schneggenport.de an!

— Andreas Nicolai

# 1. Überblick über die Qwt Bibliothek

*Qwt - Qt Widgets for Technical Applications* ist eine Open-Source Bibliothek für technische Anwendungen und stellt bestimmte Widgets für Anzeigen und Kontrollkomponenten bereit. Die wohl wichtigste Komponente der Qwt Bibliothek ist das `QwtPlot`, eine sehr flexible und mächtige Diagrammkomponente.

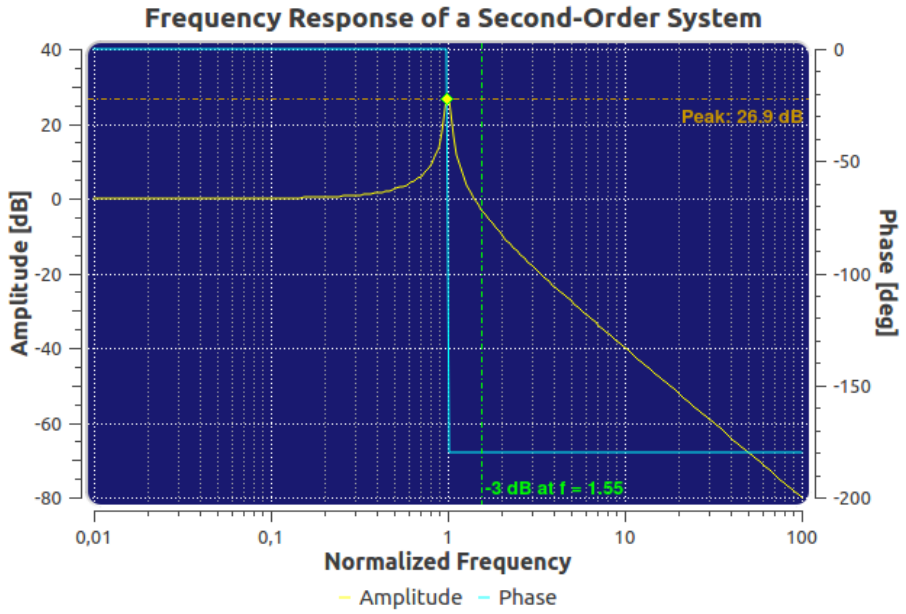



*Abbildung 1. Beispiel für die QwtPlot-Komponente*

Die Qwt Bibliothek steht unter einer Open-Source-Lizenz, wurde und wird aktiv vom Entwickler *Uwe Rathmann* gepflegt und wird auf SourceForge.net gehostet:

- Qwt Webseite (englisch)
- Qwt SourceForge Projektseite

## 1.1. Entwicklungsgeschichte

- die erste Version der Qwt-Bibliothek stammt noch aus dem Jahr 1997 von Josef Wilgen
- seit 2002 wird die Bibliothek von *Uwe Rathmann* entwickelt und gepflegt
- Version 5 ist wohl am weitesten verbreitet (erstes Release vom 26.02.2007)
- Version 6 (erstes Release vom 15.04.2011, kein Qt3 Support mehr) enthält wesentliche API-Änderungen
- aktuelle stabile Version 6.3.0 (Stand Mai 2025)
- im trunk gibt es zum Teil bereits wesentlich mehr und fortgeschrittene Funktionen

### 1.1.1. Download der Bibliothek

Die Qwt Bibliothek kann von der Qwt SourceForge Projektseite als Quelltextarchiv geladen werden. Unter Linux wird Qwt bei vielen Distributionen als Paket gehalten. Genau genommen gibt es mehrere Pakete für die unterschiedlichen Qwt-Bibliotheksversionen bzw. Qt Versionen. Details zur Installation und Verwendung der Bibliothek gibt es im Kapitel 16.

## 1.2. Widget-Konzept und Erscheinungsbild

Die Qwt Bibliothek liefert Komponenten, welche analog zu den normalen Qt-Widgets in Desktopanwendungen verwendet werden können. Die Komponenten verwenden die Qt Palette, sodass die Qwt-Widgets in die jeweilige Oberfläche passen. Dadurch integrieren sich die Widgets nahtlos in Programmoberflächen. Einzelne Komponenten des `QwtPlot` unterstützen auch Styles. So ermöglichen z.B. Abrundungseffekte beim Plot-Widget das Immitieren klassischer Anzeigen.

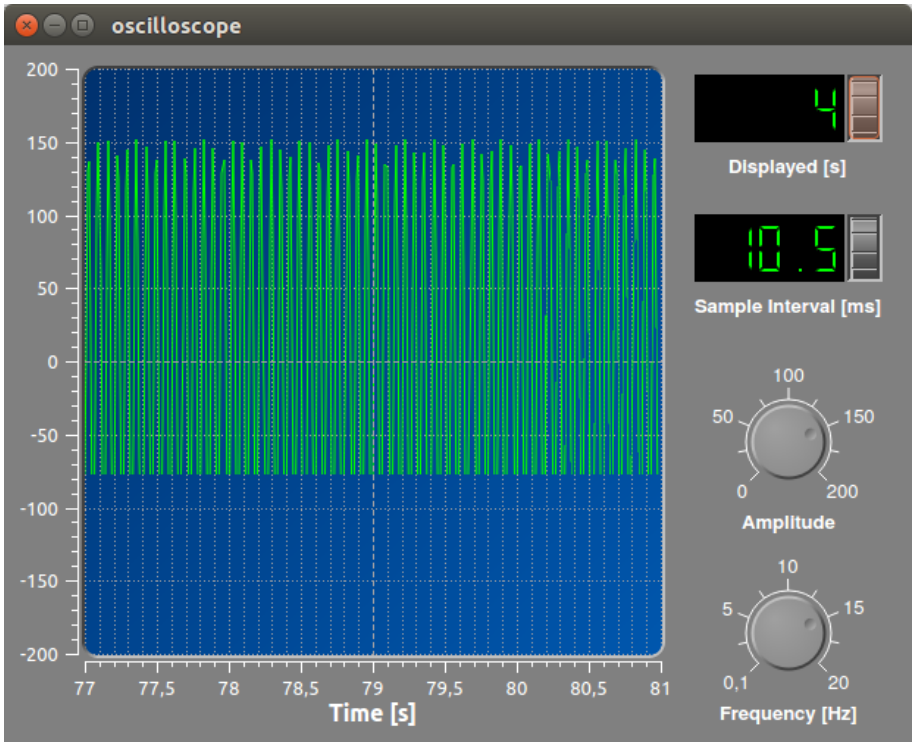



*Abbildung 2. QwtPlot mit abgerundeten Ecken*

Details zum Styling und zur Anpassung des Erscheinungsbildes sind im Kapitel 13 zu finden.

## 1.3. Besitzer/Eigentümer-Konzept des QwtPlot-Widgets

Eine grundlegende Eigenschaft der `QwtPlot`-Klasse ist die Besitzübername hinzugefügter Elemente. Dies gilt allgemein für alle Elemente des Plots (Linien, Marker, Legende, …). D.h. nach Übertragung der Eigentümerschaft kümmert sich das `QwtPlot` um das Aufräumen des Speichers.

Einmal hinzugefügte Elemente werden nicht wieder losgelöst werden (bzw. nur über einen Trick, wie im Kapitel 15.1 beschrieben wird). Daher ist es sinnvoll, bei veränderlichen Diagrammelementen einen Mechanismus zur jeweiligen Neuerstellung eines Zeichenobjekts vorzusehen (Factory-Konzept).

## 1.4. Zeichenobjekte und deren Achsenabhängigkeit

Ein wesentliches Designmerkmal beim `QwtPlot` ist die Möglichkeit, beliebige Zeichenobjekte (Kurven, Marker, Legende, …) dem Plot zu übergeben. Damit sich diese Zeichenobjekte (engl. *PlotItem*) am Koordinatengitter ausrichten können, wird ihnen eine Achsenabhängigkeit gegeben. Dadurch erhalten diese Zeichenobjekte eine Information, wann immer sich die Achsenskalierung ändert (durch Zoomen, oder Änderung der Wertebereiche etc.).

Diese Funktionalität definiert die zentrale Bedeutung der (bis zu) 4 Achsen im Diagramm. Deswegen sind diese auch fest im `QwtPlot` verankert und werden nicht wie andere Zeichenobjekte beliebig hinzugefügt.

## 1.5. Vererbungskonzept

Grundsätzlich ist das `QwtPlot` und die beteiligten Klassen auf maximale Anpassungsfähigkeit ausgelegt, d.h. es wird (fast) überall Polymorphie unterstützt. Wenn die eingebaute Funktionalität nicht zureichend ist, kann man einfach immer die entsprechende Klasse ableiten und die jeweils anzupassende Funktion re-implementieren und verändern. Dies wird anhand von Beispielen in den individuellen Kapiteln des Handbuchs beschrieben.

## 1.6. Qwt Designer Plugins

Die Qwt Bibliothek bringt Plugins für Qt Designer bzw. Qt Creator mit, welche das Einfügen von Qwt-Komponenten in ui-Dateien erleichtert. Es lassen sich jedoch keine QwtPlot-Eigenschaften festlegen oder Kurven hinzufügen. Die eigentliche Anpassung und Ausgestaltung des Plots erfolgt im Quelltext.

Die API der Qt Designer Plugins hat sich in jüngeren Qt Creator-Versionen geändert, weswegen die Qwt Designer Plugins mit aktuellen Qt und Qt Creator Versionen nicht mehr funktionieren selbst wenn man die mit den passenden Bibliotheksversionen kompiliert.

Auch deswegen wird die Konfiguration und Anpassung des `QwtPlot` in diesem Handbuch ausschließlich durch normale API-Aufrufe demonstriert.

Soll das `QwtPlot` auch ohne Designer-Plugins im grafischen QtDesigner-Editor eingefügt werden, kann man einfach ein QWidget einfügen und dieses als Platzhalter für die `QwtPlot`-Klasse definieren, siehe Kapitel 16.5.

# 2. Erste Schritte und ein interaktives Diagramm

Um mit der Qwt-Bibliothek warm zu werden, erstellen wir in einem einfachen Beispiel ein interaktives Diagramm mit der `QwtPlot`-Komponente. Der komplette Beispielquelltext ist als 7z-Archiv herunterladbar: tutorial1.7z

## 2.1. Programmrohbau

### 2.1.1. QMake Projektdatei

Wir beginnen mit der qmake-Projektdatei, in der wir den Pfad für die Header-Dateien der Bibliothek und die zu linkende Bibliothek festlegen. Hier gehe ich davon aus, dass Qwt aus dem 6.3.0er Quelltextarchiv gebaut und lokal in die Standardverzeichnisse (`C:\qwt-6.3.0` unter Windows und `/usr/local/qwt-6.3.0` unter Linux/Mac) installiert wurde. Infos über das Compilieren der Bibliothek aus dem Quelltext und Installation gibt es in Kapitel 16.

```
TARGET   = Tutorial1
QT       += core gui widgets
CONFIG   += c++11

win32 {
    # Pfad zu den Qwt Headerdateien hinzufügen
    INCLUDEPATH += C:/qwt-6.3.0/include
    CONFIG(debug, debug|release) {
        QWTLIB = qwtd
    }
    else {
        QWTLIB = qwt
    }
    # Linkerpfad
    LIBS += -LC://qwt-6.3.0/lib -l$$QWTLIB
}
else {
    # Pfad zu den Qwt Headerdateien hinzufügen
    INCLUDEPATH += /usr/local/qwt-6.3.0/include/
    # Linkerpfad, unter Linux wird standardmäßig nur die release-Version der Lib gebaut und installiert
    LIBS += -L/usr/local/qwt-6.3.0/lib -lqwt
}

SOURCES += main.cpp
```

Dies ist eine `.pro`-Datei für eine Qwt-6.3.0-Installation aus dem Quelltext mit Standardeinstellungen (siehe Kapitel 16.2).

Beachte, dass die im Debug-Modus kompilierte Qwt-Bibliothek ein angehängtes *d* hat. Unter Linux wird standardmäßig nur die release-Version gebaut und installiert, daher braucht man hier die Fallunterscheidung nicht.

### 2.1.2. Minimalistisches Hauptprogramm

Für die Verwendung des `QwtPlot` braucht man nur eine sehr minimalistische `main.cpp`.

*Hauptprogramm, in dem nur das nackige Plot selbst erstellt und angezeigt wird*

```
#include <QApplication>

#include <QwtPlot>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QwtPlot plot;
    plot.resize(800,500);
    plot.show();
```

```
    return a.exec();
}
```

Wenn man das Programm compiliert hat und ausführen will, beklagt sich Windows über eine fehlende DLL. Dazu in den Projekteinstellungen, unter "Ausführen", im Abschnitt "Umgebung" die PATH-Variable bearbeiten und dort den Pfad `C:\qwt-6.3.0\lib` hinzufügen.

Das Programm zeigt ein ziemlich langweiliges (und hässliches) Diagrammfenster (später wird das noch ansehnlicher gestaltet).

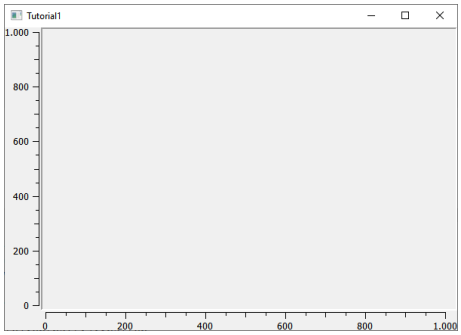



*Abbildung 3. Das nackte Plotwidget*

Ein Hinweis zu den Header-Dateien der Qwt-Bibliothek.

Analog zu Qt Klassen werden die Qwt-Klassen über den gleichnamigen Header eingebunden, also:

```
#include <QwtPlot>       // für Klasse QwtPlot
#include <QwtPlotCurve>  // für Klasse QwtPlotCurve
#include <QwtLegend>     // für Klasse QwtLegend
// ...
```

Diese Header-Dateien sind aber nur Wrapper um die eigentlichen Include-Dateien, mit dem Benennungsschema:

```
#include <qwt_plot.h>        // für Klasse QwtPlot
#include <qwt_plot_curve.h>  // für Klasse QwtPlotCurve
#include <qwt_legend.h>      // für Klasse QwtLegend
// ...
```

In früheren Versionen der Qwt-lib (auch der Debian-Paket-Version `libqwt-qt5-dev`) wurden die Wrapper-Headerdateien nach dem neuen Namensschema nicht installiert, sodass man die originalen `qwt_xxx.h` Includes verwenden muss. Wenn man also auch ältere Qwt-Versionen unterstützen möchte, bzw. unter Linux die Paketversion verwenden will, sollte die originalen Headerdateinamen verwenden.

## 2.2. Diagrammelemente hinzufügen

### 2.2.1. Linie hinzufügen

Als erstes fügen wir eine Linie bzw. Diagrammkurve hinzu (Header `QwtPlotCurve` bzw. `qwt_plot_curve.h`):

*Hauptprogramm mit einer Linie*

```
#include <QApplication>

#include <QwtPlot>
#include <QwtPlotCurve>
```

```
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QwtPlot plot;
    plot.resize(500,300);

    // etwas Abstand zwischen Rand und Achsentiteln
    plot.setContentsMargins(8,8,8,8);
    // Hintergrund der Zeichenfläche soll weiß sein
    plot.setCanvasBackground( Qt::white );

    // Daten zum Darstellen einlesen
    QVector<double> x, y;
    QFile f("spektrum.tsv");  // Datei enthält 2 Spalten
    f.open(QFile::ReadOnly);
    QTextStream strm(&f);
    strm.readLine(); // Kopfzeile überspringen
    while (!strm.atEnd()) {
        double xval, yval;
        strm >> xval >> yval;
        x.append(xval);
        y.append(yval);
    }

    QwtPlotCurve *curve = new QwtPlotCurve();
    curve->setPen(QColor(180,40,20), 0);
    curve->setTitle("Gamma-Spektrum");
    curve->setRenderHint( QwtPlotItem::RenderAntialiased, true ); // Antialiasing verwenden
    curve->setSamples(x, y);
    curve->attach(&plot); // Plot takes ownership

    plot.show();
    return a.exec();
}
```

Im erweiterten Hauptprogramm wird zunächst der Header für die `QwtPlotCurve` eingebunden. Das Kurvenobjekt selbst wird mit `new` auf dem Heap erstellt. Die Daten der Kurve lesen wir aus einer Textdatei (2 Spalten, mit Kopfzeile) aus. Die Datei `spektrum.tsv` ist im Archiv des Tutorialquelltextes enthalten.

Grundsätzlich gilt beim `QwtPlot`: Alle Plotelemente *müssen* via new auf dem Heap erstellt werden und dem Plot dann übergeben werden. Dieses wird dann Besitzer und gibt den Speicher frei. Deshalb dürfen Linien, Legende, Marker etc. *niemals* als Stack-Variablen erstellt werden, sonst gibt es (je nach Destruktoraufrufreihenfolge) einen Speicherzugriffsfehler.

Attribute wie Linienfarbe, Titel (wird später in der Legende angezeigt), und Antialising werden gesetzt (im Kapitel 5 werden alle Eigenschaften von Linien im Detail erläutert).

Die Funktion `setSamples()` setzt die Daten der Linie. Wichtig ist hier, dass die übergebenen Vectoren die gleiche Länge haben. Es handelt sich um eine parametrische Kurve, d.h. weder x noch y Werte müssen monoton sein oder sonstwelchen Regeln folgen. Jedes x,y Wertepaar definiert einen Punkt und diese Punkte werden mit der Linie verbunden.

Die Funktion `attach()` fügt das `QwtPlotCurve`-Objekt zum Diagramm hinzu.

Beim Hinzufügen der Linie mittels `attach()` zum Diagramm wird das Plot neuer Eigentümer und kümmert sich um das Aufräumen des Speichers. Man muss also nicht mehr manuell `delete` für das `QwtPlotCurve`-Objekt aufrufen.

Zusätzlich zu dem Code, welcher die Linie hinzufügt, wurden noch 2 kleine Anpassungen am Erscheinungsbild vorgenommen:

- Ränder wurden mittels `setContentsMargins()` hinzugefügt (siehe auch `QWidgdet::setContentsMargins()` )
- der Hintergrund der Zeichenfläche (*canvas*) wurde weiß gefärbt.

Das Ergebnis sieht schon eher nach Diagramm aus.

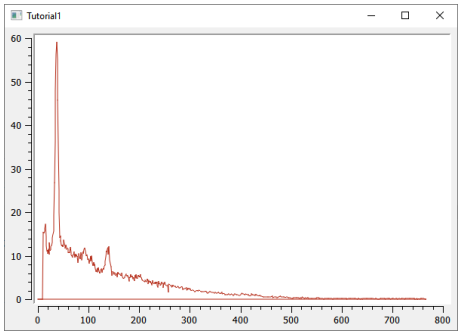



*Abbildung 4. Diagramm mit Linie*

### 2.2.2. Legende hinzufügen

Als nächstes wird eine Legende eingefügt (Header `QwtLegend` bzw. `qwt_legend.h`):

```
// Legende anzeigen
QwtLegend * legend = new QwtLegend();
QFont legendFont;
legendFont.setPointSize(8);
legend->setFont(legendFont);
plot.insertLegend( legend , QwtPlot::BottomLegend); // plot takes ownership
```

Auch hier wird oben wieder der Header für die Klasse `QwtLegend` eingebunden.

Die Legende bekommt hier noch einen veränderten Font. Das weitere Anpassen der Legende wird in Kapitel 8 beschrieben.

Die Legende kann links, rechts, oberhalb oder unterhalb der Zeichenfläche liegen, oder in der Zeichenfläche selbst. Die Platzierung wird beim Aufruf von `insertLegend()` festlegegt.

Das Plot nimmt beim Aufruf von `insertLegend()` wiederum Besitz vom Legendenobjekt und kümmert sich um das Aufräumen des Speichers.

### 2.2.3. Diagrammtitel hinzufügen

```
// Titel hinzufügen
QwtText text("Gamma-Spektrum");
QFont titleFont;
titleFont.setBold(true);
titleFont.setPointSize(10);
text.setFont(titleFont);
plot.setTitle(text);
```

Die Klasse `QwtText` (Header `QwtText` bzw. `qwt_text.h`) kapselt einen QString und ergänzt Funktionalität zum Rendern von mathematischen Symbolen mittels MathML (siehe Kapitel 11.1).

### 2.2.4. Diagrammraster hinzufügen

Gitterlinien werden durch das Zeichenobjekt `QwtPlotGrid` gezeichnet (Header `QwtPlotGrid` bzw. `qwt_plot_grid.h`):

```
// Haupt- und Nebengitter anzeigen
QwtPlotGrid *grid = new QwtPlotGrid();
QPen gridPen(Qt::gray);
gridPen.setStyle(Qt::DashLine);
```

```
grid->setMajorPen(gridPen);
// Minor grid
grid->enableYMin( true );
gridPen.setColor(Qt::lightGray);
gridPen.setStyle(Qt::DotLine);
grid->setMinorPen(gridPen);
grid->attach( &plot ); // plot takes ownership
```

Das Raster selbst kann hinsichtlich der Stifts (QPen) für das Haupt- und Nebengitter angepasst werden. Die Funktion `enableYMin()` schaltet das Nebengitter für die Y-Achse ein. Wie auch bei den Plotkurven übergibt `attach()` das `QwtPlotGrid` Objekt an das `QwtPlot`, welches sich dann um die Speicherverwaltung kümmert.

Ein Raster wird standardmäßig an eine x- und y-Achse gebunden, wobei man aber auch die Gitterlinien für eine der Achsen ausblenden kann. Wenn man z.B. ein Diagramm mit 2 y-Achsen hat und für jede ein Gitterraster anzeigen möchte (auch wenn das meistens verwirrend aussieht), dann braucht man zwei `QwtPlotGrid`-Objekte.

Inzwischen sieht das Diagramm schon ganz ansehnlich aus.

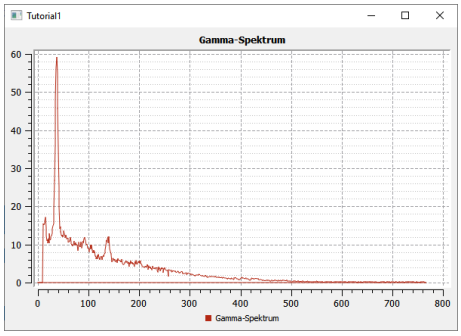



*Abbildung 5. Diagramm mit Linie, Legende, Titel und Gitterlinien*

### 2.2.5. Achsenkonfiguration

Das `QwtPlot` hat 4 Achsen eingebaut, genannt:

- `QwtPlot::yLeft` und `QwtPlot::yRight`
- `QwtPlot::xBottom` und `QwtPlot::xTop`

Standardmäßig sind die Achsen `xBottom` und `yLeft` sichtbar, wie im bisher verwendeten Plot.

Jedes Zeichenelement im Plot (Kurven, Marker, …) wird einer oder mehrerer Achsen zugeordnet. In unserem Einführungsbeispiel verwendet die `QwtPlotCurve` standardmäßig die Achsen `xBottom` und `yLeft`.

Die Achsen können wie folgt konfiguriert werden.

```
// Achsen formatieren
QFont axisFont;
axisFont.setPointSize(8);
axisFont.setBold(true);
QFont axisLabelFont;
axisLabelFont.setPointSize(8);
// X-Achse
QwtText axisTitle("Kanal");
axisTitle.setFont(axisFont);
// Titel Text und Font setzen
plot.setAxisTitle(QwtPlot::xBottom, axisTitle);
// Font für Achsenzahlen setzen
plot.setAxisFont(QwtPlot::xBottom, axisLabelFont);
// Y-Achse
```

```
axisTitle.setText("Ereignisse");
plot.setAxisTitle(QwtPlot::yLeft, axisTitle);
plot.setAxisFont(QwtPlot::yLeft, axisLabelFont);
```

Der Titel jeder Achse wird wiederum über ein `QwtText`-Objekt (enthält Text und Font) gesetzt. Der Font für die Zahlen an den Achsen selbst wird über `setAxisFont()` geändert.

Die Achsen selbst lassen sich vielfältig anpassen, siehe Kapitel 10.

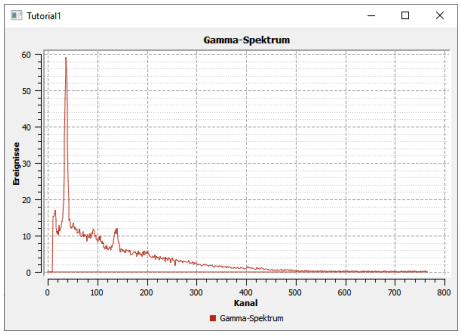



*Abbildung 6. Vollständig formatiertes Diagramm*

Die Achsen passen sich standardmäßig automatisch an den Wertebereich der angezeigten Kurven an. Das kann man natürlich auch ändern, siehe Kapitel 10.

### 2.2.6. Logarithmische Achsen

Das `QwtPlot` kann auch logarithmische Achsen verwenden. Dazu muss man eine anderen Skalenberechnungsklasse einbinden, die `QwtLogScaleEngine` (Header `QwtLogScaleEngine` bzw. `qwt_scale_engine.h`):

```
// Logarithmische Y-Achse
QwtLogScaleEngine * logScale = new QwtLogScaleEngine();
plot.setAxisScaleEngine(QwtPlot::yLeft, logScale); // plot takes ownership
// manuelle Achsenlimits festlegen, da autoscale bei log-Achsen nicht sinnvoll funktioniert
plot.setAxisScale(QwtPlot::yLeft, 1e-3,1000);
```

Beim Aufruf von `setAxisScaleEngine()` nimmt das Plot wiederum das Objekt in Besitz und kümmert sich dann um das Speicheraufräumen.

Kapitel 10 beschreibt die Details der *ScaleEngine* und gibt weitere Beispiele.

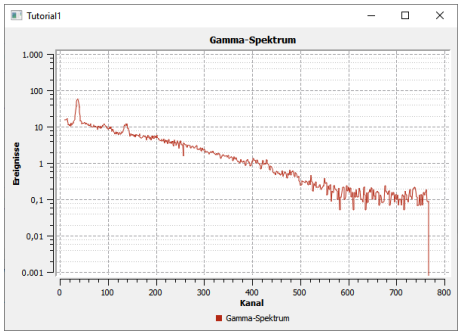



*Abbildung 7. Diagramm mit logarithmischer Y-Achse*

### 2.2.7. Markierungslinien

Ein weiteres Zeichenelement, das man hin und wieder braucht, sind horizontale oder vertikale Markierungslinien. Beispielhaft fügen wir eine solche Linie mal dem Plot hinzu (Header `QwtPlotMarker` bzw. `qwt_plot_marker.h`):

```
QwtPlotMarker * marker = new QwtPlotMarker;
marker->setLabelOrientation(Qt::Vertical); // Vertikale Linie
marker->setLabelAlignment(Qt::AlignRight | Qt::AlignBottom); // Label unten und rechts von der Linie
marker->setValue(36, 0); // bei vertikalen Linien muss die x-Koordinate festgelegt werden
QPen markerPen(QColor(40,60,255));
markerPen.setStyle(Qt::SolidLine);
marker->setLinePen(markerPen);
marker->setLineStyle(QwtPlotMarker::VLine);
QwtText markerLabel("207,50 keV");
QFont markerFont;
markerFont.setPointSize(8);
markerLabel.setFont(markerFont);
marker->setLabel(markerLabel);
marker->attach(&plot); // plot takes ownership
```

Auch bei den Markern gibt es vielfältige Einstellungsmöglichkeiten, siehe Kapitel 9.

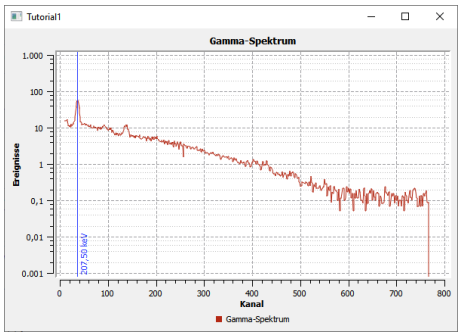



*Abbildung 8. Diagramm mit logarithmischer Y-Achse und vertikaler Peak-Markierung*

Nun ist das Diagramm selbst fertig und wir widmen uns der Nutzerinteraktion.

## 2.3. Interaktion mit dem Diagramm

Das `QwtPlot` bietet die üblichen Interaktionsmöglichkeiten für den Anwender, wie z.B. Herein- und Herauszoonmen, oder Verschieben des Plotausschnitts.

### 2.3.1. Zoomfunktionalität mit QwtPlotZoomer

Die Zoom-Funktionalität wird über die Klasse `QwtPlotZoomer` hinzugefügt (Header `QwtPlotZoomer` bzw. `qwt_plot_zoomer.h`):

```
// Zoomer hinzufügen
// Achtung: NICHT QwtPlot selbst als 3 Argument übergeben, sonder das canvas()
QwtPlotZoomer * zoomer = new QwtPlotZoomer(QwtPlot::xBottom, QwtPlot::yLeft, plot.canvas());  // plot takes ownership
zoomer->setTrackerMode( QwtPlotPicker::AlwaysOn ); // Kurvenvwerte unterm Cursor anzeigen
```

Wenn man mit der Maus über das Diagramm fährt, sieht man bereits einen veränderten Cursor und dank des Aufrufs `setTrackerMode(QwtPlotPicker::AlwaysOn)` sieht man nun auch die x- und y-Werte (des Achsen `xBottom` und `yLeft`) unter dem Cursor.

Hineinzoomen kann man, indem man die Linke Maustaste gedrückt hält, und ein Zoom-Rechteck aufzieht. Das kann man auch mehrmals hintereinander machen. Das `QwtPlot` merkt sich intern diese Zoomstufen. Herauszoomen kann durch Klick auf die rechte Maustaste, wobei immer eine Zoomstufe hinausgezoomt wird.

