Manual for Package PGFPLOTS

2D/3D Plots in LaTeX, Version 1.5.1

http://sourceforge.net/projects/pgfplots

Dr. Christian Feuersänger
ludewich@users.sourceforge.net

December 29, 2011

Abstract

PGFPLOTS draws high–quality function plots in normal or logarithmic scaling with a user-friendly interface directly in TeX. The user supplies axis labels, legend entries and the plot coordinates for one or more plots and PGFPLOTS applies axis scaling, computes any logarithms and axis ticks and draws the plots. It supports line plots, scatter plots, piecewise constant plots, bar plots, area plots, mesh– and surface plots, patch plots, contour plots, quiver plots, histogram plots, polar axes, ternary diagrams, smith charts and some more. It is based on Till Tantau's package PGF/TikZ.

Contents

1 Introduction **4**

2 About PGFPlots: Preliminaries **5**
2.1 Components . . . 5
2.2 Upgrade remarks . . . 5
2.2.1 New Optional Features . . . 5
2.2.2 Old Features Which May Need Attention . . . 6
2.3 The Team . . . 7
2.4 Acknowledgements . . . 7
2.5 Installation and Prerequisites . . . 8
2.5.1 Licensing . . . 8
2.5.2 Prerequisites . . . 8
2.5.3 Installation in Windows . . . 8
2.5.4 Installation of Linux Packages . . . 8
2.5.5 Installation in Any Directory - the TEXINPUTS Variable . . . 8
2.5.6 Installation Into a Local TDS Compliant texmf-Directory . . . 9
2.5.7 Installation If Everything Else Fails... . . . 9
2.6 Troubleshooting – Error Messages . . . 9
2.6.1 Problems with available Dimen-registers . . . 9
2.6.2 Dimension Too Large Errors . . . 9
2.6.3 Restrictions for DVI-Viewers and dvipdfm . . . 10
2.6.4 Problems with TeX's Memory Capacities . . . 10
2.6.5 Problems with Language Settings and Active Characters . . . 10
2.6.6 Other Problems . . . 11

3 User's Guide: Drawing Axes and Plots **12**
3.1 TeX-dialects: LaTeX, ConTeXt, plain TeX . . . 12
3.2 A First Plot . . . 14
3.3 Two Plots in the Same Axis . . . 14
3.4 Logarithmic Plots . . . 15
3.5 Cycling Line Styles . . . 17
3.6 Scaling Plots . . . 18

4 The Reference **20**
4.1 The Axis-Environments . . . 20
4.2 The \addplot Command: Coordinate Input . . . 21
4.2.1 Coordinate Lists . . . 24
4.2.2 Reading Coordinates From Files . . . 25
4.2.3 Reading Coordinates From Tables . . . 26
4.2.4 Computing Coordinates with Mathematical Expressions . . . 31
4.2.5 Mathematical Expressions And File Data . . . 34
4.2.6 Computing Coordinates with Mathematical Expressions (gnuplot) . . . 36
4.2.7 Computing Coordinates with External Programs (shell) . . . 38
4.2.8 Using External Graphics as Plot Sources . . . 39
4.3 About Options: Preliminaries . . . 48
4.3.1 PGFPLOTS and TikZ Options . . . 50
4.4 Two Dimensional Plot Types . . . 51
4.4.1 Linear Plots . . . 51
4.4.2 Smooth Plots . . . 51
4.4.3 Constant Plots . . . 52
4.4.4 Bar Plots . . . 54
4.4.5 Histograms . . . 61
4.4.6 Comb Plots . . . 64
4.4.7 Quiver Plots (Arrows) . . . 65
4.4.8 Stacked Plots . . . 68
4.4.9 Area Plots . . . 71
4.4.10 Scatter Plots . . . 74
4.4.11 1D Colored Mesh Plots . . . 83
4.4.12 Interrupted Plots . . . 84
4.4.13 Patch Plots . . . 85
4.5 Three Dimensional Plot Types . . . 85
4.5.1 Before You Start With 3D . . . 86
4.5.2 The \addplot3 Command: Three Dimensional Coordinate Input . . . 86
4.5.3 Line Plots . . . 90
4.5.4 Scatter Plots . . . 91
4.5.5 Mesh Plots . . . 93
4.5.6 Surface Plots . . . 96
4.5.7 Contour Plots . . . 101
4.5.8 Parameterized Plots . . . 108
4.5.9 3D Quiver Plots (Arrows) . . . 109
4.5.10 About 3D Const Plots and 3D Bar Plots . . . 109
4.5.11 Patch Plots . . . 109
4.6 Markers, Linestyles, (Background-) Colors and Colormaps . . . 116
4.6.1 Markers . . . 116
4.6.2 Line Styles . . . 120
4.6.3 Edges and Their Parameters . . . 121
4.6.4 Font Size and Line Width . . . 122
4.6.5 Colors . . . 123
4.6.6 Color Maps . . . 124
4.6.7 Cycle Lists – Options Controlling Line Styles . . . 129
4.6.8 Axis Background . . . 137
4.7 Providing Color Data - Point Meta . . . 137
4.8 Axis Descriptions . . . 142
4.8.1 Placement of Axis Descriptions . . . 142
4.8.2 Alignment of Axis Descriptions . . . 147
4.8.3 Labels . . . 149
4.8.4 Legends . . . 152
4.8.5 Legend Appearance . . . 155
4.8.6 Legends with \label and \ref . . . 163
4.8.7 Legends Outside Of an Axis . . . 165
4.8.8 Legends with Customized Texts or Multiple Lines . . . 167

4.8.9 Axis Lines . . . 168
4.8.10 Two Ordinates . . . 173
4.8.11 Axis Discontinuities . . . 173
4.8.12 Color Bars . . . 175
4.8.13 Color Bars Outside Of an Axis . . . 185
4.8.14 Scaling Descriptions: Predefined Styles . . . 186
4.9 Scaling Options . . . 189
4.10 3D Axis Configuration . . . 198
4.10.1 View Configuration . . . 198
4.10.2 Styles Used Only For 3D Axes . . . 200
4.10.3 Appearance Of The 3D Box . . . 201
4.10.4 Axis Line Variants . . . 204
4.11 Error Bars . . . 204
4.11.1 Input Formats of Error Coordinates . . . 207
4.12 Number Formatting Options . . . 207
4.12.1 Frequently Used Number Printing Settings . . . 208
4.12.2 PGFPlots-specific Number Formatting . . . 209
4.13 Specifying the Plotted Range . . . 212
4.14 Tick Options . . . 218
4.14.1 Tick Coordinates and Label Texts . . . 218
4.14.2 Tick Alignment: Positions and Shifts . . . 229
4.14.3 Tick Scaling - Common Factors In Ticks . . . 230
4.14.4 Tick Fine-Tuning . . . 234
4.15 Grid Options . . . 235
4.16 Custom Annotations . . . 236
4.16.1 Accessing Axis Coordinates in Graphical Elements . . . 237
4.16.2 Placing Nodes on Coordinates of a Plot . . . 241
4.16.3 Placing Decorations on Top of a Plot . . . 245
4.17 Style Options . . . 246
4.17.1 All Supported Styles . . . 246
4.17.2 (Re)Defining Own Styles . . . 253
4.18 Alignment Options and Bounding Box Control . . . 253
4.18.1 Basic Alignment . . . 253
4.18.2 Vertical Alignment with `baseline` . . . 256
4.18.3 Horizontal Alignment . . . 257
4.18.4 Alignment In Array Form (Subplots) . . . 258
4.18.5 Miscellaneous for Alignment . . . 262
4.18.6 Bounding box restrictions . . . 262
4.19 Closing Plots (Filling the Area Under Plots) . . . 265
4.20 Symbolic Coordinates and User Transformations . . . 266
4.20.1 String Symbols as Input Coordinates . . . 267
4.20.2 Dates as Input Coordinates . . . 268
4.21 Skipping Or Changing Coordinates – Filters . . . 270
4.22 Transforming Coordinate Systems . . . 274
4.23 Fitting Lines – Regression . . . 275
4.24 Miscellaneous Options . . . 278
4.25 Ti*k*Z Interoperability . . . 284

5 Related Libraries **288**
5.1 Clickable Plots . . . 288
5.1.1 Overview . . . 288
5.1.2 Requirements for the Library . . . 291
5.1.3 Customization . . . 292
5.1.4 Using the Clickable Library in Other Contexts . . . 293
5.2 Colormaps . . . 294
5.3 Dates as Input Coordinates . . . 299
5.4 Image Externalization . . . 299
5.5 Grouping plots . . . 299

5.5.1 Grouping options . . . 302
5.6 Patchplots Library . . . 305
5.6.1 Additional Patch Types . . . 305
5.6.2 Automatic Patch Refinement and Triangulation . . . 312
5.6.3 Peculiarities of Flat Shading and High Order Patches . . . 313
5.6.4 Drawing Grids . . . 314
5.7 Polar Axes . . . 316
5.7.1 Polar Axes . . . 316
5.7.2 Using Radians instead of Degrees . . . 318
5.7.3 Mixing With Cartesian Coordinates . . . 319
5.7.4 Special Polar Plot Types . . . 319
5.7.5 Partial Polar Axes . . . 320
5.8 Smith Charts . . . 322
5.8.1 Smith Chart Axes . . . 322
5.8.2 Size Control . . . 323
5.8.3 Working with Prepared Data . . . 327
5.8.4 Appearance Control and Styles . . . 328
5.8.5 Controlling Arcs and Their Stop Points . . . 329
5.9 Ternary Diagrams . . . 331
5.9.1 Ternary Axis . . . 331
5.9.2 Tieline Plots . . . 340
5.10 Units in Labels . . . 342
5.10.1 Preset SI prefixes . . . 344

6 Memory and Speed considerations **346**
6.1 Memory Limits of TEX . . . 346
6.2 Memory Limitations . . . 347
6.2.1 MikTEX . . . 347
6.2.2 TEXLive or similar installations . . . 347
6.3 Reducing Typesetting Time . . . 348

7 Import/Export From Other Formats **349**
7.1 Export to pdf/eps . . . 349
7.1.1 Using the Automatic Externalization Framework of Tikz . . . 349
7.1.2 Using the Externalization Framework of PGF By Hand . . . 355
7.2 Importing From Matlab . . . 356
7.2.1 Importing Mesh Data From Matlab To PGFPlots . . . 356
7.2.2 matlab2pgfplots.m . . . 357
7.2.3 matlab2pgfplots.sh . . . 357
7.2.4 Importing Colormaps From Matlab . . . 357
7.3 SVG Output . . . 357
7.4 Generate PGFPLOTS Graphics Within Python . . . 358

8 Utilities and Basic Level Commands **359**
8.1 Utility Commands . . . 359
8.2 Commands Inside Of PGFPlots Axes . . . 361
8.3 Path Operations . . . 362
8.4 Specifying Basic Coordinates . . . 363
8.5 Accessing Axis Limits . . . 366

Index **367**

1 Introduction

This package provides tools to generate plots and labeled axes easily. It draws normal plots, logplots and semi-logplots, in two and three dimensions. Axis ticks, labels, legends (in case of multiple plots) can be added with key-value options. It can cycle through a set of predefined line/marker/color specifications. In

summary, its purpose is to simplify the generation of high-quality function and/or data plots, and solving the problems of

- consistency of document and font type and font size,
- direct use of TEX math mode in axis descriptions,
- consistency of data and figures (no third party tool necessary),
- inter-document consistency using preamble configurations and styles.

Although not necessary, separate `.pdf` or `.eps` graphics can be generated using the `external` library developed as part of Ti*k*Z.

You are invited to use PGFPLOTS for visualization of medium sized data sets in two and three dimensions.

2 About PGFPLOTS: Preliminaries

This section contains information about upgrades, the team, the installation (in case you need to do it manually) and troubleshooting. You may skip it completely except for the upgrade remarks.

PGFPLOTS is built completely on Ti*k*Z/PGF. Knowledge of Ti*k*Z will simplify the work with PGFPLOTS, although it is not required.

However, note that this library requires at least PGF version 2.00. At the time of this writing, many TEX-distributions still contain the older PGF version 1.18, so it may be necessary to install a recent PGF prior to using PGFPLOTS.

2.1 Components

PGFPLOTS comes with two components:

1. the plotting component (which you are currently reading) and
2. the PGFPLOTSTABLE component which simplifies number formatting and postprocessing of numerical tables. It comes as a separate package and has its own manual pgfplotstable.pdf.

2.2 Upgrade remarks

This release provides a lot of improvements which can be found in all detail in `ChangeLog` for interested readers. However, some attention is useful with respect to the following changes.

2.2.1 New Optional Features

PGFPLOTS has been written with backwards compatibility in mind: old TEX files should compile without modifications and without changes in the appearance. However, new features occasionally lead to a different behavior. In such a case, PGFPLOTS will deactivate the new feature[1].

Any new features or bugfixes which cause backwards compatibility problems need to be activated *manually* and *explicitly*. In order to do so, you should use

```
\usepackage{pgfplots}
\pgfplotsset{compat=1.5.1}
```

in your preamble. This will configure the compatibility layer.

Here is a list of changes introduced in recent versions of PGFPLOTS:

1. PGFPLOTS 1.5.1 interpretes circle- and ellipse radii as PGFPLOTS coordinates (older versions used PGF unit vectors which have no direct relation to PGFPLOTS). In other words: starting with version 1.5.1, it is possible to write `\draw circle[radius=5]` inside of an axis. This requires `\pgfplotsset{compat=1.5.1}` or higher.

   Without this compatibility setting, circles and ellipses use low–level canvas units of PGF as in earlier versions.

[1] In case of broken backwards compatibility, we apologize – and ask you to submit a bug report. We will take care of it.

2. PGFPLOTS 1.5 uses `log origin=0` as default (which influences logarithmic bar plots or stacked logarithmic plots). Older versions keep `log origin=infty`. This requires `\pgfplotsset{compat=1.5}` or higher.

3. PGFPLOTS 1.4 has fixed several smaller bugs which might produce differences of about 1–2pt compared to earlier releases. This requires `\pgfplotsset{compat=1.4}` or higher.

4. PGFPLOTS 1.3 comes with user interface improvements. The technical distinction between "behavior options" and "style options" of older versions is no longer necessary (although still fully supported).

   This is always activated.

5. PGFPLOTS 1.3 has a new feature which allows to *move axis labels tight to tick labels* automatically. This is strongly recommended. It requires `\pgfplotsset{compat=1.3}` or higher.

   Since this affects the spacing, it is not enabled be default.

6. PGFPLOTS 1.3 now supports reversed axes. It is no longer necessary to use workarounds with negative units.

   Take a look at the `x dir=reverse` key.

   Existing workarounds will still function properly. Use `\pgfplotsset{compat=1.3}` or higher together with `x dir=reverse` to switch to the new version.

2.2.2 Old Features Which May Need Attention

1. The `scatter/classes` feature produces proper legends as of version 1.3. This may change the appearance of existing legends of plots with `scatter/classes`.

2. Starting with PGFPLOTS 1.1, `\tikzstyle` should *no longer be used* to set PGFPLOTS options.

   Although `\tikzstyle` is still supported for some older PGFPLOTS options, you should replace any occurance of `\tikzstyle` with `\pgfplotsset{`⟨*style name*⟩`/.style={`⟨*key-value-list*⟩`}}` or the associated `/.append style` variant. See Section 4.17 for more detail.

I apologize for any inconvenience caused by these changes.

`/pgfplots/compat=1.5.1|1.5|1.4|1.3|pre 1.3|default|newest` (initially `default`)

The preamble configuration

```
\usepackage{pgfplots}
\pgfplotsset{compat=1.5.1}
```

allows to choose between backwards compatibility and most recent features.

Occasionally, you might want to use different versions in the same document. Then, provide

```
\begin{figure}
    \pgfplotsset{compat=1.4}
    ...
    \caption{...}
\end{figure}
```

in order to restrict the compatibility setting to the actual context (in this case, the `figure` environment).

Use `\pgfplotsset{compat=default}` to restore the factory settings.

The setting `\pgfplotsset{compat=newest}` will always use features of the most recent version. This might result in small changes in the document's appearance. Note that it is generally a good idea to select the most recent version which is compatible with your document. This ensures that future changes will still be compatible with your document.

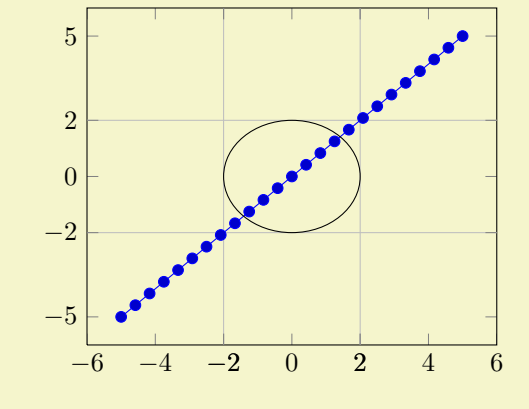
Although typically unnecessary, it is also possible to activate only selected changes and keep compatibility to older versions in general:

`/pgfplots/compat/path replacement=`⟨*version*⟩
`/pgfplots/compat/labels=`⟨*version*⟩
`/pgfplots/compat/scaling=`⟨*version*⟩

/pgfplots/compat/general=⟨*version*⟩

Let us assume that we have a document with `\pgfplotsset{compat=1.3}` and you want to keep it this way.

In addition, you realized that version 1.5.1 supports circles and ellipses. Then, use




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  preamble:
\pgfplotsset{compat=1.3,compat/path replacement=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    extra x ticks={-2,2},
    extra y ticks={-2,2},
    extra tick style={grid=major}]
    \addplot {x};
    \draw (axis cs:0,0) circle[radius=2];
\end{axis}
\end{tikzpicture}
```

All of these keys accept the possible values of the `compat` key.

The `compat/path replacement` key controls how radii of circles and ellipses are interpreted.

The `compat/labels` key controls how axis labels are aligned: either uses adjacent to ticks or with an absolute offset.

The `compat/scaling` key controls some bugfixes introduced in version 1.4: they might introduce slight scaling differences in order to improve the accuracy.

The `compat/general` key currently only activates `log origin`.

The detailed effects can be seen on the beginning of this section.

2.3 The Team

PGFPLOTS has been written mainly by Christian Feuersänger with many improvements of Pascal Wolkotte and Nick Papior Andersen as a spare time project. We hope it is useful and provides valuable plots.

If you are interested in writing something but don't know how, consider reading the auxiliary manual TeX-programming-notes.pdf which comes with PGFPLOTS. It is far from complete, but maybe it is a good starting point (at least for more literature).

2.4 Acknowledgements

I thank God for all hours of enjoyed programming. I thank Pascal Wolkotte and Nick Papior Andersen for their programming efforts and contributions as part of the development team. I thank Stefan Tibus, who contributed the `plot shell` feature. I thank Tom Cashman for the contribution of the `reverse legend` feature. Special thanks go to Stefan Pinnow whose tests of PGFPLOTS lead to numerous quality improvements. Furthermore, I thank Dr. Schweitzer for many fruitful discussions and Dr. Meine for his ideas and suggestions. Special thanks go to Markus Böhning for proof-reading all the manuals of PGF, PGFPLOTS, and PGFPLOTSTABLE. Thanks as well to the many international contributors who provided feature requests or identified bugs or simply improvements of the manual!

Last but not least, I thank Till Tantau and Mark Wibrow for their excellent graphics (and more) package PGF and Ti*k*Z, which is the base of PGFPLOTS.

2.5 Installation and Prerequisites

2.5.1 Licensing

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

A copy of the GNU General Public License can be found in the package file

`doc/latex/pgfplots/gpl-3.0.txt`

You may also visit http://www.gnu.org/licenses.

2.5.2 Prerequisites

PGFPLOTS requires PGF with **at least version** 2.0. It is used with

```
\usepackage{pgfplots}
```

in your preamble (see Section 3.1 for information about how to use it with ConTEXt and plain TEX).

There are several ways how to teach TEX where to find the files. Choose the option which fits your needs best.

2.5.3 Installation in Windows

Windows users often use MikTEX which downloads the latest stable package versions automatically. You do not need to install anything manually here.

However, MikTEX provides a feature to install packages locally in its own TEX-Directory-Structure (TDS). This is the preferred way if you like to install newer version than those of MikTEX. The basic idea is to unzip PGFPLOTS in a directory of your choice and configure the MikTEX Package Manager to use this specific directory with higher priority than its default paths. If you want to do this, start the MikTEX Settings using "Start ≫ Programs ≫ MikTEX ≫ Settings". There, use the "Roots" menu section. It contains the MikTEX Package directory as initial configuration. Use "Add" to select the directory in which the unzipped PGFPLOTS tree resides. Then, move the newly added path to the list's top using the "Up" button. Then press "Ok". For MikTEX 2.8, you may need to uncheck the "Show MikTEX-maintained root directories" button to see the newly installed path.

MikTEX complains if the provided directory is not TDS conform (see Section 2.5.6 for details), so you can't provide a wrong directory here. This method does also work for other packages, but some packages may need some directory restructuring before MikTEX accepts them.

2.5.4 Installation of Linux Packages

At the time of this writing, I am unaware of PGFPLOTS packages for recent stable Linux distributions. For Ubuntu, there are unofficial Ubuntu Package Repositories which can be added to the Ubuntu Package Tools. The idea is: add a simple URL to the Ubuntu Package Tool, run update and the installation takes place automatically. These URLs are maintained as PPA on Ubuntu Servers.

The PGFPLOTS download area on sourceforge contains recent links about Ubuntu Package Repositories, go to http://sourceforge.net/projects/pgfplots/files and download the readme files with recent links.

2.5.5 Installation in Any Directory - the `TEXINPUTS` Variable

You can simply install PGFPLOTS anywhere on your harddrive, for example into

`/foo/bar/pgfplots.`

Then, you set the `TEXINPUTS` variable to

```
TEXINPUTS=/foo/bar/pgfplots//:
```

The trailing ':' tells TEX to check the default search paths after `/foo/bar/pgfplots`. The double slash '//' tells TEX to search all subdirectories.

If the `TEXINPUTS` variable already contains something, you can append the line above to the existing `TEXINPUTS` content.

Furthermore, you should set `TEXDOCS` as well,

```
TEXDOCS=/foo/bar/pgfplots//:
```

so that the TEX-documentation system finds the files `pgfplots.pdf` and `pgfplotstable.pdf` (on some systems, it is then enough to use `texdoc pgfplots`).

Please refer to your operating systems manual for how to set environment variables.

2.5.6 Installation Into a Local TDS Compliant `texmf`-Directory

PGFPLOTS comes in a "TEX Directory Structure" (TDS) conforming directory structure, so you can simply unpack the files into a directory which is searched by TEX automatically. Such directories are `~/texmf` on Linux systems, for example.

Copy PGFPLOTS to a local `texmf` directory like `~/texmf`. You need at least the PGFPLOTS directories `tex/generic/pgfplots` and `tex/latex/pgfplots`. Then, run `texhash` (or some equivalent path-updating command specific to your TEX distribution).

The TDS consists of several sub directories which are searched separately, depending on what has been requested: the sub directories `doc/latex/`⟨*package*⟩ are used for (LATEX) documentation, the sub-directories `doc/generic/`⟨*package*⟩ for documentation which apply to LATEX and other TEX dialects (like plain TEX and ConTEXt which have their own, respective sub-directories) as well.

Similarly, the `tex/latex/`⟨*package*⟩ sub-directories are searched whenever LATEX packages are requested. The `tex/generic/`⟨*package*⟩ sub-directories are searched for packages which work for LATEX *and* other TEX dialects.

Do not forget to run `texhash`.

2.5.7 Installation If Everything Else Fails...

If TEX still doesn't find your files, you can copy all `.sty` and all `.code.tex`-files (perhaps all `.def` files as well) into your current project's working directory. In fact, you need everything which is in the `tex/latex/pgfplots` and `tex/generic/pgfplots` sub directories.

Please refer to http://www.ctan.org/installationadvice/ for more information about package installation.

2.6 Troubleshooting – Error Messages

This section discusses some problems which may occur when using PGFPLOTS. Some of the error messages are shown in the index, take a look at the end of this manual (under "Errors").

2.6.1 Problems with available Dimen-registers

To avoid problems with the many required TEX-registers for PGF and PGFPLOTS, you may want to include

```
\usepackage{etex}
```

as first package. This avoids problems with "no room for a new dimen" in most cases. It should work with any modern installation of TEX (it activates the e-TEX extensions).

2.6.2 Dimension Too Large Errors

The core mathematical engine of PGF relies on TEX registers to perform fast arithmetics. To compute $50 + 299$, it actually computes `50pt+299pt` and strips the `pt` suffix of the result. Since TEX registers can only contain numbers up to ± 16384, overflow error messages like "Dimension too large" occur if the result leaves the allowed range. Normally, this should never happen – PGFPLOTS uses a floating point unit with data range $\pm 10^{324}$ and performs all mappings automatically. However, there are some cases where this fails. Some of these cases are:

1. The axis range (for example, for x) becomes *relatively* small. It's no matter if you have absolutely small ranges like $[10^{-17}, 10^{-16}]$. But if you have an axis range like $[1.99999999, 2]$, where a lot of significant digits are necessary, this may be problematic.

   I guess I can't help here: you may need to prepare the data somehow before PGFPLOTS processes it.

2. This may happen as well if you only view a very small portion of the data range.

   This happens, for example, if your input data ranges from $x \in [0, 10^6]$, and you say `xmax=10`.

   Consider using the `restrict x to domain*=`$\langle min \rangle$`:`$\langle max \rangle$ key in such a case, where the $\langle min \rangle$ and $\langle max \rangle$ should be (say) four times of your axis limits (see page 272 for details).

3. The `axis equal` key will be confused if x and y have a very different scale.

4. You may have found a bug – please contact the developers.

2.6.3 Restrictions for DVI-Viewers and `dvipdfm`

PGF is compatible with

- `latex/dvips`,
- `latex/dvipdfm`,
- `pdflatex`,
- ⋮

However, there are some restrictions: I don't know any DVI viewer which is capable of viewing the output of PGF (and therefor PGFPLOTS as well). After all, DVI has never been designed to draw something different than text and horizontal/vertical lines. You will need to view the postscript file or the pdf-file.

Then, the DVI/pdf combination doesn't support all types of shadings (for example, the `shader=interp` is only available for `dvips` and `pdftex` drivers).

Furthermore, PGF needs to know a *driver* so that the DVI file can be converted to the desired output. Depending on your system, you need the following options:

- `latex/dvips` does not need anything special because `dvips` is the default driver if you invoke `latex`.
- `pdflatex` will also work directly because `pdflatex` will be detected automatically.
- `latex/dvipdfm` requires to use