Die äußerste Zoomstufe wird im Konstruktor der `QwtPlotZoomer`-Klasse basierend auf den aktuellen Wertebereichen der *bereits hinzugefügten Kurven* bestimmt. Sollte man die Werte der Kurven nachträglich ändern, oder den Zoomer hinzufügen, *bevor* man dem Plot Kurven gegeben hat, so kann man die Funktion `QwtPlotZoomer::setZoomBase()` aufrufen. Details dazu gibt es im Kapitel 12.

Im Quelltext gibt es noch eine Besonderheit. Während die bisherigen Plotelemente immer mit Memberfunktionen der `QwtPlot`-Klasse hinzugefügt wurde, bzw. mittels `attach()`, wird das Zoomerobjekt analog zu Qt Klassen als Kindobjekt der Zeichenfläche gegeben und registriert sich darüber als interaktives Element bei Plot.

Es ist wichtig darauf zu achten, dass man beim Konstruktor der Klasse `QwtPlotZoomer` als 3. Argument das Canvas-Objekt des Plots übergibt. Dieses erhält man mit der Funktion `QwtPlot::canvas()`. Wenn man hier stattdessen das Plot selbst übergibt, führt dies zu einem Speicherzugriffsfehler.

Im Konstruktor der `QwtPlotZoomer` Klasse registriert sich das Objekt als Kind des Canvas-Widgets, wodurch das QObject-System sich um die Speicherverwaltung kümmert. Man muss also das `QwtPlotZoomer` Objekt nicht freigeben.

Damit der Zoomer weiß, welche Achsen beim Zoom manipuliert werden sollen, muss man die x- und y-Achse im Konstruktor angeben. Möchte man z.B. beide y-Achsen gleichzeitig zoomen, braucht man zwei `QwtPlotZoomer`-Objekte.

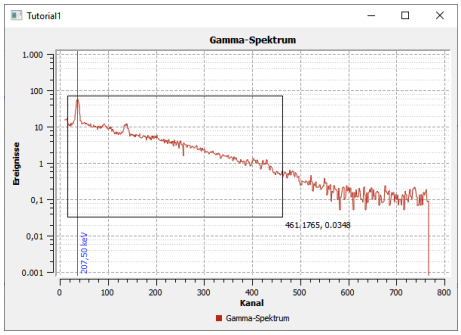



*Abbildung 9. Diagramm mit aufgezogenem Zoom-Rechteck*

### 2.3.2. Plotausschnitt verschieben mit QwtPlotPanner

Wenn man Ausschnitt eines hineingezoomten Plots interaktiv verschieben möchte, kann man den `QwtPlotPanner` hinzufügen (Header `QwtPlotZoomer` bzw. `qwt_plot_zoomer.h`):

```
// Panner hinzufügen, wie auch beim PlotZoomer muss das Canvas-Objekt als Argument übergeben werden
QwtPlotPanner * panner = new QwtPlotPanner(plot.canvas());  // plot takes ownership
panner->setMouseButton(Qt::MiddleButton); // Mittlere Maustaste verschiebt
```

Wie beim `QwtPlotZoomer` wird das Objekt als Kindobjekt des Canvas-Widgets hinzugefügt. Üblich ist das Verschieben von Bildschirminhalten mit gedrückter mittlerer Maustaste, also legt man das mit `setMouseButton()` fest.

---

Damit ist das Einstiegstutorial beendet. Mit dem `QwtPlot` kann man bereits mit wenigen Handgriffen ein voll funktionsfähiges und interaktives Diagramm erstellen.

In diesem Tutorial war das `QwtPlot` gleichzeitig das Anwendungs-Widget. Wenn man das `QwtPlot` aber in bestehende Designer-Formularklassen einfügen will, gibt es verschiedene Techniken:

- die Verwendung von Platzhalter-Widgets
- die Einbindung von Qt Designer Plugins für die Qwt Bibliothek

Diese Methoden sind in Kapitel 16.5 beschrieben.

Allen Diagrammtypen und weiteren Plot-Eigenschaften sind einzelne Kapitel gewidmet. Beim `QwtPlot` wird dabei zunächst nur auf die mitgelieferte Funktionalität des `QwtPlot` und der dazugehörigen Klassen eingegangen. In späteren Kapiteln wird die Erweiterung der Plot-Funktionalität durch Überschreiben/Ersetzen der eingebauten Funktionen gezeigt.

# 3. QWT Widgets und Eingabekomponenten

Neben dem `QwtPlot` gibt es in der Qwt-Bibliothek noch eine Reihe anderer Eingabekomponenten, die in diesem Kapitel kurz vorgestellt werden. Viele dieser Komponenten sind klassischen Anzeigen und Einstellrädern in wissenschafltlich/technischen Geräten nachempfunden.

Für die Anzeige der Skalen verwenden die nachfolgend vorgestellten Komponenten intern zur Darstellung der Skalen die in Kapitel 10.2 näher beschriebenen Skalenberechnungs- und -zeichenklassen.

## 3.1. Schieberegler (Slider)

Die Klasse `QwtSlider` erlaubt die Darstellung verschiedener Schieberegler, welche mit der Maus oder Tastatur (Cursortasten) bedient werden können. Im Gegensatz zur `QSlider` Klasse können die Skalen viel flexibler und auch nichtlinear definiert werden.

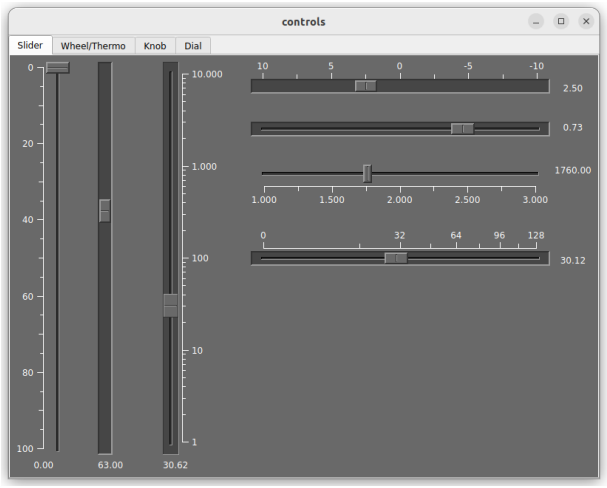



*Abbildung 10. Beispiele für* `QwtSlider`

Das Beispiel im Screenshot oben ist in der Qwt-Bibliothek als Beispiel `controls` enthalten.

## 3.2. Drehräder/Einstellräder und

Die Klasse `QwtWheel` zeigt ein horizontales oder vertikales Einstellrad. Die Klasse `QwtThermo` zeigt eine Balkenanzeige, allerdings mit einer flexibel hinterlegbaren Farbtabelle. Dies erlaubt z.B. Farbverläufe oder Farbsprünge bei Übersteigen bestimmter Schwellwerte.

Qt selbst bietet für eine Balkenanzeige die Klasse `QProgressBar` an, welches sich aber im Erscheinungsbild an den jeweiligen Plattformstil für Fortschrittsbalken orientiert und auch keine Skalen bietet.

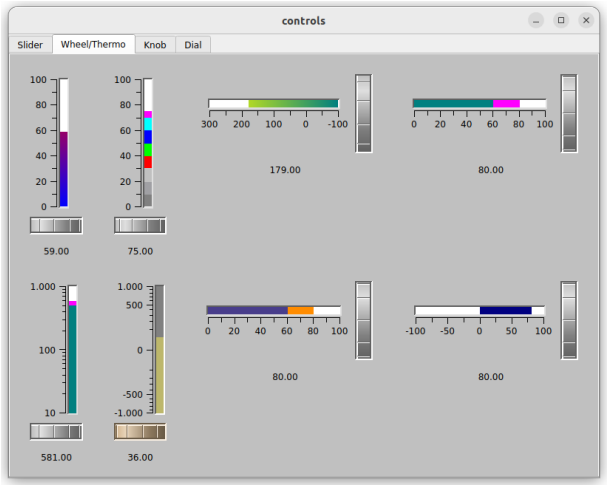



*Abbildung 11. Beispiele für* `QwtWheel` *und* `QwtThermo`

Das Beispiel im Screenshot oben ist in der Qwt-Bibliothek als Beispiel `controls` enthalten.

## 3.3. Drehknöpfe

Die Klasse `QwtKnob` zeigt einen Drehknopf, mit ebenso flexibel konfigurierbaren Skaleneinheiten. Die Qt-Klasse `QDial` bietet ebenso ein Einstellrad, jedoch wiederum viel simpler und mit weniger Einstellungsmöglichkeiten hinsichtlich der Skalendarstellung und -skalierung.

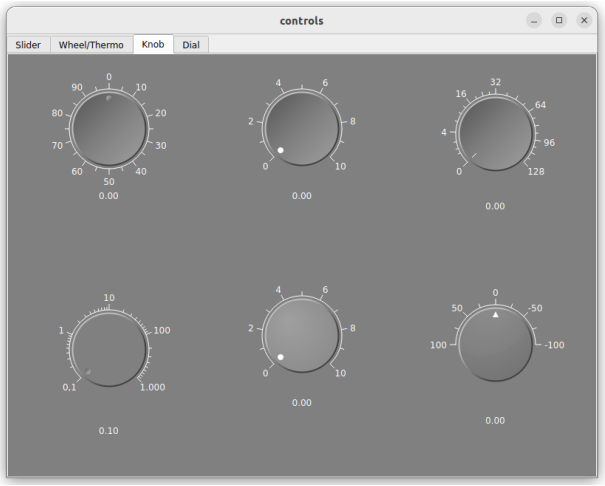



*Abbildung 12. Beispiele für* `QwtKnob`

Das Beispiel im Screenshot oben ist in der Qwt-Bibliothek als Beispiel `controls` enthalten.

## 3.4. Analoge Zeiger-Anzeigen

Die Klasse `QwtDial` zeichnet analoge Zeigeranzeigen, die aber auch mit der Maus/Tastatur verändert werden können (wenn man das aktiviert). Die Anzeigen lassen sich farblich sehr individuell konfigurieren.

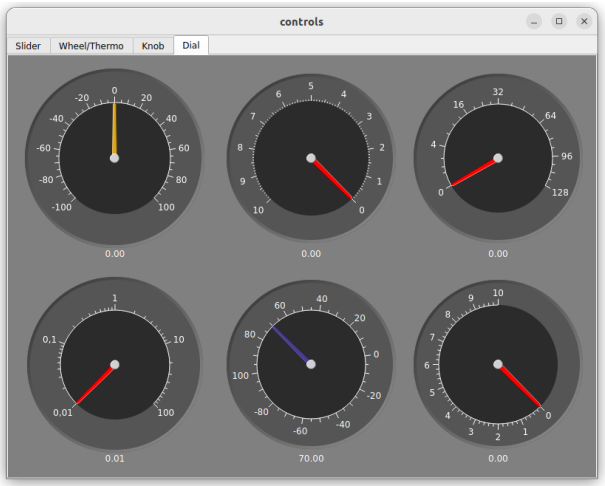



*Abbildung 13. Beispiele für* `QwtDial`

Das Beispiel im Screenshot oben ist in der Qwt-Bibliothek als Beispiel `controls` enthalten.

Bemerkenswert ist vielleicht noch, dass die Anzeigenadel selbst unabhängig von der Klasse `QwtDial` durch eine separate Klasse implementiert wird. Als Standard wird hier `QwtDialSimpleNeedle` verwendet, wie im Screenshot oben. Man kann sich hier aber auch austoben, und selber beliebige Anzeigenadeln entwerfen und integrieren.

# 4. Allgemeine Grundlagen des QwtPlot

Das `QwtPlot` ist sicher die nützlichste und am weitesten bekannte Komponente der Qwt Bibliothek. Im Gegensatz zu vielen anderen Qt-Diagrammkomponenten, kann man mit dem `QwtPlot` wirklich sehr flexibel (und effizient) alle möglichen Diagrammtypen erstellen und anzeigen.

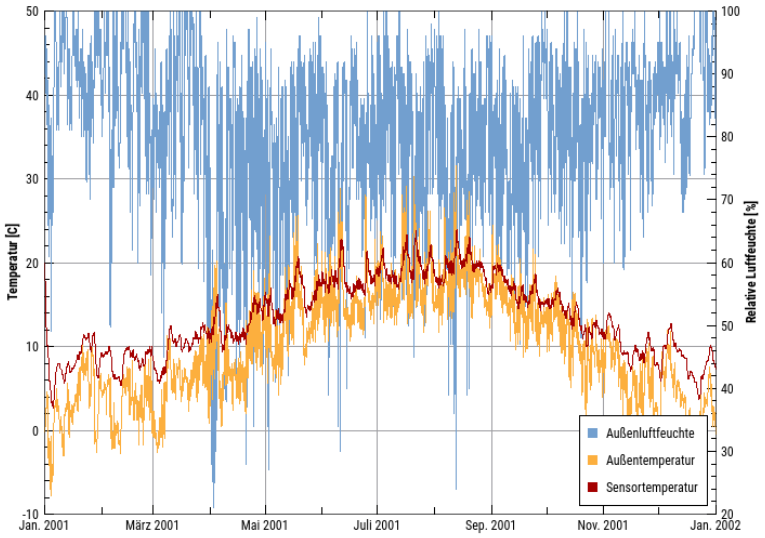



*Abbildung 14. Beispiel für ein typisches* `QwtPlot` *mit zwei y-Achsen und innenliegender Legende.*

Bevor ab Kapitel 5 die einzelnen Diagrammtypen vorgestellt werden, geht es in diesem Kapitel um die wesentlichen Grundlagen.

## 4.1. Aufbau und Elemente der Diagrammkomponente

Das `QwtPlot` besteht aus einem Titel, umliegenden Achsen, Legenden und der eigentlichen Zeichenfläche. Alle diese Elemente können konfiguriert und sichtbar/unsichtbar geschaltet werden.

### 4.1.1. Achsen und Koordinatensystem

Das Diagramm selbst ist ein kartesisches Diagramm mit maximal 4 Achsen, identifiziert über folgende Enumerationswerte:

- `QwtPlot::xBottom`
- `QwtPlot::xTop`
- `QwtPlot::yLeft`
- `QwtPlot::yRight`

Diese Achsen haben primär die Aufgabe, zwischen den *Plotkoordinaten* (x,y) und *Bildschirmkoordinaten* umzurechnen. Jede der vier Achsen kann individuell konfiguriert werden, welches sich auf die min/max-Werte und die Achsenskalierung auswirkt.

### 4.1.2. Diagramm-/Zeichenelemente

Innerhalb der Zeichenfläche kann man nun die verschiedensten Elemente zeichen, bspw.:

- Linien,
- Balken,
- Symbole,
- Markierungen,
- Legendeneinträge,
- Gitterraster,

- … und viele weitere

Alle diese Objekte sind von der Basisklasse `QwtPlotItem` abgeleitet und teilen sich dadurch gewisse gemeinsame Eigenschaften.

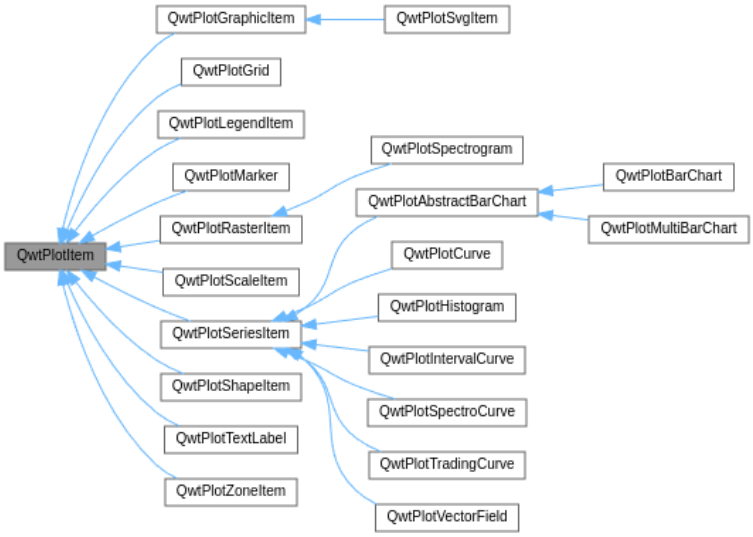



*Abbildung 15. Von QwtPlotItem abgeleitete Klassen der verschiedenen Zeichenelemente*

In einem Diagramm kann man alle möglichen Elemente kombinieren, also auch Liniendiagramme mit Balken, Symbolen und Markierungen darstellen.

Die Zeichenreihenfolge, also welches Zeichenelement ein anderes überdeckt, bestimmt das z-Attribut. Dies wird über die Funktionen `QwtPlotItem::setZ(...)` und `QwtPlotItem::z()` kontrolliert.

Die Positionierung der Diagrammelemente erfolgt über die Plotkoordinaten. Die Umrechung in Bildschirmkoordinaten bzw. die konkrete Position auf der Zeichenfläche wird mittels der jeweils zugeordneten x- und y-Achse gemacht. Da es nun jeweils zwei x- und y-Achsen gibt, braucht man also bei Anzeige eines Diagrammelements an einem bestimmtenx,y-Datenpunkt zwingend eine Zuordnung zu *einer* x- und *einer* y-Achse. Diese Eigenschaft setzt man den einzelnen Zeichenelementen, wobei standardmäßig `xBottom` und `yLeft` ausgewählt sind.

Die Zuordnung eines Zeichenelements zu einer Achse erfolgt mit den Memberfunktionen `QwtPlotItem::setAxes(...)`, `QwtPlotItem::setXAxis(...)`, `QwtPlotItem::setYAxis(...)`.

Das `QwtPlotItem` deklariert und implementiert auch die virtuellen Funktionen für das Zeichnen sowie Berechnen wichtiger Layoutdaten. Dies wird aber später in den fortgeschrittenen Kapiteln noch genauer erklärt. Nun aber zu den individuellen Zeichenelementen und daraus erstellten Diagrammtypen.

### 4.1.3. Hinzufügen/Entfernen von Zeichenelementen

Alle Zeichenelemente werden *grundsätzlich* auf dem Heap mit `new` erstellt und mittels der Memberfunktion `QwtPlotItem::attach(plot)` dem eigentlichen `QwtPlot` hinzugefügt.

Entfernen kann man Zeichenelemente mittels der Funktion `QwtPlotItem::detach()`.

Beim Aufruf von `attach()` wird das Zeichenelement in die Obhut des `QwtPlot` übergeben, welches sich dann um das Aufräumen des Speichers kümmert. Solange das `QwtPlot` noch im Besitz eines PlotElements ist, darf/sollte man das Objekt nicht löschen.

Möchte man das Zeichenelement dennoch wieder zurückerhalten, ruft man die Memberfunktion `detach()` auf. Damit übernimmt man dann selbst wieder die Verantwortung für das Speicherbereinigen.

Man kann auch alle Zeichenelemente eines bestimmten Typs wieder entfernen:

```
// alle Kurven entfernen und die Kurvenobjekte dabei löschen
plot.detachItems(QwtPlotItem::Rtti_PlotCurve, true); // detach and delete
```

Das erste Argument von `detachItems()` ist der Zeichenelementtyp. Mit dem zweiten Argument gibt man an, ob das Objekt selbst gelöscht werden soll, oder nur aus dem Plot entfernt wird. Im letzteren Fall muss man sich wieder selbst um das Speicheraufräumen kümmern.

## 4.2. Datenhaltung im QwtPlot / QwtSeriesStore

Für die Darstellen von Plotkurven/Balken oder anderen Zeichenelementen werden die Daten für mehrere Datenpunkte (*samples*) benötigt. Je nach Anforderung des Zeichenelements gibt es verschiedene Typen von *samples*. So brauchen Linienkurven normalerweise x,y-Wertepaare, aber Intervallkurven jeweils x,y1,y2-Tuples.

Alle Zeichenelemente, welche solche Reihendaten verwenden, sind Kinder der Klasse `QwtPlotSeriesItem`.

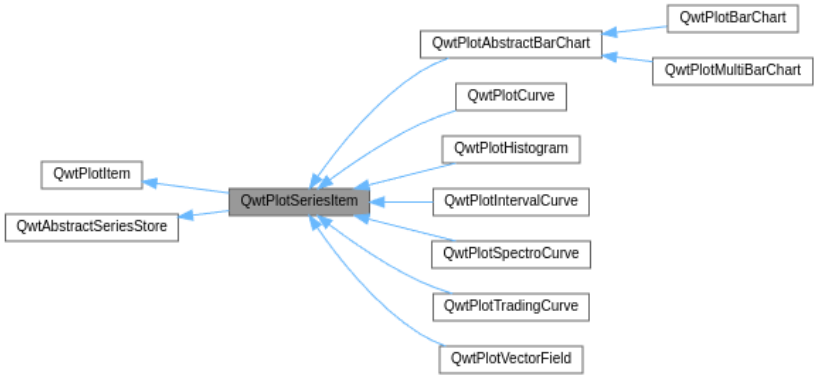



*Abbildung 16. Klasse QwtPlotSeriesItem, Elternklassen und abgeleitete Klassen*

Die Datenhaltung erfolgt in der Template-Klasse `QwtSeriesStore`, welche die abstrakte Schnittstellenklasse `QwtAbstractSeriesStore` implementiert.

Die hier in Qwt verwendet Kombination von Template-Klassen für die Datenhaltung individueller Typen und gleichzeitig Implementierung gemeinschaftlicher Klassenfunktionalität via virtueller Funktionen, zeigt mal wieder schön die Flexibilität von C++.

Allerdings macht es das Verständnis der Interaktion von Datenhalteklassen und die Zeichenelement-Klassen auf den ersten Blick etwas komplizierter. Glücklicherweise muss man das für die Verwendung des `QwtPlot` gar nicht so genau wissen.

Je nach Anforderung des individuellen Zeichenelements/Diagrammtyps werden unterschiedliche Daten benötigt:

- `QPointF` für reguläre Reihen-/Seriendiagramme (Linien)
- `QwtIntervalSample` für Histogramme und Intervallkurven
- `QwtPoint3D` für Spektrogramplots (Farbverlaufsdiagramme)
- `QwtOHLCSample` für Trading-Curves (OHLC - *Open-High-Low-Close*)
- `QwtVectorFieldSample` für Vektorfelder

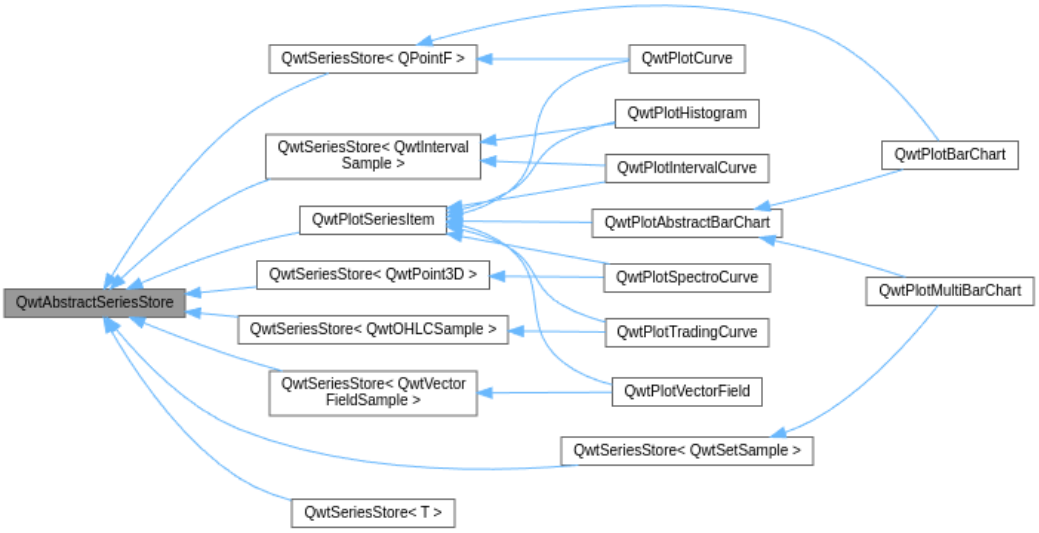



*Abbildung 17. Abstrakte Klasse QwtAbstractSeriesStore und abgeleitete Klassen*

Die meisten Diagrammelemente/Diagrammklassen haben geeignete Schnittstellenfunktionen für die Übergabe der Plotdaten ans Diagramm. Daher muss man selten direkt mit `QwtSeriesStore` arbeiten. In den nachfolgenden Kapiteln wird dies jeweils individuell erklärt.

## 4.3. Automatisches Zeichnen oder Zeichnen bei Bedarf

Das Zeichnen eines komplexen Plots kann durchaus länger dauern, daher ist es bei Anpassungen des Plots häufig nicht notwendig, bei jeweil individuellen Änderung alles neu zu zeichnen. Stattdessen reicht es aus, nach Aktualisierung aller Daten und Anpassung anderer Einstellungen (Achsen, Legenden,…) das Plot neu zu zeichnen.

*Zeichnen* bedeutet hier eigentlich zwei unterschiedliche Arbeitsschritte:

1. Neuberechnung des Layouts, d.h. Größen für Achsen, Legenden, Titel, Label, Zeichenfläche etc.. Dabei erfolgt auch eine Neuberechnung der Achsenskalierung und damit der Zuordnung von Plot-Koordinaten zu Pixelgrößen
2. Das eigentliche Zeichnen (*render*) des Plots

Schritt 1 wird ausgeführt, wenn man `QwtPlot::replot()` aufruft. Standardmäßig wird das immer gemacht, wenn man irgendeine Ploteigenschaft ändert. Diesen automatischen Aufruf kann man mit `QwtPlot::setAutoReplot()` an/ausschalten.

```
// Automatisches Neu-Layouten ausschalten
plot->setAutoReplot(false);
```

Der Aufruf von `replot()` direkt oder indirekt bei Änderungen, wenn *autoReplot* angeschaltet ist, führt nicht zu dem teils zeitaufwändigen Rendern des Plots. Je nach Komplexität des Plots und Größe der Daten ist das sogar sehr schnell. Daher ist das Ausschalten des Auto-Plots selten aus Performancegründen notwendig.

Ein Ausschalten des *autoReplot* kann aber sinnvoll sein, wenn bei Anpassungen von individuellen Ploteigenschaften ein zwischenzeitlich inkonsistenter Zustand eintreten könnte, wo ein Neulayouten nur Quatsch erzeugen könnte oder sowas wie *Division durch Null* erzeugen könnte. Dann wartet man besser, bis alle Plot-/Kurveneigenschaften fertig aktualisiert wurden und ruft dann `replot()` auf.

Nach einem Aktualisieren des Layouts in `replot()` wird ein Zeichenupdate via Qt-Event-Queue angestoßen. Gezeichnet/gerendert wird dann *erst im nächsten Bildrefresh* und auch nur *ein Mal*. Somit kann man gerne 1000 Mal `replot()` aufrufen, und es wird doch nur einmal gerendert. Dies spart sehr viel Zeit.

# 5. Kurvendiagramme

Die wohl häufiges Diagrammart werden wohl Kurvendiagramme sein. Kurvendiagramme bzw. Liniendiagramme oder Reihendiagramme sind parametrische Kurven, bei denen die einzelnen Punkte nacheinander gezeichnet werden, und im Fall von Liniendiagrammen durch Linien verbunden werden. Weder x- noch y-Werte müssen monoton steigen.

Das Zeichenelement `QwtPlotCurve` wird jedoch nicht nur für Liniendiagramme im klassischen Sinn benutzt, sondern auch für Stufen, Stäbchen, Punkt/Symbol-Diagramme und so weiter. Alle diese Varianten haben jedoch gemein, dass sie als Daten einen x- und einen y-Vector mit Daten in Plotkoordinaten erwarten.

Der individuelle Stil der Kurve wird mit der Funktion `QwtPlotCurve::setStyle()` festgelegt. Je nach Stil können/müssen weitere Parameter festgelegt werden. In den folgenden Abschnitten werden die möglichen Diagrammtypen für jeweils die gleichen x/y-Daten im Vergleich gezeigt.

## 5.1. Datenübergabe

Wie in Kapitel 4.2 erklärt, wird für die interne Datenhaltung in `QwtPlotCurve` die Klasse `QwtSeriesStore<QPointF>` verwendet.

Daten kann man nun auf mehrere Arten der Plot-Kurve übergeben:

*Verwendung zweier QVector<double> (oder analog QVector<float>)*

```
QVector<double> x{1,2,5,6,10,12,15,16,17};
QVector<double> y{5,4,8,8, 4, 5, 8, 9,11};
curve->setSamples(x, y);
```

*Verwendung eines QVector<QPointF>*

```
QVector<QPointF> samples{
    QPointF(1,5),
    QPointF(2,4),
    QPointF(5,8)
};
curve->setSamples(samples);
```

Wenn die Daten in einem C-Array oder `std::vector` vorliegen, bietet sich `QwtPlotCurve::setSamples()` an.

*Übergabe der Daten direkt im Speicher*

```
std::vector<double> x{1,2,5,6,10,12,15,16,17};
std::vector<double> y{5,4,8,8, 4, 5, 8, 9,11};
const double * xdata = x.data();
const double * ydata = y.data();
unsigned int count = x.size();

curve->setSamples(xdata, ydata, count);
```

Bei der Verwendung von `QwtPlotCurve::setSamples()` werden die Daten *immer* in den interne `QwtSeriesStore` der Plot-Kurve *kopiert*, auch wenn man hier direkt die Adresse des Speichers mit den Daten übergibt.

Bei sehr großen Datenmengen und begrenztem Hauptspeicher kann es sinnvoll sein, die Daten nicht in das Plot hineinzukopieren, sondern die Plotkurven direkt auf den Speicher zugreifen zu lassen. Dafür gibt es die Funktion `QwtPlotCurve::setRawSamples()`. Die Syntax ist wie beim vorherigen Beispiel:

*Direkte Verwendung der im Speicher befindlichen Daten durch as Plot*

```
const double * xdata = x.data(); // x ist ein std::vector
const double * ydata = y.data(); // y ist ein std::vector
```

```
unsigned int count = x.size();
curve->setRawSamples(xdata, ydata, count);
```

Die Variablen und deren Speicherbereich, welche im Aufruf von `setRawSamples()` verwendet werden, müssen eine längere Lebensdauer haben, als das Plot bzw. die Plotkurve selbst.

Direkte Änderung der Daten im Speicher wird beim nächsten Rendern des Plots direkt sichtbar. Allerdings müssen das Plot und dessen Zeichenelemente, die ein Interesse an den Wertebereichen der Plotkurven haben, manuell über eine Änderung der Daten information werden. Dazu einfach `QwtPlot::replot()`.

## 5.2. Kurvenarten

### 5.2.1. Liniendiagramm

Konfiguration einer `QwtPlotCurve` als Linie:

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Lines);
```

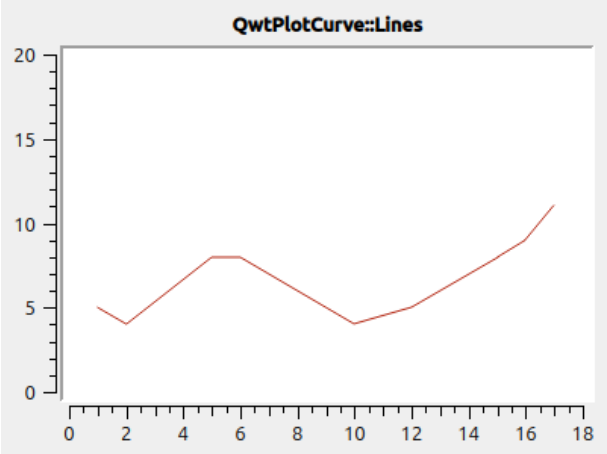



*Abbildung 18. Liniendiagramm*

### 5.2.2. Stäbchen (Sticks)

Konfiguration einer `QwtPlotCurve` als vertikale Stäbchen:

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Lines);
curve->setOrientation(Qt::Vertical);
```

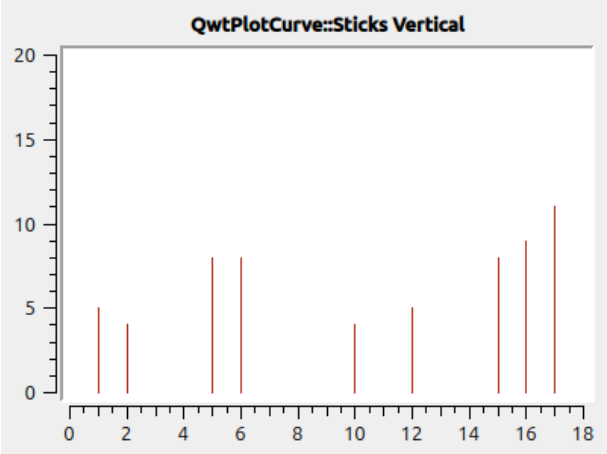



*Abbildung 19. Vertikale Stäbchen*

Alternativ kann man die Stäbchen auch horizontal zeichnen. Dazu muss man zusätzlich die Ausrichtung mit `QwtPlotSeriesItem::setOrientation()` setzen:

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Lines);
curve->setOrientation(Qt::Horizontal);
```

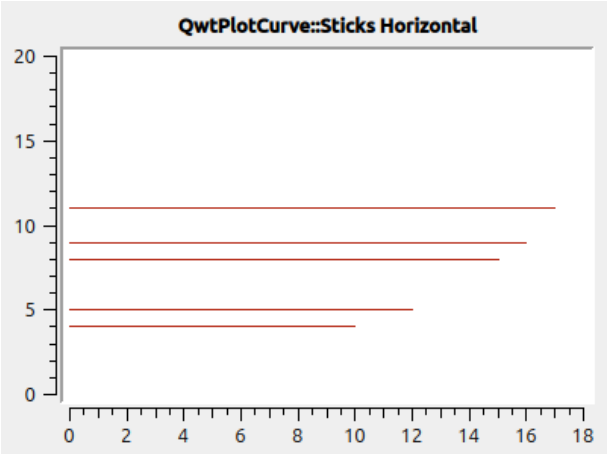



*Abbildung 20. Horizontale Stäbchen*

### 5.2.3. Stufendiagramme

Wenn die Daten nicht linear verbunden werden sollen, sondern eher Stufen darstellen, kann man den Linientyp `Steps` verwenden. Das Zusatz-Attribut `QwtPlotCurve::Inverted` gibt dabei an, ob die Stufe am Ende des Intervals oder Anfang des Intervals sein soll. Kurvenattribute werden mit `QwtPlotCurve::setCurveAttribute()` gesetzt:

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Steps);
curve->setCurveAttribute(QwtPlotCurve::Inverted, false);
```

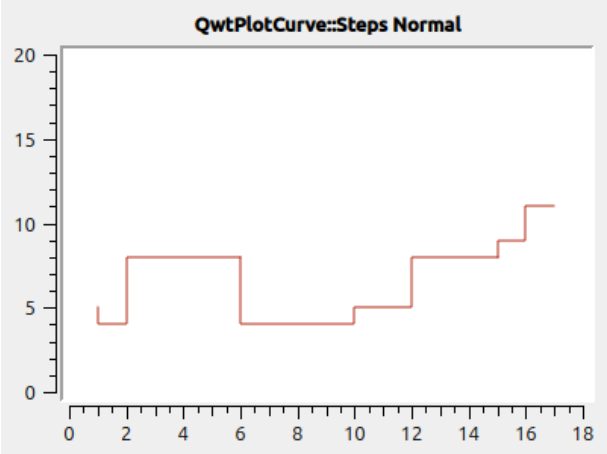



*Abbildung 21. Stufendiagramm (normal)*

Betrachtet man die Eingangsdaten:

```
x   y
1   5
2   4
5   8
...
```

so fällt auf, dass im ersten Intervall, also zwischen x=1..2, der Wert y2=4 gezeichnet wird und an der Stelle x1=1 die Verbindungslinie zwischen y1=5 und y2=4 gezeichnet wird.

Will man direkt den ersten y-Wert im ersten Intervall zeichnen (das wäre eher die natürliche Erwartungshaltung), so muss man das Attribut `Inverted` setzen:

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Steps);
curve->setCurveAttribute(QwtPlotCurve::Inverted, true);
```

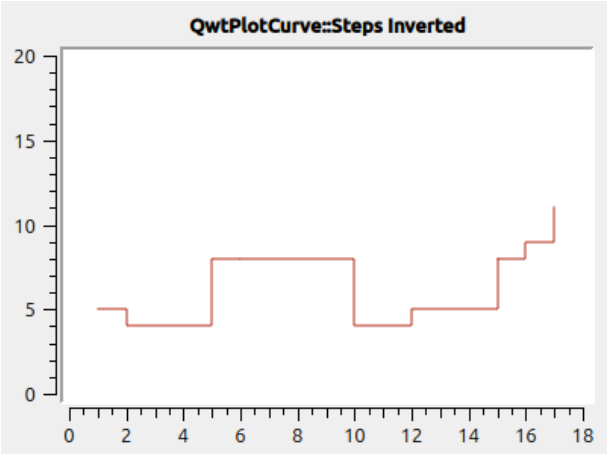



*Abbildung 22. Stufendiagramm (invertiert)*

### 5.2.4. Punkte (Dots)

Man kann an den jeweiligen x,y-Koordinaten auch einfach nur Punkte (auch nur einzelne Pixel) zeichnen. Das geht *sehr schnell* vergleichen mit dem Zeichnen von Symbolen (siehe Kapitel 5.3) und kann für größere Punktwolken verwendet werden.

Bei der Verwendung von `QwtPlotCurve::Dots` sollte man bei heute üblichen Bildschirmauflösungen immer einen `QPen` mit größerer Breite einstellen, da mein *einzelne Pixel* sonst nur noch schwer sehen kann. Für die Visualisierung großer Punktwolken (> 100000 Pixel) kann die Verwendung von einzelnen Pixeln durchaus noch einen Mehrwert bieten. Für alle besonderen Formen (Kreuze, Rauten, Ringe, Sterne, …) ist die Verwendung von Symbolen (siehe Kapitel 5.3) sinnvoll.

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Dots);
curve->setPen(QColor(180,40,20), 4); // width of 4 makes points better visible
```

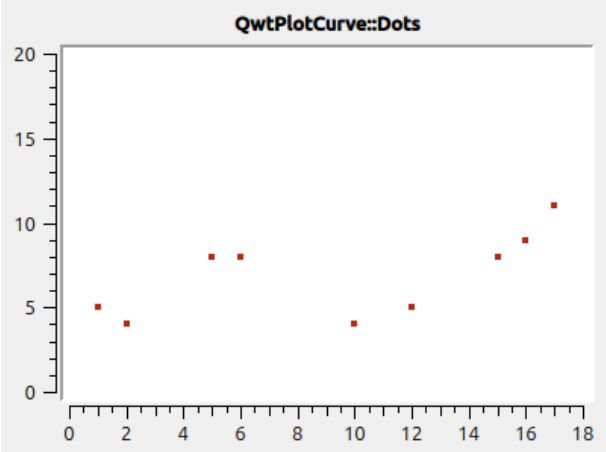



*Abbildung 23. Punktediagramm*

Bei der Visualisierung von Punktwolken kann es hilfreich sein, Transparenz/Alphablending zu benutzen. Dazu einfach bei der Zeichenfarbe noch einen Alphawert kleiner als 255 setzen.

```
QwtPlotCurve *curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Dots);
curve->setPen(QColor(0,40,180,32), 2); // 2 pixels wide, alpha value 32
```

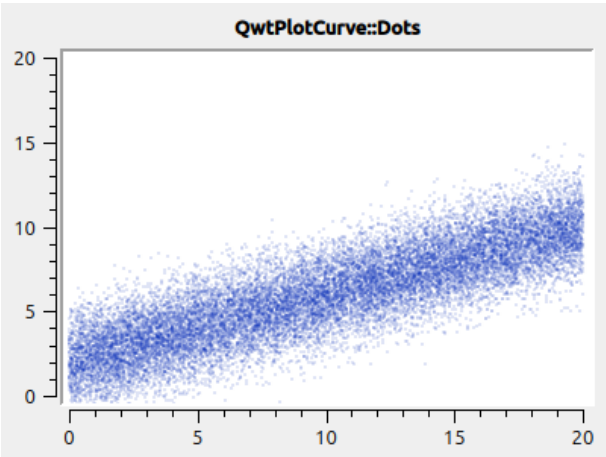



*Abbildung 24. Punktwolke mit halbtransparenten Punkten*

Wenn man ein Liniendiagramm mit Visualisierung von Stützstellen erhalten will, so kann man natürlich *zwei* Linien ins Diagramm einfügen: eine mit Stil `QwtPlotCurve::Lines` und die Zweite mit Stil `QwtPlotCurve::Dots` darüber zeichnen, d.h. mit höherem z-Wert. Dann muss man aber auch zwei Mal die Daten ins Diagramm geben und man erhält auch zwei Legendeneinträge (die kann man zwar auch individuell für einzelne Kurven abschalten, siehe Kapitel 8), aber zunächst gibt es für jede Kurve je einen Legendeneintrag). Besser ist hier die Verwendung von nur einer Kurve und

Symbolen an den Stützstellen (siehe Kapitel 5.3).

### 5.2.5. Keine Linie

Möchte man eine Kurve ausschließlich mit Symbolen zeichnen (siehe Kapitel 5.3), so kann man das Zeichnen des Linienzugs auch komplett ausstellen:

```
curve->setStyle(QwtPlotCurve::NoCurve);
```

## 5.3. Symbole/Punkte

An den jeweiligen x,y-Koordinaten einer Kurve kann man auch Symbole zeichnen. Dafür bietet die Qwt-Bibliothek die Klasse `QwtSymbol` an.

Ein Symbol fügt man zu eine Kurve wie folgt hinzu:

```
// Symbol hinzufügen
QwtSymbol * symbol = new QwtSymbol(QwtSymbol::Ellipse);
symbol->setSize(8);
symbol->setPen(QColor(0,0,160), 2);
symbol->setBrush(QColor(120,170,255));
curve->setSymbol(symbol); // Curve takes ownership of symbol
```

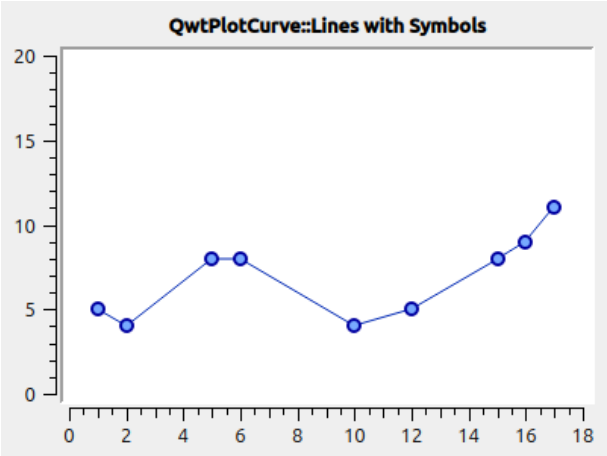



*Abbildung 25. Liniendiagramm mit ausgefüllten Kreissymbolen*

Zuerst wird das zu verwendende Symbol auf dem Heap mit new erzeugt. Der Konstruktor übernimmt den Typ des Symbols (siehe auch Galerie unten). Man kann das aber auch später über `QwtSymbol::setStyle()` setzen.

Wichtig ist auch die Größe des Symbols, gesetzt mittels `QwtSymbol::setSize()` in Pixeln. Diese Größe skaliert das Symbol je nach Form.

Außerdem wichtig sind die Eigenschaften Pen und Brush (`QwtSymbol::setPen()` und `QwtSymbol::setBruch()`). Der Pen wird für das Zeichnen des Umrisses verwendet und der Brush, so gesetzt, für das Ausfüllen der Form. Manche Symbole wie das Kreuz sind nicht ausgefüllt, daher hat hier der Brush keine Wirkung.

Schließlich wird das Symbol der Kurve mit `QwtPlotCurve::setSymbol()` gegeben.

Beim Aufruf von `QwtPlotCurve::setSymbol()` übernimmt die Plot-Kurve die Verantwortung für's Speicheraufräumen.

Die Symbolklasse ist ziemlich mächtig und kann verschiedenste Symbole zeichnen:

- vorgefertigte Formen wie Kreise, Rechtecke, Kreuze, etc. (Stil `QwtSymbol::Ellipse`…`QwtSymbol::Hexagon`
- nutzerdefinierte Bilder/Pixmaps (Stil `QwtSymbol::Pixmap`)
- spezifische Grafiken gekapselt in Klasse `QwtGraphic` und erzeugt durch eine Anzahl von `QwtPainterCommand` Anweisungen (Stil `QwtSymbol::Graphic`) (siehe auch [sec:qwtGraphic])
- SVG-Dokumente (Stil `QwtSymbol::SvgDocument`)
- nutzerdefinierte Formen, welche durch einen QPainterPath definiert sind (Stil `QwtSymbol::Path`)

### 5.3.1. Symbolstile/Eingebauten Symbolformen

Es gibt zahlreiche eingebaute Symbolformen (fett gedruckt im Diagrammtitel ist jeweils der `QwtSymbol::Style` Enumerationsname):

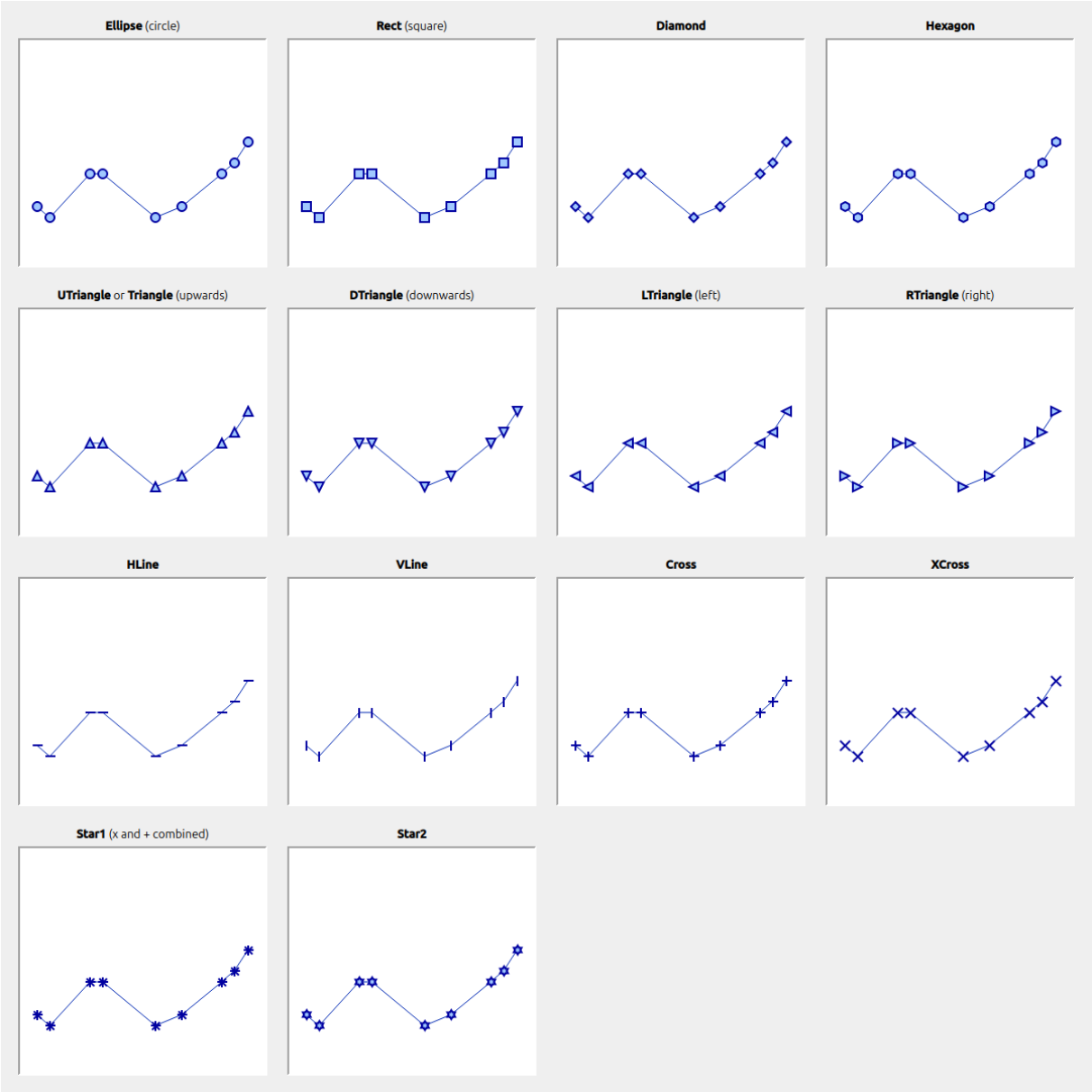



*Abbildung 26. Eingebaute Symboltypen/Stile und deren Enumerationstypen*

Symbole müssen nicht immer quadratisch sein. Wenn man die Größe eines Symbols mit

```
symbol->setSize(10);
```

setzt, wird automatisch `width=height=10` verwendet. Alternativ kann man aber auch ein Rechteck als Größe definieren:

```
symbol->setSize(w,h);
// oder via QSize
QSize s(w,h);
symbol->setSize(s);
```

Deshalb gibt es auch keine separaten Linienstile für Kreis und Ellipse oder Rechteck und Quadrat.

### 5.3.2. Nutzerdefinierte Formen via QPainterPath

Man kann beliebige eigene Symbolformen setzen, indem man die Klasse `QPainterPath` verwendet. Folgendes Beispiel generiert ein Glühlampensymbol:

```
// Symbol hinzufügen
QwtSymbol * symbol = new QwtSymbol(QwtSymbol::Path);
QPainterPath p;
p.addEllipse(QRectF(-10,-10,20,20));
p.moveTo(-7,-7);
p.lineTo(7,7);
p.moveTo(7,-7);
p.lineTo(-7,7);
symbol->setPath(p);
symbol->setPen(QColor(0,0,120), 2);
symbol->setBrush(QColor(160,200,255));
curve->setSymbol(symbol); // Curve takes ownership of symbol
```

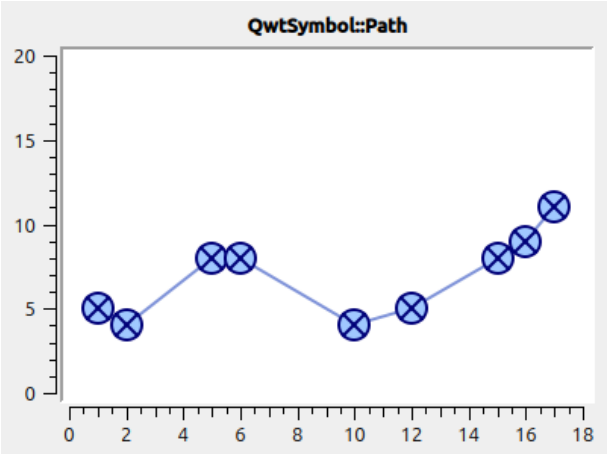



*Abbildung 27. Eigenes Symbol definiert mittels QPainterPath*

Wenn man eine nicht-rechteckige Geometrie mit QPainterPath definiert, sollte man beim Ändern der Größe mittels `QwtSymbol::setSize()` die Variante mit den zwei Argumenten aufrufen, also `QwtSymbol::setSize(width,height)`. Die Variante mit nur einem Argument transformiert den Pfad onst auf ein Quadrat. Da man üblicherweise keine nicht-quadratischen Formen als Symbol definiert, dürfte das aber nur selten ein Problem sein.

### 5.3.3. SVG-Symbole

Man kann eigene SVG-Dateien rendern und anzeigen lassen. Dafür muss man nur eine SVG-Datei einlesen/definieren und als Symbol setzen:

```
QwtSymbol * symbol = new QwtSymbol(QwtSymbol::SvgDocument);
QFile f("symbol.svg");
```

```
f.open(QFile::ReadOnly);
QTextStream strm(&f);
QByteArray svgDoc = strm.readAll().toLatin1();
symbol->setSvgDocument(svgDoc);
curve->setSymbol(symbol); // Curve takes ownership of symbol
```

Auch hier ist beim Festlegen der Größe wieder auf das Seitenverhältnis zu achten und zumeist die Variante `QwtSymbol::setSize(width,height)` zu verwenden.

Manchmal liegt der Ankerpunkt des SVG-Bildes nicht im Zentrum, wie in obigem Beispiel:

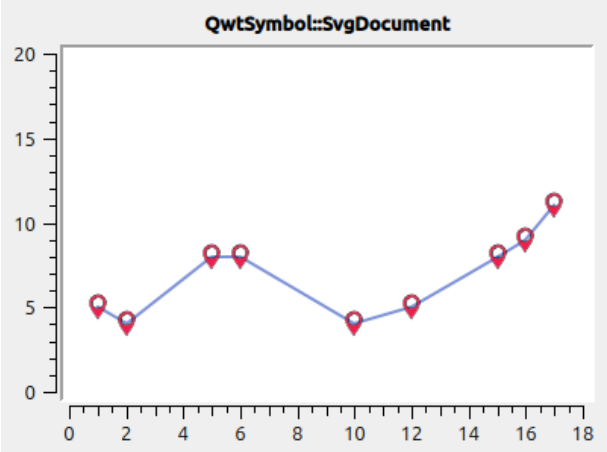



*Abbildung 28. Zentriertes SVG-Symbol, welches eigentlich aber nach oben verschoben sein sollte*

Man kann den Ankerpunkt bzw. den Zentrierpunkt des Symbols aber mit `QwtSymbol::setPinPoint()` ändern. Die Koordinaten des PinPoint werden dabei von links/oben des SVG-Bildes gemessen:

```
...
QRect br = symbol->boundingRect(); // size of symbol
symbol->setPinPoint(QPointF(br.width()/2-1,br.height()-3));
```

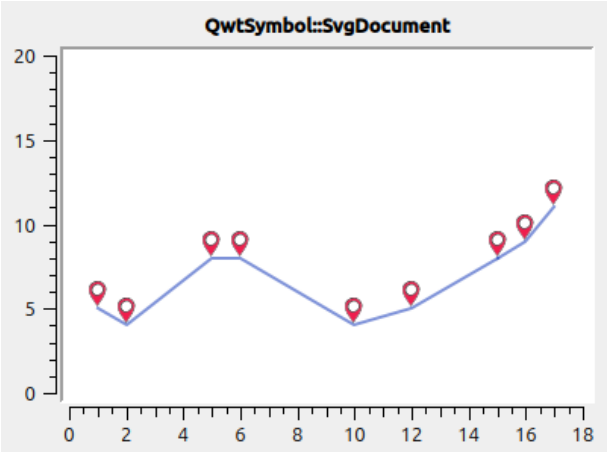



*Abbildung 29. SVG-Symbol mit korrekter Ausrichtung des "Stecknadel"-Punktes*

Man kann die manuell gesetzten Ankerpunkt auch wieder mit `QwtSymbol::setPinPointEnabled(false)` deaktivieren.

### 5.3.4. Bild-Symbole (Pixmaps)

Alternativ zu eigenen Vektorgrafiksymbolen kann man auch beliebige Bilder als Symbole verwenden. Dies geschieht analog zu den SVG-Symbolen:

```
QwtSymbol * symbol = new QwtSymbol(QwtSymbol::Pixmap);
QwtText t("QwtSymbol::Pixmap");
QPixmap pixmap;
pixmap.load("symbol.png");
symbol->setPixmap(pixmap);
QRect br = symbol->boundingRect(); // size of symbol
symbol->setPinPoint(QPointF(br.width()/2,br.height()-1));
curve->setSymbol(symbol); // Curve takes ownership of symbol
```

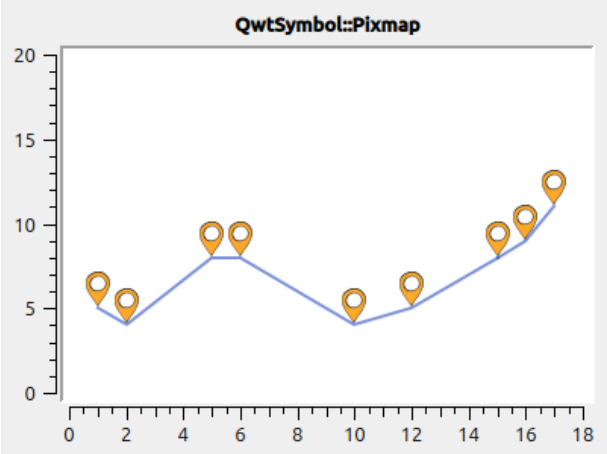



*Abbildung 30. Pixmap-Symbol, auch mit manuell festgelegtem "Stecknadel"-Punkt*

## 5.4. Ausgefüllte Kurven

Eine Kurve kann neben dem Zeichenstift auch noch einen Brush übernehmen. Dann wird die Kurve bis zur x-Achse gefüllt:

```
curve->setBrush(QColor(0xa0d0ff));
```

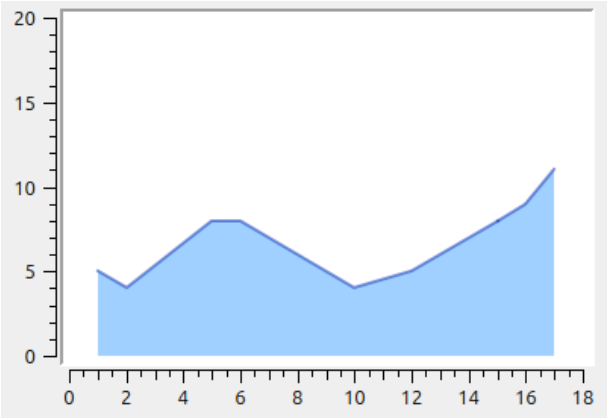



*Abbildung 31. Gefüllte Linienkurve*

Man kann die Bezugslinie für die Füllung auch noch verschieben:

```
curve->setBaseLine(8);
```

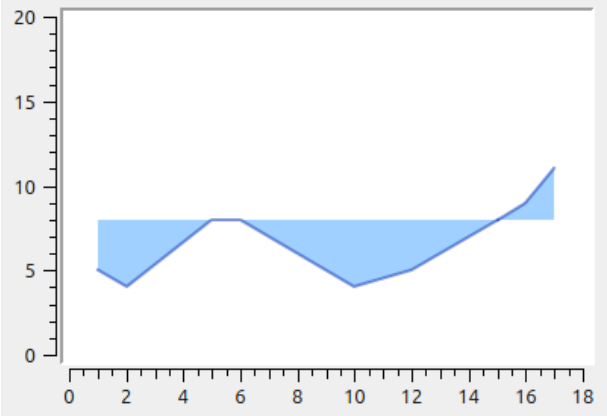



*Abbildung 32. Gefüllte Linienkurve mit verschobener Bezugslinie*

## 5.5. Legendeneinträge

Jede `QwtPlotCurve` erstellt ihr eigenes Icon zur Anzeige in der Legende. Der in der Legende angezeigte Text wird mit `QwtPlotItem::setTitle()` gesetzt. Um die Legende anzuzeigen, muss man wie in Kapitel 8 beschrieben zunächst eine Legende erstellen und ins Plot einfügen.

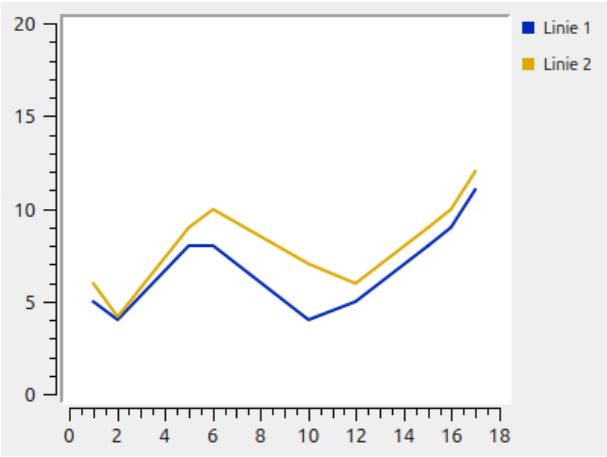



*Abbildung 33. Standardlegende für Datenreihen*

Zusätzlich kann noch mit `QwtPlotCurve::setLegendAttribute()` festgelegt werden, ob ein ausgefülltes Rechteck, eine Line oder das Reihensymbol in der Legende gezeichnet wird.