```
\def\pgfsysdriver{pgfsys-dvipdfm.def}
%\def\pgfsysdriver{pgfsys-pdftex.def}
%\def\pgfsysdriver{pgfsys-dvips.def}
\usepackage{pgfplots}.
```

  The uncommented commands could be used to set other drivers explicitly.

Please read the corresponding sections in [5, Section 7.2.1 and 7.2.2] if you have further questions. These sections also contain limitations of particular drivers.

The choice which won't produce any problems at all is `pdflatex`.

2.6.4 Problems with TEX's Memory Capacities

PGFPLOTS can handle small up to medium sized plots. However, TEX has never been designed for data plots – you will eventually face the problem of small memory capacities. See Section 6.1 for how to enlarge them.

2.6.5 Problems with Language Settings and Active Characters

Both PGF and PGFPLOTS use a lot of active characters – which may lead to incompatibilities with other packages which define active characters. Compatibility is better than in earlier versions, but may still be an issue. The manual compiles with the `babel` package for english and french, the `german` package does also work. If you experience any trouble, let me know. Sometimes it may work to disable active characters temporarily (`babel` provides such a command).

2.6.6 Other Problems

Please read the mailing list at
http://sourceforge.net/projects/pgfplots/support.
Perhaps someone has also encountered your problem before, and maybe he came up with a solution.

Please write a note on the mailing list if you have a different problem. In case it is necessary to contact the authors directly, consider the addresses shown on the title page of this document.

3 User's Guide: Drawing Axes and Plots

The user interface of PGFPLOTS consists of three components: a `tikzpicture` environment, an `axis` and the `\addplot` command.

Each axis is generated as part of a picture environment (which can be used to annotate plots afterwards, for example). The axis environment encapsulates one or more `\addplot` commands and controls axis-wide settings (like limits, legends, and descriptions). The `\addplot` command supports several coordinate input methods (like table input or mathematical expressions) and allows various sorts of visualization options with straight lines as initial configuration.

The rest of PGFPLOTS is a huge set of key–value options to modify the initial configuration or to select plot types. The reference manual has been optimized for electronical display: a lot of examples illustrate the features, and reference documentation can be found by clicking into the sourcecode text fields. Note that most pdf viewers also support to jump back from a hyperlink: for Acrobat Reader, open the menu View≫Toolbars≫More Tools and activate the "Previous View" and "Next View" buttons (which are under "Page Navigation Toolbar"). Thus, knowledge of all keys is unnecessary; you can learn them when it is necessary.

To learn PGFPLOTS, you should learn about the `\addplot` command and its coordinate input methods. The most important input methods are `\addplot table` and `\addplot expression`.

The following sections explain the basics of PGFPLOTS, namely how to work with the `\addplot` commands and `axis` environments and how line styles are assigned automatically.

3.1 TEX-dialects: LATEX, ConTEXt, plain TEX

The starting point for PGFPLOTS is an `axis` enviroment like `axis` or the logarithmic variants `semilogxaxis`, `semilogyaxis` or `loglogaxis`.

Each environment is available for LATEX, ConTEXt and plain TEX:

LATEX: `\usepackage{pgfplots}` and

```
\begin{tikzpicture}
\begin{axis}
...
\end{axis}
\end{tikzpicture}
```

```
\begin{tikzpicture}
\begin{semilogxaxis}
...
\end{semilogxaxis}
\end{tikzpicture}
```

```
\documentclass[a4paper]{article}

%  for dvipdfm:
% \def\pgfsysdriver{pgfsys-dvipdfm.def}
\usepackage{pgfplots}
\pgfplotsset{compat=1.3}%  <-- moves axis labels near ticklabels (respects tick label widths)

\begin{document}
\begin{figure}
    \centering
    \begin{tikzpicture}
        \begin{loglogaxis}[xlabel=Cost,ylabel=Error]
        \addplot coordinates {
            (5,     8.31160034e-02)
            (17,    2.54685628e-02)
            (49,    7.40715288e-03)
            (129,   2.10192154e-03)
            (321,   5.87352989e-04)
            (769,   1.62269942e-04)
            (1793,  4.44248889e-05)
            (4097,  1.20714122e-05)
            (9217,  3.26101452e-06)
        };
        \addplot coordinates {
            (7,     8.47178381e-02)
            (31,    3.04409349e-02)
            (111,   1.02214539e-02)
            (351,   3.30346265e-03)
            (1023,  1.03886535e-03)
            (2815,  3.19646457e-04)
            (7423,  9.65789766e-05)
            (18943, 2.87339125e-05)
            (47103, 8.43749881e-06)
        };
        \legend{Case 1,Case 2}
        \end{loglogaxis}
    \end{tikzpicture}
    \caption{A larger example}
\end{figure}
\end{document}
```

ConTEXt: `\usemodule[pgfplots]` and

```
\starttikzpicture
\startaxis
...
\stopaxis
\stoptikzpicture
```

```
\starttikzpicture
\startsemilogxaxis
...
\stopsemilogxaxis
\stoptikzpicture
```

A complete ConTEXt–example file can be found in

```
doc/context/pgfplots/pgfplotsexample.tex.
```

plain TEX: `\input pgfplots.tex` and

```
\tikzpicture
\axis
...
\endaxis
\endtikzpicture
```

```
\tikzpicture
\semilogxaxis
...
\endsemilogxaxis
\endtikzpicture
```

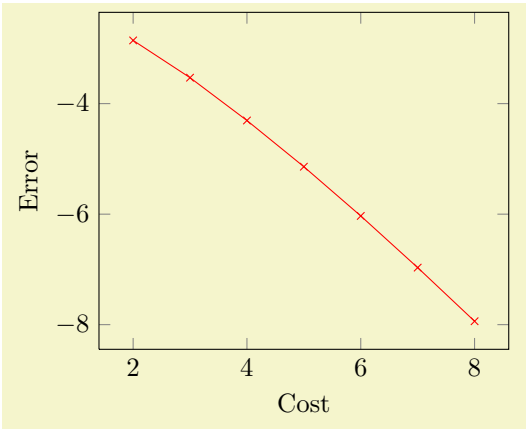
A complete plain–TEX–example file can be found in

```
doc/plain/pgfplots/pgfplotsexample.tex.
```

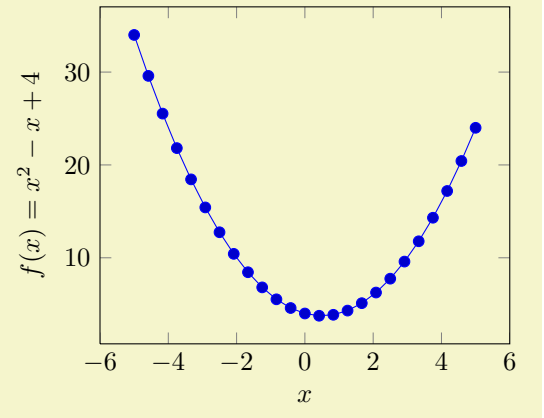
If you use `latex` / `dvips` or `pdflatex`, no further modifications are necessary. For `dvipdfm`, you should use the `\def\pgfsysdriver` line as indicated above in the examples (see also Section 2.6.3).

3.2 A First Plot

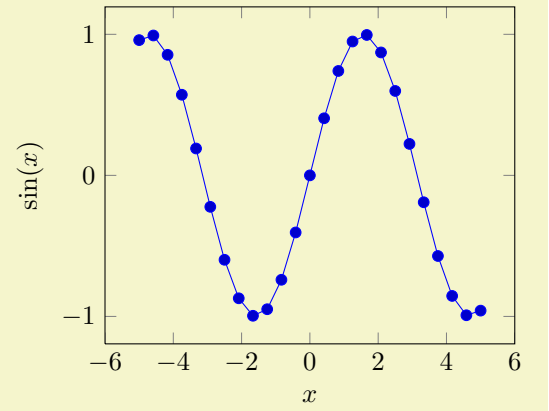
Plotting is done using `\begin{axis} ... \addplot ...; \end{axis}`, where `\addplot` is the main interface to perform plotting operations.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xlabel=Cost,
        ylabel=Error]
    \addplot[color=red,mark=x] coordinates {
        (2,-2.8559703)
        (3,-3.5301677)
        (4,-4.3050655)
        (5,-5.1413136)
        (6,-6.0322865)
        (7,-6.9675052)
        (8,-7.9377747)
    };
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xlabel=$x$,
        ylabel={$f(x) = x^2 - x +4$}
    ]
    %  use TeX as calculator:
    \addplot {x^2 - x +4};
    \end{axis}
\end{tikzpicture}
```



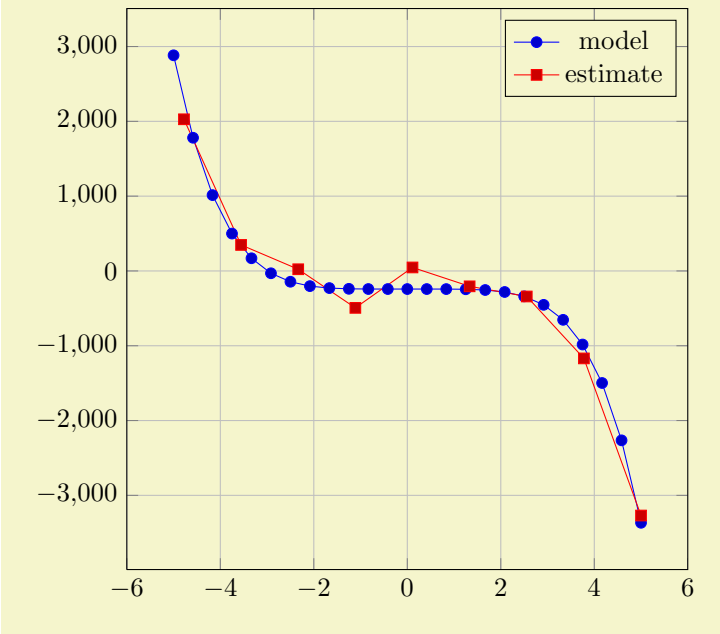

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xlabel=$x$,
        ylabel=$\sin(x)$
    ]
    %  invoke external gnuplot as
    %  calculator:
    \addplot gnuplot[id=sin]{sin(x)};
    \end{axis}
\end{tikzpicture}
```

The `plot coordinates`, `plot expression` and `plot gnuplot` commands are three of the several supported ways to create plots, see Section 4.2 for more details[2] and the remaining ones (`plot file`, `plot shell`, `plot table` and `plot graphics`). The options '`xlabel`' and '`ylabel`' define axis descriptions.

3.3 Two Plots in the Same Axis

Multiple `\addplot`-commands can be placed into the same axis, and a `cycle list` is used to automatically select different line styles:

[2]Please note that you need `gnuplot` installed to use `plot gnuplot`.

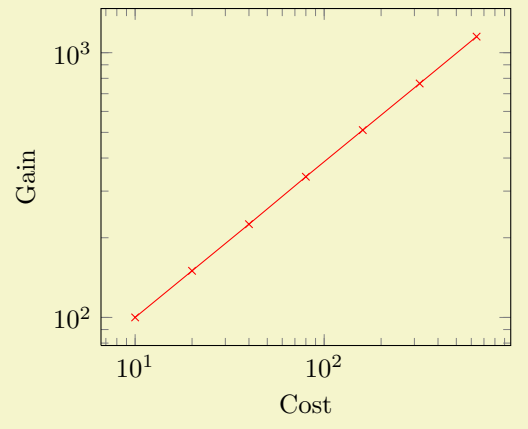



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        height=9cm,
        width=9cm,
        grid=major,
    ]

    \addplot {-x^5 - 242};
    \addlegendentry{model}

    \addplot coordinates {
        (-4.77778,2027.60977)
        (-3.55556,347.84069)
        (-2.33333,22.58953)
        (-1.11111,-493.50066)
        (0.11111,46.66082)
        (1.33333,-205.56286)
        (2.55556,-341.40638)
        (3.77778,-1169.24780)
        (5.00000,-3269.56775)
    };
    \addlegendentry{estimate}
    \end{axis}
\end{tikzpicture}
```

A legend entry is generated if there are `\addlegendentry` commands (or one `\legend` command).

3.4 Logarithmic Plots

Logarithmic plots show $\log x$ versus $\log y$ (or just one logarithmic axis). PGFPLOTS normally uses the natural logarithm, i.e. basis $e \approx 2.718$ (see the key `log basis x`). Now, the axis description also contains minor ticks and the labels are placed at 10^i.

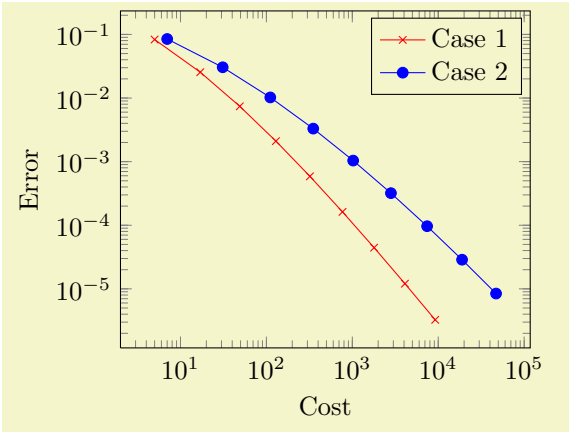



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[xlabel=Cost,ylabel=Gain]
\addplot[color=red,mark=x] coordinates {
    (10,100)
    (20,150)
    (40,225)
    (80,340)
    (160,510)
    (320,765)
    (640,1150)
};
\end{loglogaxis}
\end{tikzpicture}
```

A common application is to visualise scientific data. This is often provided in the format $1.42 \cdot 10^4$, usually written as 1.42e+04. Suppose we have a numeric table named `pgfplots.testtable`, containing

```
Level Cost  Error
1     7     8.471e-02
2     31    3.044e-02
3     111   1.022e-02
4     351   3.303e-03
5     1023  1.038e-03
6     2815  3.196e-04
7     7423  9.657e-05
8     18943 2.873e-05
9     47103 8.437e-06
```

then we can plot `Cost` versus `Error` using




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel=Cost,
    ylabel=Error]
\addplot[color=red,mark=x] coordinates {
    (5,    8.31160034e-02)
    (17,   2.54685628e-02)
    (49,   7.40715288e-03)
    (129,  2.10192154e-03)
    (321,  5.87352989e-04)
    (769,  1.62269942e-04)
    (1793, 4.44248889e-05)
    (4097, 1.20714122e-05)
    (9217, 3.26101452e-06)
};

\addplot[color=blue,mark=*]
    table[x=Cost,y=Error] {pgfplots.testtable};

\legend{Case 1,Case 2}
\end{loglogaxis}
\end{tikzpicture}
```

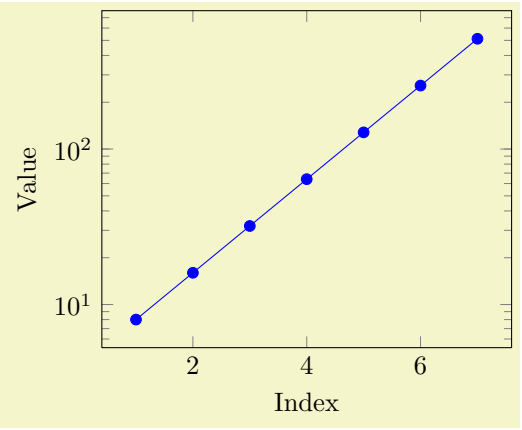
The first plot employs inline coordinates; the second one reads numerical data from file and plots column '`Cost`' versus '`Error`'.
Besides the environment "`loglogaxis`" you can use

- `\begin{axis}...\end{axis}` for normal plots,
- `\begin{semilogxaxis}...\end{semilogxaxis}` for plots which have a normal y axis and a logarithmic x axis,
- `\begin{semilogyaxis}...\end{semilogyaxis}` the same with x and y switched,
- `\begin{loglogaxis}...\end{loglogaxis}` for double–logarithmic plots.

You can also use

```
\begin{axis}[xmode=normal,ymode=log]
...
\end{axis}
```

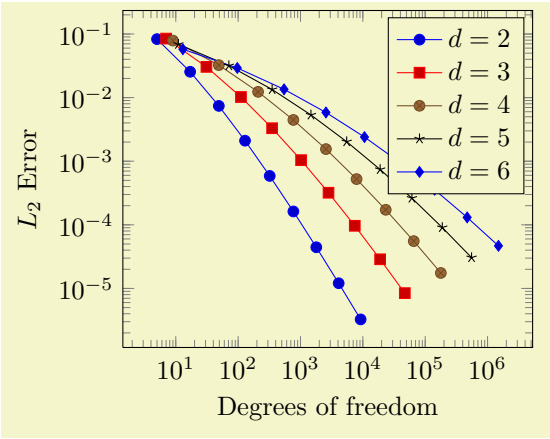
which is the same as `\begin{semilogyaxis}...\end{semilogyaxis}`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{semilogyaxis}[
        xlabel=Index,ylabel=Value]

    \addplot[color=blue,mark=*] coordinates {
        (1,8)
        (2,16)
        (3,32)
        (4,64)
        (5,128)
        (6,256)
        (7,512)
    };
    \end{semilogyaxis}%
\end{tikzpicture}%
```

3.5 Cycling Line Styles

You can skip the style arguments for `\addplot[...]` to determine plot specifications from a predefined list:

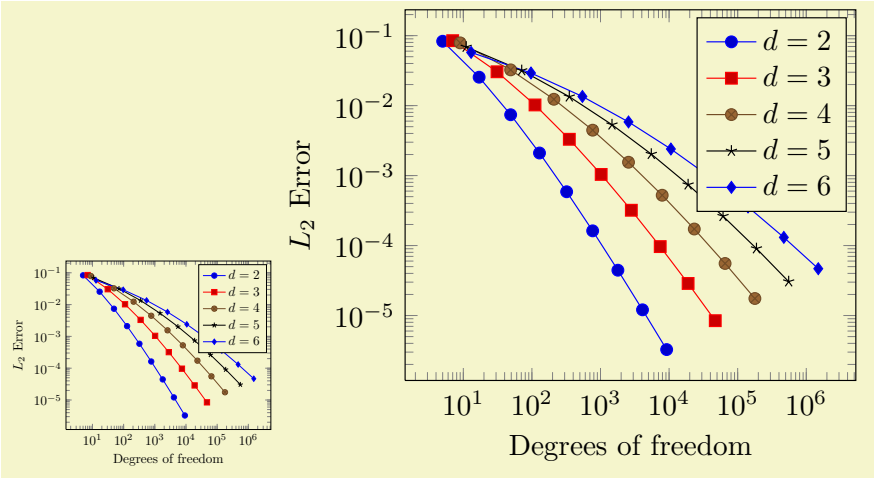


```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel={Degrees of freedom},
    ylabel={$L_2$ Error}
]
\addplot coordinates {
    (5,8.312e-02)    (17,2.547e-02)   (49,7.407e-03)
    (129,2.102e-03)  (321,5.874e-04)  (769,1.623e-04)
    (1793,4.442e-05) (4097,1.207e-05) (9217,3.261e-06)
};

\addplot coordinates{
    (7,8.472e-02)    (31,3.044e-02)    (111,1.022e-02)
    (351,3.303e-03)  (1023,1.039e-03)  (2815,3.196e-04)
    (7423,9.658e-05) (18943,2.873e-05) (47103,8.437e-06)
};

\addplot coordinates{
    (9,7.881e-02)     (49,3.243e-02)    (209,1.232e-02)
    (769,4.454e-03)   (2561,1.551e-03)  (7937,5.236e-04)
    (23297,1.723e-04) (65537,5.545e-05) (178177,1.751e-05)
};

\addplot coordinates{
    (11,6.887e-02)    (71,3.177e-02)     (351,1.341e-02)
    (1471,5.334e-03)  (5503,2.027e-03)   (18943,7.415e-04)
    (61183,2.628e-04) (187903,9.063e-05) (553983,3.053e-05)
};

\addplot coordinates{
    (13,5.755e-02)     (97,2.925e-02)     (545,1.351e-02)
    (2561,5.842e-03)   (10625,2.397e-03)  (40193,9.414e-04)
    (141569,3.564e-04) (471041,1.308e-04) (1496065,4.670e-05)
};
\legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
\end{loglogaxis}
\end{tikzpicture}
```

The `cycle list` can be modified, see the reference below.

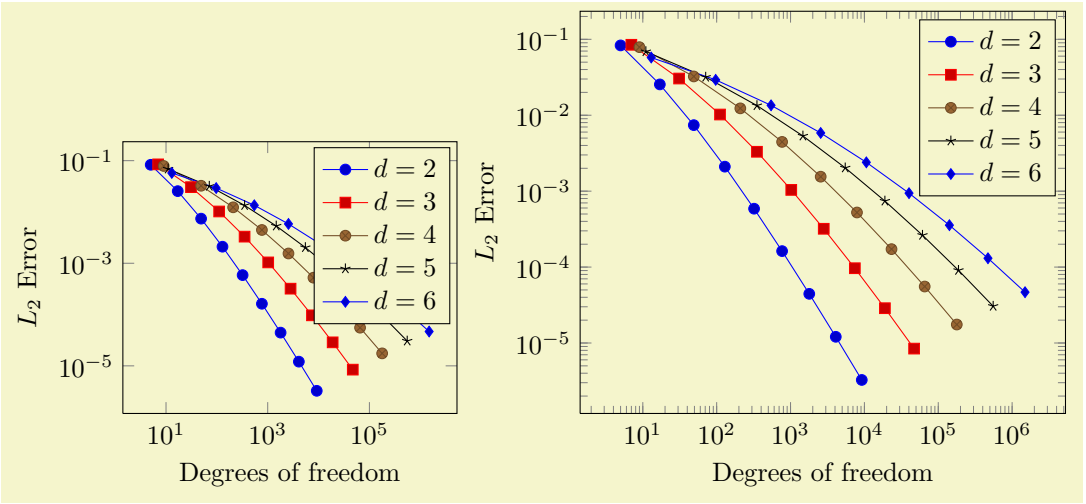
3.6 Scaling Plots

You can use any of the Ti*k*Z options to modify the appearance. For example, the "`scale`" transformation takes the picture as such and scales it (just like `\includegraphics`):



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}[scale=0.5]
    \begin{loglogaxis}[
        xlabel={Degrees of freedom},
        ylabel={$L_2$ Error}
    ]
    \plotcoords
    \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
    \end{loglogaxis}
\end{tikzpicture}

\begin{tikzpicture}[scale=1.1]
    \begin{loglogaxis}[
        xlabel={Degrees of freedom},
        ylabel={$L_2$ Error}
    ]
    \plotcoords
    \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
    \end{loglogaxis}
\end{tikzpicture}
```

However, you can also scale plots by assigning a `width=5cm` and/or `height=3cm` argument. This only affects the distance of point coordinates, no font sizes or axis descriptions:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[
        width=6cm,
        xlabel={Degrees of freedom},
        ylabel={$L_2$ Error}
    ]
    \plotcoords
    \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
    \end{loglogaxis}
\end{tikzpicture}

\begin{tikzpicture}
    \begin{loglogaxis}[
        width=8cm,
        xlabel={Degrees of freedom},
        ylabel={$L_2$ Error}
    ]
    \plotcoords
    \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
    \end{loglogaxis}
\end{tikzpicture}
```

Use the predefined styles `normalsize`, `small`, `footnotesize` to adopt font sizes and ticks automatically. Use the `/pgfplots/scale` key to rescale the axis without affecting fonts.

4 The Reference

4.1 The Axis-Environments

There is an axis environment for linear scaling, two for semi-logarithmic scaling and one for double-logarithmic scaling.

`\begin{tikzpicture}[`⟨*options of tikz*⟩`]`
⟨*environment contents*⟩
`\end{tikzpicture}`

This is the graphics environment of Ti*k*Z. It produces a single picture and encloses also every axis.

Instead of using the environment version, there is also a shortcut command

`\tikz{`⟨*picture content*⟩`}`

which can be used alternatively.

`\begin{axis}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{axis}`

The axis environment for normal plots with linear axis scaling.

The '`every linear axis`' style key can be modified with

```
\pgfplotsset{every linear axis/.append style={...}}
```

to install styles specifically for linear axes. These styles can contain both Ti*k*Z- and PGFPLOTS options.

`\begin{semilogxaxis}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{semilogxaxis}`

The axis environment for logarithmic scaling of x and normal scaling of y. Use

```
\pgfplotsset{every semilogx axis/.append style={...}}
```

to install styles specifically for the case with `xmode=log`, `ymode=normal`.

The logarithmic scaling means to apply the natural logarithm (base e) to each x coordinate. Furthermore, ticks will be typeset as $10^{\langle exponent \rangle}$, see Section 4.12 for more details.

`\begin{semilogyaxis}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{semilogyaxis}`

The axis environment for normal scaling of x and logarithmic scaling of y,

The style '`every semilogy axis`' will be installed for each such plot.

The same remarks as for `semilogxaxis` apply here as well.

`\begin{loglogaxis}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{loglogaxis}`

The axis environment for logarithmic scaling of both, x and y axes. As for the other axis possibilities, there is a style '`every loglog axis`' which is installed at the environment's beginning.

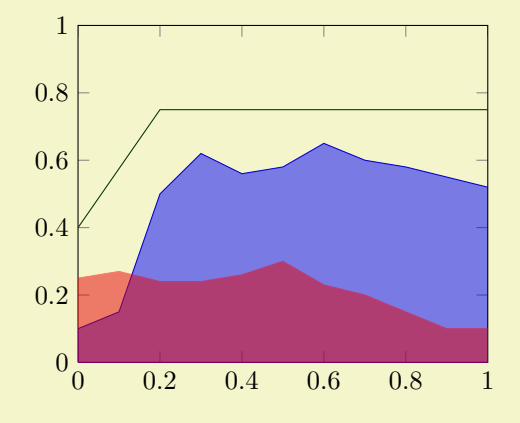
The same remarks as for `semilogxaxis` apply here as well.

They are all equivalent to

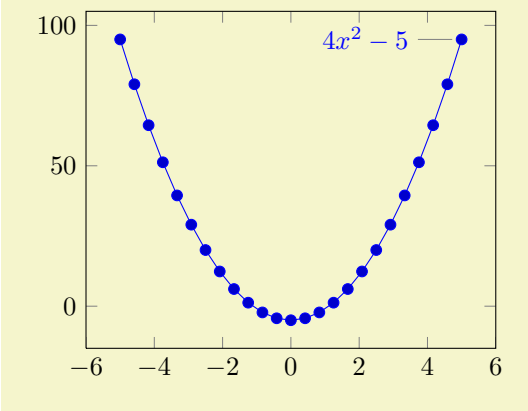
```
\begin{axis}[
    xmode=log|normal,
    ymode=log|normal]
...
\end{axis}
```

with properly set variables '`xmode`' and '`ymode`' (see below).

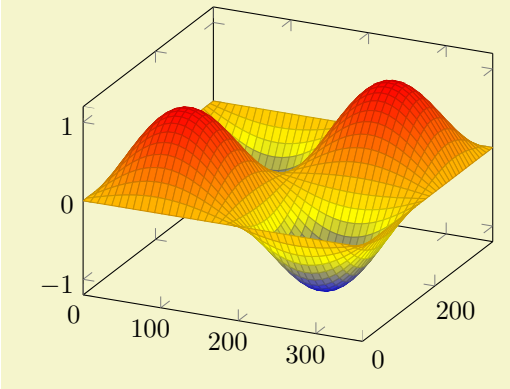
4.2 The \addplot Command: Coordinate Input



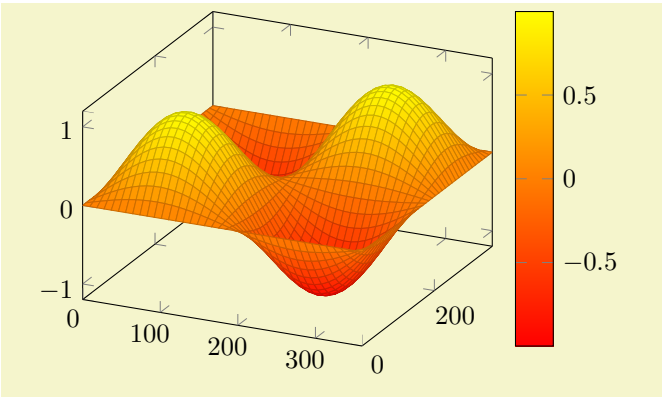

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[ymin=0,ymax=1,enlargelimits=false]
\addplot
    [blue!80!black,fill=blue,fill opacity=0.5]
coordinates
{(0,0.1)    (0.1,0.15)  (0.2,0.5)   (0.3,0.62)
 (0.4,0.56) (0.5,0.58)  (0.6,0.65)  (0.7,0.6)
 (0.8,0.58) (0.9,0.55)  (1,0.52)}
|- (axis cs:0,0) -- cycle;

\addplot
    [red,fill=red!90!black,opacity=0.5]
coordinates
{(0,0.25)   (0.1,0.27)  (0.2,0.24)  (0.3,0.24)
 (0.4,0.26) (0.5,0.3)   (0.6,0.23)  (0.7,0.2)
 (0.8,0.15) (0.9,0.1)   (1,0.1)}
|- (axis cs:0,0) -- cycle;

\addplot[green!20!black] coordinates
    {(0,0.4) (0.2,0.75) (1,0.75)};
\end{axis}
\end{tikzpicture}
```



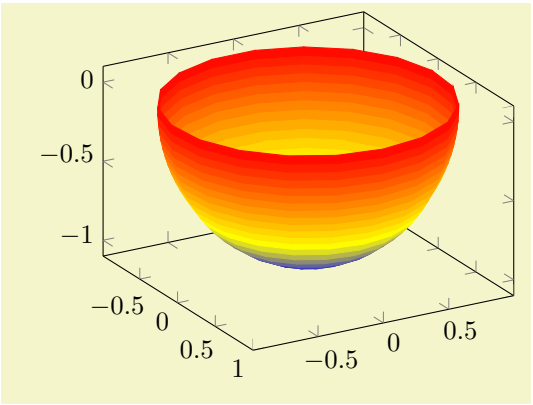

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[id=parable,domain=-5:5]
    gnuplot{4*x**2 - 5}
    node[pin=180:{$4x^2-5$}]{};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,domain=0:360,samples=40]
        {sin(x)*sin(y)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colormap/redyellow,colorbar]
    \addplot3[surf,
        domain=0:360,samples=40]
        {sin(x)*sin(y)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[view={60}{30}]
    \addplot3[surf,shader=flat,
        samples=20,
        domain=-1:0,y domain=0:2*pi,
        z buffer=sort]
        ({sqrt(1-x^2) * cos(deg(y))},
         {sqrt( 1-x^2 ) * sin(deg(y))},
         x);
\end{axis}
\end{tikzpicture}
```

Inside of an axis environment, the \addplot command is the main user interface. It comes in two variants: \addplot for two–dimensional visualization and \addplot3 for three–dimensional visualization.

\addplot[⟨*options*⟩] ⟨*input data*⟩ ⟨*trailing path commands*⟩;

This is the main plotting command, available within each axis environment. It can be used one or more times within an axis to add plots to the current axis. There is also an \addplot3 command which is described in Section 4.5.

It reads point coordinates from one of the available input sources specified by ⟨*input data*⟩, updates limits, remembers ⟨*options*⟩ for use in a legend (if any) and applies any necessary coordinate transformations (or logarithms).

The ⟨*options*⟩ can be omitted in which case the next entry from the cycle list will be inserted as ⟨*options*⟩. These keys characterize the plot's type like linear interpolation with sharp plot, smooth plot, constant interpolation with const plot, bar plot, mesh plots, surface plots or whatever and define colors, markers and line specifications[3]. Plot variants like error bars, the number of samples or a sample domain can also be configured in ⟨*options*⟩.

The ⟨*input data*⟩ is one of several coordinate input tools which are described in more detail below. Finally, if \addplot successfully processed all coordinates from ⟨*input data*⟩, it generates TikZ paths to realize the drawing operations. Any ⟨*trailing path commands*⟩ are appended to the final drawing command, allowing to continue the TikZ path (from the last plot coordinate).

Some more details:

[3] In version 1.2.2 and earlier, there was an explicit distinction between "behaviour" options like error bars, domain, number of samples etc. and "style options" like color, line width, markers etc. This distinction is obsolete now, simply collect everything into ⟨*options*⟩.

- The style `/pgfplots/every axis plot` will be installed at the beginning of ⟨*options*⟩. That means you can use

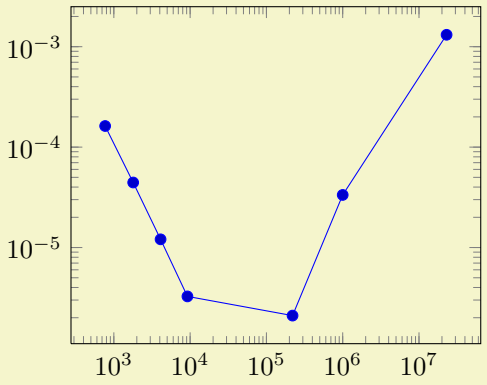
```
\pgfplotsset{every axis plot/.append style={...}}
```

to add options to all your plots - maybe to set line widths to `thick`. Furthermore, if you have more than one plot inside of an axis, you can also use

```
\pgfplotsset{every axis plot no 3/.append style={...}}
```

to modify options for the plot with number 3 only. The first plot in an axis has number 0.

- The ⟨*options*⟩ are remembered for the legend. They are available as '`current plot style`' as long as the path is not yet finished or in associated error bars.
- See Subsection 4.6 for a list of available markers and line styles.
- For log plots, PGFPLOTS will compute the natural logarithm $\log(\cdot)$ numerically using a floating point unit developed for this purpose[4]. For example, the following numbers are valid input to `\addplot`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}
\addplot coordinates {
    (769,   1.6227e-04)
    (1793,  4.4425e-05)
    (4097,  1.2071e-05)
    (9217,  3.2610e-06)
    (2.2e5, 2.1E-6)
    (1e6,   0.00003341)
    (2.3e7, 0.00131415)
};
\end{loglogaxis}
\end{tikzpicture}
```

  You can represent arbitrarily small or very large numbers as long as its logarithm can be represented as a TEX-length (up to about 16384). Of course, any coordinate $x \leq 0$ is not possible since the logarithm of a non-positive number is not defined. Such coordinates will be skipped automatically (using the initial configuration `unbounded coords=discard`).

- For normal (non–logarithmic) axes, PGFPLOTS applies floating point arithmetics to support large or small numbers like 0.00000001234 or $1.234 \cdot 10^{24}$. Its number range is much larger than TEX's native support for numbers. The relative precision is between 4 and 7 significant decimal digits for the mantissa.

  As soon as the axes limits are completely known, PGFPLOTS applies a transformation which maps these floating point numbers into TEX-precision using transformations

$$T_x(x) = 10^{s_x} \cdot x - a_x \text{ and } T_y(y) = 10^{s_y} \cdot y - a_y \text{ and (for 3D plots) } T_z(y) = 10^{s_z} \cdot z - a_z$$

  with properly chosen integers $s_x, s_y, s_z \in \mathbb{Z}$ and shifts $a_x, a_y, a_z \in \mathbb{R}$. Section 4.24 contains a description of `disabledatascaling` and provides more details about the transformation.

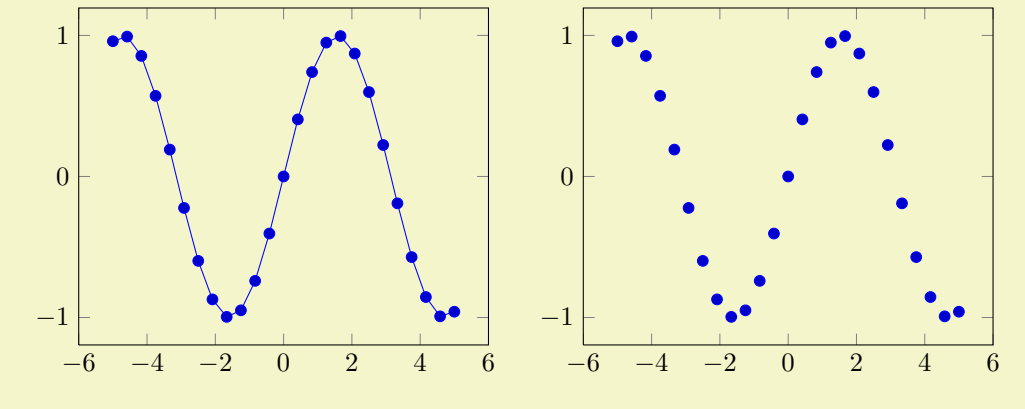
- Some of the coordinate input routines use the powerful `\pgfmathparse` feature of PGF to read their coordinates, among them `plot coordinates`, `plot expression` and `plot table`. This allows to use mathematical expressions as coordinates which will be evaluated using the floating point routines (this applies to logarithmic and linear scales).
- PGFPLOTS automatically computes missing axis limits. The automatic computation of axis limits works as follows:
  1. Every coordinate will be checked. Care has been taken to avoid TEX's limited numerical capabilities.
  2. Since more than one `\addplot` command may be used inside of `\begin{axis}...\end{axis}`, all drawing commands will be postponed until `\end{axis}`.

[4] This floating point unit is available as Ti*k*Z library as part of Ti*k*Z.

`\addplot+[⟨options⟩] ...;`

Does the same like `\addplot[⟨options⟩] ...;` except that ⟨*options*⟩ are *appended* to the arguments which would have been taken for `\addplot ...` (the element of the default list).

Thus, you can combine `cycle list` and ⟨*options*⟩.



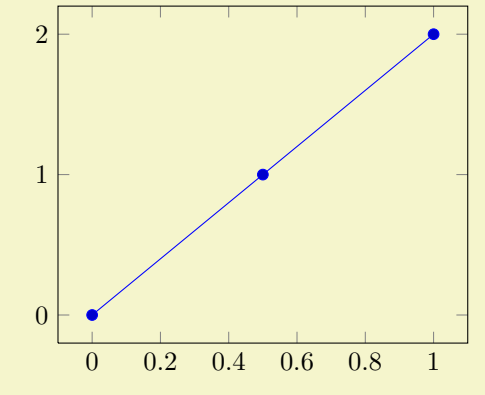

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot {sin(deg(x))};
\end{axis}
\end{tikzpicture}

\begin{tikzpicture}
\begin{axis}
\addplot+[only marks] {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```

The distinction is as follows: `\addplot ...` (without options) lets PGFPLOTS select colors, markers and linestyles automatically (using `cycle list`). The variant `\addplot+[⟨option⟩] ...` will use the same automatically determined styles, but in addition it uses ⟨*options*⟩. Finally, `\addplot[⟨options⟩]` (without the `+`) uses only the manually provided ⟨*options*⟩.

4.2.1 Coordinate Lists

`\addplot coordinates {⟨coordinate list⟩};`
`\addplot[⟨options⟩] coordinates {⟨coordinate list⟩} ⟨trailing path commands⟩;`
`\addplot3 ...`

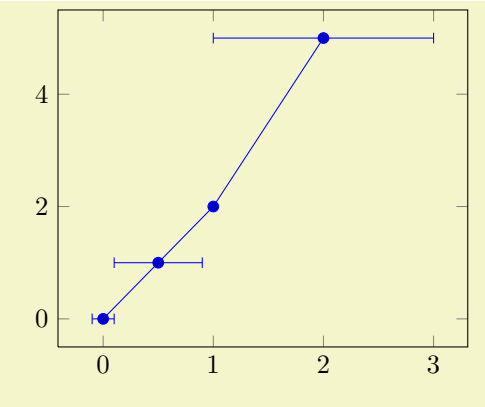
The '`plot coordinates`' command is like that provided by Ti*k*Z and reads its input data from a sequence of point coordinates, encapsulated in round braces.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot coordinates {
    (0,0)
    (0.5,1)
    (1,2)
};
\end{axis}
\end{tikzpicture}
```

The coordinates can be numbers, but they can also contain mathematical expressions like `sin(0.5)` or `\h*8` (assuming you defined `\h` somewhere). However, expressions which involve round braces need to be encapsulated in a further set of curly braces, for example `({sin(0.5)},{cos(0.1)})`.

You can also supply error coordinates (reliability bounds) if you are interested in error bars. Simply append the error coordinates with '`+-` (⟨*ex*,*ey*⟩)' (or `+-` (⟨*ex*,*ey*,*ez*⟩)) to the associated coordinate:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[error bars/.cd,x dir=both,x explicit]
coordinates {
    (0,0)   +- (0.1,0)
    (0.5,1) +- (0.4,0.2)
    (1,2)
    (2,5)   +- (1,0.1)
};
\end{axis}
\end{tikzpicture}
```

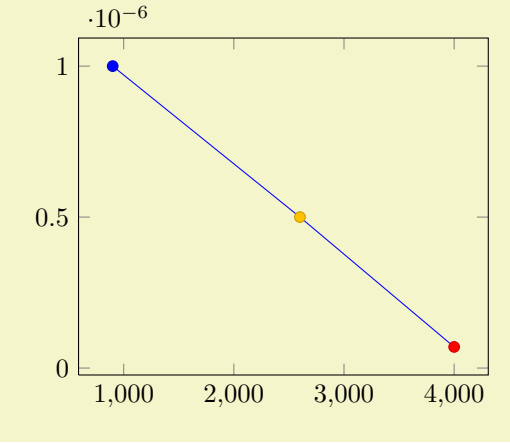
or

```
\addplot coordinates {
     (900,1e-6) +- (0.1,0.2)
    (2600,5e-7) +- (0.2,0.5)
    (4000,7e-8) +- (0.1,0.01)
};
```

These error coordinates are only used in case of error bars, see Section 4.11. You will also need to configure whether these values denote absolute or relative errors.

The coordinates as such can be numbers as `+5`, `-1.2345e3`, `35.0e2`, `0.00000123` or `1e2345e-8`. They are not limited to TEX's precision.

Furthermore, `coordinates` allows to define "meta data" for each coordinate. The interpretation of meta data depends on the visualization technique: for scatter plots, meta data can be used to define colors or style associations for every point (see page 77 for an example). Meta data (if any) must be provided after the coordinates and after error bar bounds (if any) in square brackets:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[scatter,scatter src=explicit] coordinates {
     (900,1e-6) [1]
    (2600,5e-7) [2]
    (4000,7e-8) [3]
};
\end{axis}
\end{tikzpicture}
```

Please refer to the documentation of `point meta` on page 137 for more information about per point meta data.

`/pgfplots/plot coordinates/math parser=true|false` (initially `true`)

Allows to turn off support for mathematical expressions in every coordinate inside of `plot coordinates`. This might be necessary if coordinates are not in numerical form (or if you'd like to improve speed).

It is necessary to disable `plot coordinates/math parser` if you use some sort of symbolic transformations (i.e. text coordinates).

4.2.2 Reading Coordinates From Files

`\addplot file {`⟨*name*⟩`};`

\addplot[⟨*options*⟩] file {⟨*name*⟩} ⟨*trailing path commands*⟩;
\addplot3 ...

PGFPLOTS supports two ways to read plot coordinates of external files, and one of them is similar to the Ti*k*Z-command '`plot file`'. It is to be used like

```
\addplot file {datafile.dat};
```

where ⟨*name*⟩ is a text file with at least two columns which will be used as x and y coordinates. Lines starting with '`%`' or '`#`' are ignored. Such files are often generated by GNUPLOT:

```
#Curve 0, 20 points
#x y type
0.00000 0.00000 i
0.52632 0.50235 i
1.05263 0.86873 i
1.57895 0.99997 i
...
9.47368 -0.04889 i
10.00000 -0.54402 i
```

This listing has been copied from [5, section 16.4].

Plot file accepts one optional argument,

```
\addplot file[skip first] {datafile.dat};
```

which allows to skip over a non-comment header line. This allows to read the same input files as `plot table` by skipping over column names. Please note that comment lines do not count as lines here.

The input method `plot file` can also read meta data for every coordinate. As already explained for `plot coordinates` (see above), meta data can be used to change colors or other style parameters for every marker separately. Now, if `point meta` is set to `explicit` or to `explicit symbolic` and the input method is `plot file`, one further element will be read from disk – for every line. Meta data is always the last element which is read. See page 75 for information and examples about per point meta data and page 77 for an application example using `scatter/classes`.

Plot file is very similar to `plot table`: you can achieve the same effect with

```
\addplot table[x index=0,y index=1,header=false] {datafile.dat};
```

Due to its simplicity, `plot file` is slightly faster while `plot table` allows higher flexibility.

Technical note: every opened file will be protocolled into your log file.

/pgfplots/plot file/skip first=true|false (initially `false`)
/pgfplots/plot file/ignore first=true|false (initially `false`)

The two keys can be provided as arguments to `\addplot file[`⟨*options*⟩`] {`⟨*filename*⟩`};` to skip the first non-comment entry in the file. They are equivalent. If you provide them in this context, the prefix `/pgfplots/plot file` can be omitted.

4.2.3 Reading Coordinates From Tables

\addplot table [⟨*column selection*⟩]{⟨*file*⟩};
\addplot[⟨*options*⟩] table [⟨*column selection*⟩]{⟨*file*⟩} ⟨*trailing path commands*⟩;
\addplot3 ...

The use of '`plot table`' is similar in spirit to '`plot file`', but its flexibility is higher. Given a data file like

```
dof     L2              Lmax            maxlevel
5       8.31160034e-02  1.80007647e-01  2
17      2.54685628e-02  3.75580565e-02  3
49      7.40715288e-03  1.49212716e-02  4
129     2.10192154e-03  4.23330523e-03  5
321     5.87352989e-04  1.30668515e-03  6
769     1.62269942e-04  3.88658098e-04  7
1793    4.44248889e-05  1.12651668e-04  8
4097    1.20714122e-05  3.20339285e-05  9
9217    3.26101452e-06  8.97617707e-06  10
```

one may want to plot ‘dof’ versus ‘L2’ or ‘dof’ versus ‘Lmax’. This can be done by

```
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel=Dof,
    ylabel=$L_2$ error]
\addplot table[x=dof,y=L2] {datafile.dat};
\end{loglogaxis}
\end{tikzpicture}
```

or, for the Lmax column, using

```
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel=Dof,
    ylabel=$L_\infty$ error]
\addplot table[x=dof,y=Lmax] {datafile.dat};
\end{loglogaxis}
\end{tikzpicture}
```

It is also possible to provide the data inline, i.e. directly as argument in curly braces:

```
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel=Dof,
    ylabel=$L_\infty$ error]
\addplot table[x=dof,y=Lmax] {
    dof     L2              Lmax            maxlevel
    5       8.31160034e-02  1.80007647e-01  2
    17      2.54685628e-02  3.75580565e-02  3
    49      7.40715288e-03  1.49212716e-02  4
    129     2.10192154e-03  4.23330523e-03  5
    321     5.87352989e-04  1.30668515e-03  6
    769     1.62269942e-04  3.88658098e-04  7
    1793    4.44248889e-05  1.12651668e-04  8
    4097    1.20714122e-05  3.20339285e-05  9
    9217    3.26101452e-06  8.97617707e-06  10
};
\end{loglogaxis}
\end{tikzpicture}
```

Inline table may be convenient together with ‘\\’ and `row sep=\\`, see below for more information.

Alternatively, you can load the table *once* into an internal structure and use it *multiple* times[5]:

```
\pgfplotstableread{datafile.dat}\loadedtable %  use any custom name in place of ‘\loadedtable’
...
\addplot table[x=dof,y=L2] {\loadedtable};
...
\addplot table[x=dof,y=Lmax] {\loadedtable};
...
```

I am not really sure how much time can be saved, but it works anyway. The `\pgfplotstableread` command is documented in all detail in the manual for PgfplotsTable. As a rule of thumb, decide as follows:

1. If tables contain few rows and many columns, the ⟨*macro*⟩ framework will be more efficient.
2. If tables contain more than 200 data points (rows), you should always use file input (and reload if necessary).

Occasionally, it might be handy to load a table, apply manual preparation steps (for example `\pgfplotstabletranspose`) and plot the result tables afterwards.

If you do prefer to access columns by column indices instead of column names (or your tables do not have column names), you can also use

```
\addplot table[x index=2,y index=3] {datafile.dat};
\addplot table[x=dof,y index=2] {datafile.dat};
```

[5] In earlier versions, there was an addition keyword ‘from’ before the argument like `\addplot table from {\loadedtable}`. This keyword is still accepted, but no longer required.

Summary and remarks:

- Use `\addplot table[x={`⟨*column name*⟩`},y={`⟨*column name*⟩`}]` to access column names. Those names are case sensitive and need to exist.
- Use `\addplot table[x index={`⟨*column index*⟩`},y index={`⟨*column index*⟩`}]` to access column indices. Indexing starts with 0. You may also use an index for x and a column name for y.
- Use `\addplot table[x expr=\coordindex,y={`⟨*column name*⟩`}]` to plot the coordinate index versus some y data.
- Use `\addplot table[header=false] {`⟨*file name*⟩`}` if your input file has no column names. Otherwise, the first non-comment line is checked for column names: if all entries are numbers, they are treated as numerical data; if one of them is not a number, all are treated as column names.
- It is possible to read error coordinates from tables as well. Simply add options '`x error`', '`y error`' or '`x error index`'/'`y error index`' to ⟨*source columns*⟩. See Section 4.11 for details about error bars.
- It is possible to read per point meta data (usable in `scatter src`, see page 75) as has been discussed for `plot coordinates` and `plot file` above. The meta data column can be provided using the `meta` key (or the `meta index` key).
- Use `\addplot table[`⟨*source columns*⟩`] {`⟨*\macro*⟩`}` to use a pre–read table. Tables can be read using