Diese Eigenschaften werden für jede `QwtPlotCurve` individuell gesetzt:

```
curve->setTitle("Linie 1");
curve->setLegendAttribute(QwtPlotCurve::LegendShowLine, true);
```

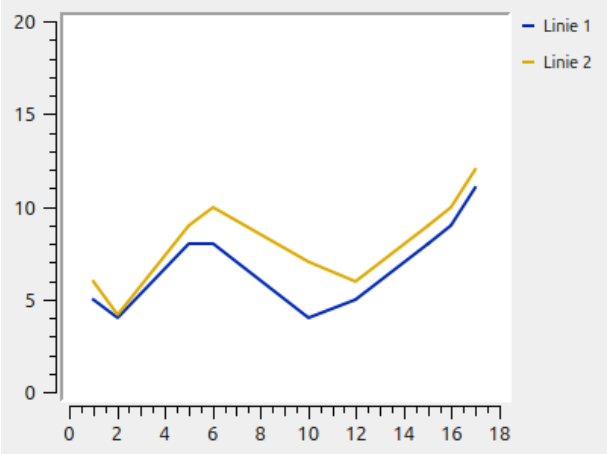



*Abbildung 34. Legende mit Linien als Icons*

Bei Linien mit Markierungen kann man auch die Markierungen zeichnen:

```
curve->setTitle("Linie 1");
curve->setLegendAttribute(QwtPlotCurve::LegendShowLine, true);
curve->setLegendAttribute(QwtPlotCurve::LegendShowSymbol, true);

QwtSymbol * symbol = new QwtSymbol(QwtSymbol::Rect);
symbol->setSize(6);
symbol->setPen(QColor(0,0,160), 1);
symbol->setBrush(QColor(160,200,255));
curve->setSymbol(symbol); // Curve takes ownership of symbol
```

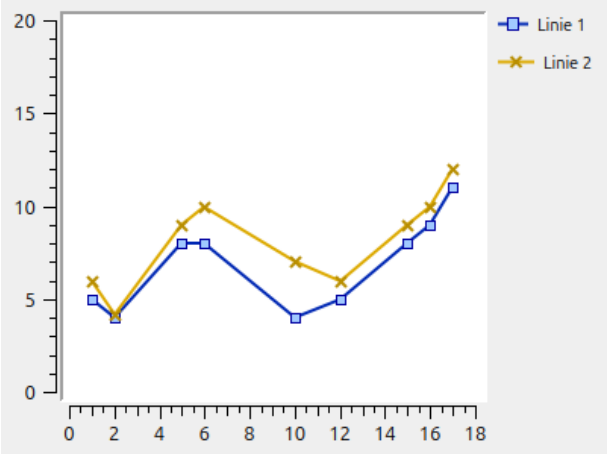



*Abbildung 35. Legende mit Linien und Symbolen als Legendenicons*

Wenn man unterschiedliche Liniensymbole verwendet, dann wirkt sich das auf die automatisch bestimmte Legendenicongröße aus. Somit sind dann die Legendentitel manchmal wie im Screenshot oben nicht perfekt ausgerichtet.

Man kann aber die Größe der Legendenicons mit `QwtPlotItem::setLegendIconSize()` einheitlich ändern.

```
curve->setTitle("Linie 1");
curve->setLegendAttribute(QwtPlotCurve::LegendShowLine, true);
curve->setLegendAttribute(QwtPlotCurve::LegendShowSymbol, true);

QwtSymbol * symbol = new QwtSymbol(QwtSymbol::Rect);
symbol->setSize(6);
```

```
symbol->setPen(QColor(0,0,160), 1);
symbol->setBrush(QColor(160,200,255));
curve->setSymbol(symbol); // Curve takes ownership of symbol

// einheitliche Legendenicon-Breite unabhängig vom gewählten Symbol
curve->setLegendIconSize(QSize(30,16));
```

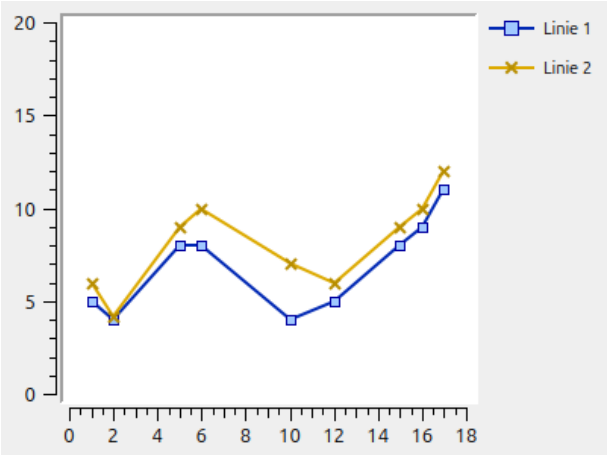



*Abbildung 36. Legende mit gleichgroßen Legendenicons*

Da die Legendenicons beim Setzen von Kurvensymbolen konfiguriert werden, muss die Änderung der Legendenicongrößen stets *nach* dem Setzen/Anpassen von Kurvensymbolen gemacht werden. Also der Aufruf von `setLegendIconSize()` muss *nach* `setSymbol()` erfolgen!

Alle Informationen zu Legenden und auch spezifische Anpassungen, z.B. wie man eigene Icons zeichnet, gibt es in Kapitel 8.

## 5.6. Zeichenattribute, Zeichengeschwindigkeit und Performanceoptimierung bei großen Datenreihen

Das Zeichnen großer Datenreihen kann mitunter einige Zeit beanspruchen. Vor allem beim Größenändern des Plotfensters wirkt sich das dann durch merkliche Verzögerung beim Bildschirmaufbau aus. Je größer die Datenreihen sind, d.h. umsomehr Punkte enthalten sind, umso länger dauert im Allgemeinen das Zeichnen des Plots. In diesem Abschnitt soll es um verschiedene Eigenschaften und Fähigkeiten der `QwtPlotCurve` gehen, die sich direkt auf die Zeichengeschwindigkeit auswirken.

### 5.6.1. Zeichenattribute (PaintAttribute / PaintAttributes)

Die Klasse `QwtPlotCurve` hat verschiedene Zeichenattribute, die verwendete Algorithmen näher konfigurieren. Hier ist zunächst eine kurze Liste, wobei die individuellen Attribute und deren Auswirkung später näher beschrieben werden:

- `ClipPolygons` - damit werden zu zeichnende Datenreihen ausgedünnt, sodass nur im aktuellen Zoomfenster sichtbare Kurventeile gezeichnet werden. Ist standardmäßig aktiv und ist für den SVG-Export notwendig. Sonst ist dieses Herausfiltern außerhalb liegender Punkte vor allem beim Hineinzoomen wirkungsvoll.
- `FilterPoints` - damit werden Punkte/Liniensegmente aus den Datenreihen herausgefiltert, die sowieso nicht gezeichnet werden würden, weil sie übereinander liegen.
- `FilterPointsAggressive` - *Nur* für Linientyp `QwtPlotCurve::Linien`: Ersetzt viele kurze übereinanderliegende Polygonstücke durch einzelne Liniensegmente
- `MinimizeMemory` - *Nur* beim Linientyp `QwtPlotCurve::Dots`: Verhindert wenn eingeschaltet, dass bei der Transformation der Datenreihe von Plotkoordinaten zu Renderkoordinaten eine eventuell große Polygon-Kopie erstellt wird. Stattdessen werden alle Punktkoordinaten einzeln transformiert, geclippt und gezeichnet. Da hierbei für jeden Punkt ein gewisser Extraaufwand benötigt wird, ist dies in der Regel etwas langsamer.
- `ImageBuffer` - *Nur* beim Linientyp `QwtPlotCurve::Dots`: Eine sehr spezielle Optimierung, wobei die Punkte (jeweils nur ein Pixel pro


Andreas Nicolai <nicolai@gmx.de>, https://schneggenport.de

Punkt) erst in ein `QImage` gezeichnet werden. Dies erfolgt parallelisiert und kann daher sehr schnell sein, ist allerdings nur effektiv bei sehr großen Punktmengen.

Das Zeichenattribut `FilterPointsAggressive` wirkt sich nur auf Linientyp `QwtPlotCurve::Lines` aus. Die Attribute `MinimizeMemory` und `ImageBuffer` sind sehr spezielle Optimierungen für den Linientyp `QwtPlotCurve::Dots`.

### 5.6.2. Punktfilter

Durch Verwendung des Datenfiltes `FilterPoints` kann man bei sehr großen Reihendaten nur die Daten wirklich zeichnen, die sich auf das Diagramm auswirken. Wenn bspw. 100 Datenpunkte auf dem gleichen Pixel landen, braucht man die nicht alle zeichnen. Das Ausfiltern nicht sichtbarer Punkte und Liniensegmente hängt natürlich von der Zoomstufe ab und ist deshalb in die Zeichenroutine der `QwtPlotCurve` integriert.

Neben dem Filtern überflüssiger Punkte durch Setzen des Zeichenattributs `FilterPoints` gibt es auch noch die Möglichkeit, den `QwtWeedingCurveFitter` (siehe Kapitel 5.7.3) zu verwenden. Dieser ist aber eigentlich für die Datenreduktion beim Export gedacht und wegen des zusätzlichen Berechnungsaufwands dauert das Zeichnen auf dem Bildschirm in der Regel länger.

Das Zeichenattribut `QwtPlotCurve::FilterPoints` ist standardmäßig gesetzt und führt dazu, dass beim Transformieren der Datenreihenkoordinaten in Bildschirmkoordinaten bereits doppelte Punkte herausgefiltert werden (dies macht die Klasse `QwtPointMapper`). Enthält eine Datenreihe bspw. 100000 gleichverteilte Punkte entlang der x-Achse und wird auf einem 1000 Pixel breiten Plot gezeichnet, so teilen sich jeweils 100 Punkte eine x-Pixelkoordinate. Wenn nach der Umrechnung der Koordinaten zwei aufeinanderfolgende Punkte identische Pixelkoordinaten haben, werden die doppelten Punkte entfernt. Diese standardmäßig eingeschaltete Funktion macht bereits einen großen Unterschied bei der Zeichengeschwindigkeit.

Um das mal zu verdeutlichen, ist hier ein kleines Testprogramm für die Filterfunktionen:

*Testprogramm für den Filteralgorithmus mit 10 Millionen Punkten in der Datenreihe*

```
#include <QApplication>
#include <QElapsedTimer>
#include <QDebug>

#include <cmath>

#include <qwt_plot.h>
#include <qwt_plot_curve.h>
#include <qwt_weeding_curve_fitter.h>

// Spezialisierte QwtPlotCurve mit Zeitmessung um drawCurve()
class BenchmarkedPlotCurve : public QwtPlotCurve {
protected:
    void drawCurve(QPainter *p, int style,
        const QwtScaleMap & xMap, const QwtScaleMap & yMap,
        const QRectF & canvasRect, int from, int to) const override
    {
        QElapsedTimer timer;
        timer.start();
        QwtPlotCurve::drawCurve(p, style, xMap, yMap, canvasRect, from, to);
        qDebug() << "QwtPlotCurve::drawCurve(): " << timer.elapsed() << "ms";
    }
};


int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QwtPlot plot;
    plot.setContentsMargins(8,8,8,8);
    plot.setCanvasBackground( Qt::white );

    // Daten zum Darstellen generieren
    QVector<double> x, y;
```

```
    for (unsigned int i=0; i<10000000; ++i) {
        x.append(i);
        y.append(std::sin(i*0.00001));
    }

    QwtPlotCurve *curve = new BenchmarkedPlotCurve();
    curve->setPen(QColor(180,40,20), 1);
    curve->setRenderHint( QwtPlotItem::RenderAntialiased, true); // Antialiasing verwenden
    curve->setPaintAttribute(QwtPlotCurve::FilterPoints, false); // Punktefilter ausschalten
    curve->setSamples(x, y);
    curve->attach(&plot); // Plot takes ownership

    plot.resize(1000,800);
    plot.show();

    return a.exec();
}
```

Das Testprogramm generiert 10 Millionen Datenpunkte (mehrere Sinuswellen) und zeichet diese dann in ein 1000x800 Pixel großes Diagramm, zunächst *ohne* Punktefilter. Dazu wird in Zeile

```
curve->setPaintAttribute(QwtPlotCurve::FilterPoints, false);
```

das Zeichenattribut `FilterPoints` ausgeschaltet. Die Zeichenzeit für die Kurve einschließlich der Zeit für die Datenfilterung wird in der abgeleiteten `QwtPlotCurve` und dem kleinen Wrapper um die zentrale `drawCurve()`-Funktion gemessen.

Auf meinem Rechner gibt das Programm im Release-Build ca. **650 ms** aus. Dabei werden wirklich alle 10 Millionen Punkte in den zu zeichnenden Polygonzug übernommen und auf dem Painter gezeichnet (der natürlich Liniensegmente mit Länge 0 verwirft, aber dafür etwas Zeit braucht).

Schaltet man nun das Attribut `FilterPoints` wieder ein, so enthält das Polygon *nach* dem Filtern nur noch ca. 24500 Punkte und das Zeichnen dauert ca. **300 ms**.

Mit dem Zeichenattribut `FilterPoints` werden Liniensegmente mit Länge 0 herausgefiltert, was bei sehr großen Datenreihen eine merkliche Zeichenbeschleunigung bewirkt.

### 5.6.3. Aggressives Punktefiltern

Wenn Datenreihen sehr stark rauschen und gleichzeitig sehr große Datenmengen enthalten, dann können sich mehrere Liniensegmente überlagern. Zum Beispiel werden dann mehrere Linien mit der gleichen x-Bildschirmkoordinate übereinander gezeichnet, obwohl ja eine Linie gezeichnet zwischen minimaler und maximaler y-Koordinate ausreichen würde.

Das Zeichenattribut `FilterPointsAggressive` schaltet eine Vorberechnung ein, die genau diese Art der Optimierung durchführt (ebenfalls in Klasse `QwtPointMapper` implementiert) und aus mehreren sich überlagernden, vertikalen Liniensegmenten nur eine einzige Linie macht (deswegen ist diese Optimierung auch nur für den Linientyp `QwtPlotCurve::Lines` sinnvoll).

Um das mal zu testen, wandeln wir das Programm oben ab und generieren eine stark rauschende Kurve:

```
QVector<double> x, y;
for (unsigned int i=0; i<1000000; ++i) {
    x.append(i);
    y.append(QRandomGenerator64::global()->generateDouble());
}
```

Ohne `FilterPointsAggressive` dauert das Zeichnen ca. **1500 ms**. Obwohl die Datenreihe "nur" aus 1 Mio Punkten besteht, sind die Liniensegmente deutlich länger als bei der Sinuswelle und es dauert länger, diese zu zeichnen. Das Zeichenattribut `FilterPoints` kann auch wegen der stark schwankenden y-Werte kaum Punkte rausschmeißen (die Liniensegmente haben fast nie eine Länge von 0).

Wenn man nun das Attribut mit

```
curve->setPaintAttribute(QwtPlotCurve::FilterPointsAggressive, true);
```

einschaltet, verkürzt sich die Zeichenzeit auf ca. **32 ms** !!!! Nach dem Filtern hat das zu zeichnende Polygon nur noch ca. 3670 Punkte.

Die Verwendung des Zeichenattributs `FilterPointsAggressive` beschleunigt die Darstellung drastisch, vor allem beim Kurven mit vielen Datenpunkten und stark rauschenden Werten.

Wenn man sich die Diagramme mit und ohne `FilterPointsAggressive` im Vergleich anschaut, so sieht man kleinere Unterschiede.

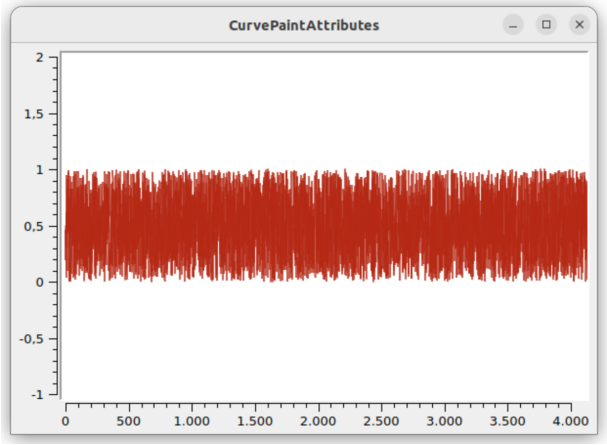



*Abbildung 37. Ausgabe mit* `FilterPoints`*, Antialiasing eingeschaltet*

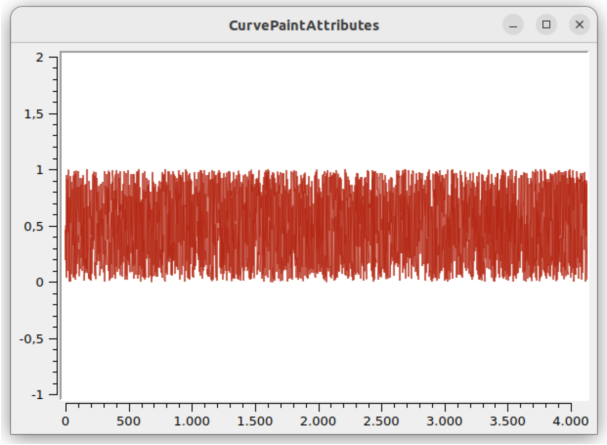



*Abbildung 38. Ausgabe* `FilterPointsAggressive`*, Antialiasing eingeschaltet*

Mit Antialiasing sieht man Unterschiede in den teiltransparenten Segmenten. Ohne Antialiasing muss man schon sehr genau hinsehen, um die kleinen Unterschiede zu sehen.

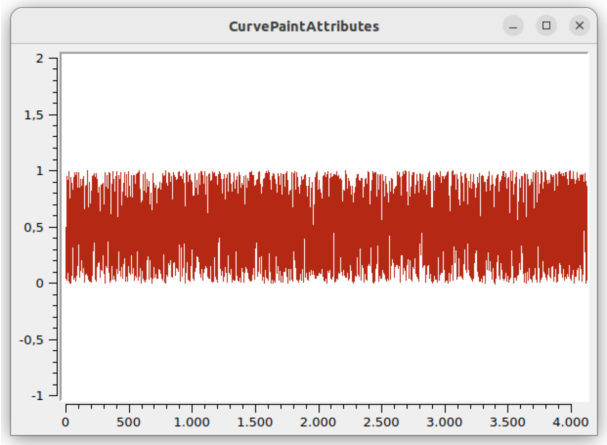



*Abbildung 39. Ausgabe mit* `FilterPoints`*, Antialiasing ausgeschaltet*

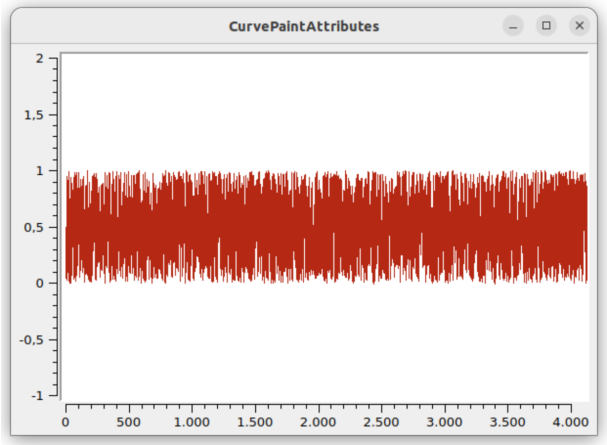



*Abbildung 40. Ausgabe* `FilterPointsAggressive`*, Antialiasing ausgeschaltet*

(Zum Vergleich beide Bilder herunterladen und im Bildbetrachter abwechselnd anzeigen, dann sieht man die kleinen Unterschiede.)

Im Gegensatz zu `FilterPoints` ist das Ergebnis des `FilterPointsAggressive` aber eine leicht veränderte Darstellung. Das liegt an der Art, wie Liniensegmente zusammengefasst werden.

Nach der Skalierung von Plotkoordinaten zu Bildschirmkoordinaten liegt beispielsweise folgendes Polygon zum Filtern vor:

```
x      y
----------
6      230
6      379
6      602
7      304
7      602
7       81
7      155
8      424
...
```

so enthält das Polygon *nach* dem Filtern nur noch die Punkte:

```
x      y
----------
6      230
```

```
6       602
7       304
7       602
7       81
8       424
...
```

Die Liniensegmente auf den x-Zeichenkoordinaten 6 und 7 wurden hier zusammengefasst.

Aber während vorher eine Linie von (7,155) (8,424) gezeichnet wurde, wird nun eine Linie von (7,81) (8,424) gezeichnet, welches im Detail ein minimal anderes Erscheinungsbild gibt. Die Unterschiede in der Ausgabe sind aber so klein, dass man für die relevanten Anwendungsfälle, also größere Datenreihen und stark schwankende Werte, das Einschalten des Zeichenattributs `FilterPointsAggressive` durchaus generell empfehlen kann.

### 5.6.4. Polygon-Clipping

Je nach Wahl des Plotausschnitts, festzulegen über Achsenskalierung (siehe Kapitel 10) oder interaktiv durch den Plotzoomer (siehe Kapitel 12), werden mitunter nur Teile von Kurven gezeichnet. In diesem Fall ist es sinnvoll, das Polygon nur auf die sichtbaren Bereiche zu begrenzen. Diese Funktionalität bietet die Klasse `QwtPlotCurve` durch Einschalten des Zeichenattributs `QwtPlotCurve::ClipPolygons`.

Beim Zeichnen des Plots auf dem Bildschirm könnte man das Polygon-Clipping auch weglassen, da der Qt-Painter selbst ein Clipping durchführt. Beim Datenexport in eine SVG-Datei (siehe Kapitel 14) ist das Clipping aber zwingend notwendig.

Standardmäßig ist das Attribut `QwtPlotCurve::ClipPolygons` eingeschaltet. Das Clipping wird übrigends erst *nach* dem Punktfilteralgorithmus angewendet, sodass eine Datenreduktion in diesem Schritt sich positiv auf die Zeit für das Clipping auswirkt.

Interessanterweise wirkt sich das Polygon-Clipping beim Rendern auf den Bildschirm nur unmerklich auf die Renderperformance aus.

Beim 10 Mio Sinuskurvenplot oben (Antialiasing an, FilterPoints an, FilterPointsAggressive aus, ClipPolygons an) ist die Zeichenzeit für das komplette Diagramm ca. **300 ms**. Hineingezoomt mit

```
plot.setAxisScale(QwtPlot::xBottom, 150000, 160000);
plot.setAxisScale(QwtPlot::yLeft, 0.99, 1);
```

erhöht sich die Renderzeit auf ca. **370 ms**. Wenn man nun `ClipPolygons` ausschaltet, so dauert das **290 ms** beim vollständigen Plot und **410 ms** beim hineingezoomten Plot.

Im hineingezoomten Zustand dauert der Clipping-Algorithmus länger, dafür ist das Zeichnen etwas schneller. Beide Effekte arbeiten gegeneinander aber im Ergebnis ist es bei meinem Testfall minimal langsamer als im herausgezoomten Zustand).

Der Unterschied zwischen **300 ms** und **290 ms** beim vollständigen Plot liegt an dem Oberhead für die Polygon-Clipping-Funktion, die natürlich beim vollständig sichtparen Plot keine Wirkung hat.

Im stark hineingezoomten Zustand zeichnet das Plot mit eingeschaltetem Polygon-Clipping immerhin ca. **40 ms** schneller, was aber selten signifikant sein sollte.

Dein Einfluss des Attributs `ClipPolygons` auf die Zeichenperformance ist marginal, sodass man dieses Attribut getrost immer eingeschaltet lassen kann.

Lediglich beim Ausschalten des `FilterPoints`-Attributs und sehr großen Datenmengen könnte das `ClipPolygons` im hineingezoomten Zustand etwas bewirken. Zum Beispiel rendert das Plot *ohne* `FilterPoints` und *ohne* `ClipPolygons` im hineingezoomten Zustand (siehe oben) in **550 ms**, gegenüber **360 ms** *mit* `ClipPolygons`. Aber selbst in diesem eher unrealistischen Szenario ist die Auswirkung des Attributs klein.

Fazit: `ClipPolygons` eingeschaltet lassen!

## 5.7. Kurvenfilter/Kurvenanpasser

Die `QwtPlotCurve` kann die übergebenen Rohdaten vor dem eigentlichen Rendern noch an einen Algorithmus übergeben, der die Daten glättet oder eine kontinuierliche Kuve durch die Datenpunkte legt. Diese Operationen hängen vom aktuellen Zoomlevel und der Plotgröße ab, denn je nach Auflösung wird der Verlauf der angepassten Kurve neu berechnet. Dies gibt eine bessere Qualität als beim Vorberechnen der Daten und Plotten eines Linienzugs durch vorberechnete Kurven. Deshalb ist diese Funktionalität direkt in das `QwtPlot` integriert.

Einen solchen Anpass-/Filteralgorithmus übergibt man der Kurve, indem man eine Klasse, abgeleitet von `QwtCurveFitter` der Kurve übergibt.

Es werden verschiedene Implementierungen dieser Schnittstelle mitgeliefert:

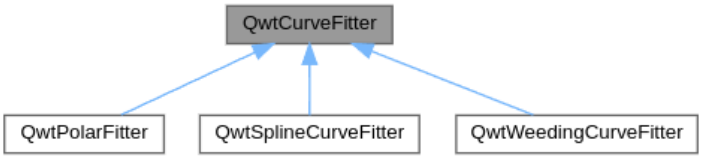



*Abbildung 41. Kind-Klassen der Basisklasse* `QwtCurveFitter`

### 5.7.1. Kurvenglättung/Spline-Interpolation

Schauen wir uns zuerst den `QwtSplineCurveFitter` an. Diese Klasse kapselt einen Algorithmus, der aus den gegebenen Stützstellen der Kurve einen weichen Verlauf berechnet. Dafür gibt es verschiedene mathematischen Algorithmen.

Jeder dieser Algorithmen ist in einer von `QwtSpline` abgeleiteten Klasse implementiert.

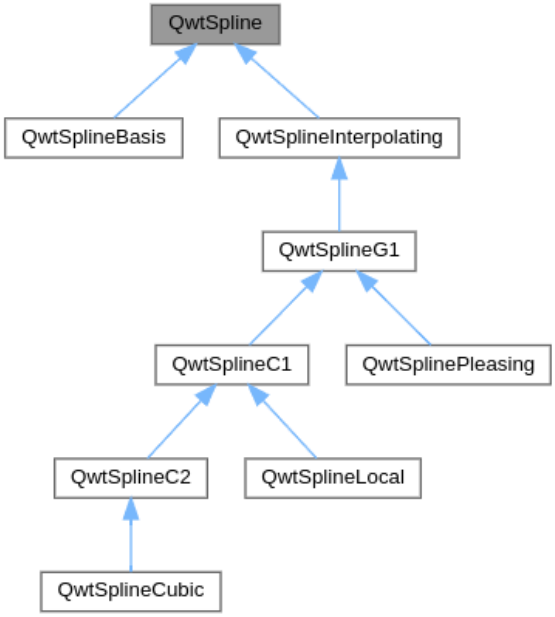



*Abbildung 42. Kind-Klassen der Basisklasse* `QwtSpline`

Das Ergebnis lässt sich am Besten mit einem einfachen parametrischen Datensatz veranschaulichen:

```
QVector<double> x{1,2,5,6,10,12,15,16,8};
QVector<double> y{5,4,8,8, 4, 5, 8, 9,10};
// Kurve hinzufügen
curve = new QwtPlotCurve();
curve->setStyle(QwtPlotCurve::Lines);
curve->setPen(QColor(0,220,20), 2);
```

```
curve->setRenderHint( QwtPlotItem::RenderAntialiased, true ); // Antialiasing verwenden
curve->setSamples(x, y);

// SplineFitter-Objekt erstellen
QwtSplineCurveFitter * splineFitter = new QwtSplineCurveFitter;
// Spline-Implementierung auswählen, hier QwtSplinePleasing
QwtSplinePleasing * spline = new QwtSplinePleasing();
// Splinealgorithmus setzen
splineFitter->setSpline(spline); // takes ownership
// SplineFitter-Objekt der Kurve geben
curve->setCurveFitter(splineFitter); // takes ownership
// fitting einschalten
curve->setCurveAttribute(QwtPlotCurve::Fitted, true);

curve->attach(&plot); // takes ownership
```

Wichtig ist, dass die Verwendung des Kurvenanpassers/Filters explizit mit

```
curve->setCurveAttribute(QwtPlotCurve::Fitted, true);
```

eingeschaltet wird.

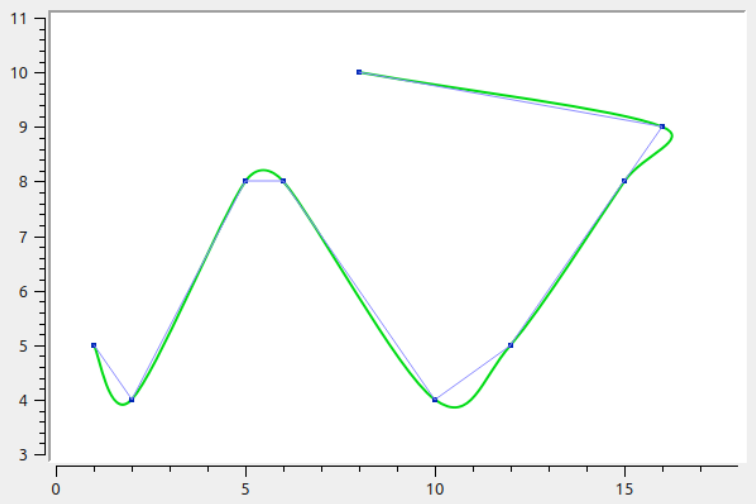



*Abbildung 43. QwtSplineCurveFitter mit QwtSplinePleasing*

Die Qwt-Bibliothek bringt eine ganze Reihe verschiedener Algorithmen und passender Implementierungen mit:

- `QwtSplinePleasing`: *"QwtSplinePleasing is some sort of cardinal spline, with non C1 continuous extra rules for narrow angles. It has a locality of 2. The algorithm is the one offered by a popular office package."*
- `QwtSplineLocal`: *"QwtSplineLocal offers several standard algorithms for interpolating a curve with polynomials having C1 continuity at the control points. All algorithms are local in a sense, that changing one control point only few polynomials."*

  `Cardinal`: *"The cardinal spline interpolation is a very cheap calculation with a locality of 1"*

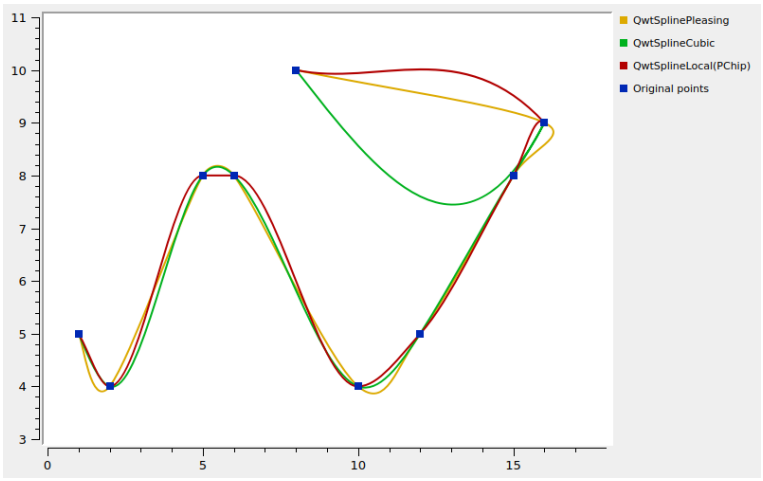
  `ParabolicBlending`: *"Parabolic blending is a cheap calculation with a locality of 1. Sometimes is also called Cubic Bessel interpolation."*

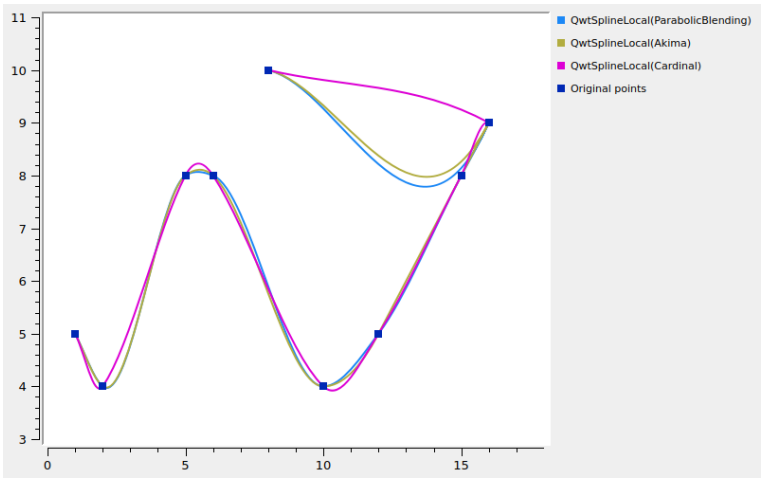
  `Akima`: *"The algorithm of H.Akima is a calculation with a locality of 2."*

  `PChip`: *"Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) is an algorithm that is popular because of being offered by MATLAB. It preserves the shape of the data and respects monotonicity. It has a locality of 1."*
- `QwtSplineCubic`: *"A cubic spline is a spline with C2 continuity at all control points. It is a non local spline, what means that all polynomials are changing when one control point has changed."*

Nur `QwtSplinePleasing` funktioniert für wirklich parametrische Kurven, auch wenn die x-Werte nicht monoton steigen.

Zum Vergleich sind die verschiedenen Spline-Algorithmen dargestellt:




Obwohl die verschiedenen Spline-Implementierungen primär für das *QwtPlot* gedacht sind, spricht nichts dagegen, diese für allgemeine Splineinterpolationsaufgaben einzusetzen. Was die Spline-Klassen in der Qwt-Bibliothek können, ist im Detail im Kapitel 15.2 beschrieben.

### 5.7.2. PolarCurveFitter

Dieser Kurvenfilter ist für Kreisdiagramme gedacht, siehe [sec:polarPlots].

### 5.7.3. Datenreduktionsfilter

Wie ja schon in Kapitel 5.6 beschrieben, bringt die Klasse `QwtPlotCurve` bereits eine Reihe eigener Funktionen mit, um unnötige Punkte und Liniensegmente hinauszufiltern. Diese sind aber primär für Rasterausgaben sinnvoll, also auf dem Bildschirm.

Bei der Ausgabe in Vektorformate (PDF/SVG) werden die Zeichenattributsfilter `QwtPlotCurve::FilterPoints` oder `QwtPlotCurve::FilterPointsAggressive` nicht verwendet, da sie nur bei gerundeten Koordinaten funktionieren.

Die Datenreduktionsmethoden durch Setzen der Zeichenattribute `QwtPlotCurve::FilterPoints` oder `QwtPlotCurve::FilterPointsAggressive` wirken sich nur bei Rasterplots aus. Beim Rendern in Vektorformate (PDF/SVG) werden bei `QwtPlotCurve::FilterPoints` *nur wirklich identische* Punkte im originalen Datensatz herausgefiltert.

Man kann aber zur Datenreduktion bei Vektorexporten einen alternativen Punktfilteralgorithmus verwenden, um nicht notwendige Punkte zu entfernen. Ein solcher Datenreduktionsfilter ist in der Klasse `QwtWeedingCurveFitter` implementiert. Dieser ist wirklich ein *Filter* und entfernt Datenpunkte nach gewissen Regeln.

Konkret versucht der Algorithmus alle Punkte zu entfernen, die bereits bei linearer Interpolation zwischen den benachbarten Punkten hinreichend gut approximiert werden.

Testweise wird wieder eine Sinuswelle generiert:

```
QVector<double> x, y;
for (unsigned int i=0; i<10000000; ++i) {
    x.append(i);
    y.append(std::sin(i*0.000001)*330);
}
```

Das Beispiel aus Kapitel 5.6 wird nun um den `QwtWeedingCurveFitter` erweitert:

```
QwtWeedingCurveFitter * weedingFitter = new QwtWeedingCurveFitter;
curve->setCurveFitter(weedingFitter);
curve->setCurveAttribute(QwtPlotCurve::Fitted, true);
```

Verwendet wird dieser Kurvenfilter wie die bisherigen Kurvenfilter, wobei man auch hier nicht vergessen darf, das Kurvenattribut `QwtPlotCurve::Fitted` zu setzen.

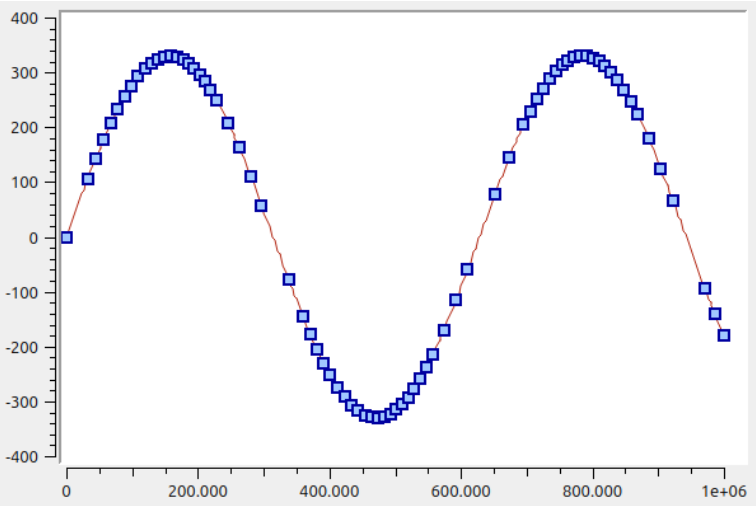



*Abbildung 44. Datenreduktion mit dem QwtWeedingCurveFitter*

Der WeedingCurveFitter reduziert die Anzahl der Datenpunkte wie hier bei eher weichen Kurven deutlich. Im Beispiel oben bleiben von den ursprünglich 10 Mio Punkten gerade mal 78 übrig. Zur Visualisierung der resultierenden Stützstellen wurde eine zweite Datenreihe erzeugt, wobei aus den Datenpunkten mit der `QwtWeedingCurveFitter`-Klasse direkt die Daten vorab reduziert wurden:

```
// use curve fitter to fit the curve
QPolygonF poly;
for (int i=0; i<x.count(); ++i)
    poly << QPointF(x[i],y[i]);
poly = weedingFitter->fitCurve(poly);
symbolCurve->setSamples(poly);
```

Die Funktion `QwtWeedingCurveFitter::fitCurve(QPolygonF)` liefert das entsprechend reduzierte Polynom zurück.

Ein wichtiger Parameter ist die erlaubte Toleranz, zu setzen im Konstruktor oder mittels `QwtWeedingCurveFitter::setTolerance(double)`. Diese Toleranz ist ein *Absolutwert* basierend auf den y-Koordinaten im übergebenen Polygon. Je nach Größenordnung der y-Werte der Punkte werden die Punkte anders gefiltert. Dies bewirkt, dass bei größeren Plots mit entsprechend größerer Pixelauflösung in y-Richtung auch mehr Stützstellen generiert werden und die Qualität des Diagramms nicht leidet.

Der Parameter `tolerance` wirkt sich dabei grob wie folgt aus, wenn die y-Werte in der Größenordnung 100 vorliegen (wie im obigen Beispiel).

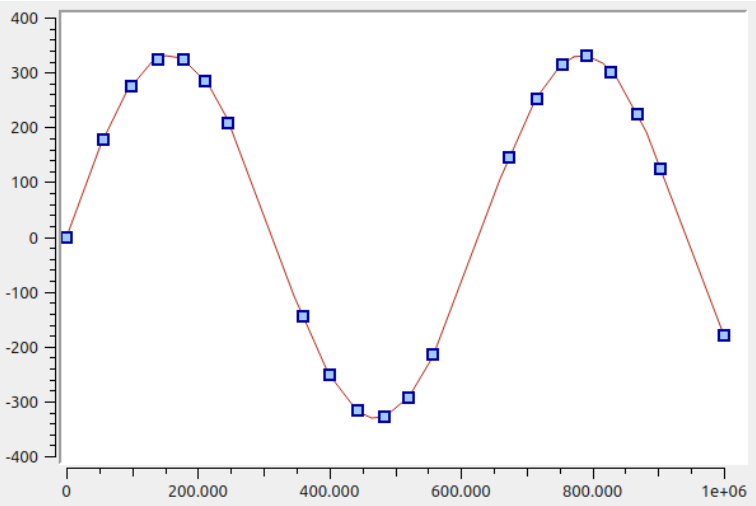



*Abbildung 45. Toleranz beim QwtWeedingCurveFitter 10, resultierende Punktzahl 21*

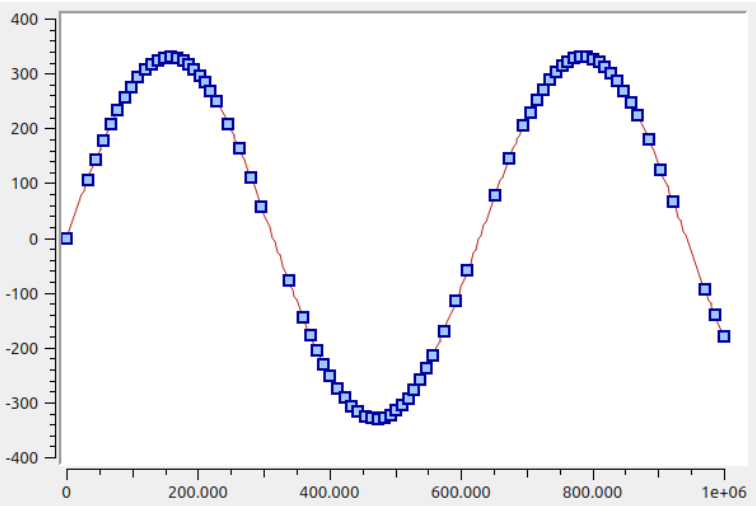



*Abbildung 46. Toleranz beim QwtWeedingCurveFitter 1, resultierende Punktzahl 78*

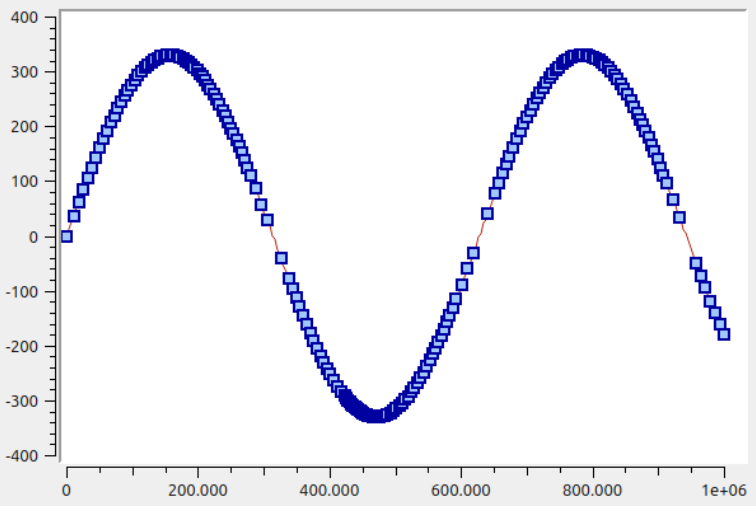



*Abbildung 47. Toleranz beim QwtWeedingCurveFitter 0.1, resultierende Punktzahl 194*

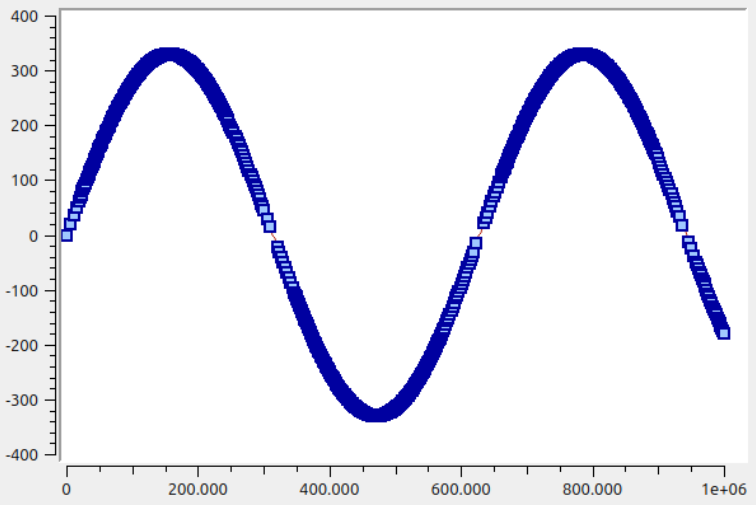



*Abbildung 48. Toleranz beim QwtWeedingCurveFitter 0.01, resultierende Punktzahl 684*

Der zweite Parameter der `QwtWeedingCurveFitter`-Klasse ist die *chunk size*, gesetzt mit `QwtWeedingCurveFitter::setChunkSize(int)`.

Das Originalpolynom wird dabei in Stücke mit der gegebenen Brockengröße zerteilt und der Algorithmus wird auf die einzelnen Stücke individuel angewendet. Dabei wird dann pro Brocken aber *mindestens ein Punkt* generiert. Würde man also beim Beispiel oben eine Chunksize von 100000 wählen, so erhält man 200 Punkte (selbst bei Toleranz 10). Der Algorithmus hat dafür etwas weniger zu tun und ist nach ca. **50 ms** fertig, statt wie bisher nach ca. **270 ms**. Wenn man aber bedenkt, dass hier aus 10 Millionen Punkten ein sehr kleines Polynom mit weniger als 100 Punkten gemacht wird, ohne dass die Datenqualität sichtbar leidet, dann ist der Algorithmus (vor allem auf heutiger Hardware) insgesamt sehr schnell.

Der `QwtWeedingCurveFitter` reduziert wirkungsvoll die Anzahl der gezeichneten Punkte im Diagramm, welches vor allem beim Export von Daten in ein Vektorformat (svg, pdf, siehe Kapitel 14) zur Reduktion der Datengröße beiträgt.

### Optimierung der Zeichengeschwindigkeit mit `QwtWeedingCurveFitter`

Man könnte nun auf die Idee kommen, und den `QwtWeedingCurveFitter` zur Beschleunigung der Bildschirmdarstellung einsetzen - immerhin sind ja weniger Punkte zu zeichnen. Das kann man mit einem kleinen Benchmarkprogramm testen:

*Beispielquelltext für das Messen der Zeichengeschwindigkeit auf dem Bildschirm bei verschiedenen Auflösungen mit und ohne QwtWeedingCurveFitter*

```
#include <QApplication>
#include <QPen>
#include <QElapsedTimer>
```

```
#include <QDebug>
#include <QTimer>

#include <cmath>

#include <qwt_plot.h>
#include <qwt_plot_curve.h>
#include <qwt_weeding_curve_fitter.h>
#include <qwt_plot_canvas.h>

class BenchmarkedPlotCanvas : public QwtPlotCanvas {
public slots:
    // slot, when called, resizes parent QwtPlot window
    void resizePlot() {
        ((QWidget*)parent())->resize(2400,1200);
    }

protected:
    void paintEvent(QPaintEvent * event) override {
        QElapsedTimer timer;
        timer.start();
        QwtPlotCanvas::paintEvent(event);
        qDebug() << "QwtPlotCanvas::paintEvent(): " << timer.elapsed() << "ms";
    }
};


class BenchmarkedWeedingCurveFitter : public  {
public:
    QPolygonF fitCurve(const QPolygonF & polygon) const override {
        QElapsedTimer timer;
        timer.start();
        const QPolygonF & stripped = QwtWeedingCurveFitter::fitCurve(polygon);
        qDebug() << "QwtWeedingCurveFitter::fitCurve():"
                 << polygon.count() << "->" << stripped.count()
                 << "points: " << timer.elapsed() << "ms";
        return stripped;
    }
};


int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QwtPlot plot;

    plot.setContentsMargins(8,8,8,8);
    // eigene Canvas-Klasse mit Benchmark-Wrapper einsetzen
    BenchmarkedPlotCanvas * canvas = new BenchmarkedPlotCanvas;
    plot.setCanvas(canvas);
    plot.setCanvasBackground( Qt::white );

    // Daten zum Darstellen generieren
    QVector<double> x, y;
    for (unsigned int i=0; i<10000000; ++i) {
        x.append(i);
        y.append(std::sin(i*0.00001));
    }

    QwtPlotCurve *curve = new QwtPlotCurve();
    curve->setPen(QColor(180,40,20), 1);
    curve->setRenderHint( QwtPlotItem::RenderAntialiased, true); // Antialiasing verwenden
    curve->setSamples(x, y);
    curve->attach(&plot); // Plot takes ownership

    // WeedingCurveFitter mit Benchmark-Wrapper einsetzen
```

```
    QwtWeedingCurveFitter * weedingFitter = new BenchmarkedWeedingCurveFitter;
    curve->setCurveFitter(weedingFitter);
    curve->setCurveAttribute(QwtPlotCurve::Fitted, true);

    plot.show();
    plot.resize(1000,800);

    QTimer::singleShot(1000, canvas, &BenchmarkedPlotCanvas::resizePlot);
    QTimer::singleShot(4000, &plot, &BenchmarkedPlotCanvas::close);

    return a.exec();
}
```

Im obigen Benchmarkprogramm sind die Klassen `QwtPlotCanvas` und `QwtWeedingCurveFitter` abgeleitet und die relevanten Zeichenroutinen mit Zeitmessung umgeben.

Wichtig zu vermerken ist, dass die Zeichenattribute `FilterPoints` und `ClipPolygons` (siehe <<>>) standardmäßig eingeschaltet sind.

Führt man das Programm aus, so sieht man beim großen Plot mit Auflösung 2400x1200, dass aus den 10 Mio Punkten durch den `QwtWeedingCurveFitter` nur noch 284 Punkte übrig bleiben. Das müsste dann doch zu einem deutlich schneller gezeichneten Plot führen, oder?

*Tabelle 1. Geschwindigkeitsvergleich mit/ohne QwtWeedingCurveFitter*

| CurveFitter | FilterPoints | ClipPolygons | Fitter-Algorithmus [ms] | Zeichnen insgesamt [ms] |
|---|---|---|---|---|
| an | an | an | 670 | 1260 |
| an | aus | an | 660 | 1200 |
| an | an | aus | 660 | 860 |
| an | aus | aus | 660 | 820 |
| aus | an | an | --- | 320 |
| aus | aus | an | --- | 870 |
| aus | an | aus | --- | 320 |
| aus | aus | aus | --- | 550 |

In keiner Variante ist die Option *mit* schneller. Außerdem fällt auf, dass unabhängig von der Option `FilterPoints` der CurveFitter immer alle 10 Mio Punkte übergeben bekommt, so wie auch der Polygon-Clipping-Algorithmus.

*Qwt 6.3.x Einschränkung*

Wenn man das Kurvenattribut `QwtPlotCurve::Fitted` einschaltet, werden automatisch die Zeichenattribute `FilterPoints` und `FilterPointsAggressive` unwirksam, da intern *das Runden auf ganzzahlige Bildschirmkoordinaten abgeschaltet wird*. Dadurch werden natürlich die Polygone nicht mehr sinnvoll ausgedünnt (siehe Kapitel 5.6) und die nachfolgenden Algorithmen müssen mit sehr sehr viel mehr Datenpunkten arbeiten.

Auch wenn der `QwtWeedingCurveFitter` letztlich das Polygon sehr stark verkleinert, ist der Mehraufwand in den Algorithmen durch den Wegfall des Punktefilterns enorm.

*Inside Qwt*

Wenn man das *Rounding-Alignment* direkt im Qwt-Quelltext trotz eingeschaltetem `Fitted` Attribut wieder aktiviert, und zwar in `qwt_plot_curve.cpp:487`:

```
const bool doFit = ( m_data->attributes & Fitted ) && m_data->curveFitter;
// original Qwt 6.3.0
// const bool doAlign = !doFit && QwtPainter::roundingAlignment( painter );
// new: alignment depends only on paint device
```

```
const bool doAlign = QwtPainter::roundingAlignment( painter );
```

sieht man den wirklichen Einfluss des `QwtWeedingCurveFitter` (`QwtWeedingCurveFitter` an, `FilterPoints` an, `ClipPolygons` an):

- `FilterPoints` reduziert die 10 Mio Punkte zu 38666
- `QwtWeedingCurveFitter` benötigt nur noch **1 ms** für die weitere Reduktion auf 200 Punkte
- Gesamtzeichenzeit liegt bei **300 ms**

Das ist marginal schneller als die Variante *ohne* `QwtWeedingCurveFitter`.

Wie man sieht, bringt selbst ein Patchen des Qwt-Quelltextes keine signifikante Verbesserung. Daher lautet das finale Fazit: der `QwtWeedingCurveFitter` ist sinnvoll als Datenreduktionsfilter für Vektorplot-Datenexporte.

### Verwendung zur Datenreduktion vorab

Man kann die Klasse `QwtWeedingCurveFitter` jedoch auch zu Datenreduktion *außerhalb* der QwtPlotCurve verwenden, d.h. man reduziert die der Kurve übergebenen Daten vorab. Und hier liegt das wirkliche Performancesteigerungspotential.

In diesem Fall benutzt man den `QwtWeedingCurveFitter`-Algorithmus, um damit die originale Datenreihe auszudünnen und der PlotCurve bereits ein kleineres Polygon zu übergeben. Folgende Quelltext nimmt die Punkte der originalen Datenreihe und überträgt sie in ein Polygon, welches dann dem `QwtWeedingCurveFitter` übergeben wird.

*Verwendung des* `QwtWeedingCurveFitter` *zur Datenreduktion außerhalb des QwtPlot*

```
// use curve fitter to reduce data to plot
QPolygonF poly;
for (int i=0; i<x.count(); ++i)
    poly << QPointF(x[i],y[i]);
QwtWeedingCurveFitter weedingAlgorithm(0.001);
poly = weedingAlgorithm.fitCurve(poly);
// set weeeded-out polygon in curve
curve->setSamples(poly);
```

Wenn man den `QwtWeedingCurveFitter` nur als Utilityklasse benutzt und nicht als Objekt der `QwtPlotCurve` übergibt (und damit das Objekt in das Eigentum der PlotCurve überträgt), sollte man das Objekt auf dem Stack erstellen oder sich selbst um das Speicheraufräumen kümmern.

Nun benötigt der Algorithmus einmalig ca. **800 ms**, aber das Zeichnen auf dem Bildschirm und Aktualisieren des Plots beim Plot-Größenänderungen dauert minimale **2 ms**, und *das* ist eine wahrhaftige Performancesteigerung.

Eine wirklich enorme Performancesteigerung ist mit dem `QwtWeedingCurveFitter` möglich, wenn man ihn *vorab* zur Datenreduktion der Datenreihendaten verwendet, *bevor* man diese der Plotkurve übergibt.

# 6. Intervallkurven

Eine spezielle Kurvenart ist die *Intervallkurve*, bereitgestellt über die Klasse `QwtPlotIntervalCurve`.

Im Prinzip ist das eine Kurve mit zwei y-Werten pro x-Koordinate im Datensatz. Es werden zwei reguläre Kurven gezeichnet und dazwischen wird die Fläche ausgefüllt. Dies kann man auch gut dazu nutzen, um gestackte Liniendiagramme zu zeichnen.

*Beispiel für eine Intervallkurve*

```
QVector<double> x{1,2,5,6,10,12,15,16,17};
QVector<double> y1{2,2,3,4, 2, 4, 4, 5,11};
QVector<double> y2{6,4.4,9,10, 5.5, 5.7, 9, 11,12};

QVector<QwtIntervalSample> intervalSamples;
for (int i=0; i<x.count(); ++i)
    intervalSamples.append(QwtIntervalSample(x[i],y1[i],y2[i]));

QwtPlotIntervalCurve *curve = new QwtPlotIntervalCurve();
curve->setStyle(QwtPlotIntervalCurve::Tube);
curve->setPen(QColor(0,40,180), 2);
curve->setBrush( QColor(60,200,255) );
curve->setRenderHint( QwtPlotItem::RenderAntialiased, true ); // Antialiasing verwenden
curve->setSamples(intervalSamples);
curve->attach(&plot); // Plot takes ownership
```

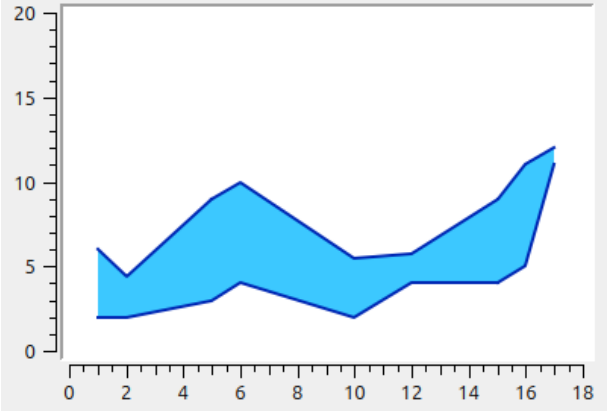



*Abbildung 49. Intervallkurve*

Die Funktion `setSamples()` gibt es in zwei Varianten:

- `QwtPlotIntervalCurve::setSamples( const QVector< QwtIntervalSample >& )` : erwartet einen Vektor mit Interval-Samples, bestehend aus x-Koordinate, unterem und oberen y-Wert
- `QwtPlotIntervalCurve::setSamples( QwtSeriesData< QwtIntervalSample >* )` : erwartet ein `QwtSeriesData` Objekt (siehe Kapitel 4.2) welches Eigentum der Intervallkurve wird. Diese Funktion entspricht der Funktion `setData()` der Elternklasse `QwtSeriesStore`.