  ```
  \pgfplotstableread{datafile.dat}\macroname.
  ```

  If you like, you can insert the optional keyword '`from`' before `\macroname`.
- The accepted input format of tables is as follows:
  - Rows are separated by new line characters.
    Alternatively, you can use `row sep=\\` which enables '`\\`' as row separator. This might become necessary for inline table data, more precisely: if newline characters have been converted to white spaces by TEX's character processing before PGFPLOTS had a chance to see them. This happens if inline tables are provided inside of macros. Use `row sep=\\` and separate the rows by '`\\`' if you experience such problems.
  - Columns are usually separated by white spaces (at least one tab or space).
    If you need other column separation characters, you can use the
    col sep=`space`|`tab`|`comma`|`colon`|`semicolon`|`braces`|`&`|`ampersand`
    option documented in all detail in the manual for PGFPLOTSTABLE which is part of PGFPLOTS.
  - Any line starting with '#' or '%' is ignored.
  - The first line will be checked if it contains numerical data. If there is a column in the first line which is *no* number, the complete line is considered to be a header which contains column names. Otherwise it belongs to the numerical data and you need to access column indices instead of names.
  - There is future support for a second header line which must start with '`$flags `'. Currently, such a line is ignored. It may be used to provide number formatting hints like precision and number format if those tables shall be typeset using `\pgfplotstabletypeset` (see the manual for PGFPLOTSTABLE).
  - The accepted number format is the same as for '`plot coordinates`', see above.
  - If you omit column selectors, the default is to plot the first column against the second. That means `plot table` does exactly the same job as `plot file` for this case.
  - If you need unbalanced columns, simply use `nan` as "empty cell" placeholder. These coordinates will be skipped in plots.
- It is also possible to use **mathematical expressions** together with '`plot table`'. This is documented in all detail in Section 4.2.5, but the key idea is to use one of `x expr`, `y expr`, `z expr` or `meta expr` as in '`plot table[x expr=\thisrow{maxlevel}+3,y=L2]`'.
- The PGFPLOTSTABLE package coming with PGFPLOTS has a the feature "Postprocessing Data in New Columns" (see its manual).

  This allows to compute new columns based on existing data. One of these features is `create col/linear regression` (described in Section 4.23).

You can invoke all the `create col/`⟨*key name*⟩ features directly in `\addplot table` using `\addplot table[x={create col/`⟨*key name*⟩`=`⟨*arguments*⟩`}]`.

In this case, a new column will be created using the functionality of ⟨*key name*⟩. This column generation is described in all detail in PgfplotsTable. Finally, the resulting data is available as x coordinate (the same holds for `y=` or `z=`).

One application (with several examples how to use this syntax) is line fitting with `create col/linear regression`, see Section 4.23 for details.

- Technical note: every opened file will be protocolled into your log file.

Keys To Configure Table Input

The following list of keys allow different methods to select input data or different input formats. Note that the common prefix '`table/`' can be omitted if these keys are set after `\addplot table[`⟨*options*⟩`]`. The `/pgfplots/` prefix can always be omitted when used in a pgfplots method.

`/pgfplots/table/header=true|false` (initially `true`)

Allows to disable header identification for `plot table`. See above.

`/pgfplots/table/x={`⟨*column name*⟩`}`
`/pgfplots/table/y={`⟨*column name*⟩`}`
`/pgfplots/table/z={`⟨*column name*⟩`}`
`/pgfplots/table/x index={`⟨*column index*⟩`}`
`/pgfplots/table/y index={`⟨*column index*⟩`}`
`/pgfplots/table/z index={`⟨*column index*⟩`}`

These keys define the sources for `plot table`. If both column names and column indices are given, column names are preferred. Column indexing starts with 0. The initial setting is to use `x index=0` and `y index=1`.

Please note that column *aliases* will be considered if unknown column names are used. Please refer to the manual of PgfplotsTable which comes with this package.

`/pgfplots/table/x expr={`⟨*expression*⟩`}`
`/pgfplots/table/y expr={`⟨*expression*⟩`}`
`/pgfplots/table/z expr={`⟨*expression*⟩`}`
`/pgfplots/table/meta expr={`⟨*expression*⟩`}`

These keys allow to combine the mathematical expression parser with file input. They are listed here to complete the list of table keys, but they are described in all detail in Section 4.2.5.

The key idea is to provide an ⟨*expression*⟩ which depends on table data (possibly on all columns in one row). Only data within the same row can be used where columns are referenced with `\thisrow{`⟨*column name*⟩`}` or `\thisrowno{`⟨*column index*⟩`}`.

Please refer to Section 4.2.5 for details.

`/pgfplots/table/x error={`⟨*column name*⟩`}`
`/pgfplots/table/y error={`⟨*column name*⟩`}`
`/pgfplots/table/z error={`⟨*column name*⟩`}`
`/pgfplots/table/x error index={`⟨*column index*⟩`}`
`/pgfplots/table/y error index={`⟨*column index*⟩`}`
`/pgfplots/table/z error index={`⟨*column index*⟩`}`
`/pgfplots/table/x error expr={`⟨*math expression*⟩`}`
`/pgfplots/table/y error expr={`⟨*math expression*⟩`}`
`/pgfplots/table/z error expr={`⟨*math expression*⟩`}`

These keys define input sources for error bars with explicit error values.

The `x error` method provides an input column name (or alias), the `x error index` method provides input column *indices* and `x error expr` works just as `table/x expr`: it allows arbitrary mathematical expressions which may depend on any number of table columns using `\thisrow{`⟨*col name*⟩`}`.

Please see Section 4.11 for details about the usage of error bars.

`/pgfplots/table/meta={`⟨*column name*⟩`}`

/pgfplots/table/meta index={⟨*column index*⟩}

These keys define input sources for per point meta data. Please see page 75 for details about meta data or the documentation for `plot coordinates` and `plot file` for further information.

/pgfplots/table/row sep=newline|\\ (initially `newline`)

Configures the character to separate rows.

The choice `newline` uses the end of line as it appears in the table data (i.e. the input file or any inline table data).

The choice `\\` uses '\\' to indicate the end of a row.

Note that `newline` for inline table data is "fragile": you can't provide such data inside of TEX macros (this does not apply to input files). Whenever you experience problems, proceed as follows:

1. First possibility: call `\pgfplotstableread{`⟨*data*⟩`}\yourmacro` *outside* of any macro declaration.
2. Use `row sep=\\`.

The same applies if you experience problems with inline data and special `col sep` choices (like `col sep=tab`).

The reasons for such problems is that TEX scans the macro bodies and replaces newlines by white spaces. It does other substitutions of this sort as well, and these substitutions can't be undone (maybe not even found).

/pgfplots/table/col sep=space|tab|comma|semicolon|colon|braces|&|ampersand (initially `space`)

Allows to choose column separators for `plot table`. Please refer to the manual of PgfplotsTable which comes with this package for details about `col sep`.

/pgfplots/table/read completely={⟨*auto,true,false*⟩} (initially `auto`)

Allows to customize `\addplot table{`⟨*file name*⟩`}` such that it always reads the entire table into memory.

This key has just one purpose, namely to create postprocessing columns on-the-fly and to plot those columns afterwards. This "lazy evaluation" which creates missing columns on-the-fly is documented in the PgfplotsTable manual (in section "Postprocessing Data in New Columns").

The initial configuration `auto` checks whether one of the keys `table/x`, `table/y`, `table/z` or `table/meta` contains a `create on use` column. If so, it enables `read completely`, otherwise it prefers to load the file in the normal way.

Attention: Usually, `\addplot table` only picks required entries, requiring linear runtime complexity. As soon as `read completely` is activated, tables are loaded completely into memory. Due to datastructures issues ("macro append runtime"), the runtime complexity for `read completely` is $O(N^2)$ where N is the number of rows. Thus: use this feature only for "small" tables[6].

/pgfplots/table/ignore chars={⟨*comma-separated-list*⟩} (initially `empty`)

Allows to silently remove a set of single characters from input files. The characters are separated by commas. The documentation for this command, including cases like '`\%,\#,\ `' or binary character codes like '`\^^ff`' can be found in the manual for PgfplotsTable.

This setting applies to `\addplot file` as well.

/pgfplots/table/white space chars={⟨*comma-separated-list*⟩} (initially `empty`)

Allows to define a list of single characters which are actually treated like white spaces (in addition to tabs and spaces). Please refer to the manual of PgfplotsTable for details.

This setting applies to `\addplot file` as well.

/pgfplots/table/comment chars={⟨*comma-separated-list*⟩} (initially `empty`)

Allows to add one or more *additional* comment characters. Each of these characters has a similar effect as the `#` character, i.e. all following characters of that particular input line are skipped.

[6] This remark might be deprecated; many of the slow routines have been optimized in the meantime to have at least pseudo-linear runtime.

For example, `comment chars=!` uses '!' as additional comment character (which allows to parse Touchstone files).

Please refer to the manual of PgfplotsTable for details.

`/pgfplots/table/skip first n={`⟨*integer*⟩`}` (initially 0)

Allows to skip the first ⟨*integer*⟩ lines of an input file. The lines will not be processed.

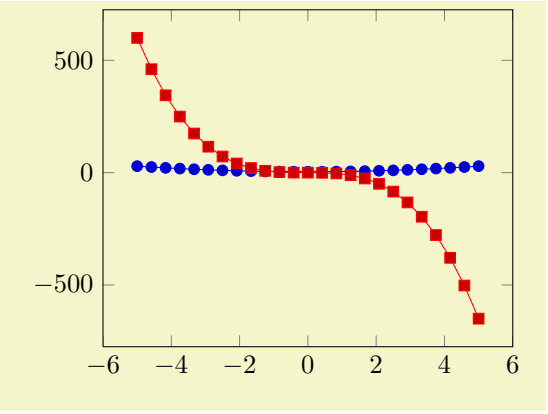
Please refer to the manual of PgfplotsTable for details.

4.2.4 Computing Coordinates with Mathematical Expressions

`\addplot {`⟨*math expression*⟩`} ;`
`\addplot[`⟨*options*⟩`] {`⟨*math expression*⟩`}` ⟨*trailing path commands*⟩`;`
`\addplot3` ...

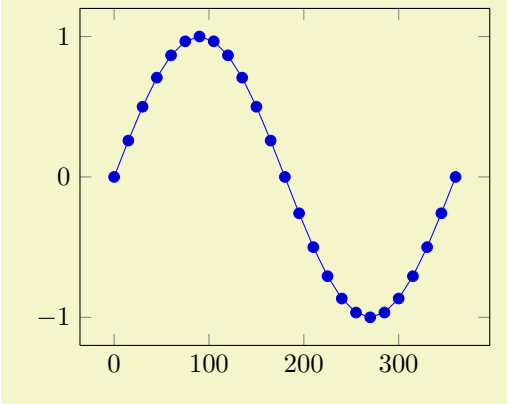
This input method allows to provide mathematical expressions which will be sampled. But unlike `plot gnuplot`, the expressions are evaluated using the math parser of pgf, no external program is required.

Plot expression samples `x` from the interval $[a, b]$ where a and b are specified with the `domain` key. The number of samples can be configured with `samples=`⟨N⟩ as for plot gnuplot.



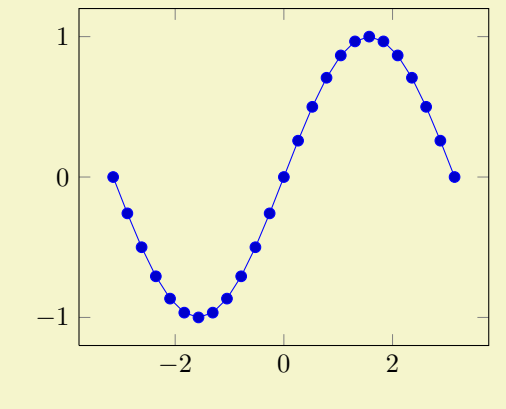

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot {x^2 + 4};
    \addplot {-5*x^3 - x^2};
\end{axis}
\end{tikzpicture}
```

Please note that pgf's math parser uses degrees for trigonometric functions:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot+[domain=0:360]
        {sin(x)};
\end{axis}
\end{tikzpicture}
```

If you want to use radians, use




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot+[domain=-pi:pi]
        {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```

to convert the radians to degrees. The plot expression parser also accepts some more options like `samples at={`⟨*coordinate list*⟩`}` or `domain=`⟨*first*⟩`:`⟨*last*⟩ which are described below.

Remarks

1. What really goes on is a loop which assigns the current sample coordinate to the macro `\x`. PGFPLOTS defines a math constant `x` which always has the same value as `\x`.

   In short: it is the same whether you write `\x` or just `x` inside of math expressions.

   The variable name can be customized using `variable=\t` (the backslash is necessary!). Then, `t` will be the same as `\t`.

2. The complete set of math expressions can be found in the PGF manual. The most important mathematical operations are `+`, `-`, `*`, `/`, `abs`, `round`, `floor`, `mod`, `<`, `>`, `max`, `min`, `sin`, `cos`, `tan`, `deg` (conversion from radians to degrees), `rad` (conversion from degrees to radians), `atan`, `asin`, `acos`, `cot`, `sec`, `cosec`, `exp`, `ln`, `sqrt`, the constants `pi` and `e`, `^` (power operation), `factorial`[7], `rand` (random between −1 and 1), `rnd` (random between 0 and 1), number format conversions `hex`, `Hex`, `oct`, `bin` and some more. The math parser has been written by Mark Wibrow and Till Tantau [5], the FPU routines have been developed as part of PGFPLOTS. The documentation for both parts can be found in [5].

   Please note, however, that trigonometric functions are defined in degrees. The character '`^`' is used for exponentiation (not '`**`' as in gnuplot).

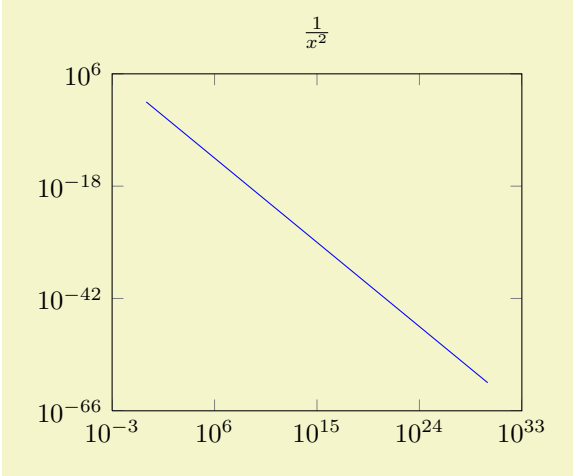
3. If the x axis is logarithmic, samples will be drawn logarithmically.

4. Please note that plot expression does not allow separate per point meta data (color data). You can, of course, use `point meta=f(x)` or `point meta=x`.

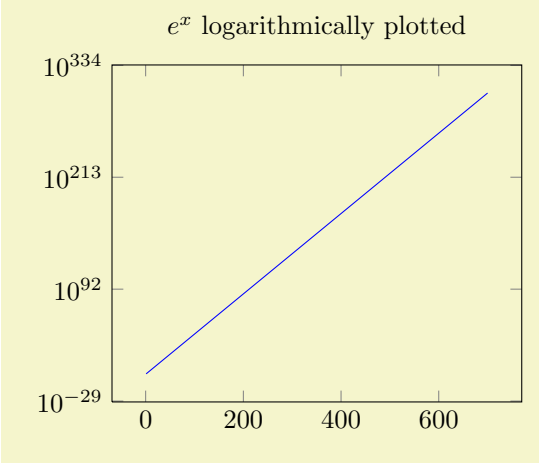
About the precision and number range: Starting with version 1.2, `plot expression` uses a floating point unit. The FPU provides the full data range of scientific computing with a relative precision between 10^{-4} and 10^{-6}. The `/pgf/fpu` key provides some more details.

In case the `fpu` does not provide the desired mathematical function or is too slow[8], you should consider using the `plot gnuplot` method which invokes the external, freely available program `gnuplot` as desktop calculator.

[7] Starting with PGF versions newer than 2.00, you can use the postfix operator `!` instead of `factorial`.
[8] Or in case you find a bug...




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[
        title={$\frac{1}{x^2}$}]
    \addplot[blue,domain=1:1e30]
        {x^-2};
    \end{loglogaxis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{semilogyaxis}[
        title={$e^x$ logarithmically plotted}]
    \addplot[blue,domain=1:700]
        {exp(x)};
    \end{semilogyaxis}
\end{tikzpicture}
```

`\addplot` **expression** {⟨*math expr*⟩};
`\addplot[`⟨*options*⟩`]` **expression** {⟨*math expr*⟩} ⟨*trailing path commands*⟩;
`\addplot3` ...

The syntax

`\addplot` {⟨*math expression*⟩};

as short-hand equivalent for

`\addplot` expression {⟨*math expression*⟩};

`\addplot` (⟨*x expression*⟩,⟨*y expression*⟩) ;
`\addplot[`⟨*options*⟩`]` (⟨*x expression*⟩,⟨*y expression*⟩) ⟨*trailing path commands*⟩;
`\addplot3` ...

A variant of `\addplot expression` which allows to provide different coordinate expressions for the x and y coordinates. This can be used to generate parametrized plots.

Please note that `\addplot (x,x^2)` is equivalent to `\addplot expression {x^2}`.

Note further that since the complete point expression is surrounded by round braces, round braces for either ⟨*x expression*⟩ or ⟨*y expression*⟩ need special attention. You will need to introduce curly braces additionally to allow round braces:

`\addplot ({`⟨*x expr*⟩`}, {`⟨*y expr*⟩`}, {`⟨*z expr*⟩`});`

/pgfplots/**domain**=⟨x_1⟩:⟨x_2⟩ (initially `[-5:5]`)
/pgfplots/**y domain**=⟨y_1⟩:⟨y_2⟩
/pgfplots/**domain y**=⟨y_1⟩:⟨y_2⟩

Sets the function's domain(s) for `plot expression` and `plot gnuplot`. Two dimensional plot expressions are defined as functions $f\colon [x_1, x_2] \to \mathbb{R}$ and ⟨x_1⟩ and ⟨x_2⟩ are set with `domain`. Three dimensional

plot expressions use functions $f\colon [x_1,x_2]\times[y_1,y_2]\to\mathbb{R}$ and $\langle y_1\rangle$ and $\langle y_2\rangle$ are set with `y domain`. If `y domain` is empty, $[y_1,y_2]=[x_1,x_2]$ is assumed for three dimensional plots (see page 89 for details about three dimensional plot expressions).

The keys `y domain` and `domain y` are the same.

The `domain` key won't be used if `samples at` is specified; `samples at` has higher precedence.

Please note that `domain` is not necessarily the same as the axis limits (which are configured with the `xmin`/`xmax` options).

The `domain` keys are *only* relevant for `gnuplot` and `plot expression`. In case you'd like to plot only a subset of other coordinate input routines, consider using the coordinate filter `restrict x to domain`.

Remark for Ti*k*Z-users: `/pgfplots/domain` and `/tikz/domain` are independent options. Please prefer the PGFPLOTS variant (i.e. provide `domain` to an axis, `\pgfplotsset` or a plot command). Since older versions also accepted something like `\begin{tikzpicture}[domain=...]`, this syntax is also accepted as long as no PGFPLOTS `domain` key is set.

`/pgfplots/samples=`{⟨*number*⟩} (initially 25)
`/pgfplots/samples y=`{⟨*number*⟩}

Sets the number of sample points for `plot expression` and `plot gnuplot`. The `samples` key defines the number of samples used for line plots while the `samples y` key is used for mesh plots (three dimensional visualisation, see page 89 for details). If `samples y` is not set explicitly, it uses the value of `samples`.

The `samples` key won't be used if `samples at` is specified; `samples at` has higher precedence.

The same special treatment of `/tikz/samples` and `/pgfplots/samples` as for the `domain` key applies here. See above for details.

`/pgfplots/samples at=`{⟨*coordinate list*⟩}

Sets the x coordinates for `plot expression` explicitly. This overrides `domain` and `samples`.

The ⟨*coordinate list*⟩ is a `\foreach` expression, that means it can contain a simple list of coordinates (comma–separated), but also complex `...` expressions like[9]

```
\pgfplotsset{samples at={5e-5,7e-5,10e-5,12e-5}}
\pgfplotsset{samples at={-5,-4.5,...,5}}
\pgfplotsset{samples at={-5,-3,-1,-0.5,0,...,5}}
```

The same special treatment of `/tikz/samples at` and `/pgfplots/samples at` as for the `domain` key applies here. See above for details.

Attention: `samples at` overrides `domain`, even if `domain` has been set *after* `samples at`! Use `samples at={}` to clear ⟨*coordinate list*⟩ and re-activate `domain`.

`/pgfplots/variable=`{⟨*variable name*⟩} (initially `x`)
`/pgfplots/variable y=`{⟨*variable name*⟩} (initially `y`)

Defines the variables names which will be sampled in `domain` (with `variable`) and in `domain y` (with `variable y`).

The same variables are used for parametric and for non-parametric plots. Use `variable=t` to change them if you like (for `gnuplot`, there is such a distinction; see `parametric/var 1d`).

Technical remark: Ti*k*Z also uses the `variable` key. However, it expects a *macro* name, i.e. `\x` instead of just `x`. Both possibilities are accepted here.

4.2.5 Mathematical Expressions And File Data

PGFPLOTS allows to combine '`plot table`' and '`plot expression`' to get both file input and modifications by means of mathematical expressions.

`\addplot table [`⟨*column selection and expressions*⟩`]{`⟨*file*⟩`};`

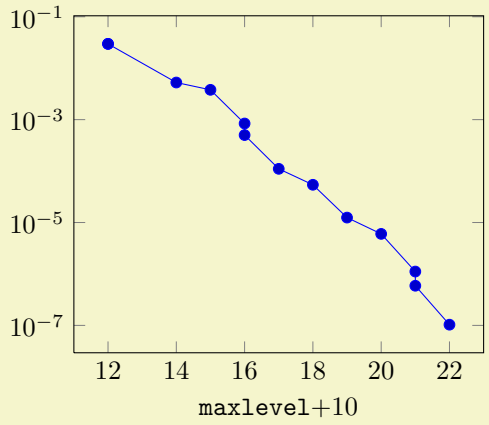
[9] Unfortunately, the `...` is somewhat restrictive when it comes to extended accuracy. So, if you have particularly small or large numbers (or a small distance), you have to provide a comma–separated list (or use the `domain` key).

`\addplot[`⟨*options*⟩`] table [`⟨*column selection and expressions*⟩`]{`⟨*file*⟩`}` ⟨*trailing path commands*⟩`;`
`\addplot3` ...

Besides the already discussed possibility to provide a column selection by means of column names (`x=`⟨*name*⟩ or `x index=`⟨*index*⟩, see Section 4.2.3), it is also possible to provide mathematical expressions as arguments.

Mathematical expressions are specified with `x expr=`⟨*expression*⟩ inside of ⟨*column selection and expressions*⟩. They can depend on zero, one or more columns of the input file. A column is referenced using the special command '`\thisrow{`⟨*column name*⟩`}`' within ⟨*expression*⟩ (or `\thisrowno`⟨*column index*⟩).

maxlevel	L2
2	$2.97 \cdot 10^{-2}$
2	$2.97 \cdot 10^{-2}$
4	$5.27 \cdot 10^{-3}$
5	$3.8 \cdot 10^{-3}$
6	$8.41 \cdot 10^{-4}$
6	$5.01 \cdot 10^{-4}$
7	$1.11 \cdot 10^{-4}$
8	$5.41 \cdot 10^{-5}$
9	$1.25 \cdot 10^{-5}$
10	$6.01 \cdot 10^{-6}$
11	$1.11 \cdot 10^{-6}$
11	$5.9 \cdot 10^{-7}$
12	$1.03 \cdot 10^{-7}$




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotstabletypeset[columns={maxlevel,L2}]{plotdata/newexperiment1.dat}

\begin{tikzpicture}
    \begin{semilogyaxis}[
        xlabel=\texttt{maxlevel}$ + 10$
    ]
    \addplot table
        [x expr=\thisrow{maxlevel}+10, y=L2]
        {plotdata/newexperiment1.dat};
    \end{semilogyaxis}
\end{tikzpicture}
```

Besides `x expr`, there are keys `y expr`, `z expr` and `meta expr` where the latter allows to provide point meta data (which is used as `scatter src` or color data for surface plots etc.).

Inside of ⟨*expression*⟩, the following macros can be used to access numerical data cells inside of the input file:

`\thisrow{`⟨*column name*⟩`}`

Yields the value of the column designated by ⟨*column name*⟩. There is no limit on the number of columns which can be part of a mathematical expression, but only values inside of the currently processed *table row* can be used.

It is possible to provide column aliases for ⟨*column name*⟩ as described in the manual of PgfplotsTable.

The argument ⟨*column name*⟩ has to denote either an existing column or one for which a column alias exists (see the manual of PgfplotsTable). If it can't be resolved, the math parser yields an "Unknown function" error message.

`\thisrowno{`⟨*column index*⟩`}`

Similar to `\thisrow`, this command yields the value of the column with index ⟨*column index*⟩ (starting with 0).

`\coordindex`

Yields the current index of the table row (starting with 0). This does *not* count header or comment lines.

`\lineno`

Yields the current line number (starting with 0). This does also count header and comment lines.

If `x index`, `x` and `x expr` (or the corresponding keys for `y`, `z` or `meta`) are combined, this is how they interact:

1. Column access via `x` has higher precedence than index access via `x index`.
2. Even if `x expr` is provided, the values of `x index` and `x` are still checked. Any value found using column name access or column index access is made available as `\columnx` (or `\columny`, `\columnz`, `\columnmeta`, resp.). However, the result of `x expr` is used as plot coordinate.

   This allows to access the cell values identified by `x` or `x index` using the "pointer" `\columnx`. I am not sure if this yields any advantage, but it is possible nevertheless. If in doubt, prefer using `\thisrow{`⟨*column name*⟩`}`.

Attention: If your table has less than two rows, you may need to set `x index={},y index={}` explicitly. This is a consequence of the fact that column name/index access is still applied even if an expression is provided.

4.2.6 Computing Coordinates with Mathematical Expressions (gnuplot)

`\addplot gnuplot [`⟨*further options*⟩`]{`⟨*gnuplot code*⟩`};`
`\addplot[`⟨*options*⟩`] gnuplot [`⟨*further options*⟩`]{`⟨*gnuplot code*⟩`}` ⟨*trailing path commands*⟩`;`
`\addplot3` ...

In contrast to `plot expression`, the `plot gnuplot` command[10] employs the external program `gnuplot` to compute coordinates. The resulting coordinates are written to a text file which will be plotted with `plot file`. PGF checks whether coordinates need to be re-generated and calls `gnuplot` whenever necessary (this is usually the case if you change the number of samples, the argument to `plot gnuplot` or the plotted domain[11]).

The differences between `plot expression` and `plot gnuplot` are:

- `plot expression` does not require any external programs and requires no additional command line options.
- `plot expression` does not produce a lot of temporary files.
- `plot gnuplot` uses radians for trigonometric functions while `plot expression` has degrees.
- `plot gnuplot` is faster.
- `plot gnuplot` has a larger mathematical library.
- `plot gnuplot` has a higher accuracy. However, starting with version 1.2, this is no longer a great problem. The new floating point unit for TEX provides reasonable accuracy and the same data range as `gnuplot`.

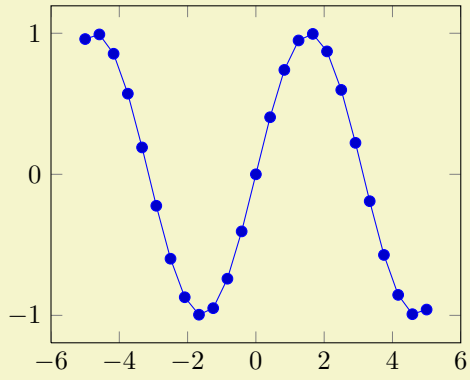
[10] Note that `plot gnuplot` is actually a re-implementation of the `plotfunction` method known from PGF. It also invokes PGF basic layer commands.

[11] Please note that PGFPLOTS produces slightly different files than Ti*k*Z when used with `plot gnuplot` (it configures high precision output). You should use different `id` for PGFPLOTS and Ti*k*Z to avoid conflicts in such a case.

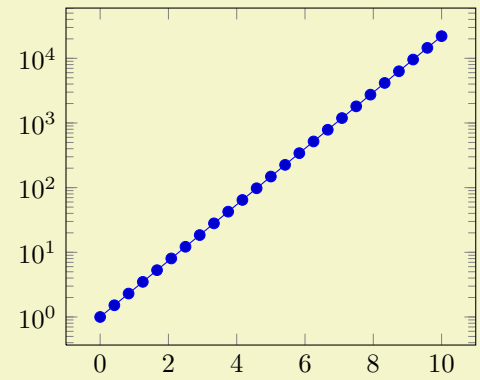
Since system calls are a potential danger, they need to be enabled explicitly using command line options, for example

```
pdflatex -shell-escape filename.tex.
```

Sometimes it is called `shell-escape` or `enable-write18`. Sometimes one needs two hyphens – that all depends on your TEX distribution.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot
    gnuplot[id=sin]{sin(x)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}
\addplot gnuplot
    [id=exp,domain=0:10]{exp(x)};
\end{semilogyaxis}
\end{tikzpicture}
```

The ⟨*options*⟩ determine the appearance of the plotted function; these parameters also affect the legend. There is also a set of options which are specific to the gnuplot interface. These options are described in all detail in [5, section 18.6]. A short summary is shown below.

Some remarks:

- The independent variable for one-dimensional plots can be changed with the `variable` option, just as for `plot expression`. Similarly, the second variable for two dimensional plots can be changed with `variable y`.

  For `parametric` plots, the variable names need to be adjusted with `parametric/var 1d` and `parametric/var 2d` (since gnuplot uses `t` and `u,v` as initial values for `parametric` plots).
- Please note that `plot gnuplot` does not allow separate per point meta data (color data for each coordinate). You can, however, use `point meta=f(x)` or `point meta=x`.
- The generated output file name can be customized with `id`, see below.

Please refer to [5, section 18.6] for more details about `plot function` and the `gnuplot` interaction.