Man kann das Erscheinungsbild noch etwas Aufwerten, wenn man für die Füllung der Kurve einen Gradienten verwendet. Dafür gibt man der Kurve einfach einen `QBrush`, der mit einem Gradienten erstellt wurde.

```
...
QLinearGradient grad(0,90,0,220);
grad.setColorAt(0, QColor(60,200,255));
grad.setColorAt(1, QColor(0,60,120));
curve->setBrush( QBrush(grad));
...
```

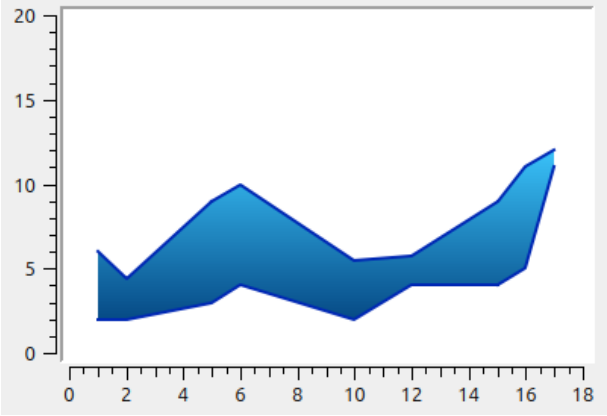
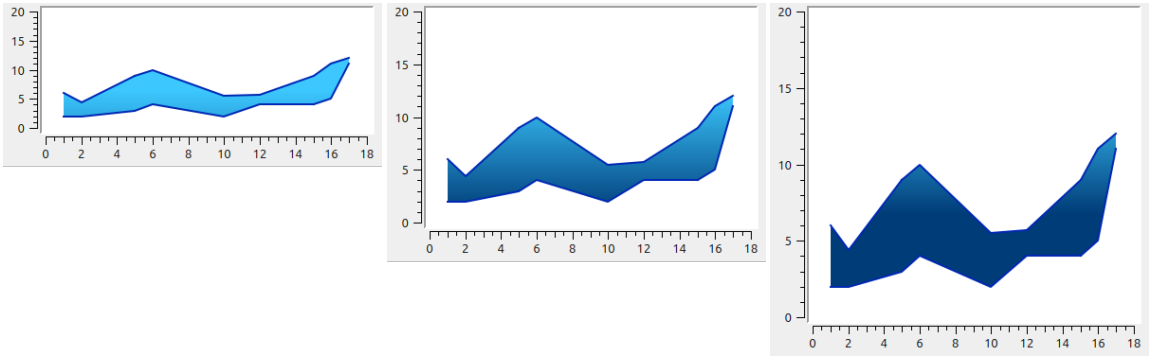



*Abbildung 50. Intervallkurve mit Gradientenfüllung*

Die Colorstops der Gradienten in Qt werden in Pixelkoordinaten angegeben. Wenn man also das Plot-Fenster vergrößert/verkleinert, dann führt das zu witzigen Effekten:

*Intervallkurve mit festem Gradienten bei Änderung der Plotgeometrie*



Die Lösung für das Problem besteht darin, die Klasse `QwtPlotIntervalCurve` abzuleiten und die Funktion je nach aktueller Skal und die Zeichenfunktion selbst zu implementieren.

```
class OwnPlotIntervalCurve : public QwtPlotIntervalCurve {
public:
    void draw(QPainter * painter,
              const QwtScaleMap & xMap, const QwtScaleMap & yMap,
              const QRectF & canvasRect) const override
    {
        // min/max y-Pixel berechnen
        QRectF br = boundingRect();
        double topPixel = yMap.transform(br.top());
        double bottomPixel = yMap.transform(br.bottom());
        QLinearGradient grad(0,bottomPixel,0,topPixel);
        grad.setColorAt(0, QColor(60,200,255));
        grad.setColorAt(1, QColor(0,60,120));
        const_cast<OwnPlotIntervalCurve*>(this)->setBrush( QBrush(grad));
        // originale Zeichenfunktion aufrufen
        QwtPlotIntervalCurve::draw(painter, xMap, yMap, canvasRect);
    }
};
```

Nun bleibt beim Zoomen, Verschieben oder beim Anpassen der Fenstergröße der Gradient an Ort und Stelle.

## 6.1. Gestapelte (Intervall-)Kurven/Flächendiagramme

Man kann die Intervallkurven auch nutzen, um gestapelte, ausgefüllte Kurven bzw. Flächendiagramme zu erstellen. Dazu erstellt man einfach mehrere `QwtPlotIntervalCurve` Zeichenelemente, die sich jeweils die gleichen Y-Werte teilen.

```
QVector<double> x{1,2,5,6,10,12,15,16,17};
QVector<QVector<double> >  y;
// 3 curves, 4 lines
y.append( QVector<double>{0,  0, 0, 0,  0,  0, 0,  0,  0} );
y.append( QVector<double>{2,  2, 3, 4,  2,  4, 4,  5, 11} );
y.append( QVector<double>{6,4.4, 9, 8,5.5,5.7, 9, 11, 12} );
y.append( QVector<double>{7,6.6,12,10,  9, 11,12, 12, 13} );

const QColor cols[] = { QColor(96,60,20),
                        QColor(156,39,6),
                        QColor(212,91,18),
                        QColor(242,188,43)
                      };

for (int j=0;j<y.count()-1; ++j) {
    QwtPlotIntervalCurve *curve = new QwtPlotIntervalCurve();
    // generate intervals for current curve
    QVector<QwtIntervalSample> intervalSamples;
    for (int i=0; i<x.count(); ++i)
        intervalSamples.append(QwtIntervalSample(x[i],y[j][i],y[j+1][i]));
    curve->setStyle(QwtPlotIntervalCurve::Tube);
    curve->setPen(QColor(96,60,20), 1);
    curve->setBrush(cols[j+1]);
    curve->setRenderHint( QwtPlotItem::RenderAntialiased, true ); // Antialiasing verwenden
    curve->setSamples(intervalSamples);
    curve->attach(&plot); // Plot takes ownership
}
```

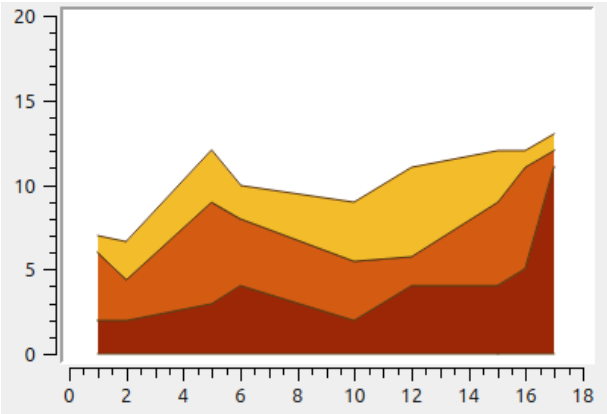



*Abbildung 51. Diagramm mit gestapelten Kurven/Flächendiagramm*

# 7. Balkendiagramme

Balkendiagramme sind mit `QwtPlot` ebenfalls einfach möglich. Dabei gibt es verschiedene Varianten, je nach Datenlage:

- ein Balken pro Intervall, nebeneinander oder gestapelt
- mehrere Balken pro Intervall, nebeneinander oder gestapelt
- mehrere Balken pro Intervall, nebeneinander und gestapelt (das geht z.B. mit Excel nicht!)

Als Zeichenelement/Diagrammtyp werden entweder `QwtPlotBarChart` oder `QwtPlotMultiBarChart` verwendet. Beide Klassen implementieren die Schnittstelle der abstrakten Basisklasse `QwtPlotAbstractBarChart`.

## 7.1. Grundlegende Eigenschaften der Plots

Für einfache Balkendiagramme verwendet man die Klasse `QwtPlotBarChart`. Wie auch schon bei `QwtPlotCurve` wird das Balkendiagramm-Zeichenelement auf dem Heap erstellt und dem Diagramm mit `attach()` übergeben.

```
QwtPlotBarChart * curve = new QwtPlotBarChart();
QVector<double> y{10,20,15,14,18,12};
curve->setSamples(y);
curve->attach(&plot); // Plot takes ownership
```

Ohne weitere Anpassung sieht das Diagramm noch recht langweilig aus.

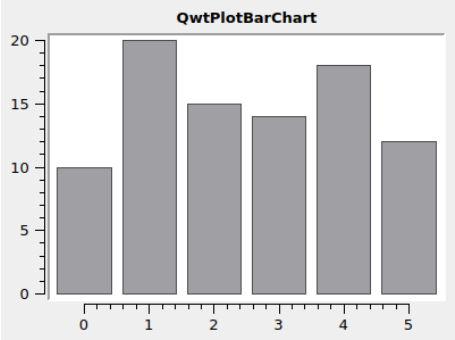



*Abbildung 52. Minimalistisches Balkendiagramm*

Die Funktion `QwtPlotBarChart::setSamples()` generiert aus dem Vektor der y-Werte automatisch einen Vektor mit x,y-Wertepaaren, wobei die x-Werte von 0 beginnend durchnummeriert werden. Die Balken entsprechen intern einer normalen Plotkurve mit equidistanten Stützstellen auf der x-Achse.

## 7.2. Bezugslinie

Standardmäßig beginnen die Balken bei 0. Mitunter will man aber relative Unterschiede bezogen auf eine Grundlinie einstellen. Dazu verwendet man `setBaseLine(yPlotCoordinate)`.

```
QVector<double> y{10,20,15,14,18,12};
curve->setSamples(y);
curve->setBaseLine(15);
```

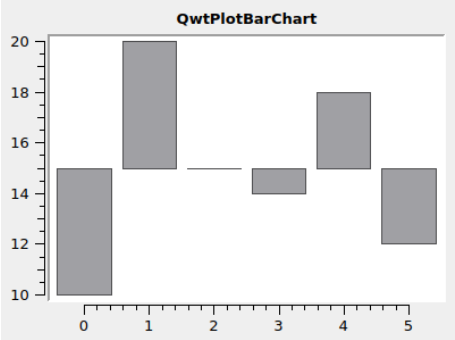



*Abbildung 53. Balkendiagramm mit Bezugslinie bei y=15*

## 7.3. Layout und Abstände

Das Erscheinungsbild des Diagramms lässt sich vielfältig anpassen. Wenn man das Diagramm vergrößert und verkleinert, werden dabei die Achsen skaliert und passend dazu auch die Balken.

In der Standardeinstellung werden die Balken mit minimalem Abstand zueinander und vom Rand gezeichnet. Den Randabstand kann man mit `setMargin(pixels)` ändern und den Abstand zwischen den einzelnen Balken definiert man mit `setSpacing(pixels)`.

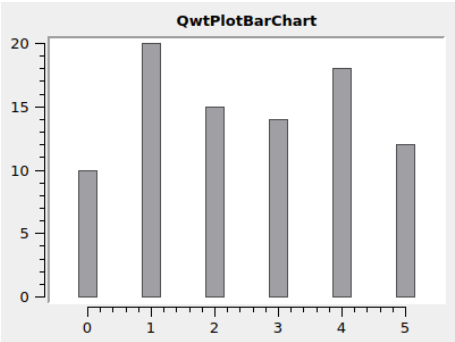



*Abbildung 54. Balkendiagramm mit angepassten Abständen und Rändern*

Die Breite der Balken selbst wird über Layoutvorgaben und konkret über die Funktionen `QwtPlotAbstractBarChart::setLayoutPolicy()` und `QwtPlotAbstractBarChart::setLayoutHint()` kontrolliert.

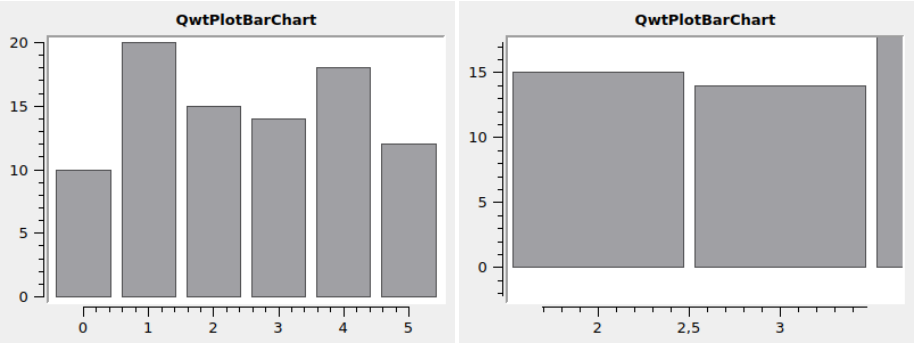
### 7.3.1. AutoAdjustSamples

In diesem Modus, gesetzt durch

```
curve->setLayoutPolicy(QwtPlotAbstractBarChart::AutoAdjustSamples);
```

wird die Größe der Balken basierend auf der Größe der Zeichenfläche und der gesetzen Rand- und Zwischenabstände bestimmt.

*Balkendiagram im Layoutmodus AutoAdjustSamples, normal (links) und hineingezoomt (rechts)*



Die Balken, die Zwischenräume (spacing) und Randabstände (margin) füllen die Zeichenfläche komplett aus. Wie man am rechten Diagramm sieht, bleibt das auch beim Hineinzoomen ins Diagramm so.

Der zusätzliche Parameter `setLayoutHint()` definiert die Anzahl der Pixel, die ein Balken mindestens breit sein sollte. Damit kann man verhindern, dass beim Verkleinern der Plotgröße die Balken irgendwann komplett verschwinden. Folgendes Beispiel zeigt, was bei einem größeren LayoutHint im Modus `AutoAdjustSamples` passiert:

```
curve->setLayoutPolicy(QwtPlotAbstractBarChart::AutoAdjustSamples);
curve->setLayoutHint(100);  // minimum width of bars is 100 pixels
```

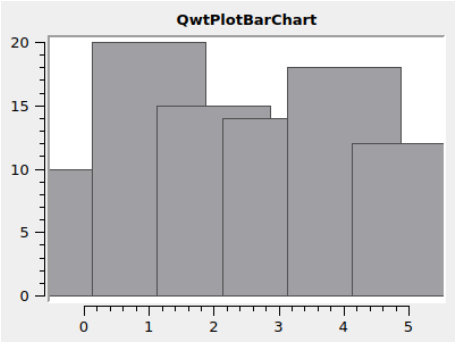


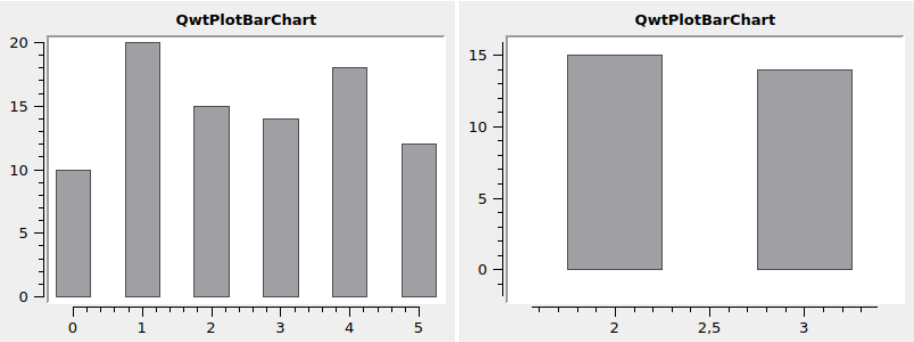

*Abbildung 55. Durch LayoutHint definierte Mindestbalkenbreite*

### 7.3.2. ScaleSamplesToAxes

In diesem Layoutmodus wird die Balkenbreite basierend auf der aktuellen x-Achsenskalierung festgelegt. Wenn man in diesem Layoutmodus den LayoutHint auf 0.5 setzt, dann wird ein Balken genau halb so breit wie ein Achsentick gezeichet und dann mittig am Achsentick ausgerichtet. Es wird also die X-Achse benutzt, um die 0.5 in Plotkoordinaten in Pixelbreiten der Zeichenfläche umzurechnen.

```
curve->setLayoutPolicy(QwtPlotAbstractBarChart::ScaleSamplesToAxes);
curve->setLayoutHint(0.5); // 0.5 axis scale as bar width
```

*Balkendiagram im Layoutmodus ScaleSamplesToAxes, normal (links) und hineingezoomt (rechts)*




Auch beim Hereinzoomen orientiert sich die Balkenbreite stets an der X-Achsenskalierung.

Der Balkenabstand wird hier ausschließlich über den LayoutHint definiert und der Balkenabstand, den man mittels `setSpacing()` setzt, wird in diesem Layoutmodus nicht berücksichtigt. Wenn man also eine Balkenbreite 1 (in x-Achsenskaleneinheiten) setzt, dann werden die Balken dicht-an-dicht gezeichnet, unabhängig vom *spacing*. Der Randabstand *margin* wird hingegen wie bisher angewendet.

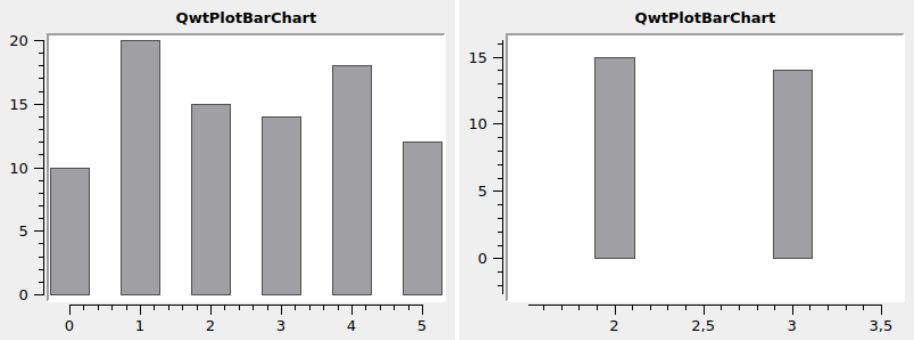
### 7.3.3. ScaleSampleToCanvas

In diesem Modus werden die Balkenbreiten in Abhängigkeit (als Prozentsatz) zur Zeichenflächengröße bestimmt. Man sollte sich zur

Festlegung des LayoutHint-Parameters überlegen, wie viele Balken denn maximal im Diagramm sichtbar sein werden.

```
curve->setLayoutPolicy(QwtPlotAbstractBarChart::ScaleSampleToCanvas);
curve->setLayoutHint(0.1); // bar width 10% of canvas width
```

*Balkendiagram im Layoutmodus ScaleSampleToCanvas, mit 10% Zeichenflächebreite als Balkenbreite*




Beim Hereinzoomen bleibt die Balkenbreite konstant und ändert sich nur bei Größenänderung des Plots.

### 7.3.4. Layout-Empfehlung

Abgesehen von speziellen Anforderungen ist für die meisten Fälle die LayoutPolicy `QwtPlotAbstractBarChart::ScaleSamplesToAxes` empfehlenswert. Sowohl beim Größenändern des Plots als auch beim Zoomen verhält sich das Plot so, wie man es erwartet.

```
curve->setLayoutPolicy(QwtPlotAbstractBarChart::ScaleSamplesToAxes);
curve->setLayoutHint(0.8); // 0.8 axis scale as bar width
curve->setMargin(10); // 10 pixel margin
```

Möchte man wirklich immer die gleichen Abstände zwischen den Balken haben, egal wie groß das Plot ist oder wie weit man hineinzoomt, dann ist die LayoutPolicy `QwtPlotAbstractBarChart::AutoAdjustSamples` zu empfehlen.

## 7.4. Balkenformen und Farben

Die Balken selbst werden durch die Klasse `QwtColumnSymbol` gezeichnet. Diese kann verschiedenartig konfiguriert werden. Standardmäßig wird der Symboltyp `QwtColumnSymbol::Box` verwendet, wie in nachfolgendem Beispiel:

```
QwtColumnSymbol* symbol = new QwtColumnSymbol( QwtColumnSymbol::Box );
symbol->setLineWidth( 2 );
symbol->setFrameStyle( QwtColumnSymbol::Raised );
symbol->setPalette( QPalette( QColor(0xff0040) ) );
curve->setSymbol( symbol );
```

Angepasst werden können die Form des Rechteckrahmens (`Raised`, `Plain`, `NoFrame`), die Füllfarbe und Linienfarbe.

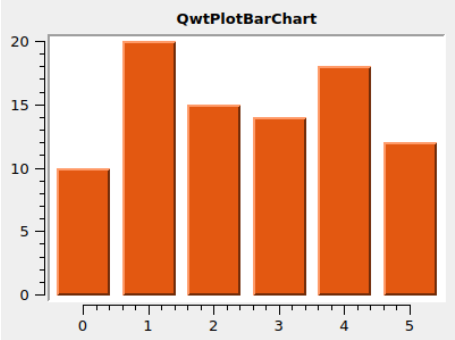



*Abbildung 56. Balkendiagram mit angepasstem Stil für die Balken*

Wenn man ein `QPalette` Objekt mit einer einzelnen Farbe im Konstruktor erstellt, dann werden die Farben für die einzelnen Palettenrollen automatisch berechnet. `QwtColumnSymbol` verwendet die Rollen `QPalette::Window`, `QPalette::Dark` und `QPalette::Light` für die individuelle Elemente. Beim Stil `plain` wird der Rahmen mit der Palettenrolle `Dark` gezeichnet.

Durch Anpassung individueller Palettenrollen kann man das Zeichnen der Balken anpassen.

```
QwtColumnSymbol* symbol = new QwtColumnSymbol( QwtColumnSymbol::Box );
symbol->setFrameStyle(QwtColumnSymbol::Plain);
symbol->setLineWidth(1);
QPalette palette(QColor(0xc1e311));
palette.setBrush(QPalette::Dark, Qt::black); // black frame
symbol->setPalette(palette);
curve->setSymbol( symbol );
```

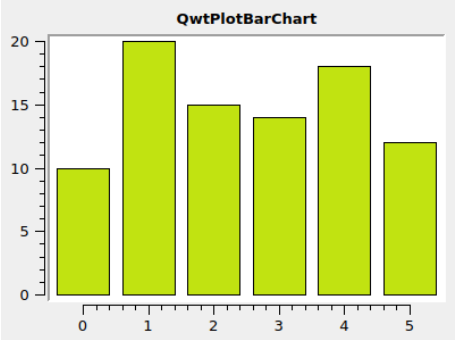



*Abbildung 57. Balkendiagram mit schwarz umrandeten, einfarbigen Balken*

## 7.5. Balkenbeschriftung auf der X-Achse

Die in den bisherigen Beispieldiagrammen gezeigte X-Achse ist etwas ungewöhnlich für Balkendiagramme. Ohne jetzt auf Kapitel 10 vorgreifen zu wollen, soll hier doch die für Balkendiagramme typische Anpassung der x-Achse gezeigt werden.

Zunächst wird die Achsenzeichenfunktionalität angepasst, wofür man sich zunächst Zugriff auf die aktuelle Zeichenklassen `QwtScaleDraw` (Header `QwtScaleDraw` bzw. `qwt_scale_draw.h`) mit `QwtPlot::axisScaleDraw()` holt. Dann schaltet man die Unterteilungsstriche (*Ticks*) und die Achsenlinie (*Backbone*) ab.

Um die unterschiedlichen Randeinstellungen des Plots besser zu verstehen, werden Balken mit `ScaleSamplesToAxes` und LayoutHint 1 (komplette Breite) gewählt, die Balkendiagrammränder (*margins*) auf 20 und der Zeichenflächenrand (umlaufend) auf 10 Pixel gesetzt.

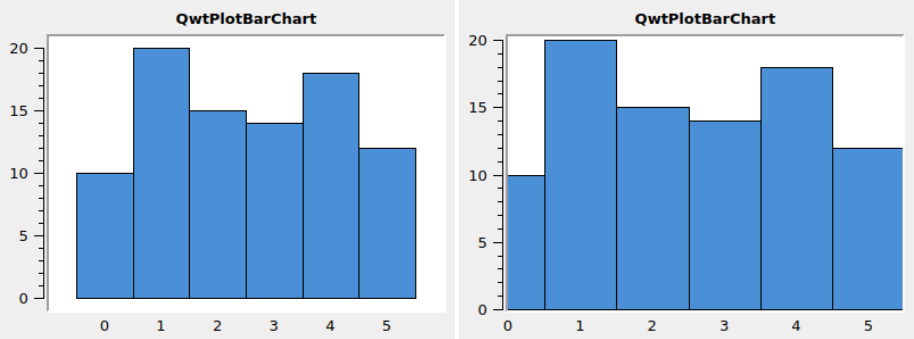
Schließlich stellen wir noch sicher, dass das Plotlayout die y-Achse nicht direkt über dem x=0 Wert an den linken Rand der Zeichenfläche legt. Dies macht man durch Anpassung des `QwtPlotLayout` (Header `QwtPlotLayout` bzw. `qwt_plot_layout.h`). `QwtPlotLayout::setAlignCanvasToScale()` legt fest, ob die gewählte Achse direkt am Rand der Zeichenfläche liegt und damit die jeweils zugeordnete Achse (hier die x-Achse) eben mit dem 0-Wert direkt am linken Rand der Zeichenfläche liegt.

```
QwtScaleDraw* scaleDraw1 = plot.axisScaleDraw( QwtPlot::xBottom );
scaleDraw1->enableComponent( QwtScaleDraw::Backbone, false );
scaleDraw1->enableComponent( QwtScaleDraw::Ticks, false );

curve->setMargin(20); // margin left/right of bars
plot.plotLayout()->setCanvasMargin( 10 ); // canvas margin all around

// do not fix y-axis at 0 and left edge of canvas
plot.plotLayout()->setAlignCanvasToScale( QwtPlot::yLeft, false );
plot.updateCanvasMargins();
```

Zum Vergleich nochmal das gleiche Diagramm *ohne* Ränder und mit `setAlignCanvasToScale(yLeft, true)`.

```
QwtScaleDraw* scaleDraw1 = plot.axisScaleDraw( QwtPlot::xBottom );
scaleDraw1->enableComponent( QwtScaleDraw::Backbone, false );
scaleDraw1->enableComponent( QwtScaleDraw::Ticks, false );

curve->setMargin(0);
plot.plotLayout()->setCanvasMargin(0);
plot.plotLayout()->setAlignCanvasToScale( QwtPlot::yLeft, true );
plot.updateCanvasMargins();
```

*Balkendiagram mit typischer X-Achsenbeschriftung, mit Rändern (links), ohne Ränder und y-Achse schneidet x-Achse bei x=0 direkt am linken Rand*




## 7.6. Balkenbeschriftungen

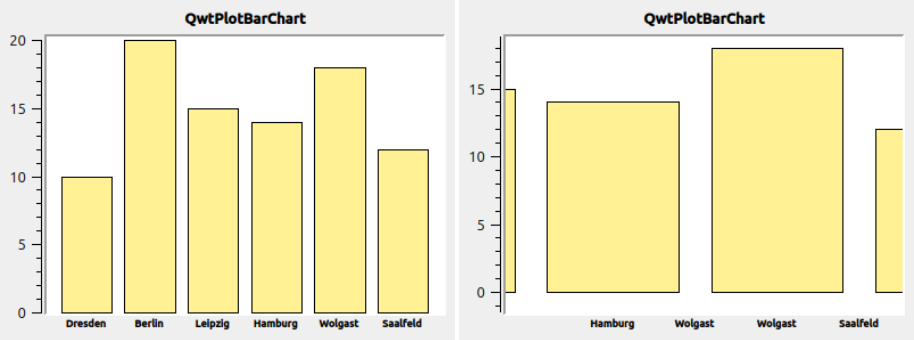
Möchte man statt der Zahlen am unteren Bildschirmrand Beschriftungen für die einzelnen Balken haben, musst man die Klasse `QwtScaleDraw` ableiten und dann die virtuelle Funktionen `QwtScaleDraw::label()` überschreiben.

```
class ScaleDraw : public QwtScaleDraw {
public:
    ScaleDraw(const QStringList& labels ) : m_labels( labels ) {
        enableComponent( QwtScaleDraw::Ticks, false );
        enableComponent( QwtScaleDraw::Backbone, false );
        setLabelAlignment( Qt::AlignHCenter | Qt::AlignVCenter );
    }

    virtual QwtText label( double value ) const QWT_OVERRIDE {
        const int index = qRound( value );
        if ( index >= 0 && index < m_labels.size() )
            return m_labels[index];
        return QwtText();
    }

    QStringList m_labels;
```

```
};
```

Diese Klassenimplementierung konfiguriert die Darstellung der Unterteilungsstriche und Skalenlinie wie bisher, sorgt für korrekte Ausrichtung der Labels und merkt sich die im Konstruktor übergebenen Texte als indizierte Liste. Entscheidend ist die Implementierung der Funktion `QwtScaleDraw::label()`. Diese Funktion hat die Aufgabe, passend zu den übergebenen Zahlenwerten (hier Werte der x-Achse) entsprechende Beschriftungen anzuzeigen. Wie am Anfang dieses Kapitels erwähnt, wird jedem Balken eine fortlaufende Nummer zugeordnet. Wenn die Achse nun also eine Zahl zeichnen will, also z.B. die 4, dann wird in der Funktion der Wert gerundet und als Index benutzt, um den dazugehörigen Text zurückzuliefern.

*Balkendiagram mit Textbeschriftung der Balken, rechts hineingezoomed mit fehlerhafter Beschriftung der Balken*




Das Problem mit dem hineinzoomen lässt sich durch eine minimale Erweiterung des Codes lösen (nur Labels dort zeichnen, wo der x-Achsen-Skalenwert gerundet ziemlich exakt einer ganzen Zahl entspricht):

```
    if ( index >= 0 && index < m_labels.size() && qAbs(index-value) < 1e-6 )
        return m_labels[index];
```

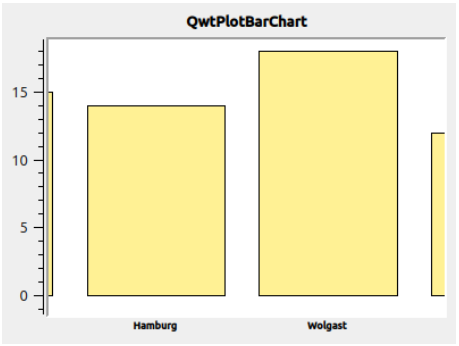



*Abbildung 58. Hineingezoomt in ein Balkendiagramm mit Textbeschriftung*

## 7.7. Mehrfarbige Balken

Wenn man die Balken nun auch noch individuell einfärben möchte, so kommt man um eine Re-Implemetierung des `QwtPlotBarChart` Zeichenelements nicht herum. Man muss eigentlich nur die Funktion `QwtPlotBarChart::specialSymbol()` reimplementieren und hier unterschiedlich gefärbte Balkensymbole zurückliefern.

```
class MultiColorBarChart : public  {
public:
    MultiColorBarChart() {
        setLayoutPolicy(QwtPlotBarChart::ScaleSamplesToAxes);
        setLayoutHint(0.8);
    }

    // we want to have individual colors for each bar
    virtual QwtColumnSymbol* specialSymbol(
        int sampleIndex, const QPointF&) const QWT_OVERRIDE
```

```
    {
        // generate symbol with color for each bar
        QwtColumnSymbol* symbol = new QwtColumnSymbol( QwtColumnSymbol::Box );
        symbol->setLineWidth( 2 );
        symbol->setFrameStyle( QwtColumnSymbol::Raised );

        QColor c( Qt::white );
        if ( sampleIndex >= 0 && sampleIndex < m_colors.size() )
            c = m_colors[ sampleIndex ];
        symbol->setPalette( c );

        return symbol;
    }

    QList<QColor> m_colors;
};
```

Für jeden Balken im Diagramm speichern wir uns eine Farbe in der Membervariable `m_colors`. In der überschriebenen Funktion `QwtPlotBarChart::specialSymbol()` erstellen wir nun jeweils ein `QwtColumnSymbol` wie im vorangehenden Kapitel und liefern dieses zurück. Die Funktion übernimmt sowohl den Index des Balkens (oder *samples*) als Argument, wie auch die Plotkoordinaten des Balkens, wobei der x-Wert der Punktes wieder der Balkenindex ist, und der y-Wert dem Funktionswert des Balkens entspricht.

Die Funktion `QwtPlotBarChart::specialSymbol()` ist eine *factory function* und überträgt die Eigentümerschaft des auf dem Heap erstellten Objekts an den Aufrufer. Deshalb *muss* man hier in jedem Aufruf der Funktion ein neues `QwtColumnSymbol` Objekt mit `new` erstellen. Nachdem der Balken gezeichnet wurde, löscht das Balkendiagrammelement das erzeugte Säulensymbolobjekt automatisch wieder (man muss und darf das generierte Objekt also *nicht* selbst später löschen).

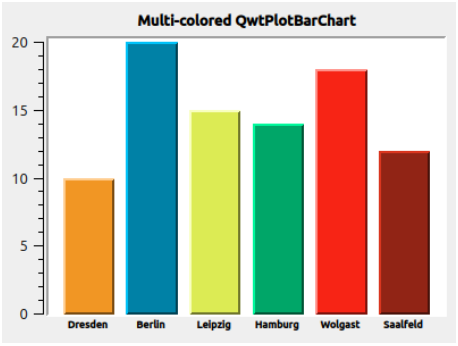



*Abbildung 59. Balkendiagramm mit unterschiedlich gefärbten Balken*

Wenn man nur bestimmte Balken einfärben möchte, die anderen aber im Standarddesign zeichnen lassen will, so kann man die Funktion `QwtPlotBarChart::specialSymbol()` auch einfach einen *nullptr* zurückgeben lassen. Dann verwendet das Balkendiagramm das Standardsymbol.

## 7.8. Legendeneinträge

Für Balkendiagramme kann man zwei Arten von Legendeneinträgen erstellen:

- einen Eintrag für das gesammte Balkendiagramm-Zeichenelement, oder
- individuelle Einträge für jeden einzelnen Balken

Die erste Variante ist dann sinnvoll, wenn neben dem Balkendiagramm-Zeichenelement noch weitere Zeichenelemente mit Legendeneinträgen angezeigt werden. Im vorliegenden Fall ist eine individuelle Bezeichnung der Balken sinnvoll.

Dafür muss die virtuelle Funktion `QwtPlotBarChart::barTitle()` reimplementiert werden (die Standardimplementierung liefert immer einen leeren Titeltext). Außerdem muss der Legendentyp auf "einzelne Balken" durch Aufruf der Funktion `QwtPlotBarChart::setLegendMode( QwtPlotBarChart::LegendBarTitles )` umgeschaltet werden. Zusätzlich kann man noch mit `QwtPlotBarChart::setLegendIconSize()` die Größe der Legendensymbole anpassen:

```
class MultiColorBarChart : public QwtPlotBarChart {
public:
    MultiColorBarChart() {
        setLayoutPolicy(QwtPlotBarChart::ScaleSamplesToAxes);
        setLayoutHint(0.8);
        // Legende zeigt individuelle Balkentitel
        setLegendMode( QwtPlotBarChart::LegendBarTitles );
        setLegendIconSize( QSize( 10, 14 ) );
    }

    // Individuelle Farben für die einzelnen Balken
    virtual QwtColumnSymbol* specialSymbol(
        int sampleIndex, const QPointF&) const QWT_OVERRIDE
    {
        // ... wie bisher
    }

    virtual QwtText barTitle( int sampleIndex ) const QWT_OVERRIDE {
        if ( sampleIndex >= 0 && sampleIndex < m_titles.size() )
            return m_titles[ sampleIndex ];
        return QwtText();
    }

    QStringList   m_titles;
    QList<QColor> m_colors;
};
```

Zusätzlich zu den Farben werden nun die Titel der Balken in der Membervariable `m_titles` gehalten und in jedem Aufruf von `barTitle()` zurückgeliefert.

Damit die Legende überhaupt gezeichnet wird, muss man diese für das Plot anzeigen:

```
// Legende anzeigen
QwtLegend * legend = new QwtLegend();
QFont legendFont;
legendFont.setPointSize(7);
legend->setFont(legendFont);
plot.insertLegend(legend, QwtPlot::RightLegend); // plot takes ownership

// x-Achse verstecken
plot.setAxisVisible(QwtPlot::xBottom, false);
```

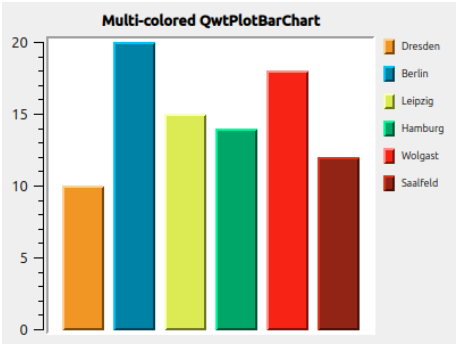



*Abbildung 60. Balkendiagramm mit unterschiedlich gefärbten Balken und Legendeneinträgen*

## 7.9. Gestapelte Balkendiagramme oder Balkendiagramme mit mehreren Balken pro Gruppe

# 8. Legende

## 8.1. Außenseitige Legende

…

## 8.2. Legenden-Zeichenelement

…

## 8.3. Eigene Legenden-Icons

Manchmal will man aber auch ganz eigene Icons anzeigen. Das kann man machen, wenn man eine der Kindklassen von `QwtPlotItem` ableitet und die virtuelle Funktion `QwtPlotItem::legendIcon()` überschreibt. In dieser kann man dann nach Herzenslust ein Bild malen/generieren und als `QwtGraphic` zurückliefern.

Das `QwtGraphic` Objekt wird wie ein gewöhnliches PaintDevice benutzt, d.h. man erzeugt einen `QPainter` damit und zeichnet drauflos.

Nachfolgendes Beispiel zeigt so eine Implementierung am Beispiel einer `QwtPlotCurve`:

```
class OwnPlotCurve : public QwtPlotCurve {
public:
    QwtGraphic legendIcon(int, const QSizeF & ) const override {
        QwtGraphic graphic;

        QSizeF s(30,16); // Icongröße fixieren
        graphic.setDefaultSize( s );
        graphic.setRenderHint( QwtGraphic::RenderPensUnscaled, true );

        QPainter painter( &graphic );
        painter.setRenderHint( QPainter::Antialiasing, false);

        // Mittellinie im Iconrechteck
        const double y = 0.5 * s.height();

        // Hintergrund zeichnen (für den schwarzen Rahmen)
        QPen backgroundPen(Qt::black);
        backgroundPen.setWidth(5);
        backgroundPen.setCapStyle( Qt::FlatCap );
        painter.setPen( backgroundPen );
        QwtPainter::drawLine( &painter, 0.0, y, s.width(), y );

        // Linienfarbe zeichnen)
        QPen pn = pen();
        pn.setCapStyle( Qt::FlatCap );
        pn.setWidth(3);
        painter.setPen( pn );
        QwtPainter::drawLine( &painter, 1, y, s.width()-1, y );

        return graphic;
    }
};
```

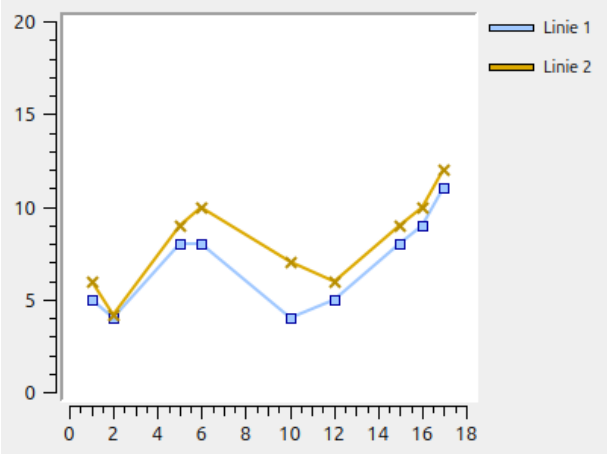



*Abbildung 61. Legende mit eigenem Legendenicon (schwarzumrandete Linien)*

Wenn man eher die Standard-Rechtecke bevorzugt, dann ist folgender Zeichencode ausreichend:

```
// ...
QRect r(0, 0, s.width(), s.height() );
painter.setPen(Qt::black);
painter.setBrush(pen().color());
painter.drawRect(r);
return graphic;
```

was dann auch ganz schick aussieht:

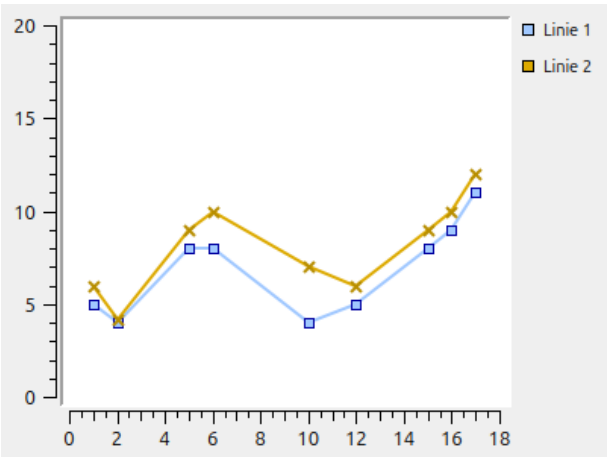



*Abbildung 62. Legende mit eigenem Legendenicon (schwarzumrandete Box)*

Wenn man eigene Icons zeichnet, kann man auch wie im Beispiel oben die Größe des Icons fixieren. So hat man dann die komplette Kontrolle über das Erscheinungsbild und kann auch DPI-Skalierungen (für Highres-Bildschirme) mit einbeziehen.

# 9. Markierungslinien

# 10. Plotachsen

Die Achsen/Skalen eines Plots (insgesamt 4, open, unten, links und rechts) können bereits in den mitgeliferten Klassenimplementierungen vielfältig angepasst und verändert werden. Und natürlich können die beteiligten Klassen auch abgeleitet und so beliebig modifiziert/geändert werden.

Die wichtigsten Klassen in Bezug auf die Achsen sind:

- `QwtAxis`
- `QwtAbstractScaleDraw` und die Spezialisierungen `QwtScaleDraw` und `QwtDateScaleDraw`
- `QwtScaleEngine`und die Spezialisierungen `QwtLinearScaleEngine`und `QwtLogScaleEngine`

## 10.1. Allgemeine Achsenformatierung

## 10.2. Skalen

# 11. QwtText und Sonderformatierungen

## 11.1. MathML

# 12. Interaktiver Zoom und Verschieben von Diagrammausschnitten

# 13. Anpassung/Styling der Qwt Komponenten

## 13.1. Allgemeines zu Farbpaletten

Die Qwt-Komponenten verwenden die Qt Palette und deren Farbrollen für die Einfärbung.

## 13.2. Rahmen und Zeichenfläche des Diagramms

Beim QwtPlot können verschiedene Elemente angepasst werden. Nachfolgend ist ein QwtPlot zu sehen, welches in einem äußeren Widget (dunkelgrau) eingebettet ist. Die hellgraue Fläche ist das eigentliche `QwtPlot`:

![Rahmen und Zeichenfläche](imgs/plotStylingOverview.png)

Im Screenshot sind die wichtigsten Attribute markiert:

1. Innenabstand (siehe `QWidget::setContentsMargins()`)
2. Rahmen (hauptsächlich für den Druck wichtig)
3. Hintergrund des Plot-Widgets
4. Zeichenfläche (engl. *Canvas*) (betrifft Hintergrundfarbe und Rahmen)

### 13.2.1. Farbe und Rahmen des Plots

Die Farbe des äußeren Bereichs des Plots wird über die Paletteneigenschaft des `QwtPlot` kontrolliert. Standardmäßig wird der äußere Rand des Plot-Widgets transparant gezeichnet, d.h. die Farbe des darunterliegenden Widgets ist sichtbar. Um eine eigene Farbe zu setzen, muss daher ```setAutoFillBackground(true)``` aufgerufen werden:

```
QPalette pal = plot.palette();
// Die QPalette::Window Farbrolle definiert die Einfärbung
// des äußeren Plotbereichs
pal.setColor(QPalette::Window, QColor(196,196,220));
plot->setPalette(pal);
// die Eigenschaft "autoFillBackground" muss dafür eingeschaltet sein
plot->setAutoFillBackground(true);
```

![](imgs/plotStyling1.png)

> **Hinweis:** In Abschnitt [Gradient als Plot-Hintergrund](customization/#gradient-als-plot-hintergrund) wird beschrieben, wie man einen Farbverlauf im Plothintergrund umsetzt, und diesen bei Größenänderung entsprechend anpasst.

Der Rahmen wird wie bei einem normalen Widget angepasst:

```
plot->setFrameStyle(QFrame::Box | QFrame::Sunken);
```

Normalerweise ist ein solcher Rahmen nicht notwendig für die Bildschirmdarstellung oder für das Einbetten des QwtPlot in eine Programmoberfläche. Der Rahmen ist jedoch häufig beim [Export/Druck](export) des Widgets sinnvoll.

### 13.2.2. Zeichenfläche

Die Zeichenfläche kann eingefärbt werden:

```
plot->setCanvasBackground(Qt::darkGray);
```

 (imgs/plotStyling3.png)

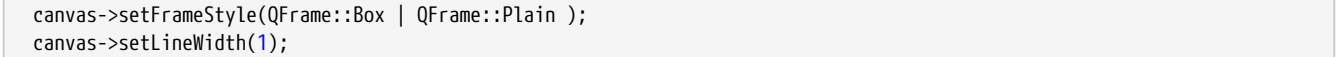
Der Randabstand zwischen Achsenbeschriftung und Titel zum Rand kann definiert werden:

```
plot->setContentsMargins(15,10,35,5);
```

 (imgs/plotStyling4.png)

Die Rahmen um die Zeichenfläche kann durch Anpassen des Zeichenflächenobjekts (`QwtPlotCanvas`) verändert werden. `QwtPlotCanvas` ist von QFrame abgeleitet, wodurch es entsprechend angepasst werden kann. Es wird einfach neues Objekt erstellt, konfiguriert und dem Plot übergeben (das QwtPlot wird neuer Besitzer des Zeichenflächenobjekts):

```
QwtPlotCanvas * canvas = new QwtPlotCanvas(&plot);
canvas->setPalette(Qt::white);
canvas->setFrameStyle(QFrame::Box | QFrame::Plain );
canvas->setLineWidth(1);
plot->setCanvas(canvas);
```

 (imgs/plotStyling5.png)

Einfacher geht es durch Setzen des Stylesheets für das Canvas-Widget (siehe Qt-Widgets Dokumentation, welche Attribute unterstützt werden):

```
plot->canvas()->setStyleSheet(
    "border: 1px solid Black;"
    "border-radius: 15px;"
    "background-color: qlineargradient( x1: 0, y1: 0, x2: 0, y2: 1,"
        "stop: 0 LemonChiffon, stop: 1 PaleGoldenrod );"
);
```

(imgs/plotStyling6.png)

# 14. Exportieren und Drucken

Neben der Anzeige auf dem Bildschirm ist das Speichern schicker Diagramme und Verwendung dieser in Berichten eine nicht unwichtige Aufgabe. Allerdings ist es nicht trivial, gute Diagramme mit sinnvollen Schriftgrößen zu exportieren. Grundsätzlich muss hier zwischen Pixelgrafik-Export und Vektorgrafik unterschieden werden.

## 14.1. Exportieren des Plots als Pixelgrafik

Der naheliegenste Export des Plots ist eine 1-zu-1 Kopie in die Zwischenablage oder in eine Bitmapdatei (jpg, gif, png,…).

### 14.1.1. Erstellen einer 1-zu-1 Kopie des Plotwidgets

Jedes QWidget kann direkt in eine QPixmap gezeichnet werden. Und dieses kann dann in eine Datei gespeichert werden.

```
// Plot in Pixmap rendern
QPixmap p = plot.grab();
// Pixmap in Datei speichern
p.save("diagramm_screenshot.png");
```

Das Diagramm aus dem Tutorial 1 (Kapitel 2) sieht exportiert als PNG-Build so aus:

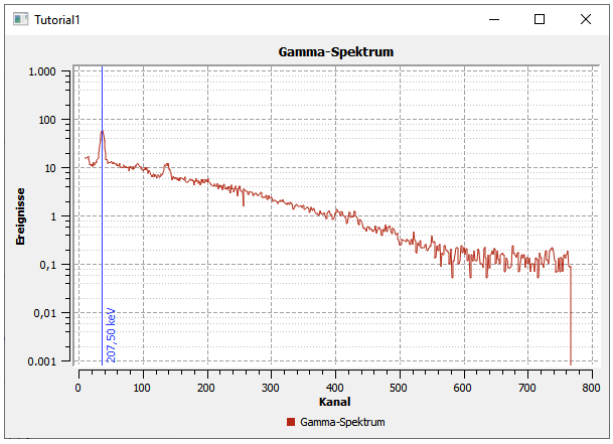



*Abbildung 63. 1-zu-1 Kopie des Plot, exportiert in ein PNG-Bild*

Die Funktion `QWidget::grab()` macht bei jedem beliebigen Qt Widget einen Screenshot oder bei Verwendung von `QWidget::grab(QRect(...))` eine Teil-Screenshot des Widgets und alle seiner Kind-Widgets. Das resultierende QPixmap kann dann beliebig weiterverarbeitet werden.

Der Widget-Titel wurde mit `QWidget::grab()` nur deshalb mit exportiert, weil das `QwtPlot` das oberste Anwendungswidget war. Ist das `QwtPlot` jedoch in ein Layout eingebettet und wird mit `QWidget::grab()` exportiert, so erhält man auch nur das Plot selbst.

### 14.1.2. Kopie in die Zwischenablage

Statt das Pixmap in eine Datei zu speichern, kann man das auch einfach in die Zwischenablage kopieren. Dazu `QClibBoard` und `QApplication` einbinden und:

```
// Plot in Pixmap rendern
QPixmap p = plot.grab();
// Pixmap in Zwischenablage kopieren
qApp->clipboard()->setImage(p.toImage());
```

### 14.1.3. QwtPlot mit anderer Auflösung abspeichern

Wenn das QwtPlot mit einer anderen Auflösung/Pixelgröße als angezeigt auf dem Bildschirm abgespeichert werden soll, so verwendet man die `QwtPlotRenderer`:

```
// Render-Objekt erstellen
QwtPlotRenderer renderer;
// Zielgröße festlegen
QRect imageRect( 0.0, 0.0, 1200, 600 );
// Bildobjekt in der entsprechenden Größe erstellen...
QImage image( imageRect.size(), QImage::Format_ARGB32 );
// und mit weißem Hintergrund füllen
image.fill(Qt::white);

// Das Diagramm in das QImage zeichnen
QPainter painter( &image );
renderer.render( &plot, &painter, imageRect );
painter.end();

// QImage zurück in Pixmap konvertieren
QPixmap plotPixmap( QPixmap::fromImage(image) );
plotPixmap.save("diagram.png");
```

Das Diagramm aus dem Tutorial 1 (Kapitel 2) sieht dann z.B. so aus:

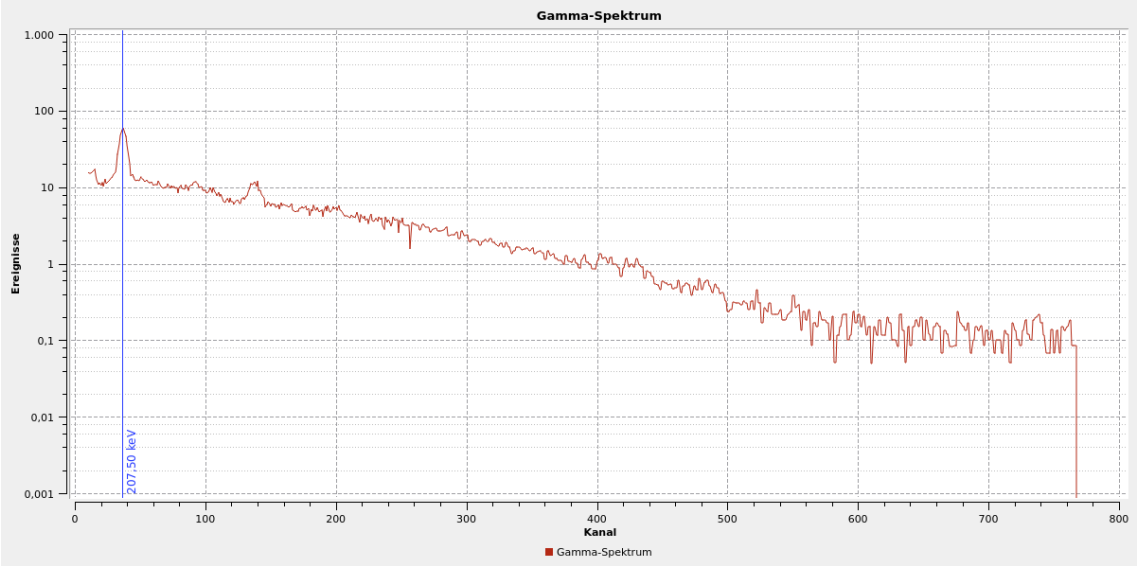



*Abbildung 64. Plot in höherer Auflösung abgespeichert*

Wenn man dieses Diagramm und das vorherige der 1-zu-1 Kopie vergleicht, so stellt man fest, dass Fontgrößen, Linienstärken und manche Randabstände (z.B. bei der vertikalen Markierung) gleich geblieben sind. Die Achsenskalierung und damit auch das Gitterraster und Achsenbeschriftungen haben sich jedoch aufgrund der höheren Auflösung verändert. Das kann man sich zu Nutze machen, wenn z.B. Achsenbeschriftungen auf dem Bildschirm nicht komplett ins Diagramm passen und nun aufgrund höherer Auflösung geschrieben werden können. Allerdings birg dies auch die Gefahr, dass die Schriftgrößen im Ausdruck zu klein werden. Das ist aber eher ein Problem beim Vektorexport und wird im Kapitel 14.1.9 thematisiert.

### 14.1.4. Drucken

Beim Drucken wird das Bild einfach auf eine Druckerzeichenfläche gerendert. Für die Druckunterstützung mit im Qt-Programm zunächst die pro-Datei erweitert werden:

```
QT += printsupport
```

Man erstellt und konfiguriert ein Printerobjekt:

```
QPrinter printer( QPrinter::HighResolution );
printer.setCreator("Ich");          // only for exporting to (PDF) files
printer.setDocName("Mein Plot");    // only for exporting to (PDF) files

printer.setPageOrientation( QPageLayout::Landscape );
```

Dann kann der Anwender noch den Drucker auswählen:

```
QPrintDialog dialog( &printer );
if ( dialog.exec() ) {
    QwtPlotRenderer renderer;
    renderer.renderTo( &plot, printer );
}
```

Und schon wird das Bild auf den Drucker oder PDF-Drucker geschoben.

Die Druckereigenschaften *creator* und *docName* werden bei PDF-Druckern als Eigenschaften in der PDF-Datei abgelegt. Für normale Drucker sind die nicht notwendig.

Wichtig bei Druckern ist die zu verwendende Auflösung. Dies wird im Konstruktor der `QPrinter`-Klasse eingestellt:

- `QPrinter::HighResolution` legt die Druckauflösung wie beim eingestellten Drucker fest (bzw. 1200 DPI bei PDFs)
- `QPrinter::ScreenResolution` legt die Auflösung wie auf dem aktuellen Bildschirm fest, üblicherweise also 72 DPI. Dadurch wird das gedruckte Diagramm ziemlich genau so aussehen, wie auf dem Bildschirm angezeigt

Man kann auch manuell die zu verwendende Auflösung durch Aufruf von `QPrinter::setResolution()` vorgeben.

Das Zusammenspiel von Auflösung und Druckgröße wird in Kapitel 14.1.9 beschrieben.

### 14.1.5. PDF Export mittels QPdfWriter

Statt PDFs via PDF-Drucker zu erzeugen, kann man auch den durch Qt bereitgestellten `QPdfWriter` verwenden:

```
QPdfWriter writer("plot.pdf");
writer.setTitle("Mein plot");
writer.setCreator("Ich");
writer.setPageSize(QPageSize::A4);
writer.setPageOrientation(QPageLayout::Landscape);
renderer.renderTo( &plot, writer);
```

### 14.1.6. SVG Export mittels QSvgDocument

SVG Export erfolgt mittels der Klasse `QSvgGenerator`. Für die SVG-Unterstützung muss zunächst die Qt pro-Datei erweitert werden:

```
QT += svg
```

Dann erstellt und konfiguriert man das Generatorobjekt und rendert in das SVG-Generator-Paintdevice.

```
 generator;
generator.setFileName("plot.svg");
generator.setSize(QSize(600, 400));
generator.setViewBox(QRect(0, 0, 600, 400));
generator.setTitle("Mein Plot");
generator.setResolution(72);
generator.setDescription("Ein SVG-Plot");
renderer.renderTo( &plot, generator);
```

Wendet man obigen Code auf das Diagramm im ersten Tutorial an, so wird beim Betrachten des SVG-Dokuments *keine* Linie angezeigt. Das liegt daran, dass der SVG-Generator im Gegensatz zum Pixel-Renderer oder PDF-Writer nicht mit NAN-Werten umgehen kann. Diese entstehen häufig unbewusst, z.B. in diesem Beispiel beim Logarithmieren von 0-Werten durch Auswahl einer entsprechenden y-Achsenskalierung. Auf dem Bildschirm wird die Linie zwar trotzdem korrekt angezeigt, aber beim SVG Export wird eine Kurve (intern Polylinie) nur bis zum ersten Erscheinen eines NAN-Werts gezeichnet. Ist zufälligerweise der erste Wert bereits ein NAN (wir hier der Fall, da aus dem erstem Wert 0 beim Logarithmieren ein NAN wird), so fehlt die Kurve im Export und man sucht dann zumeist sehr lange, bis man den Fehler gefunden hat.

Die Angabe der Auflösung mit `QSvgGenerator::setResolution()` definiert zusammen mit der gegebenen Größe die finale Auflösung des Bildes. Je höher die Auflösung, umso größer sind Font und Stiftbreiten. In diesem Zusammenhang unterscheidet sich der SVG-Export vom PDF-Export/Drucken.

Ein generiertes SVG-Dokument kann man auch in die Zwischenablage kopieren. Dazu definiert legt man im `QSvgGenerator` ein alternatives Ausgabegerät an (mit `QSvgGenerator::setOutputDevice()`) und

```
// Puffer als Ausgabegerät festlegen
QBuffer b;
generator.setOutputDevice(&b);
renderer.renderTo( &plot, generator);
// Puffer als MimeData in die Zwischenablage legen
QMimeData * d = new QMimeData();
d->setData("image/svg+xml",b.buffer());
QApplication::clipboard()->setMimeData(d,QClipboard::Clipboard);
```

### 14.1.7. EMF Export unter Windows

Qt bringt keine eigene Unterstützung für das Erstellen von EMF-Dateien mit. Diese sind mitunter aber ganz praktisch, wenn man Diagramme direkt in PowerPoint, Word oder sonstige Microsoft Software einfügen möchte.

Es gibt verschiedene EMF-Generator-Bibliotheken (open-source und kommerzielle), die man dafür gut verwenden kann. Das Vorgehen entspricht genau dem bisher gezeigten: `QwtPlotRenderer` erstellen und konfigurieren und dann in das Exportobjekt rendern (also in das `QPaintDevice` des Exportobjekts).

### 14.1.8. Anpassen des gerenderten Plots

Zwischen einem auf dem Bildschirm angezeigten Plot und einem Ausdruck gibt es häufig diverse Unterschiede. So möchte man bei schwarz-weiß-Ausdrucken wahrscheinlich keinen bunten Hintergrund des Diagramms drucken und auch der Rand des Plots soll nicht in Widget-Farben gedruckt werden. Derartige Anpassungen macht man direkt im `QwtPlotRenderer` über verschiedene Layout-Anpassungsfunktionen.