`\addplot function {`⟨*gnuplot code*⟩`};`
`\addplot[`⟨*options*⟩`] function {`⟨*gnuplot code*⟩`}` ⟨*trailing path commands*⟩`;`
`\addplot3` ...

Use

`\addplot function {`⟨*gnuplot code*⟩`};`

as alias for

`\addplot gnuplot {`⟨*gnuplot code*⟩`};`

/pgfplots/translate gnuplot=true|false (initially true)

Enables or disables automatic translation of the exponentiation operator '^' to '**'.

This features allows to use ^ in plot gnuplot instead of gnuplot's **.

/pgfplots/parametric=true|false (initially false)

Set this to true if you'd like to use parametric plots with gnuplot. Parametric plots use a comma separated list of expressions to make up $x(t)$, $y(t)$ for a line plot or $x(u,v)$, $y(u,v)$ $z(u,v)$ for a mesh plot (refer to the gnuplot manual for more information about its input methods for parametric plots).

/pgfplots/parametric/var 1d={⟨*variable name*⟩} (initially t)
/pgfplots/parametric/var 2d={⟨*variable name,variable name*⟩} (initially u,v)

Allows to change the dummy variables used by parametric gnuplot plots. The initial setting is the one of gnuplot: to use the dummy varialbe 't' for parametric line plots and 'u,v' for parametric mesh plots.

These keys are quite the same as variable and variable y, only for parametric plots. If you like to change variables for non-parametric plots, use variable and/or variable y.

In case you don't want the distinction between parametric and non-parametric plots, use

\pgfplotsset{parametric/var 1d=,parametric/var 2d=}.

/tikz/id={⟨*unique string identifier*⟩}

A unique identifier for the current plot. It is used to generate temporary filenames for gnuplot output.

/tikz/prefix={⟨*file name prefix*⟩}

A common path prefix for temporary filenames (see [5, section 18.6] for details).

/tikz/raw gnuplot (no value)

Disables the use of samples and domain.

4.2.7 Computing Coordinates with External Programs (shell)

\addplot shell [⟨*further options*⟩]{⟨*shell commands*⟩};
\addplot[⟨*options*⟩] shell [⟨*further options*⟩]{⟨*shell commands*⟩} ⟨*trailing path commands*⟩;
\addplot3 ...

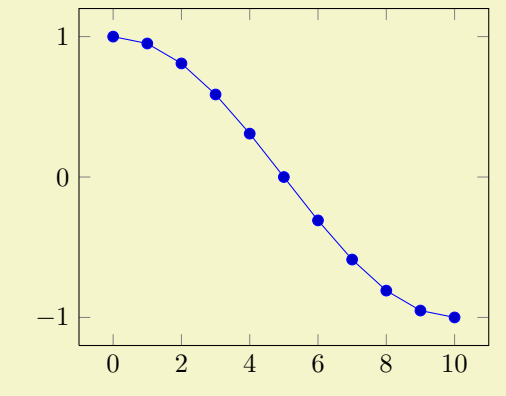
An extension by Stefan Tibus

In contrast to plot gnuplot, the plot shell command allows execution of arbitrary shell commands to compute coordinates. The resulting coordinates are written to a text file which will be plotted with plot file. PGF checks whether coordinates need to be re-generated and executes the ⟨*shell commands*⟩ whenever necessary.

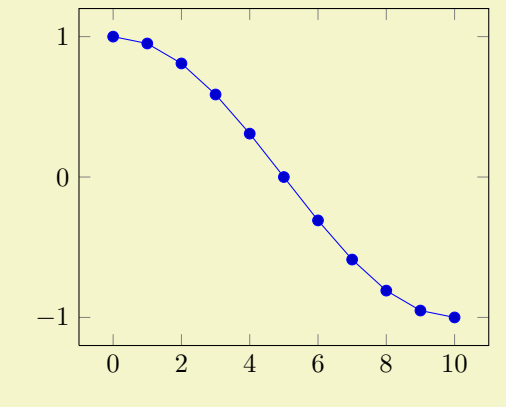
Since system calls are a potential danger, they need to be enabled explicitly using command line options, for example

```
pdflatex -shell-escape filename.tex.
```

Sometimes it is called shell-escape or enable-write18. Sometimes one needs two slashes – that all depends on your TEX distribution.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot
    shell[prefix=pgfshell_,id=cos]{awk 'BEGIN{
        pi=3.14159; N=10;
        for(i=0;i<=N;i++) print i,cos(i/N*pi);}'};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[prefix=pgfshell_,id=replot]
    shell{cat pgfshell_cos.out};
    %  just reprint the result from above
\end{axis}
\end{tikzpicture}
```

The ⟨*options*⟩ determine the appearance of the plotted function; these parameters also affect the legend. There is also a set of options which are specific to the gnuplot and the shell interface. These options are described in all detail in [5, section 19.6]. A short summary is shown below.

/tikz/**id**={⟨*unique string identifier*⟩}

A unique identifier for the current plot. It is used to generate temporary filenames for `shell` output.

/tikz/**prefix**={⟨*file name prefix*⟩}

A common path prefix for temporary filenames (see [5, section 19.6] for details).

4.2.8 Using External Graphics as Plot Sources

\addplot **graphics** {⟨*file name*⟩};
\addplot[⟨*options*⟩] **graphics** {⟨*file name*⟩} ⟨*trailing path commands*⟩;
\addplot3 ...

This plot type allows to extend the plotting capabilities of PGFPLOTS beyond its own limitations. The idea is to generate the graphics as such (for example, a contour plot, a complicated shaded surface[12] or a large point cluster) with an external program like Matlab (®) or `gnuplot`. The graphics, however, should *not* contain an axis or descriptions. Then, we use `\includegraphics` and a PGFPLOTS axis which fits exactly on top of the imported graphics.

Of course, one could do this manually by providing proper scales and such. The operation `plot graphics` is intended so simplify this process. However the *main difficulty* is to get images with correct bounding box. Typically, you will have to adjust bounding boxes manually.

Let's start with an example: Suppose we use, for example, matlab to generate a surface plot like

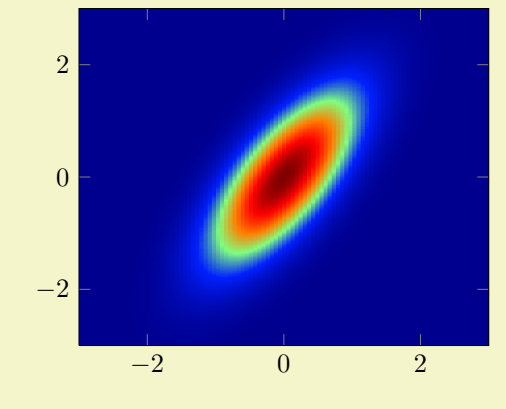
```
[X,Y] = meshgrid( linspace(-3,3,500) );
surf( X,Y, exp(-(X - Y).^2 - X.^2 ) );
shading flat; view(0,90); axis off;
print -dpng external1
```

which is then found in `external1.png`. The `surf` command of Matlab generates the surface, the following commands disable the axis descriptions, initialise the desired view and export it. Viewing the image in any image tool, we see a lot of white space around the surface – Matlab has a particular weakness in producing tight bounding boxes, as far as I know. Well, no problem: use your favorite image editor and crop the image (most image editors can do this automatically). We could use the free ImageMagick command

```
convert -trim external1.png external1.png
```

to get a tight bounding box. Then, we use

[12] See also Section 4.5.6 for an overview of PGFPLOTS methods to draw shaded surfaces.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[enlargelimits=false,axis on top]
        \addplot graphics
            [xmin=-3,xmax=3,ymin=-3,ymax=3]
            {external1};
    \end{axis}
\end{tikzpicture}
```

to load the graphics[13] just as if we would have drawn it with PGFPLOTS. The `axis on top` simply tells PGFPLOTS to draw the axis on top of any plots (see its description).

Please note that PGFPLOTS offers support for smaller surface plots as well which might be an option – unless the number of samples is too large. See Section 4.5.6 for details.

However, external programs have the following advantages here: they are faster, allow more complexity and provide real z buffering which is currently only simulated by PGFPLOTS. Thus, it may help to consider `plot graphics` for complicated surface plots.

Our first test was successful – and not difficult at all because graphics programs can automatically compute the bounding box. There are a couple of free tools available which can compute tight bounding boxes for `.eps` or `.pdf` graphics:

1. The free vector graphics program `inkscape` can help here. Its feature "File ≫ Document Properties: Fit page to selection" computes a tight bounding box around every picture element.

   However, some images may contain a rectangular path which is as large as the bounding box (Matlab (®) computes such `.eps` images). In this case, use the "Ungroup" method (context menu of `inkscape`) as often as necessary and remove such a path.

   Finally, save as `.eps`.

   However, `inkscape` appears to have problems with postscript fonts – it substitutes them. This doesn't pose problems in this application because fonts shouldn't be part of such images – the descriptions will be drawn by PGFPLOTS.

2. The tool `pdfcrop` removes surrounding whitespace in `.pdf` images and produces quite good bounding boxes.

Adjusting bounding boxes manually In case you don't have tools at hand to provide correct bounding boxes, you can still use TEX to set the bounding box manually. Some viewers like `gv` provide access to low–level image coordinates. The idea is to determine the number of units which need to be removed and communicate these units to `\includegraphics`.

I am aware of the following methods to determine bounding boxes manually:

inkscape I am pretty sure that `inkscape` can do it.

gv The ghost script viewer `gv` always shows the postscript units under the mouse cursor.

gimp The graphics program `gimp` usually shows the cursor position in pixels, but it can be configured to display postscript points (`pt`) instead.

Let's follow this approach in a further example.

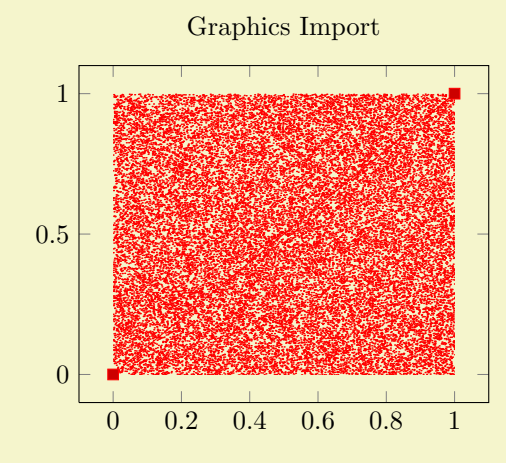
We use `gnuplot` to draw a (relatively stupid) example data set. The gnuplot script

[13] Please note that I don't have a Matlab license, so I used `gnuplot` to produce an equivalent replacement graphics.

```
set samples 30000
set parametric
unset border
unset xtics
unset ytics
set output "external2.eps"
set terminal postscript eps color
plot [t=0:1] rand(0),rand(0) with dots notitle lw 5
```

generates `external2.eps` with a uniform random sample of size 30000. As before, we import this scatter plot into PGFPLOTS using `plot graphics`. Again, the bounding box is too large, so we need to adjust it (`gnuplot` can do this automatically, but we do it anyway to explain the mechanisms):

Using `gv`, I determined that the bounding box needs to be shifted `12` units to the left and `9` down. Furthermore, the right end is `12` units too far off and the top area has about `8` units space wasted. This can be provided to the `trim` option of `\includegraphics`, and we use `clip` to clip the rest away:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[axis on top,title=Graphics Import]
        \addplot graphics
            [xmin=0,xmax=1,ymin=0,ymax=1,
            %  trim=left bottom right top
            includegraphics={trim=12 9 12 8,clip}]
            {external2};
        \addplot coordinates {(0,0) (1,1)};
    \end{axis}
\end{tikzpicture}
```

So, `plot graphics` takes a graphics file along with options which can be passed to `\includegraphics`. Furthermore, it provides the information how to embed the graphics into an axis. The axis can contain any other `\addplot` command as well and will be resized properly.

Details about `plot graphics`: The loaded graphics file is drawn with

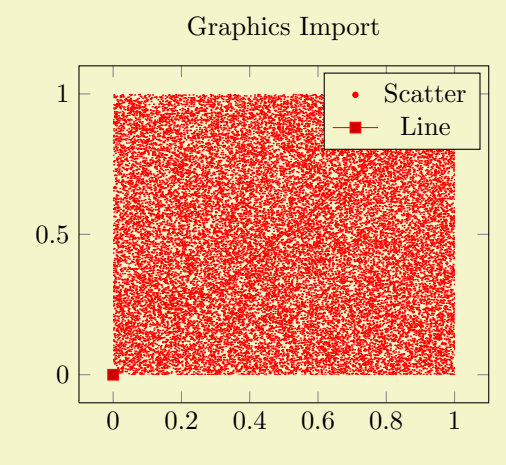
`\node[/pgfplots/plot graphics/node] {\includegraphics[`⟨*options*⟩`]{`⟨*file name*⟩`}};`

where the `node` style is a configurable style. The node is placed at the coordinate designated by `xmin`, `ymin`.

The ⟨*options*⟩ are any arguments provided to the `includegraphics` key (see below) and `width` and `height` determined such that the graphics fits exactly into the rectangle denoted by the `xmin`, `ymin` and `xmax`, `ymax` coordinates.

The scaling will thus ignore the aspect ratio of the external image and prefer the one used by PGFPLOTS. You will need to provide `width` and `height` to the PGFPLOTS axis to change its scaling. Use the `scale only axis` key in such a case.

Legends in `plot graphics`: A legend for `plot graphics` uses the current plot handler and the current plot `mark`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[axis on top,title=Graphics Import]
    %  provide options for the legend:
    \addplot[red,only marks,mark=*,mark size=1pt]
    graphics
        [xmin=0,xmax=1,ymin=0,ymax=1,
        %  trim=left bottom right top
        includegraphics={trim=12 9 12 8,clip}]
        {external2};

    \addplot coordinates {(0,0) (1,1)};

    \legend{Scatter,Line}
\end{axis}
\end{tikzpicture}
```

Keys To Configure Plot Graphics

The following list of keys configure `\addplot graphics`. Note that the common prefix '`plot graphics/`' can be omitted if these keys are set after `\addplot graphics[`⟨*options*⟩`]`. The `/pgfplots/` prefix can always be omitted when used in a PGFPLOTS method.

`/pgfplots/plot graphics/xmin=`{⟨*coordinate*⟩}
`/pgfplots/plot graphics/ymin=`{⟨*coordinate*⟩}
`/pgfplots/plot graphics/zmin=`{⟨*coordinate*⟩}
`/pgfplots/plot graphics/xmax=`{⟨*coordinate*⟩}
`/pgfplots/plot graphics/ymax=`{⟨*coordinate*⟩}
`/pgfplots/plot graphics/zmax=`{⟨*coordinate*⟩}

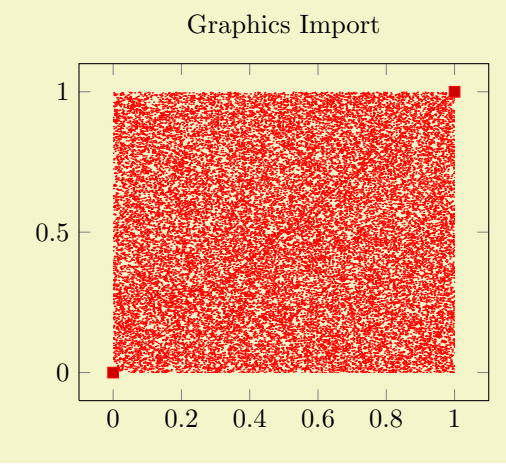
These keys are required for `plot graphics` and provide information about the external data range. The graphics will be squeezed between these coordinates. The arguments are axis coordinates; they are only useful if you provide each of them.

Alternatively, you can also use the `plot graphics/points` feature to provide the external data range, see below.

`/pgfplots/plot graphics/points=`{⟨*list of coordinates*⟩} (initially `empty`)

This key also allows to provide the external data range. It constitutes an alternative to `plot graphics/xmin` (and its variants): simply provide at least two coordinates in ⟨*list of coordinates*⟩. Their bounding box is used to determine the external data range, and the graphics is squeezed between these coordinates.

The example from above can be written equivalently as




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[axis on top,title=Graphics Import]
        \addplot graphics
            %  instead of the min/max things:
            [points={(0,1) (1,0)},
            %  trim=left bottom right top
            includegraphics={trim=12 9 12 8,clip}]
            {external2};
        \addplot coordinates {(0,0) (1,1)};
    \end{axis}
\end{tikzpicture}
```

The ⟨*list of coordinates*⟩ is a sequence of the form `(x,y)` for two–dimensional plots and `(x,y,z)` for three–dimensional ones, the ordering is irrelevant. The single elements are separated by white space.

It is possible to mix plot graphics/xmin and variants with plot graphics/points.

The plot graphics/points key has further functionality for inclusion of three–dimensional graphics which is discussed at the end of this section (on page 43). Here is a short reference on the accepted syntax for three–dimensional plot graphics: in addition to the (x,y,z) syntax, you can provide arguments of the form (x,y,z) => (X,Y). Here, the first (three–dimensional) coordinate is a logical coordinate and the second (two–dimensional) coordinate denotes the coordinates of the very same point, but inside of the included image (relative to the lower left corner of the image). Applications and examples for this syntax can be found in the section for three–dimensional plot graphics (see page 43).

/pgfplots/plot graphics/includegraphics={⟨*options*⟩}

A list of options which will be passed as–is to \includegraphics. Interesting options include the trim=⟨*left*⟩ ⟨*bottom*⟩ ⟨*right*⟩ ⟨*top*⟩ key which reduces the bounding box and clip which discards everything outside of the bounding box. The scaling options won't have any effect, they will be overwritten by PGFPLOTS.

/pgfplots/plot graphics/includegraphics cmd={⟨*macro*⟩} (initially \includegraphics)

Allows to use a different graphics routine. A possible choice could be \pgfimage. The macro should accept the width and height arguments (in brackets) and the file name as first argument.

/pgfplots/plot graphics/node (style, no value)

A predefined style used for the Ti*k*Z node containing the graphics. The predefined value is

```
\pgfplotsset{
    plot graphics/node/.style={
        transform shape,
        inner sep=0pt,
        outer sep=0pt,
        every node/.style={},
        anchor=south west,
        at={(0pt,0pt)},
        rectangle
    }
}
```

/pgfplots/plot graphics (no value)

This key belongs to the public low–level plotting interface. You won't need it in most cases.

This key is similar to sharp plot or smooth or const plot: it installs a low–level plot–handler which expects exactly two points: the lower left corner and the upper right one. The graphics will be drawn between them. The graphics file name is expected as value of the /pgfplots/plot graphics/src key. The other keys described above need to be set correctly (excluding the limits, these are ignored at this level of abstraction). This key can be used independently of an axis.

/pgfplots/plot graphics/lowlevel draw={⟨*width*⟩}{⟨*height*⟩}

A low–level interface for plot graphics which actually invokes \includegraphics. But there is no magic involved: the command is simply expected to draw a box of dimensions ⟨*width*⟩ × ⟨*height*⟩. The coordinate system has already been shifted correctly.

The initial configuration is

\includegraphics[⟨*value of* "plot graphics/includegraphics"⟩,width=#1,height=#2]
{⟨*value of* "plot graphics/src"⟩}.

Thus, you can tweak plot graphics to place any TEX box of the desired dimensions into an axis between the provided minimum and maximum coordinates. It is not necessary to make use of the graphics file name or the options in the 'includegraphics' key if you overwrite this lowlevel interface with

plot graphics/lowlevel draw/.code 2 args={⟨*code which depends on #1 and #2*⟩}.

Support for External Three-Dimensional Graphics

PGFPLOTS offers several visualization techniques for three dimensional graphics. Nevertheless, complex visualizations or specialized applications are beyond the scope of PGFPLOTS and you might want to use other tools to generate such figures.

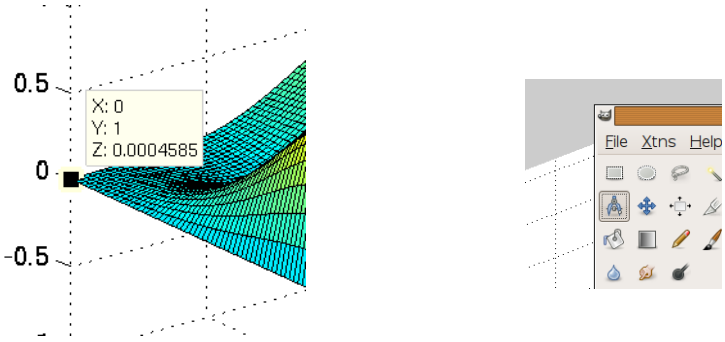


Figure 1: Using Matlab to extract image coordinates (left) and Gimp to measure distances (right).

The `plot graphics` tool of PGFPLOTS allows to include three–dimensional external graphics: it generates a three–dimensional axis on its own. The idea is to provide a graphics (without descriptions) and use PGFPLOTS to overlay a three–dimensional axis automatically. This allows to maintain document consistency (making it unnecessary to use different programs within the same document).

You are probably guessing how this is possible. Well, it needs more user input than two–dimensional external graphics. The cost to include external three dimensional images into PGFPLOTS is essentially control of a graphics program like `gimp`: you need to identify the 3D coordinates of a couple of points in your image. PGFPLOTS will then squeeze the graphics correctly, and it reconfigures the axis to ensure a correct display of the result.

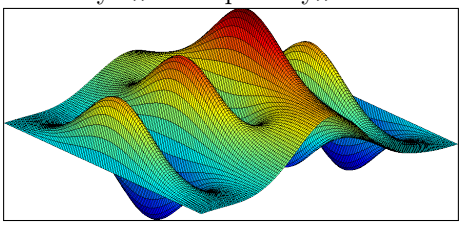
Warning: The feature is only 95% stable yet and may need some more testing. Use at your own risk.

Let's start with two examples. Suppose you generate a surface plot with Matlab and want to include it in PGFPLOTS. We have the matlab script

```
[x,y]=meshgrid(linspace(0,1,120));
surf(x,y,sin(8*pi*x).* exp(-20*(y-0.5).^2) + exp(-(x-0.5).^2*30 - (y-0.25).^2 - (x-0.5).*(y-0.25)))
xlabel('x'), ylabel('y')
axis off
print -dpng plotgraphics3dsurf
```

which generates the figure in question.

After automatically computing a tight bounding box for `plotgraphics3dsurf.png` (I used `gimp`'s Image≫Autocrop feature), and making the background color transparent (`gimp`: select the outer white space with the magic wand, then use[14] Layer≫Transparency≫Color to Transparency) we get:

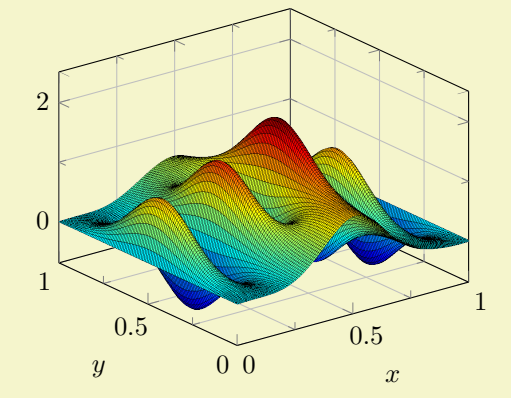


The key idea is now to identify several points in the image, and assign *both* their logical three–dimensional coordinates *and* the corresponding two–dimensional canvas coordinates in image coordinates. How? Well, the three–dimensional coordinates are known to Matlab, it can display them for you if you click somewhere into the image, compare Figure 1 (left).

The two–dimensional canvas coordinates need work; they need to be provided relative to the *lower left corner* of the image. I used `gimp` and activated "Points" as units (lower left corner). The lower left corner now displays the image coordinates in `pt` which is compatible with PGFPLOTS. An alternative to pointing onto coordinates is a measurement tool; compare Figure 1 (right) for the "Measure" tool in `gimp` which allows to compute the length of a line (in our case, the length of the lower left corner to the point of interest).

I selected four points in the graphics and noted their 2d image coordinates and their 3d logical coordinates as follows:

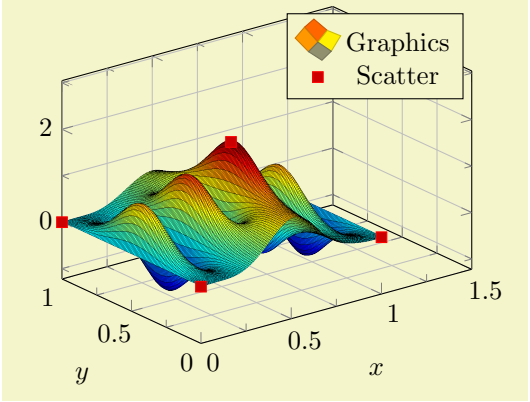
[14] I have a german version, I am not sure if the translation is correct.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    grid=both,minor tick num=1,
    xlabel=$x$,ylabel=$y$,
    ]
\addplot3 graphics[
    points={%  important
        (0,1,0) => (0,207-112)
        (1,0,0) => (446,207-133)
        (0.5546,0.5042,1.825) => (236,207)
        (0,0,0) => (194,207-202)
    }] {plotdata/plotgraphics3dsurf.png};
\end{axis}
\end{tikzpicture}
```

Here, the `points` key gets our collected coordinates as argument. It accepts a sequence of maps of the form ⟨*3d logical coordinate*⟩ `=>` ⟨*2d canvas coordinate*⟩. In our case, `(0,1,0)` has been found in the `.png` file at `(0,207-112)`. Note that I introduced the difference since `gimp` counts from the upper left, but PGFPLOTS counts from the lower left.

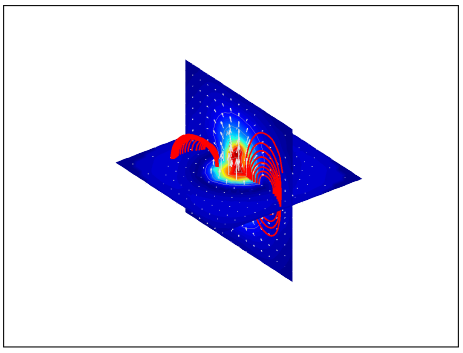
Once these four point coordinates are gathered, we find Matlab's surface plot in a PGFPLOTS axis. You can modify any appearance options, including different axis limits or further `\addplot` commands:



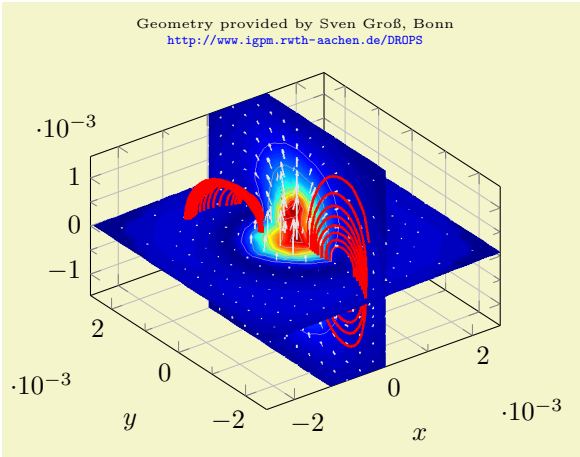

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xmax=1.5,%  extra limits
    grid=both,minor tick num=1,
    xlabel=$x$,ylabel=$y$,
    ]
\addplot3[surf] %  'surf' is only used for the legend.
    graphics[
        points={
            (0,1,0) => (0,207-112)
            (1,0,0) => (446,207-133)
            (0.5546,0.5042,1.825) => (236,207)
            (0,0,0) => (194,207-202)
    }]
    {plotdata/plotgraphics3dsurf.png};
\addlegendentry{Graphics}

\addplot3+[only marks] coordinates {
    (0,1,0) (1,0,0)
    (0.5546,0.5042,1.825) (0,0,0)
};
\addlegendentry{Scatter}
\end{axis}
\end{tikzpicture}
```

PGFPLOTS uses the four input points to compute appropriate `x`, `y` and `z` unit vectors (and the origin in graphics coordinates). These four vectors (with two components each) can be computed as a result of a linear system of size 8×8, that is why you need to provide four input points (each has two coordinates). PGFPLOTS computes the unit vectors of the imported graphics, and afterwards it rescales the result such that it fits into the specified `width` and `height`. This rescaling respects the `unit vector ratio` (more precisely, it uses `scale mode=scale uniformly` instead of `scale mode=stretch to fill`). Consequently, the freedom to change the view of a three–dimensional axis which contains a projected graphics is considerably smaller than before. Surprisingly, you can still change axis limits and `width` and `height` – PGFPLOTS will take care of a correct display of your imported graphics.

Here is a further example. Suppose we are given the three–dimensional visualization



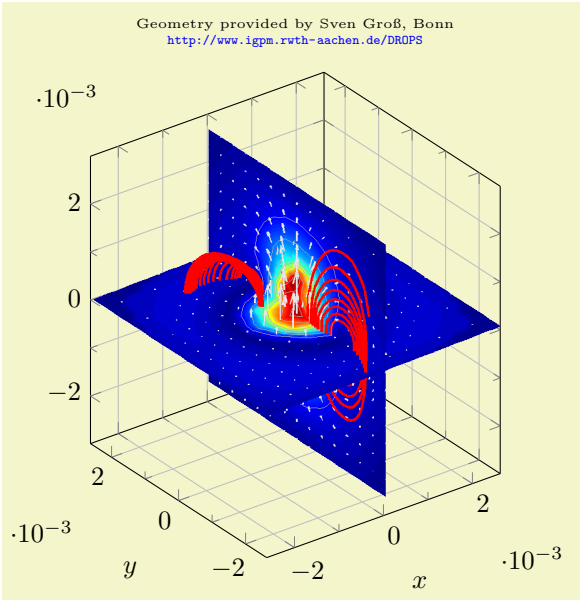
It has been generated by matlab (I only added transparency to the background with `gimp`). Besides advanced visualization techniques, it uses `axis equal`, i.e. `unit vector ratio=1 1 1`. As before, we need to identify four points, each with its 3d logical coordinates (from matlab) and the associated 2d canvas coordinates relative to the lower left corner of the graphics (note that there is a lot of white space around the graphics). Here is the output of PGFPLOTS when you import the resulting graphics:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    grid=both,minor tick num=1,
    xlabel=$x$,ylabel=$y$,
    title={\centering
      Geometry provided by Sven Gro\ss, Bonn\\
      \url{http://www.igpm.rwth-aachen.de/DROPS}\\},
    title style={text width=6cm,font=\tiny},
]
    \addplot3 graphics[
        points={
            (-0.002625,0.002625,0) => (140,234)
            (0,0.00263,0.00263)    => (230,364)
            (0,-0.00263,-0.00263)  => (366,81)
            (0,-0.00263,0.00263)   => (366,276)
            (0.002625,0.002625,0.002625)
        }
    ]
    {plotdata/risingdrop3d.png};
\end{axis}
\end{tikzpicture}
```

Note that I provided *five* three–dimensional coordinates here, but the last entry has no `=>` mapping to two–dimensional canvas coordinates. Thus, it is only used to update the bounding box (see the reference manual for the `points` key for details).

The example above is clipped because PGFPLOTS could not rescale the graphics automatically. Changing the ratio between `width` and `height` improves the display:



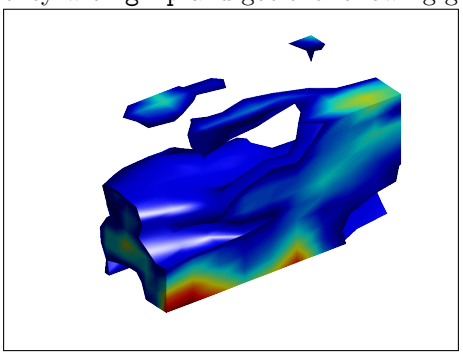

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    height=8cm,width=7cm,%  improve scaling manually
    grid=both,minor tick num=1,
    xlabel=$x$,ylabel=$y$,
    title={\centering
      Geometry provided by Sven Gro\ss, Bonn\\
      \url{http://www.igpm.rwth-aachen.de/DROPS}\\},
    title style={text width=6cm,font=\tiny},
]
    \addplot3 graphics[
        points={
            (-0.002625,0.002625,0) => (140,234)
            (0,0.00263,0.00263)    => (230,364)
            (0,-0.00263,-0.00263)  => (366,81)
            (0,-0.00263,0.00263)   => (366,276)
            (0.002625,0.002625,0.002625)
        }
    ]
    {plotdata/risingdrop3d.png};
\end{axis}
\end{tikzpicture}
```

What happens is that PGFPLOTS *only* rescales the z axis. This has the effect that the axes are all scaled by a single scaling factor (using `scale mode=scale uniformly`), skewing is avoided. In this approach, you can only modify `width` and `height` to provide more freedom.

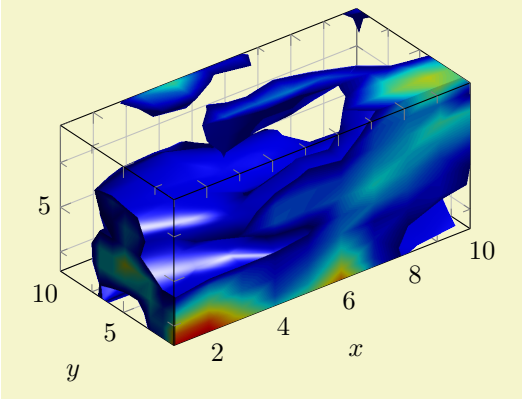
We consider a third example which has been generated by the Matlab code

```
clear all
close all
seed = sum(clock)
rand('seed',seed);
X = rand(10,10,10);
data = smooth3(X,'box',5);
p1 = patch(isosurface(data,.5), ...
   'FaceColor','blue','EdgeColor','none');
p2 = patch(isocaps(data,.5), ...
    'FaceColor','interp','EdgeColor','none');
isonormals(data,p1)
daspect([1 2 2])
view(3); axis vis3d tight
camlight; lighting phong
%  print -dpng plotgraphics3withaxis
axis off
print -dpng plotgraphics3
save  plotgraphics3.seed seed -ASCII %  to reproduce the result
```

I only added background transparency with `gimp` and got the following graphics:



We proceed as before and collect four points, each with 3d logical coordinates (by clicking into the matlab figure) and their associated 2d canvas (graphics) coordinates using the measure tool of gimp. The result is shown in the code example below.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    grid=both,minor tick num=1,
    xlabel=$x$,ylabel=$y$,
    3d box,
]
    \addplot3 graphics[
        points={
            (1,1,1)     => (205,48)
            (10,1,10)   => (503,324)
            (1,1,4.044)=> (206,102)
            (10,10,10) => (390,398)
        }
    ]
    {plotdata/plotgraphics3.png};
\end{axis}
\end{tikzpicture}
```

Technical points: the following issues might arise while working with `\addplot3 graphics`:

- It must be possible to deduce the origin and the three (two–dimensional) unit vectors from the four provide `points`; otherwise the algorithm will fail.

  The algorithm should detect any deficiancies. However, if you encounter strange "Dimension too large" messages here, you can try other arguments in `points`. Take a look into your log file, it will probably indicate the source of problems (or use the `debug` key).

- PGFPLOTS uses the first two points to squeeze the graphics into the desired coordinates (which implies that they should not have the same canvas X or Y coordinates). It verifies that the remaining `points` arguments are projected correctly.
- The resulting scaling by means of `scale mode=scale uniformly` will only rescale the z axis. You may need to adjust `width` and `height` (both of them) if the result is unsatisfactory (see the example above).
- There is a `debug` key to investigate what the algorithm is doing:

  `/pgfplots/plot graphics/debug={`⟨*true,false*⟩`}` (initially `false`)

  If you provide `\addplot3 graphics[debug,points={...}]`, PGFPLOTS will provide debug information onto your terminal and into the logfile. It will also generate extra files containing the determined unit vectors and the linear system used to derive them (one such file for every `\addplot3 graphics` statement, the filename will be the graphics file name and `.dat` appended).

  Without the `debug` key, only the log file will contain brief information what PGFPLOTS is doing behind the scenes.

4.3 About Options: Preliminaries

PGFPLOTS knows a whole lot of key–value options which can be (re)defined to activate desired features or modified to apply some fine-tuning.

A key usually has a value (like a number, a string, or perhaps some macro code). You can assign values to keys ("set keys") in many places in a LaTeX document. The value will remain effective until it is changed or until the current TeX scope ends (which happens after a closing curly brace '`}`', after `\end{`⟨*name*⟩`}` or, for example, after `\addplot`).
Most keys can be used like

```
\begin{tikzpicture}
\begin{axis}[key=value,key2=value2] %  axis-wide keys
...
\end{axis}
\end{tikzpicture}
```

which changes them for the complete axis. A `key` in this context can be any option defined in this manual, no matter if it has the `/pgfplots/` or the `/tikz/` key prefix. Note that key prefixes can be omitted in almost all cases.

A value can usually be provided without curly braces. For example, if the manual contains somethig like '`xmin={`⟨*x coordinate*⟩`}`', you can safely skip the curly braces. The curly braces are mandatory if values contain something which would otherwise confuse the key setup (for example an equal sign '`=`' or a comma '`,`').

Some keys can be changed individually for each plot:

```
\begin{tikzpicture}
\begin{axis}
%  keys valid for single plots:
\addplot ...;                          %  uses the "cycle list" to determine keys
\addplot[key=value,key2=value2] ... ;  %  uses the provided keys (not the "cycle list")
\addplot+[key=value,key2=value2] ... ; %  appends something to the "cycle list"
\end{axis}
\end{tikzpicture}
```

Besides these two possibilities, it is also possible to work with document-wide keys:

```
\section{My Section}
\pgfplotsset{
    key=value,
    key2=value2,
}
This section has a common key configuration:
\begin{tikzpicture}
    \begin{axis}%  uses the key config from above
    ...
    \end{axis}
\end{tikzpicture}
```

In the example above, the \pgfplotsset command changes keys. The changes are permanent and will be used until

- you redefine them or
- the current environment (like \end{figure}) is ended or
- TEX encounters a closing brace '}'.

This includes document–wide preamble configurations like

```
\documentclass{article}

\usepackage{pgfplots}
\pgfplotsset{
    xticklabel={$\mathsf{\pgfmathprintnumber{\tick}}$},
    every axis/.append style={
        font=\sffamily,
    },
}
...
```

The basic engine to manage key–value pairs is pgfkeys which is part of PGF. This engine always has a key name and a key "path", which is somehow similar to file name and directory of files. The common "directory" (key path) of PGFPLOTS is '/pgfplots/'. Although the key definitions below provide this full path, it is always (well, almost always) safe to skip this prefix – PGFPLOTS uses it automatically. The same holds for the prefixes '/tikz/' which are common for all Ti*k*Z drawing options and '/pgf/' which are for the (more or less) low–level commands of PGF. All these prefixes can be omitted.

One important concept is the concept of *styles*. A style is a key which contains one or more other keys. It can be redefined or modified until it is actually used by the internal routines. Each single component of Ti*k*Z and PGFPLOTS can be configured with styles.

For example,

```
\pgfplotsset{legend style={line width=1pt}}
```

sets the line width for every legend to 1pt by appending 'line width=1pt' to the existing style for legends.

There are keys like legend style, ticklabel style, and label style which allow to modify the predefined styles (in this case the styles for legends, ticklabels and axis labels, respectively). They are, in general, equivalent to a ⟨*style name*⟩/.append style={} command (the only difference is that the /.append style thing is a little bit longer). There is also the possibility to define a new style (or to overwrite an already existing one) using /.style={}.

There are several other styles predefined to modify the appearance, see Section 4.17.

\pgfplotsset{⟨*key-value-list*⟩}

Defines or sets all options in ⟨*key-value-list*⟩. The ⟨*key-value-list*⟩ can contain any of the options in this manual which have the prefix /pgfplots/ (however, you do not need to type that prefix).

Inside of ⟨*key-value-list*⟩, the prefixes '/pgfplots/' which are commonly presented in this manual can be omitted (they are checked automatically).

This command can be used to define default options for the complete document or a part of the document. For example,

```
\pgfplotsset{
    cycle list={%
        {red, mark=*}, {blue,mark=*},
        {red, mark=x}, {blue,mark=x},
        {red, mark=square*}, {blue,mark=square*},
        {red, mark=triangle*}, {blue,mark=triangle*},
        {red, mark=diamond*}, {blue,mark=diamond*},
        {red, mark=pentagon*}, {blue,mark=pentagon*}
    },
    legend style={
        at={(0.5,-0.2)},
        anchor=north,
        legend columns=2,
        cells={anchor=west},
        font=\footnotesize,
        rounded corners=2pt,
    },
    xlabel=$x$,ylabel=$f(x)$
}
```

can be used to set document-wise styles for line specifications, the legends' style and axis labels. The settings remain in effect until the end of the current environment (like `\end{figure}`) or until you redefine them or until the next closing curly brace '`}`' (whatever comes first).

You can also define new styles (collections of key–value–pairs) with `/.style` and `/.append style`.

```
\pgfplotsset{
    My Style 1/.style={xlabel=$x$, legend entries={1,2,3} },
    My Style 2/.style={xlabel=$X$, legend entries={4,5,6} }
}
```

The `/.style` and `/.append style` key handlers are described in Section 4.17 in more detail.

Key handler ⟨*key*⟩`/.code=`{⟨*TEX code*⟩}

Occasionally, the PGFPLOTS user interface offers to replace parts of its routines. This is accomplished using so called "code keys". What it means is to replace the original key and its behavior with new ⟨*TEX code*⟩. Inside of ⟨*TEX code*⟩, any command can be used. Furthermore, the `#1` pattern will be the argument provided to the key.

I've been invoked with 'this here'

```
\pgfplotsset{
    My Code/.code={I've been invoked with '#1'}}
\pgfplotsset{My Code={this here}}
```

The example defines a (new) key named `My Code`. Essentially, it is nothing else but a `\newcommand`, plugged into the key-value interface. The second statement "invokes" the code key.

Key handler ⟨*key*⟩`/.code 2 args=`{⟨*TEX code*⟩}

As `/.code`, but this handler defines a key which accepts two arguments. When the so defined key is used, the two arguments are available as `#1` and `#2`.

Key handler ⟨*key*⟩`/.cd`

Each key has a fully qualified name with a (long) prefix, like `/pgfplots/xmin`. However, if the "current directory" is `/pgfplots`, it suffices to write just `xmin`. The `/.cd` key handler changes the "current directory" in this way.

The prefixes `/tikz/` and `/pgfplots/` are checked automatically for any argument provided to `\begin{axis}[`⟨*options*⟩`]` or `\addplot`. So, you won't need to worry about them, just omit them – and look closer in case the package doesn't identify the option.

4.3.1 Pgfplots and Ti*k*Z Options

This section is more or less technical and can be skipped unless one really wants to know more about this topic.

Ti*k*Z options and PGFPLOTS options can be mixed inside of the axis arguments and in any of the associated styles. For example,

```
\pgfplotsset{every axis legend/.append style={
    legend columns=3,font=\Large}}
```

assigns the ‘legend columns’ option (a PGFPLOTS option) and uses ‘font’ for drawing the legend (a Ti*k*Z option). The point is: legend columns needs to be known *before* the legend is typeset whereas font needs to be active when the legend is typeset. PGFPLOTS sorts out any key dependencies automatically:

The axis environments will process any known PGFPLOTS options, and all ‘every’–styles will be parsed for PGFPLOTS options. Every unknown option is assumed to be a Ti*k*Z option and will be forwarded to the associated Ti*k*Z drawing commands. For example, the ‘font=\Large’ above will be used as argument to the legend matrix, and the ‘font=\Large’ argument in

```
\pgfplotsset{every axis label/.append style={
    ylabel=Error,xlabel=Dof,font=\Large}}
```

will be used in the nodes for axis labels (but not the axis title, for example).

It is an error if you assign incompatible options to axis labels, for example ‘xmin’ and ‘xmax’ can’t be set inside of ‘every axis label’.

4.4 Two Dimensional Plot Types

PGFPLOTS supports several two-dimensional line plots like piecewise linear line plots, piecewise constant plots, smoothed plots, bar plots and comb plots. Most of them use the PGF plot handler library directly, see [5, section 18.8].

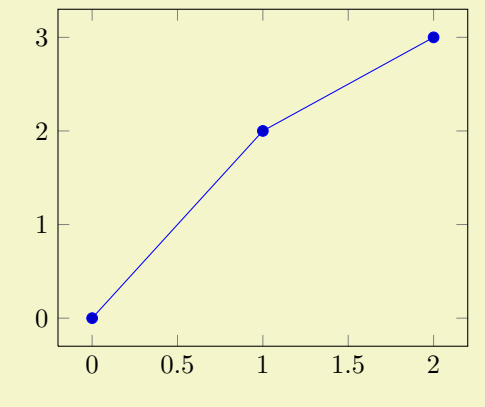
Plot types are part of the plot style, so they are set with options. Most of the basic 2d plot types are part of Ti*k*Z, see [5, section 18.8], and are probably known to users of Ti*k*Z. They are documented here as well.

4.4.1 Linear Plots

/tikz/sharp plot (no value)

\addplot+[sharp plot]

Linear (‘sharp’) plots are the default. Point coordinates are simply connected by straight lines.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot+[sharp plot] coordinates
        {(0,0) (1,2) (2,3)};
\end{axis}
\end{tikzpicture}
```

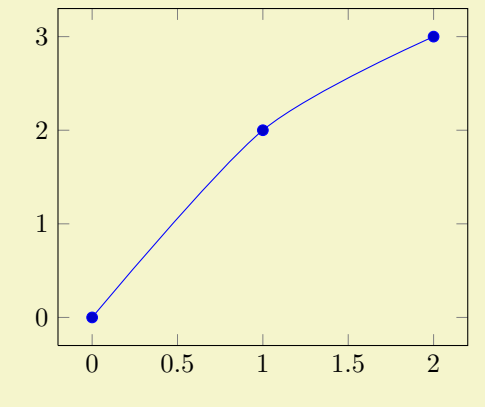
The ‘+’ here means to use the normal plot cycle list and append ‘sharp plot’ to its option list.

4.4.2 Smooth Plots

/tikz/smooth (no value)

\addplot+[smooth]

Smooth plots interpolate smoothly between successive points.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot+[smooth] coordinates
        {(0,0) (1,2) (2,3)};
\end{axis}
\end{tikzpicture}
```

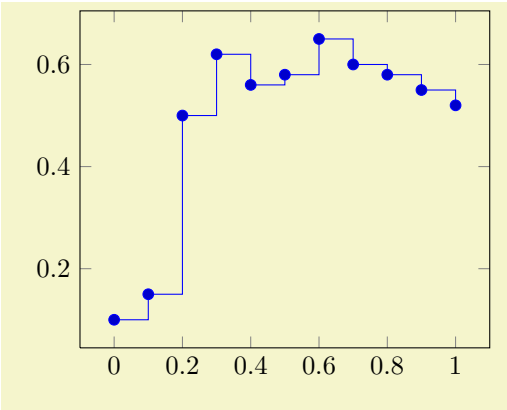
4.4.3 Constant Plots

Constant plots draw lines parallel to the x-axis to connect coordinates. The discontinuous edges may be drawn or not, and marks may be placed on left or right ends.

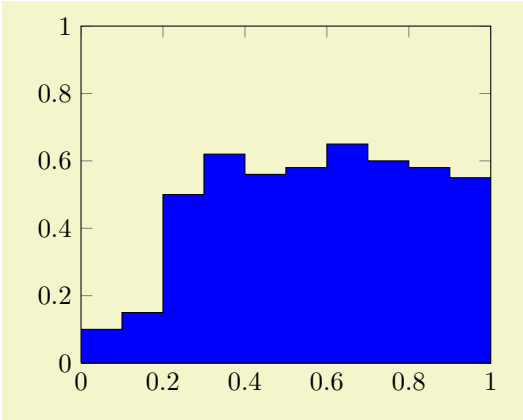
/tikz/const plot (no value)

\addplot+[const plot]

Connects all points with horizontal and vertical lines. Marks are placed left-handed on horizontal line segments, causing the plot to be right-sided continuous at all data points.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[const plot]
coordinates
{(0,0.1)    (0.1,0.15)  (0.2,0.5)   (0.3,0.62)
 (0.4,0.56) (0.5,0.58)  (0.6,0.65)  (0.7,0.6)
 (0.8,0.58) (0.9,0.55)  (1,0.52)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[ymin=0,ymax=1,enlargelimits=false]
\addplot
    [const plot,fill=blue,draw=black]
coordinates
{(0,0.1)    (0.1,0.15)  (0.2,0.5)   (0.3,0.62)
 (0.4,0.56) (0.5,0.58)  (0.6,0.65)  (0.7,0.6)
 (0.8,0.58) (0.9,0.55)  (1,0.52)}
    \closedcycle;
\end{axis}
\end{tikzpicture}
```

/tikz/const plot mark left (no value)

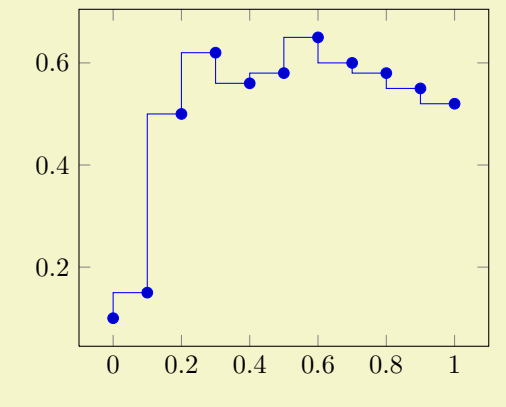
\addplot+[const plot mark left]

An alias for ‘const plot’.

/tikz/const plot mark right (no value)

\addplot+[const plot mark right]

A variant which places marks on the right of each line segment, causing plots to be left-sided continuous at the given coordinates.

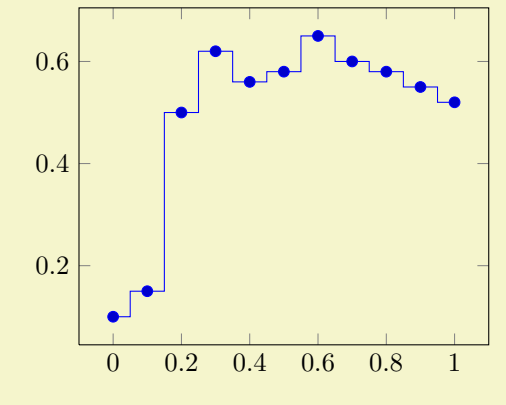



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[const plot mark right]
coordinates
{(0,0.1)    (0.1,0.15)  (0.2,0.5)   (0.3,0.62)
 (0.4,0.56) (0.5,0.58)  (0.6,0.65)  (0.7,0.6)
 (0.8,0.58) (0.9,0.55)  (1,0.52)};
\end{axis}
\end{tikzpicture}
```

/tikz/const plot mark mid (no value)

\addplot+[const plot mark mid]

A variant which places marks in the middle of each line segment, causing plots to be symmetric around its data points.



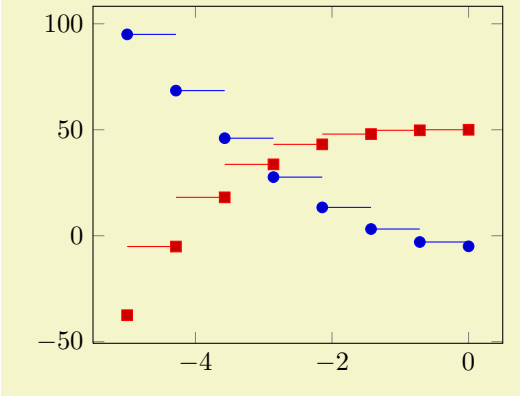

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[const plot mark mid]
coordinates
{(0,0.1)    (0.1,0.15)  (0.2,0.5)   (0.3,0.62)
 (0.4,0.56) (0.5,0.58)  (0.6,0.65)  (0.7,0.6)
 (0.8,0.58) (0.9,0.55)  (1,0.52)};
\end{axis}
\end{tikzpicture}
```

Note that "symmetric" is only true for constant mesh width: if the x–distances between adjacent data points differ, `const plot mark mid` will produce vertical lines in the middle between each pair of consecutive points.

/tikz/jump mark left (no value)

\addplot+[jump mark left]

A variant of '`const plot mark left`' which does not draw vertical lines.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[samples=8]
\addplot+[jump mark left,domain=-5:0]
    {4*x^2 - 5};

\addplot+[jump mark right,domain=-5:0]
    {0.7*x^3 + 50};
\end{axis}
\end{tikzpicture}
```

/tikz/jump mark right (no value)

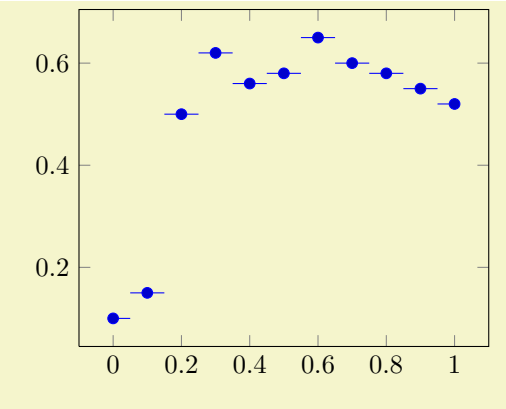
\addplot+[jump mark right]

A variant of ‘const plot mark right’ which does not draw vertical lines.