Man kann den Renderer anweisen, bestimmte Elemente nicht zu zeichnen:

```
// kein Hintergrund des Widgets
renderer.setDiscardFlag( QwtPlotRenderer::DiscardBackground );
// Keinen Zeichenflächenhintergrund
renderer.setDiscardFlag( QwtPlotRenderer::DiscardCanvasBackground );
// keinen Rahmen um die Zeichenfläche
```

```
renderer.setDiscardFlag( QwtPlotRenderer::DiscardCanvasFrame );
```

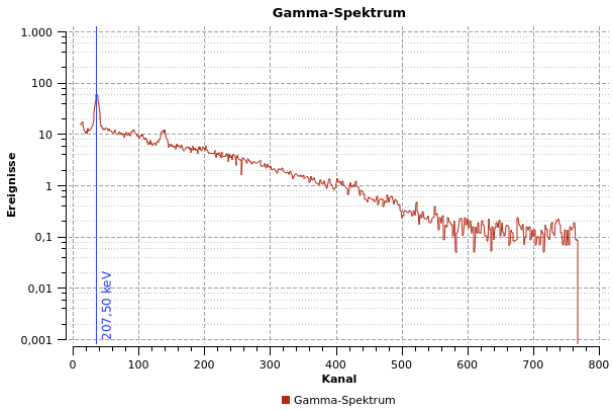



*Abbildung 65. Plot ohne Widget-Hintergrund (Rand), ohne Zeichenflächenhintergrund und ohne Rahmen*

Zusätzlich könnte man noch weitere Dinge ausblenden:

- Titel mit `QwtPlotRenderer::DiscardTitle` (da man häufig Diagrammtitel ja in die Diagrammbeschriftung schreibt)
- Legende mit `QwtPlotRenderer::DiscardLegend`
- Footer mit `QwtPlotRenderer::DiscardFooter` (falls das Plot so einen hat, siehe [sec:plotFooter])

Damit kann das Plot gedruckt schon ganz ordentlich aussehen. Man kann das Erscheinungsbild aber für einen Ausdruck noch ändern, indem man einen flachen Rahmen mit angedockten Skalen zeichnet.

```
renderer.setLayoutFlag(  );
```

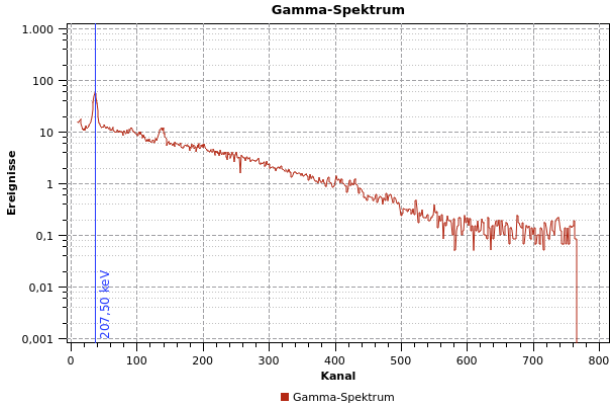



*Abbildung 66. Plot gerendert im Layoutmodus FrameWithScales*

Die linke untere Ecke des Diagramms ist hier nicht direkt am Punkt (0, 0,001). Das kann man erzwingen, wenn man die Zeichenfläche an alle vier Achsen ausrichtet:

```
plot.plotLayout()->setAlignCanvasToScale( QwtPlot::yLeft, true );
plot.plotLayout()->setAlignCanvasToScale( QwtPlot::xBottom, true );
plot.plotLayout()->setAlignCanvasToScale( QwtPlot::yRight, true );
plot.plotLayout()->setAlignCanvasToScale( QwtPlot::xTop, true );
```

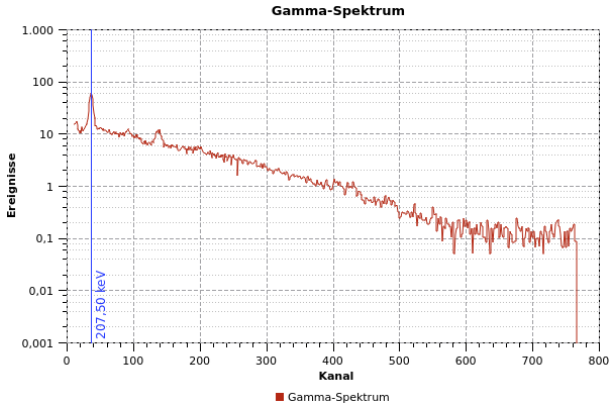



*Abbildung 67. Plot mit ausgerichteten Skalen - typisches Plotlayout für den Export*

### 14.1.9. Diagrammelemente skalieren (DPI ändern)

Wie schon oben beschrieben, wird bei unterschiedlicher Druck-Pixelgröße die Skalierung von Plotelementen neu berechnet. Bei modernen Druckern sind Auflösungen von 600 bzw. 1200 DPI normal, was bei einem A4-Ausdruck zu *sehr hohen* Pixelgrößen führt und einzelne gedruckte Pixel sind im Ausdruck nicht mehr zu erkennen.

Bei Ausgabe des Plots als Vektorgrafik, vor allem beim Drucken oder Ausgabe in PDF-Dateien, müssen *alle* Plotelement mit skalierbaren Größen festgelegt sein. Dies ist insbesondere für alle Definitionen eines `QPen()` notwendig, z.B. bei Linienkurven. Hier dürfen *keine* kosmetischen Pens mit Linienstärke 0 verwendet werden. Die resultierenden Linien mit Breiten von einem Pixel würden im Ausdruck schlicht nicht sichtbar sein.

Die Skalierung von Schriftgrößen und Linienbreiten wird über die Auflösung (angegeben in DPI) des gewählten Zeichengeräts festgelegt. Zusammen mit der Pixel-Größe des Plots ergibt sich dann das finale Erscheinungsbild der Diagramms, welches durchaus von dem Bild auf dem Bildschirm abweichen kann.

Beim Export eines Plots als Vektorgrafik (PDF oder SVG-Dokument) sind Schriftgröße und Linienstärke Eigenschaften der exportierten Vektorelemente. Wichtig ist hierbei, dass bei allen Zeichenelementen Stiftbreiten (auch als Dezimalzahl) angegeben werden.

Exportiert man das Diagramm als PDF, werden die Skalenlinien nur als kosmetische Linien mit einem Pixel Breite exportiert (d.h. beim Hineinzoomen bleiben diese Linien immer exakt ein Pixel breit). Alle anderen Diagrammelemente werden entsprechend skaliert.

*Abbildung 68. Vergrößerung des Vektorexports (PDF), Linien der Skalen werden als kosmetische Linien immer mit einem Pixel Breite gezeichnet*

Auch bei den Skalen darf man nicht vergessen, den Zeichenstift (`QPen`) entsprechend zu setzen.

```
plot.axisScaleDraw(QwtPlot::xBottom)->setPenWidthF(1);
plot.axisScaleDraw(QwtPlot::yLeft)->setPenWidthF(1);
```

Bei gewähltem Rendermodus `QwtPlotRenderer::FrameWithScales` verwendet `QwtPlot` die *breiteste* Zeichenbreite der für die einzelnen *sichtbaren* Skalen gesetzten Stifte.

Bis zur Qwt-Plot Version 6.1.3 werden die Stiftbreiten für das Zeichnen des Rahmens auf Integerwerte (ab-)gerundet. Daher führen Stiftbreiten < 1 stets zu einem kosmetischen Stift für den umliegenden Rahmen. Daher sollte man immer

eine Stiftbreite von 1 für den Rahmen verwenden, oder eben auf die Option `QwtPlotRenderer::FrameWithScales` verzichten.

Beim SVG-Generator kann man durch Veränderung des DPI-Werts das Diagramm skalieren. Analog gilt das auch, wenn mittels `QwtPlotRenderer` eine Pixelgrafik gerendert wird.

Bei den anderen Exportformaten ist die DPI-Zahl nur eine Zusatzinformation für das Anzeigeprogramm, wie das jeweilige Bild zu skalieren ist. Da aber alle Größen und Abstände relativ in der Datei abgelegt sind, und Schriftgrößen und Linienstärken Attribute der Vektorelemente sind, ändert sich das Erscheinungsbild des Diagramms beim PDF-Export/Druck nicht in Abhängigkeit der DPI-Zahl.

Beim Ausdruck/Export in PDF ist aber die Zielgröße entscheidend für das finale Erscheinungsbild, nachfolgend mal am Vergleich des Exports in A4 Format und A5 Format gezeigt:

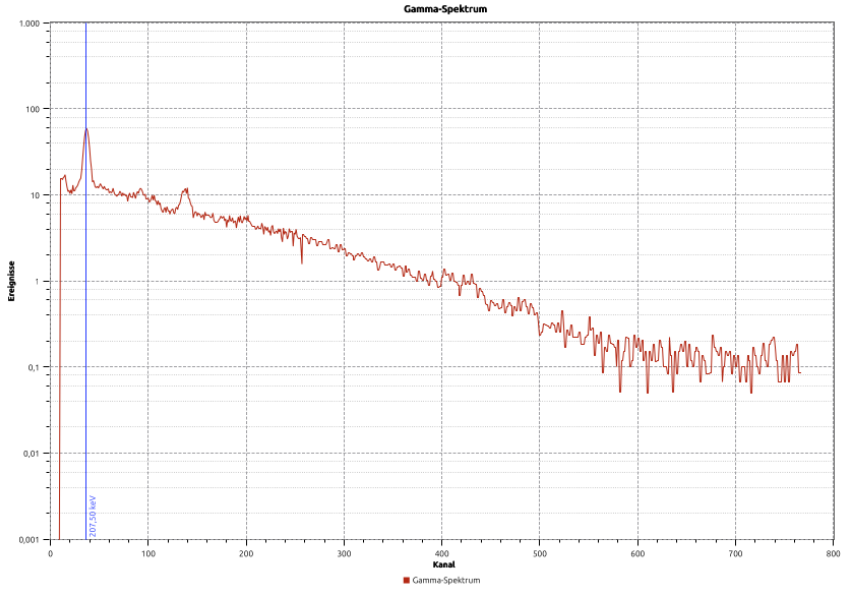



*Abbildung 69. Export in A4 Landscape Format*

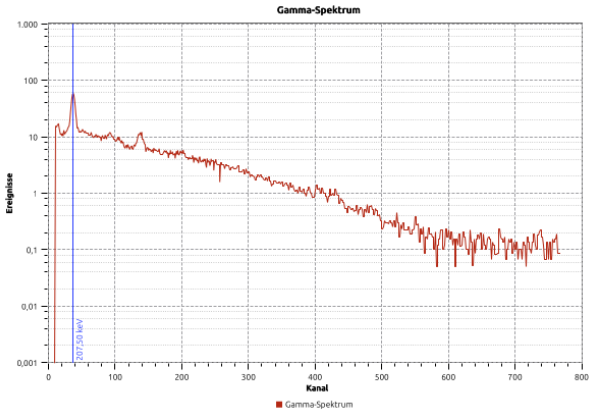



*Abbildung 70. Export in A5 Landscape Format*

Wie man gut sieht, sind Zeichenelemente und Schriftgrößen aufgrund der gleichen DPI-Zahl identisch groß. Durch die unterschiedliche Zielgröße wird aber das Diagramm unterschiedlich gelayouted und die Skalen anders berechnet.

Wenn man konsistente Diagramme in Berichten verwenden möchte, bei denen Schriftgrößen und Linienstärken immer gleich groß sind, dann braucht man beim Vektorexport nur stets die gleiche DPI-Zahl einstellen und das Bild ohne Skalierung einfügen (100% Größe).

Mitunter sind die Fonts im Diagramm aber zu klein oder zu groß, oder man möchte das Diagramm verkleinert im Bericht haben, um mehr Beschriftungsdetails zu bekommen. Die alternative wäre, im Diagramm die Fonts und Linienstärken

vor dem Export entsprechend zu vergrößern/verkleinern.

So könnte man das exportierte Diagramm eventuell mit 75% Vergrößerung in den Bericht übernehmen, oder wahlweise auf eine feste Breite (z.B. Seitenbreite) skalieren. Im letzteren Fall, also immer, wenn beim Einfügen konkrete Höhen/Breiten festgelegt werden, sollte man sicherstellen, dass das Verhältnis von Exportgeometrie und Importgeometrie stets identisch ist. Denn nur so stellt man sicher, dass Schriftgrößen in unterschiedlich großen Diagrammen dennoch stets gleich bleiben.

# 15. Fortgeschrittene Themen

Die nachfolgend vorgestellten Themen greifen in die internen Datenstrukturen der Qwt-Bibliotheksklassen ein und diese könnten sich in zukünftigen Bibliotheksversionen sicher nocheinmal deutlich ändern. Daher sind diese Techniken mit Vorsicht zu genießen!

## 15.1. Objekte aus dem QwtPlot loslösen

Die API des `QwtPlot` geht davon aus, dass Objekt beim Hinzufügen/Ersetzen existierender Plotelemente das Plot als neuen Eigentümer erhalten. Sobald ein Plotelement ein vorheriges Plotelement ersetzt, löscht das `QwtPlot` das alte Objekt automatisch. Es gibt keine *release* -Funktionen, wie man die von *shared pointer*-Implementierungen kennt. Daher kann man einmal hinzugefügte Objekte nicht entfernen, anpassen und wieder neu hinzufügen.

Bei Plotelementen, welche mittels `QwtPlotItem::attach()` hinzugefügt wurden, kann man das Element einfach mit `QwtPlotItem::detach()` wieder entfernen.

Bei der Legende geht das so jedoch nicht.

TODO den entfern-Trick erklären…

## 15.2. Splines und Bézier-Kurven

Die Qwt-Bibliothek bringt eine Reihe von Spline-Interpolationsalgorithmen mit, welche auch abseits der `QwtCurveFitter`-Funktionality (siehe Kapitel 5.7) genutzt werden können. Hier ist nochmal ein Überblick:

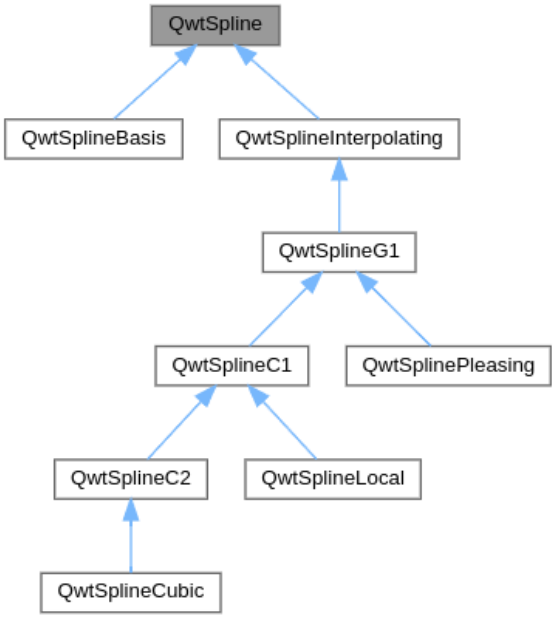



*Abbildung 71. Kind-Klassen der Basisklasse* `QwtSpline`

Direkt zu verwenden sind die Klassen:

`QwtSplinePleasing` `QwtSplineLocal` `QwtSplineCubic`

Jede dieser Spline-Implementierungen implementiert zwei Funktionen, welche man für das Generieren von Splines aus gegebenen Stützstellen benutzen kann:

- `QwtSpline::polygon()` - interpoliert das Polygon stückweise mittels Bézier-Kurven und generiert daraus einen Polygonzug als `QPolygonF` mit Punkten, welche die Spline approximieren
- `QwtSpline::painterPath()`- berechnet die Bézier-Kontrolllinien und berechnet dann stückweise kubische Funktionen zwischen den Stützstellen und fügt diese mittels `QPainterPath::cubicTo()` zum PainterPath hinzu

Die Funktion `QwtSpline::polygon( const QPolygonF&, double tolerance )` generiert eine Bézierkurve, oder genau genommen approximiert diese mittels eines Polygonzugs. Das erste Argument ist das Polygon mit den zu fittende Stützstellen und das zweite Argument die Toleranz. Die Funktion liefert selbst einen Linienzug als `QPolygonF` zurück, welcher dann weiter verwendet werden kann. Die Anzahl der Stützstellen im Polygonzug wird durch die Toleranz so definiert, dass die maximale Abweichung zwischen dem Liniensegment und der originalen Splinekurve diese Toleranz nicht überschreitet. Damit werden in Bereichen stärkerer Krümmung mehr Punkte gesetzt als in eher linearen Abschnitten der Kurve, wie man gut im nachfolgenden Beispiel sieht. Je kleiner die Toleranz, umso mehr Punkte werden verwendet.

```
QVector<double> x{1,2,5,6,10,12,15,16,8};
QVector<double> y{5,4,8,8, 4, 5, 8, 9,10};
QPolygonF poly;
for (int i=0; i<x.count(); ++i)
    poly << QPointF(x[i],y[i]);
// Spline-Implementierung (hier QwtSplinePleasing) erstellen
QwtSplinePleasing spline;
// Polygon generieren
QPolygonF splinePoly = spline.polygon(poly, 1e-2);
```

Die Stützstellen der Beispielkurve oben werden in einen Polygonzug gesteckt und durch die Spline in einen neuen Polygonzug konvertiert. Die Toleranz von 0.01 erzeugt hier 95 Punkte. Plottet man die generierte Kurve im Vergleich zu der mittels `QwtSplineCurveFitter` generierten Kurve, so sehen beide Linien auf den ersten Blick identisch aus. Wie man auch gut sehen kann, werden die Punkte in den Krümmungsstellen stärker geclustert.

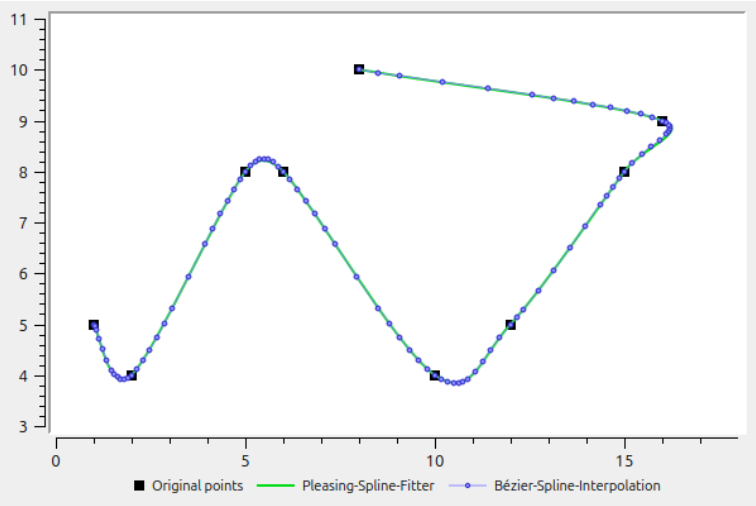



*Abbildung 72. SplinePleasing: generiert mittels polygon() und mittels painterPath()*

Hineingezoomt erkennt man jedoch Unterschiede:

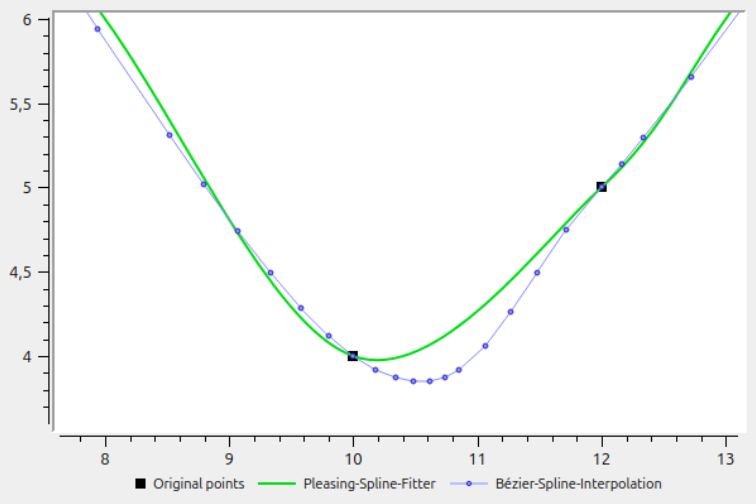



*Abbildung 73. SplinePleasing: Unterschiede zwischen generierte Spline-Polygon und PainterPath*

Auch bei `QwtSplineLocal(QwtSplineLocal::PChip)` mit (testweise veränderten Stützstellen) sieht das so aus:

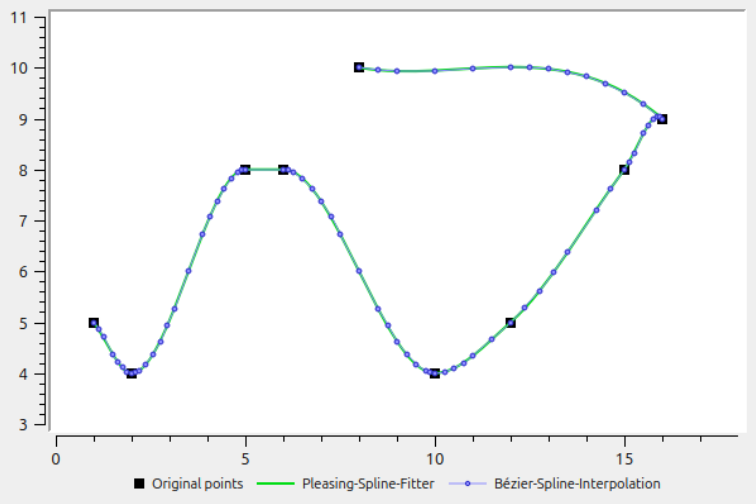



*Abbildung 74. SplineLocal: generiert mittels polygon() und mittels painterPath()*

Hineingezoomt erkennt man jedoch Unterschiede:

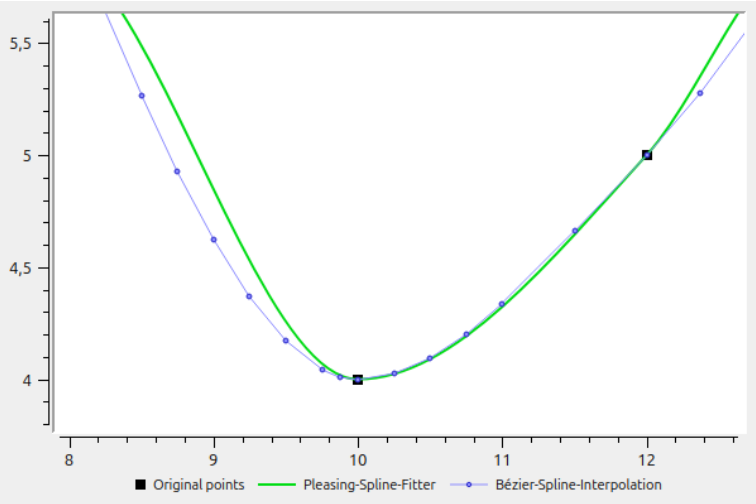



*Abbildung 75. SplineLocal: Unterschiede zwischen generierte Spline-Polygon und PainterPath*

Die Generierung der Spline als Polygonzug mittels `QwtSpline::polygon()` liefert eine genaue Approximation der jeweiligen Spline im Rahmen der geforderten Genauigkeit. Für die Visualisierung der Spline reicht häufig die Variante als PainterPath aus, wobei hier der Verlauf der Kurve zwischen den Stützstellen durch ein kubisches Polynom abgebildet wird. Qt zeichnet solche Polygonzüge sehr effizient, daher ist das für die Verwendung im Diagramm auch die optimale Variante. Aufgrund der unterschiedlichen Generierungsalgorithmen sind beide Varianten aber *nicht* identisch.

# 16. Download/Installation/Erstellung der Qwt Bibliothek

## 16.1. Download fertiger Pakete

### 16.1.1. Windows/Mac

Auf diesen Plattformen würde ich immer das Bauen aus dem Quelltext empfehlen, da das hinreichend einfach ist (siehe Kapitel 16.2 unten).

### 16.1.2. Linux

Unter Linux kann man auf die Pakete des Paketmanagers zurückgreifen.

#### Debian/Ubuntu

Beispielsweise für Ubuntu 24.04:

```
# Paket mit Headern für die Entwicklung
sudo apt install libqwt-qt5-dev
```

Headerdatei-Pfad: `/usr/include/qwt`

Für's Deployment eigener Programme und als Abhängigkeit eigener Pakete reicht es, dass Paket `libqwt-qt5-6` zu installieren.

## 16.2. Erstellung aus dem Quelltext

### 16.2.1. Windows

- Release `qwt-6.3.0.zip` herunterladen und entpacken.
- Datei `qwtconfig.pri` bearbeiten und Optionen ein-/ausschalten
- Kommandozeile mit Qt Umgebungsvariablen öffnen, z.B.: *Startmenu  Qt 5.15.2 (MinGW 8.1.0 64-bit)*, oder alternativ in der Commandozeile die benötigten Umgebungsvariablen setzen.
- Ins Verzeichnis mit der `qwt.pro` wechseln

In der Commandozeile wird erwartet, dass:

- der Compiler ausführbar d.h. im Suchpfad ist
- der QTPATH gesetzt ist

#### MinGW32/64

Es wird eine MinGW32/64 Installation mit `mingw32-make` im PATH erwartet.

```
:: Makefile erstellen
qmake qwt.pro
:: Bibliothek und Plugin/Beispiele bauen
mingw32-make -j8
:: Biblithek installieren
mingw32-make install
```

Die `-j8` sind für das parallele Bauen auf 8 CPUs.

### Visual Studio Compiler

Es gibt verschiedene Compilerversionen, wobei 2017, 2019, 2022 oder VSCode aktuell üblich sind. Eine vorbereitete Kommandozeile öffnet man am Besten über den vorbereiteten Startmenü-Link, welcher für 2019 ungefähr so heißt: *Entwickler-Eingabeaufforderung für VS 2019*

Man muss hier darauf achten, dass man die richtige Variante wählt, also x86 oder x64.

Alternativ kann man auch eine normale Kommandozeile öffnen und danach die Compilerpfade und Optionen setzen. Hier hilft es, die normalerweise über das Startmenü verknüpfte Batchdatei auszuführen:

```
"%ProgramFiles(x86)%\Microsoft Visual Studio\2019\Community\Common7\Tools\VsDevCmd.bat" -arch=amd64
```

Gebaut wird mit jom:

```
:: Makefile erstellen
qmake qwt.pro
:: Bibliothek und Plugin/Beispiele bauen
nmake
:: Biblithek installieren
nmake install
```

### Installationsverzeichnis/Relevante Pfade

Sofern nicht in der Datei `qwtconfig.pri` ein anderer Installationspräfix in der Variable `QWT_INSTALL_PREFIX` eingestellt wurde, ist die Bibliothek nach dem Erstellen unter `c:\Qwt-6.3.0` installiert:

```
c:\Qwt-6.3.0\include  - Header-Dateien
c:\Qwt-6.3.0\lib      - Bibliothek/DLLs
c:\Qwt-6.3.0\doc\html - API Dokumentation (`index.html` in diesem Verzeichnis öffnen)
```

### 16.2.2. Linux/Mac

## 16.3. Qt Designer Plugins

i. wie erstellt man die Designerplugins und bekommt die in die Komponentenpalette…

## 16.4. Verwendung des Plots in eigenen Programmen

### 16.4.1. Windows

### 16.4.2. Linux/Mac

## 16.5. Das QwtPlot in eine Designer-Oberfläche/ui-Datei integrieren

Wenn man mittels Qt Designer eine Programmoberfläche baut, möchte man da vielleicht auch ein `QwtPlot` einbetten. Das kann man auf zwei verschiedene Arten machen:

1. ein QWidget als Platzhalter einfügen und zu einem Platzhalterwidget für das `QwtPlot` machen, oder
2. die Qwt-Designer-Plugins verwenden.

### 16.5.1. Definition eines Platzhalterwidgets

Zur Erklärung wird im Qt Designer ein einfaches Widget entworfen:

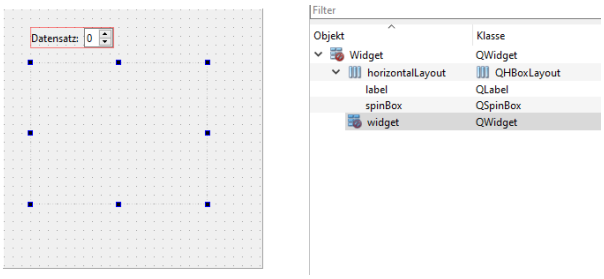
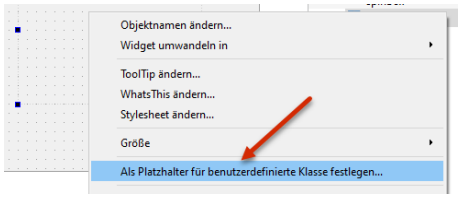
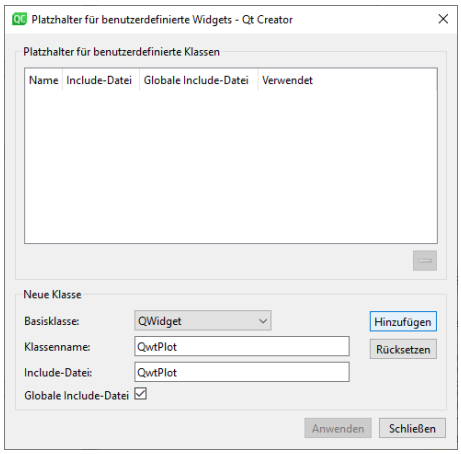



*Abbildung 76. Widget mit Platzhalter-Widget für das Diagramm*

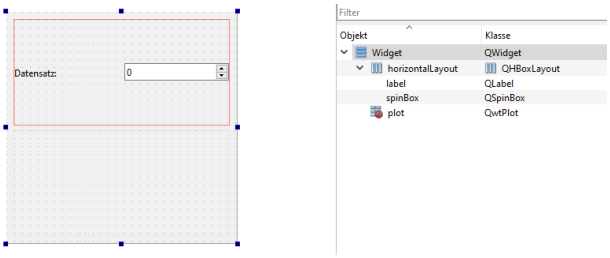
Unter der Spinbox wurde ein `QWidget` eingefügt. Dieses soll nun als Platzhalter für das `QwtPlot` dienen. Dazu im Kontextmenü des Widgets die Option "Als Platzhalter für benutzerdefinierte Klasse festlegen..." auswählen:




Und im Dialog eine neue Platzhalterklasse wie folgt definieren:




Die Eingabe mit "Hinzufügen" bestätigen und dann auf "Anwenden" klicken, um das Platzhalter-Widget in das `QwtPlot` zu wandeln. Wir benennen das noch in *plot* um, und füge das horizontale Layout und das Plotwidget in ein vertikales Layout ein:




Damit sich das Plotwidget den ganzen vertikalen Platz schnappt, wählt man das Top-Level Widget aus und scrollt in der Eigenschaftsleiste bis nach unten zu den Einstellungen für das vertikale Layout. Dort gibt man bei den Stretch-Faktoren "0,1" ein, wodurch sich das 2. Widget im Layout (das Plot) komplett ausdehnt.

### 16.5.2. Verwendung der Designer-Plugins

Dazu muss man die QtDesigner-Plugins zunächst erstellen und integrieren.

TODO :

Wenn man die erstmal installiert hat, kann man ein `QwtPlot` direkt aus der Komponentenpalette in den Entwurf zeihen und ist fertig.

# 17. Über den Autor

i. später, siehe https://schneggenport.de