/tikz/jump mark mid (no value)

\addplot+[jump mark mid]

A variant of ‘const plot mark mid’ which does not draw vertical lines.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[jump mark mid]
coordinates
{(0,0.1)    (0.1,0.15)  (0.2,0.5)   (0.3,0.62)
 (0.4,0.56) (0.5,0.58)  (0.6,0.65)  (0.7,0.6)
 (0.8,0.58) (0.9,0.55)  (1,0.52)};
\end{axis}
\end{tikzpicture}
```

4.4.4 Bar Plots

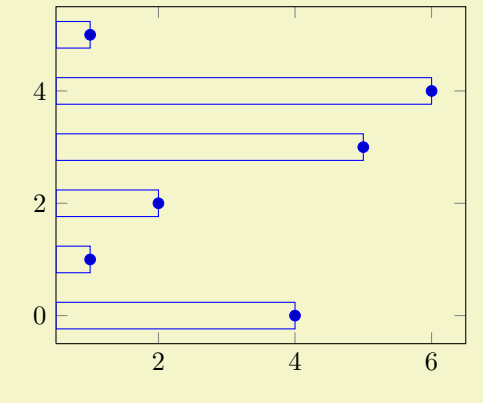
Bar plots place horizontal or vertical bars at coordinates. Multiple bar plots in one axis can be stacked on top of each other or aligned next to each other.

/tikz/xbar (no value)

\addplot+[xbar]

Places horizontal bars between the $(y = 0)$ line and each coordinate.

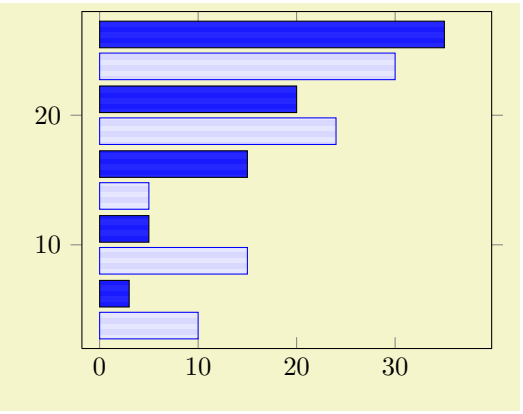
This option is used on a per-plot basis and configures only the visualization of coordinates. The figure-wide style /pgfplots/xbar also sets reasonable options for ticks, legends and multiple plots.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[xbar] coordinates
    {(4,0) (1,1) (2,2)
     (5,3) (6,4) (1,5)};
\end{axis}
\end{tikzpicture}
```

Bars are centered at plot coordinates with width bar width. Using bar plots usually means more than just a different way of how to connect coordinates, for example to draw ticks outside of the axis, change the legend’s appearance or introduce shifts if multiple \addplot commands appear.

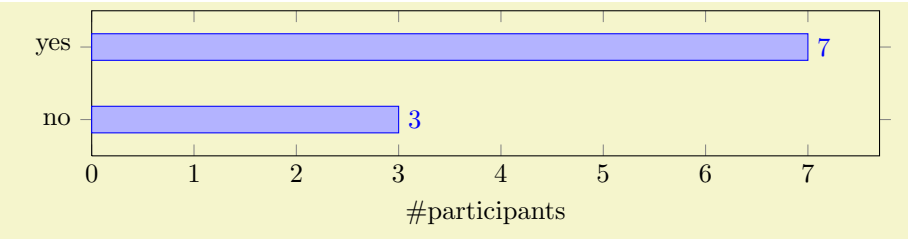
There is a preconfigured style for xbar which is installed automatically if you provide xbar as argument to the axis environment which provides this functionality.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[xbar,enlargelimits=0.15]
\addplot
[draw=blue,pattern=horizontal lines light blue]
coordinates
    {(10,5) (15,10) (5,15) (24,20) (30,25)};

\addplot
[draw=black,pattern=horizontal lines dark blue]
coordinates
    {(3,5) (5,10) (15,15) (20,20) (35,25)};
\end{axis}
\end{tikzpicture}
```

Here `xbar` yields `/pgfplots/xbar` because it is an argument to the axis, not to a single plot.

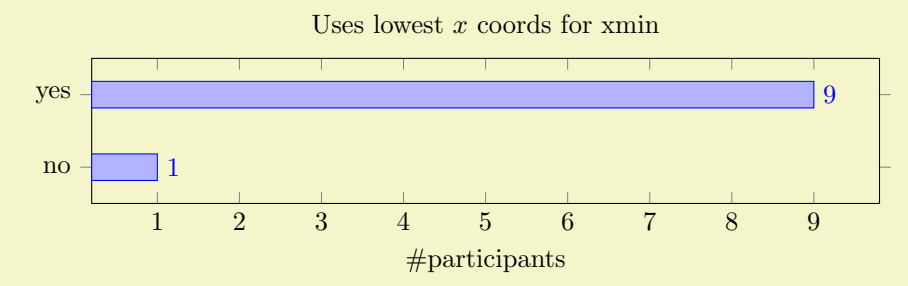
For bar plots, it is quite common to provide textual coordinates or even descriptive nodes near the bars. This can be realized using the `symbolix y coords` and `nodes near coords` keys, respectively:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{axis}[
    xbar, xmin=0,
    width=12cm, height=3.5cm, enlarge y limits=0.5,
    xlabel={\#participants},
    symbolic y coords={no,yes},
    ytick=data,
    nodes near coords, nodes near coords align={horizontal},
    ]
    \addplot coordinates {(3,no) (7,yes)};
  \end{axis}
\end{tikzpicture}
```

The `symbolic y coords` defines a dictionary of accepted coordinates which are then expected in y coordinates and the `nodes near coords` key displays values as extra nodes (see their reference documentations for details). The example employs `enlarge y limits` in order to get some more free space since the default spacing is not always appropriate for bar plots.

Note that it might be quite important to include `xmin=0` explicitly as in the example above. Without it, the lower bound will be used:



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{axis}[
      title=Uses lowest $x$ coords for xmin,
    xbar,
    width=12cm, height=3.5cm, enlarge y limits=0.5,
    xlabel={\#participants},
    symbolic y coords={no,yes},
    ytick=data,
    nodes near coords, nodes near coords align={horizontal},
    ]
    \addplot coordinates {(1,no) (9,yes)};
  \end{axis}
\end{tikzpicture}
```

Besides line, fill, and colorstyles, bars can be configured with `bar width` and `bar shift`, see below.

/pgfplots/xbar={⟨*shift for multiple plots*⟩} (style, default 2pt)

This style sets `/tikz/xbar` *and* some commonly used options concerning horizontal bars for the complete axis. This is automatically done if you provide `xbar` as argument to an axis argument, see above.

The `xbar` style defines shifts if multiple plots are placed into one axis. It draws bars adjacent to each other, separated by ⟨*shift for multiple plots*⟩. Furthermore, it sets the style `bar cycle list` and sets tick and legend appearance options.

The style is defined as follows.

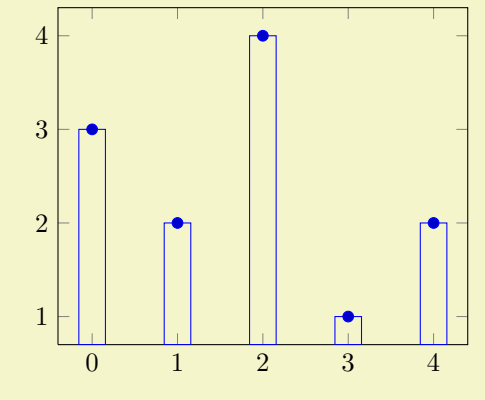
```
\pgfplotsset{
    /pgfplots/xbar/.style={
        /tikz/xbar,
        bar cycle list,
        tick align=outside,
        xbar legend,
        /pgf/bar shift={%
            %  total width = n*w + (n-1)*skip
            %  i.e. subtract half for centering
            -0.5*(\numplotsofactualtype*\pgfplotbarwidth + (\numplotsofactualtype-1)*#1)  +
            %  the '0.5*w' is for centering
            (.5+\plotnumofactualtype)*\pgfplotbarwidth + \plotnumofactualtype*#1%
        },
    },
}
```

The formula for `bar shift` assigns shifts dependent on the total number of plots and the current plot's number. It is designed to fill a total width of n·`bar width`+$(n-1)$·⟨*shift for multiple plots*⟩. The 0.5 compensates for centering.

/tikz/ybar (no value)

\addplot+[ybar]

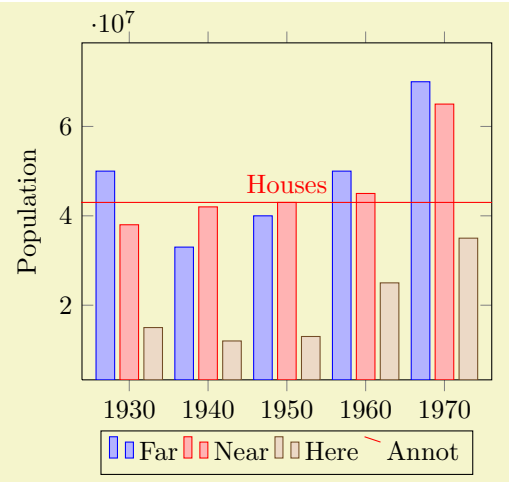
Like `xbar`, this option generates bar plots. It draws vertical bars between the $(x = 0)$ line and each input coordinate.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[ybar] plot coordinates
    {(0,3) (1,2) (2,4) (3,1) (4,2)};
\end{axis}
\end{tikzpicture}
```

The example above simply changes how input coordinates shall be visualized. As mentioned for `xbar`, one usually needs modified legends and shifts for multiple bars in the same axis.

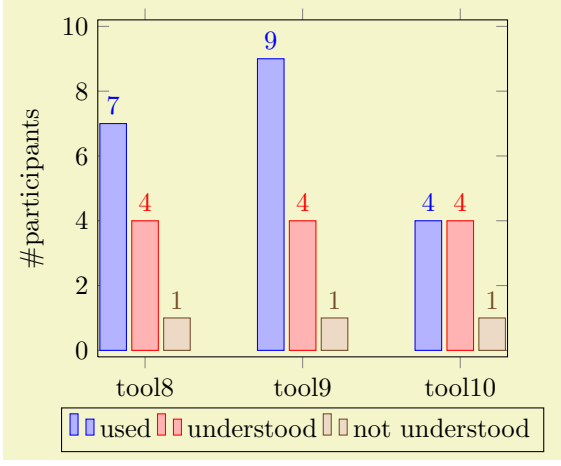
There is a predefined style which installs these customizations when provided to the axis environment:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    x tick label style={
        /pgf/number format/1000 sep=},
    ylabel=Population,
    enlargelimits=0.15,
    legend style={at={(0.5,-0.15)},
        anchor=north,legend columns=-1},
    ybar,
    bar width=7pt,
]
\addplot
    coordinates {(1930,50e6) (1940,33e6)
         (1950,40e6) (1960,50e6) (1970,70e6)};

\addplot
    coordinates {(1930,38e6) (1940,42e6)
        (1950,43e6) (1960,45e6) (1970,65e6)};

\addplot
    coordinates {(1930,15e6) (1940,12e6)
        (1950,13e6) (1960,25e6) (1970,35e6)};

\addplot[red,sharp plot,update limits=false]
    coordinates {(1910,4.3e7) (1990,4.3e7)}
    node[above] at (axis cs:1950,4.3e7) {Houses};

\legend{Far,Near,Here,Annot}
\end{axis}
\end{tikzpicture}
```

Here, `ybar` yields `/pgfplots/ybar` because it is an argument to the axis, not to a single plot. The style affects the first three `\addplot` commands. Note that it shifts them horizontally around the plot coordinates. The fourth `\addplot` command is some kind of annotation which doesn't `update limits`.

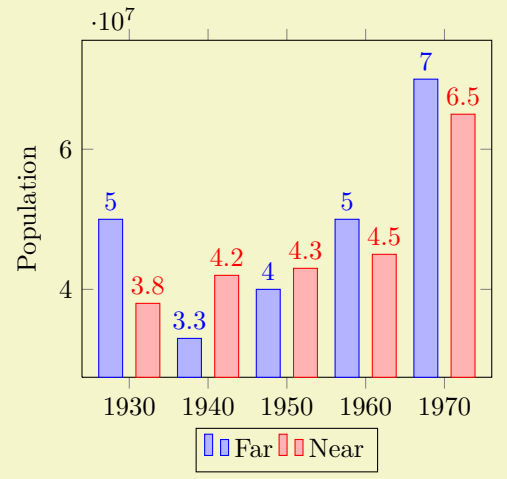
The `ybar` style can be combined with `symbolic x coords` in a similar way as described for `xbar`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    ybar,
    enlargelimits=0.15,
    legend style={at={(0.5,-0.15)},
      anchor=north,legend columns=-1},
    ylabel={\#participants},
    symbolic x coords={tool8,tool9,tool10},
    xtick=data,
    nodes near coords,
    nodes near coords align={vertical},
    ]
\addplot coordinates {(tool8,7) (tool9,9) (tool10,4)};
\addplot coordinates {(tool8,4) (tool9,4) (tool10,4)};
\addplot coordinates {(tool8,1) (tool9,1) (tool10,1)};
\legend{used,understood,not understood}
\end{axis}
\end{tikzpicture}
```

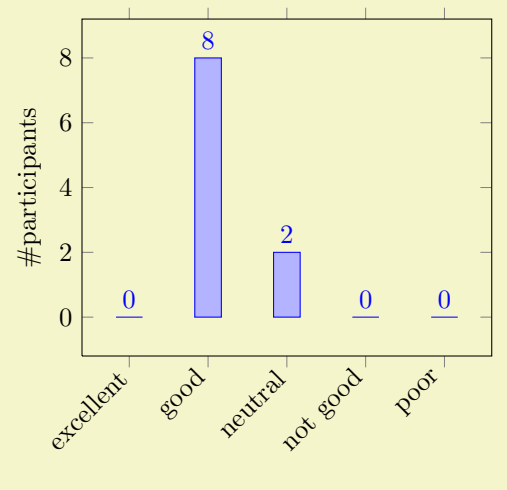
As for `xbar`, the bar width and shift can be configured with `bar width` and `bar shift`. However, the bar shift is better provided as argument to `/pgfplots/ybar` since this style will overwrite the bar shift. Thus, prefer `/pgfplots/ybar=4pt` to set the bar shift.

Sometimes it is useful to write the y values directly near the bars. This can be realized using the `nodes near coords` method:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    x tick label style={
        /pgf/number format/1000 sep=},
    ylabel=Population,
    enlargelimits=0.15,
    legend style={at={(0.5,-0.15)},
        anchor=north,legend columns=-1},
    ybar=5pt,%  configures 'bar shift'
    bar width=9pt,
    nodes near coords,
    point meta=y *10^-7 %  the displayed number
]
\addplot
    coordinates {(1930,50e6) (1940,33e6)
         (1950,40e6) (1960,50e6) (1970,70e6)};

\addplot
    coordinates {(1930,38e6) (1940,42e6)
        (1950,43e6) (1960,45e6) (1970,65e6)};

\legend{Far,Near}
\end{axis}
\end{tikzpicture}
```

Any support style changes are possible, of course. A useful example for bar plots might be to use rotated tick labels:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{axis}[
    ybar,
    enlargelimits=0.15,
    legend style={at={(0.5,-0.2)},
      anchor=north,legend columns=-1},
    ylabel={\#participants},
    symbolic x coords={excellent,good,neutral,%
        not good,poor},
    xtick=data,
    nodes near coords,
    nodes near coords align={vertical},
    x tick label style={rotate=45,anchor=east},
    ]
    \addplot coordinates {(excellent,0) (good,8)
        (neutral,2) (not good,0) (poor,0)};
  \end{axis}
\end{tikzpicture}
```

/pgfplots/ybar={⟨*shift for multiple plots*⟩} (style, default 2pt)

As /pgfplots/xbar, this style sets the /tikz/ybar option to draw vertical bars, but it also provides commonly used options for vertical bars.

If you supply ybar to an axis environment, /pgfplots/ybar will be chosen instead of /tikz/ybar.

It changes the legend, draws ticks outside of the axis lines and draws multiple \addplot arguments adjacent to each other; block–centered at the x coordinate and separated by ⟨*shift for multiple plots*⟩. It will also install the bar shift for every node near coord. Furthermore, it installs the style bar cycle list. It is defined similarly to /pgfplots/xbar.

/pgfplots/bar cycle list (no value)

A style which installs cycle lists for multiple bar plots.

```
\pgfplotsset{
    /pgfplots/bar cycle list/.style={/pgfplots/cycle list={%
        {blue,fill=blue!30!white,mark=none},%
        {red,fill=red!30!white,mark=none},%
        {brown!60!black,fill=brown!30!white,mark=none},%
        {black,fill=gray,mark=none},%
        }
    },
}
```

/pgf/bar width={⟨*dimension*⟩} (initially 10pt)

Configures the width used by xbar and ybar. It is accepted to provide mathematical expressions.

/pgf/bar shift={⟨*dimension*⟩} (initially 0pt)

Configures a shift for xbar and ybar. Use bar shift together with bar width to draw multiple bar plots into the same axis. It is accepted to provide mathematical expressions.

/tikz/ybar interval (no value)

\addplot+[ybar interval]

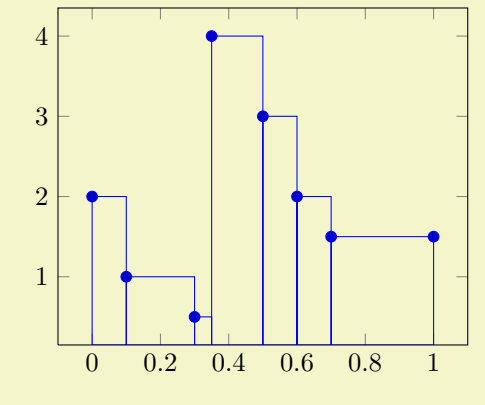
This plot type produces vertical bars with width (and shift) relatively to intervals of coordinates.

There is one conceptional difference when working with intervals: an interval is defined by *two* coordinates. Since ybar has one value for each interval, the ith bar is defined by

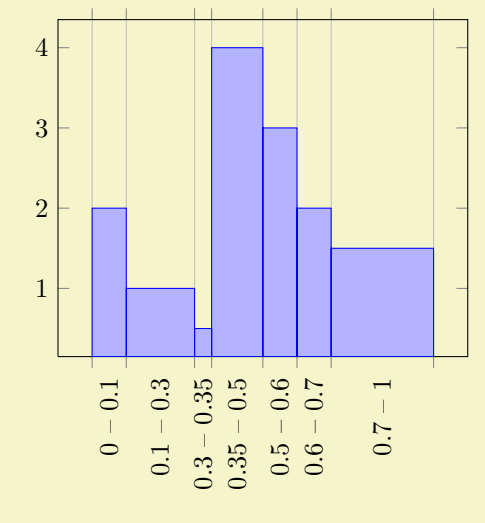
1. the y value of the ith coordinates,
2. the x value of the ith coordinate as left interval boundary,
3. the x value of the $(i+1)$th coordinate as right interval boundary.

Consequently, there is *one coordinate too much*: the last coordinate will *only* be used to determine the interval width; its y value doesn't influence the bar appearance.

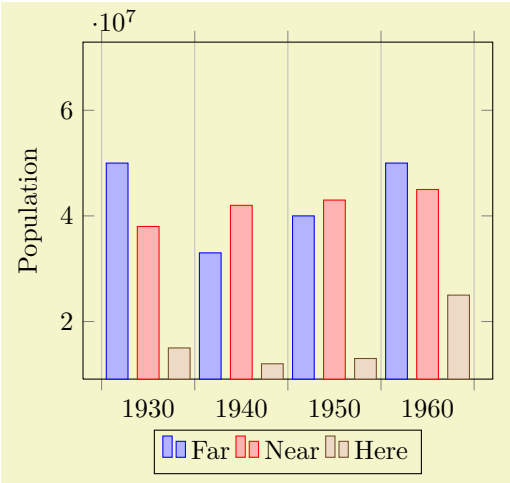
It is installed on a per-plot basis and configures *only* the visualization of coordinates. See the style /pgfplots/ybar interval which configures the appearance of the complete figure.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[ybar interval] plot coordinates
    {(0,2) (0.1,1) (0.3,0.5) (0.35,4) (0.5,3)
     (0.6,2) (0.7,1.5) (1,1.5)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[ybar interval,
    xtick=data,
    xticklabel interval boundaries,
    x tick label style=
        {rotate=90,anchor=east}
    ]
\addplot coordinates
    {(0,2) (0.1,1) (0.3,0.5) (0.35,4) (0.5,3)
     (0.6,2) (0.7,1.5) (1,1.5)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    x tick label style={
        /pgf/number format/1000 sep=},
    ylabel=Population,
    enlargelimits=0.05,
    legend style={at={(0.5,-0.15)},
        anchor=north,legend columns=-1},
    ybar interval=0.7,
]
\addplot
    coordinates {(1930,50e6) (1940,33e6)
         (1950,40e6) (1960,50e6) (1970,70e6)};

\addplot
    coordinates {(1930,38e6) (1940,42e6)
        (1950,43e6) (1960,45e6) (1970,65e6)};

\addplot
    coordinates {(1930,15e6) (1940,12e6)
        (1950,13e6) (1960,25e6) (1970,35e6)};
\legend{Far,Near,Here}
\end{axis}
\end{tikzpicture}
```

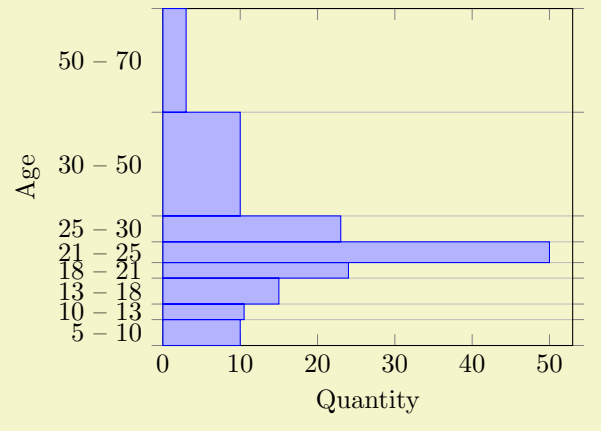
/pgfplots/ybar interval={⟨*relative width*⟩} (style, default 1)

A style which is intended to install options for ybar interval for a complete figure. This includes tick and legend appearance, management of multiple bar plots in one figure and a more adequate cycle list using the style bar cycle list.

/tikz/xbar interval (no value)

\addplot+[xbar interval]

As ybar interval, just for horizontal bars.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xmin=0,xmax=53,
    ylabel=Age,
    xlabel=Quantity,
    enlargelimits=false,
    ytick=data,
    yticklabel interval boundaries,
    xbar interval,
]
\addplot
    coordinates {(10,5) (10.5,10) (15,13)
        (24,18) (50,21) (23,25) (10,30)
        (3,50) (3,70)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/xbar interval={⟨*relative width*⟩} (style, default 1)

A style which is intended to install options for xbar interval for a complete figure, see the style /pgfplots/ybar interval for details.

/pgfplots/xticklabel interval boundaries (no value)
/pgfplots/yticklabel interval boundaries (no value)
/pgfplots/zticklabel interval boundaries (no value)

These are style keys which set x tick label as interval (see page 227 for details) and configure the tick appearance to be ⟨*start*⟩ – ⟨*end*⟩ for each tick interval.

4.4.5 Histograms

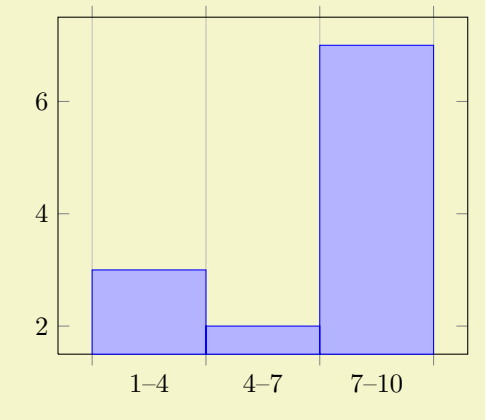
/pgfplots/hist={⟨*options with* hist/ *prefix*⟩}

\addplot+[hist={⟨*options with* hist/ *prefix*⟩}]

A histogram plot takes one-dimensional input data and counts the occurrence of values: it determines the data range $[\underline{m}, \overline{m}]$ and subdivides it into N equally sized bins with $(N+1)$ end–points. Then, it counts the number of points falling into each bin. More precisely, it computes the $N+1$ points $\underline{m} =: x_0 < x_1 < \cdots < x_N := \overline{m}$ using $x_i := \underline{m} + i \cdot (\overline{m} - \underline{m})/N$. Then, it creates the $N+1$ coordinates (x_i, y_i), $i = 0, \ldots, N-1$ by means of

$$y_i := \begin{cases} \text{bincount}\big([x_i, x_{i+1})\big) & i < N \\ y_{N-1} & i = N, \end{cases}$$

i.e. the value of the last coordinate is replicated. This set of $(N+1)$ interval boundaries is then visualized by an ybar interval plot handler.

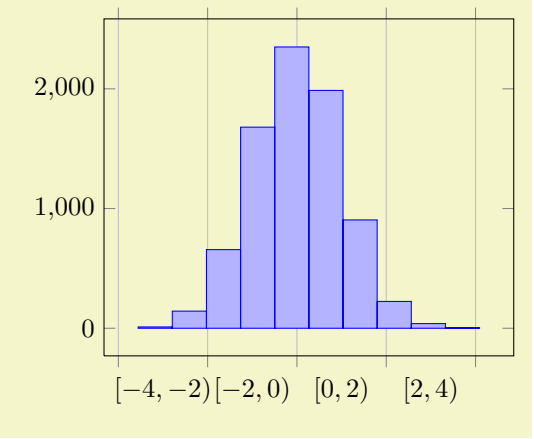



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
  ybar interval,
  xticklabel=
\pgfmathprintnumber\tick--\pgfmathprintnumber\nexttick
]
    \addplot+[hist={bins=3}]
        table[row sep=\\,y index=0] {
        data\\
        1\\ 2\\ 1\\ 5\\ 4\\ 10\\
        7\\ 10\\ 9\\ 8\\ 9\\ 9\\
    };
\end{axis}
\end{tikzpicture}
```

We see that hist={bins=3} takes a table with one column as input. The data values fall into the range $[1, 10]$ which is partitioned into 3 intervals (of equal lengths). Finally, the number of points falling into

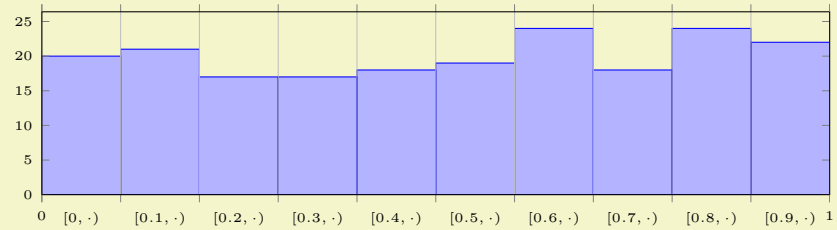
each of the three bins is plotted. The `xticklabel` key shows the range (note that it works only in conjunction with `x tick label as interval` which has been enabled by `ybar interval` before). We see that there are 3 elements in the range $[1, 4)$, 2 elements in the range $[4, 7)$ and finally 7 elements in the range $[7, 10]$.

The bins are half–open intervals, i.e. the end–point does not belong to the bin. Only the last bin contains its right end point.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
  ybar interval,
  xtick=,%  reset from ybar interval
  xticklabel=
    {$[\pgfmathprintnumber\tick,%
       \pgfmathprintnumber\nexttick)$}
]]
%  a data file containing 8000 normally distributed
%  random numbers of mean 0 and variance 1
\addplot+[hist={data=x}]
    file {plotdata/pgfplots.randn.dat};

\end{axis}
\end{tikzpicture}
```

The `hist` plot type can be combined with `plot expression` as well: provide the usual ⟨*expression*⟩ as you would for a line plot. Then, configure the value for `data=`⟨*expression*⟩ in dependence of `x`, `y`, or `z`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
  tiny,
  height=4cm,width=12cm,
  ybar interval,
  ymin=0,
  xmin=0,xmax=1,
  axis on top,
  extra x ticks={0,1},
  extra x tick style={
    grid=none,
    x tick label as interval=false,
    xticklabel=$\pgfmathprintnumber\tick$
  },
  xticklabel={$[\pgfmathprintnumber[fixed]\tick,\cdot)$}
]
    \addplot+[samples=200,hist] {rnd};
\end{axis}
\end{tikzpicture}
```

The example uses the `rnd` method of PGF which defines `y` to contain uniform random numbers in the range $[0, 1]$. Then, it configures `hist`. Note that `hist` has the default `data=y` such that it uses the `y` coordinate as input. Note furthermore that the `x` value is effectively ignored here. The options after `\begin{axis}[...]` are mainly to scale the graphics and to insert the right limits. The `extra x ticks` method is inserted to demonstrate how to add further tick marks without affecting the overall layout. Note that the `extra x tick style` sets `x tick label as interval=false` to disable the special tick handling which is active for the rest of the plot.

The following keys configure `hist`. If they are provided inside of ⟨*options*⟩, the common key prefix

`hist/` can be omitted.

/pgfplots/hist/data={⟨*expression*⟩} (initially `y`)

Tells `hist` how to get its data. The common idea is to provide a mathematical ⟨*expression*⟩ which depends on data supplied by the `\addplot` statement. For example, if you have `\addplot expression`, the ⟨*expression*⟩ may depend upon `x`, `y` or `z`. In case of an `\addplot table` input routine, the ⟨*expression*⟩ can employ `\thisrow{`⟨*colname*⟩`}` to access the currently active table row in the designated column.

It is also possible to avoid invocations of the math parser. Use `hist/data value={`⟨*value*⟩`}` instead to do so. Here, ⟨*value*⟩ should be of a numeric constant.

The initial configuration employs what would usually become the final `y` coordinate as input (to be more precise, the initial value is `data value=\pgfkeysvalueof{/data point/y}`).

/pgfplots/hist/data min={⟨*min value*⟩} (initially `/pgfplots/xmin`)
/pgfplots/hist/data max={⟨*max value*⟩} (initially `/pgfplots/xmax`)

Allows to provide the min/max values (the $\underline{m}$ and $\overline{m}$) values manually.

If empty, these v (walues will be deduced from the input data range.

The resulting interval will be splitted into `hist/bins` intervals.

The initial configuration uses any provided data limits, i.e. the (natural) choices `hist/data min=xmin` and `hist/data max=xmax`.

/pgfplots/hist/bins={⟨*number of intervals*⟩} (initially `10`)

Specifies the number of intervals to use.

/pgfplots/hist/intervals={⟨*true,false*⟩} (initially `true`)

If `intervals=true` (the initial configuration), `hist` will generate $N+1$ coordinates, with

$$\underline{m} = x_0 < x_1 < \cdots < x_N = \overline{m}$$

where $[\underline{m}, \overline{m}]$ is the data range. In this case, the data points for x_{N-1} and x_N will get the same value, namely the number of elements in the last bin. This is (only) useful in conjunction with `const plot` or `ybar interval`.

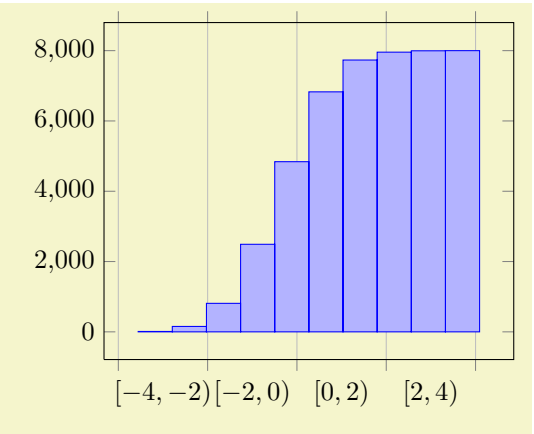
If `intervals=false`, the last data point will be omitted and exactly N coordinates will be generated. In this case, the right end point is not returned explicitly.

/pgfplots/hist/cumulative={⟨*true,false*⟩} (initially `false`)

Allows to compute a cumulative histogram.

A cumulative histogram uses the sum of all previous bins and the current one as final value.

Here is the example from above, this time with `hist/cumulative`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
  ybar interval,
  xtick=,%  reset from ybar interval
  xticklabel=
    {$[\pgfmathprintnumber\tick,
       \pgfmathprintnumber\nexttick)$}
]]
%  a data file containing 8000 normally distributed
%  random numbers of mean 0 and variance 1
\addplot+[hist={
        data=x,
        cumulative}]
    file {plotdata/pgfplots.randn.dat};

\end{axis}
\end{tikzpicture}
```

/pgfplots/hist/handler (style, initially `ybar interval`)

Allows to change the way the generated coordinates are visualized. The hist/handler key is a style, so use hist/handler/.style={const plot} to change it.

/pgfplots/hist/data filter/.code={⟨...⟩}

Allows to define coordinate filters, similar to the coordinate filter key x filter described in Section 4.21. The argument #1 is the coordinate as it has been found after processing hist/data. The code is supposed to assign \pgfmathresult to contain the result. If \pgfmathresult is empty afterwards, it will be skipped. Otherwise, it is supposed to contain a number.

This filter is applied *before* the histogram is computed. Note that x filter and y filter are applied *after* the histogram is computed.

Note that predefined styles like each nth point can also be applied to hist/data if

1. an asterisk '*' is appended to the predefined style's name and
2. the first argument to the style is hist/data.

For example, each nth point*={hist/data}{2} will skip each second input value of hist/data (try it out).

/pgfplots/hist/data coord trafo/.code={⟨...⟩}
/pgfplots/hist/data coord inv trafo/.code={⟨...⟩}

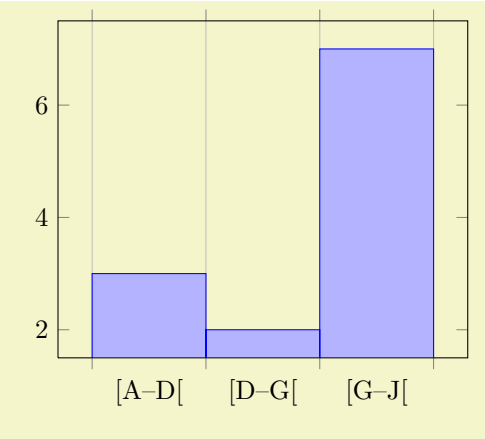
These keys work in the same way as for x coord trafo and x coord inv trafo. They are applied to the hist/data value before the histogram is evaluated and after the result value is assigned, respectively.

Note that hist will apply the hist/data coord inv trafo before it visualizes its results. Consequently, it may be necessary to assign a similar transformation to x coord trafo as well.

See the documentation of x coord trafo for more information about custom transformations.

/pgfplots/hist/symbolic coords={⟨*list*⟩}

A style which enables symbolic x coords for an axis containing hist plots:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
  ybar interval,
  hist/symbolic coords={A,B,C,D,E,F,G,H,I,J},
  xticklabel={[\tick--\nexttick[}],
]
    \addplot+[hist={bins=3}]
        table[row sep=\\,y index=0] {
        data\\
        A\\ B\\ A\\ D\\ F\\ J\\
        G\\ J\\ I\\ H\\ I\\ I\\
    };
\end{axis}
\end{tikzpicture}
```

The style does two things: first, it defines hist/data coord trafo and hist/data coord inv trafo, then, it calls symbolic x coords with the same argument.

Attention : do not use hist/data=x or other symbolic values as input when you have symbolic coords. Rather than symbolic values, you need to provide *expandable* values like \pgfkeysvalueof{/data point/x} (which has the same effect, but directly expands to the correct value).

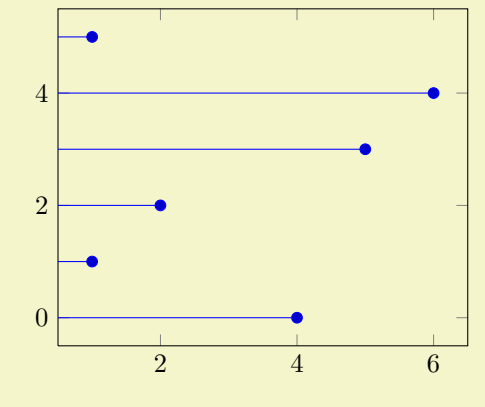
Please refer to the documentation of symbolic x coords for further details about symbolic coordinates.

4.4.6 Comb Plots

Comb plots are very similar to bar plots except that they employ single horizontal/vertical lines instead of rectangles.

/tikz/xcomb (no value)

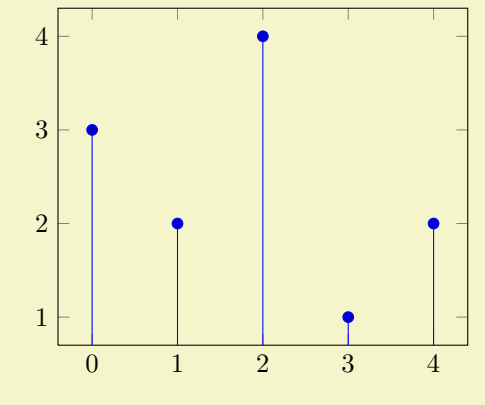
\addplot+[xcomb]




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[xcomb] coordinates
    {(4,0) (1,1) (2,2)
     (5,3) (6,4) (1,5)};
\end{axis}
\end{tikzpicture}
```

/tikz/ycomb (no value)

\addplot+[ycomb]



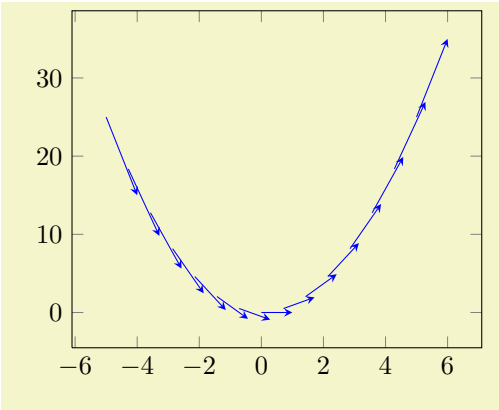

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[ycomb] plot coordinates
    {(0,3) (1,2) (2,4) (3,1) (4,2)};
\end{axis}
\end{tikzpicture}
```

4.4.7 Quiver Plots (Arrows)

/pgfplots/quiver={⟨options with 'quiver/' prefix⟩}

\addplot+[quiver={⟨options with 'quiver/' prefix⟩}]

A plot type which draws small arrows, starting at (x, y), in direction of (u, v).

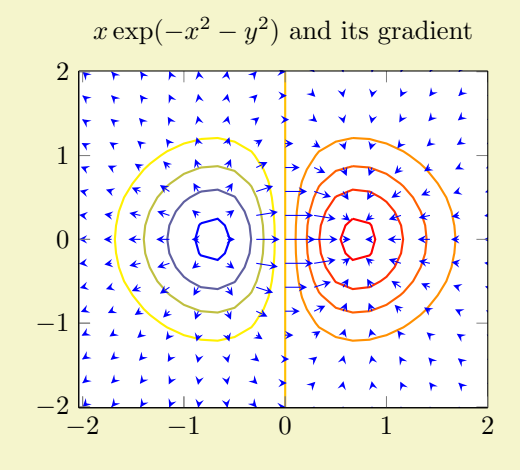



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot[blue,
        quiver={u=1,v=2*x},
        -stealth,samples=15] {x^2};
\end{axis}
\end{tikzpicture}
```

The base point (x, y) is provided as before; in the example above, it is generated by `plot expression` and yields (x, x^2). The vector direction (u, v) needs to be given in addition. Our example with

quiver/u=1 and quiver/v=2*x results in $u = 1$ and $v = 2x$. Thus, we have defined and visualized a vector field for the derivative of $f(x) = x^2$.

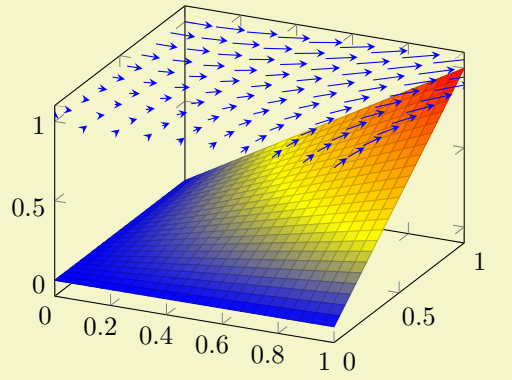
A common example is to visualize the gradient $(\partial_x f, \partial_y f)(x, y)$ of a two–dimensional function $f(x, y)$:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        title={$x \exp(-x^2-y^2)$ and its gradient},
        domain=-2:2,
        view={0}{90},
        axis background/.style={fill=white},
    ]
        \addplot3[contour gnuplot={number=9,
            labels=false},thick]
                {exp(0-x^2-y^2)*x};
        \addplot3[blue,
            quiver={
             u={exp(0-x^2-y^2)*(1-2*x^2)},
             v={exp(0-x^2-y^2)*(-2*x*y)},
             scale arrows=0.3,
            },
            -stealth,samples=15]
                {exp(0-x^2-y^2)*x};
    \end{axis}
\end{tikzpicture}
```

The example visualizes $f(x, y) = x \exp(-x^2 - y^2)$ using contour gnuplot as first step. The options contour/number and contour/labels provide fine-tuning for the contour and are not of interest here (so is the axis background which just improves visibility of contour lines). What we are interested in is the quiver= style: it defines u and v to some two–dimensional expressions. Furthermore, we used quiver/scale arrows to reduce the arrow size. The -stealth is a TikZ style which configures outgoing arrow styles of type 'stealth'. The samples=15 key configures how we get our input data. In our case, we have input data $(x_i, y_j, f(x_i, y_j))$ with 15 samples for each, i and j.

It is also possible to place quiver plots on a prescribed z value:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    domain=0:1,
    xmax=1,
    ymax=1,
]
\addplot3[surf] {x*y};
\addplot3[blue,/pgfplots/quiver,
    quiver/u=y,
    quiver/v=x,
    quiver/w=0,
    quiver/scale arrows=0.1,
    -stealth,samples=10] {1};
\end{axis}
\end{tikzpicture}
```

Here, the quiver plots is placed on top of a surf. It visualizes the gradient (using a common scale factor of 1/10 to reduce the arrow lengths). The quiver/w=0 means that arrows have no z difference, and the {1} argument indicates that all start at $(x_i, y_j, 1)$. Here, the values (x_i, y_j) are sampled in the domain=0:1 argument (with samples=10), i.e. arrows start at $(x_i, y_j, 1)$ and end at $(x_i + y_j/10, y_j + x_i/10, 1)$.

So far, quiver plots do not assume a special sequence of input points. This has two consequences: first, you can plot any vector field by considering just $(x, y) + (u, v)$ (or $(x, y, z) + (u, v, w)$) – the data doesn't necessarily need to be a two–dimensional function (as opposed to surf etc). On the other hand, you need to provide quiver/scale arrows manually since quiver doesn't know the mesh width in case you provide matrix data[15].

Note that quiver plots are currently not available together with logarithmic axes.

/pgfplots/quiver/u=⟨*expression*⟩ (initially 0)

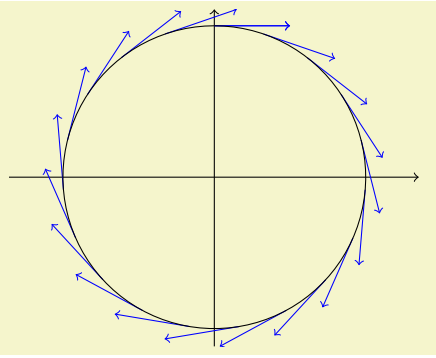
[15] Actually, I might add something like quiver/scale arrows=auto in the future, I don't know yet. Loops through input data are slow in TEX, automatic mesh widths computation even more...

/pgfplots/quiver/v=⟨*expression*⟩ (initially 0)
/pgfplots/quiver/w=⟨*expression*⟩ (initially 0)

These keys define how the vector directions (u, v) (or, for three dimensional plots, (u, v, w)) shall be set.

The ⟨*expression*⟩ can be a constant expression like `quiver/u=1` or `quiver/u=42*5`. It may also depend on the final base point values using the values `x`, `y` or `z` as in the example above. In this context, `x` yields the x coordinate of the point where the vector starts, `y` the y coordinate and so on.

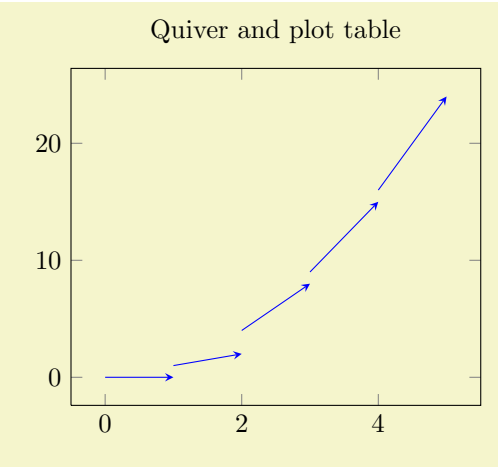
Attention: the fact that `x` refers to *the final x coordinate* means that parametric plots *should use t as variable*[16]. Consider the following example:



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[axis equal,
 axis lines=middle,
 axis line style={->},
 tick style={color=black},
 xtick=\empty,
 ytick=\empty
]
  \addplot[samples=20, domain=0:2*pi,
    %  the default choice 'variable=\x' leads to
    %  unexpected results here!
    variable=\t,
    quiver={
        u={cos(deg(t))},
        v={-sin(deg(t))},
        scale arrows=0.5},
        ->,blue]
    ({sin(deg(t))}, {cos(deg(t))});
  \addplot[samples=100, domain=0:2*pi]
    ({sin(deg(x))}, {cos(deg(x))});
\end{axis}
\end{tikzpicture}
```

Here, a parametric plot is used to draw a circle and tangent vectors. The choice `variable=\t` plays a functional role besides naming conventions: it allows to access the parametric variable within the expressions for both `u` and `v`. On the other hand, we could have used `u=y` and `v=-x` since `x` expands to the x coordinate with value `sin(deg(t))` and `y` expands to the y coordinate `cos(deg(t))`.

Another important application is to use *table column references* like `quiver/u=\thisrow{col}` in conjunction with `\addplot table`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[title=Quiver and plot table]
    \addplot[blue,
        quiver={u=\thisrow{u},v=\thisrow{v}},
        -stealth]
    table
    {
    x y u v
    0 0 1 0
    1 1 1 1
    2 4 1 4
    3 9 1 6
    4 16 1 8
    };
\end{axis}
\end{tikzpicture}
```

Here, the ⟨*expression*⟩ employs `\thisrow` which always refers to the actual row of `\addplot table`.

Note that ⟨*expression*⟩ should always be of numeric type (no symbolic input extensions are supported currently).

[16] Sorry for this usability issue.

/pgfplots/quiver/u value={⟨value⟩} (initially 0)
/pgfplots/quiver/v value={⟨value⟩} (initially 0)
/pgfplots/quiver/w value={⟨value⟩} (initially 0)

These keys have the *same function* as quiver/u and its variants. However, they don't call the math parser, so only single values are allowed (including something like \thisrow{columnname}).

/pgfplots/quiver/colored (no value)
/pgfplots/quiver/colored={⟨color⟩}

Allows to define an individual color for each arrow. Omitting the argument '⟨color⟩' is identical to quiver/colored=mapped color which uses the point meta to get colors for every arrow.

If you just want to set the same color for every arrow, prefer using \addplot[blue,quiver] which is more efficient.

/pgfplots/quiver/scale arrows={⟨scale⟩} (initially 1)

Allows to rescale the arrows by a factor. This may be necessary if the arrow length is longer than the distance between adjacent base points (x_i, y_i). There may come a feature to rescale them automatically.

/pgfplots/quiver/update limits=true|false (initially true)

A boolean indicating whether points $(x, y) + (u, v)$ shall contribute to the axis limits.

/pgfplots/quiver/every arrow (style, initially empty)

Allows to provide individual arrow styles.

The style can contain any Ti*k*Z drawing option. It might depend upon mapped color (provided point meta has been set).

Again, if you don't need individual arrow styles, prefer using a plot style (cycle list or argument to \addplot) which is more efficient.

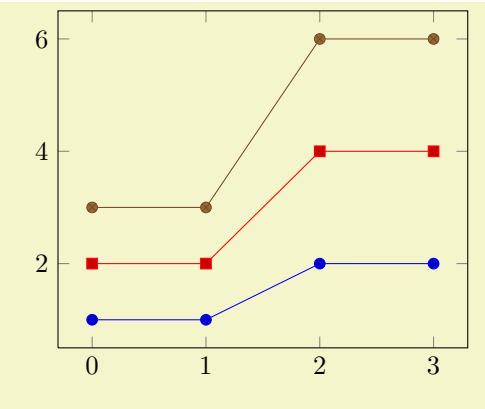
/pgfplots/quiver/before arrow/.code={⟨...⟩}
/pgfplots/quiver/after arrow/.code={⟨...⟩}

Advanced keys for more fine tuning of the display. They allow to install some TEX code manually before or after the drawing operations for single arrows. Both are initially empty.

4.4.8 Stacked Plots

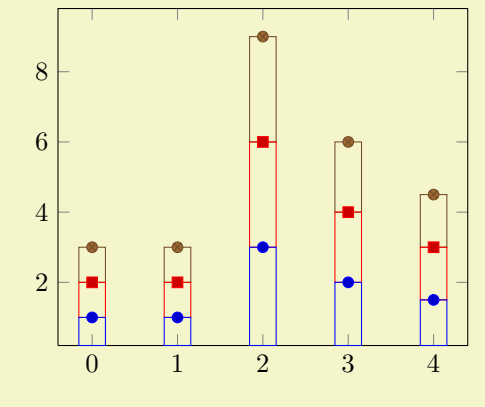
/pgfplots/stack plots=x|y|false (initially false)

Allows stacking of plots in either x or y direction. Stacking means to add either x- or y coordinates of successive \addplot commands on top of each other.

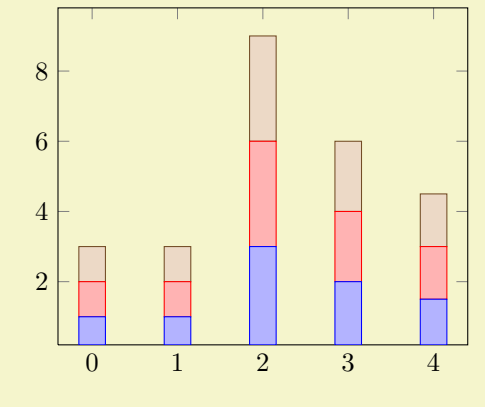



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[stack plots=y]
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)};
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)};
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)};
    \end{axis}
\end{tikzpicture}
```

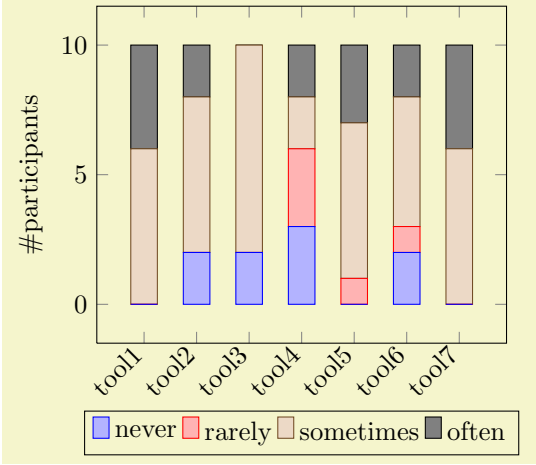
stack plots is particularly useful for bar plots. The following examples demonstrate its functionality. Normally, it is advisable to use the styles ybar stacked and xbar stacked which also set some other options.



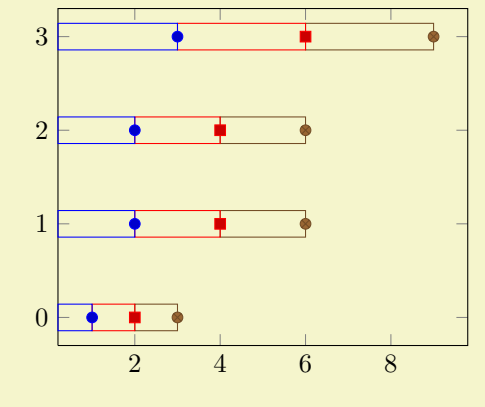

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[stack plots=y,/tikz/ybar]
    \addplot coordinates
        {(0,1) (1,1) (2,3) (3,2) (4,1.5)};
    \addplot coordinates
        {(0,1) (1,1) (2,3) (3,2) (4,1.5)};
    \addplot coordinates
        {(0,1) (1,1) (2,3) (3,2) (4,1.5)};
    \end{axis}
\end{tikzpicture}
```



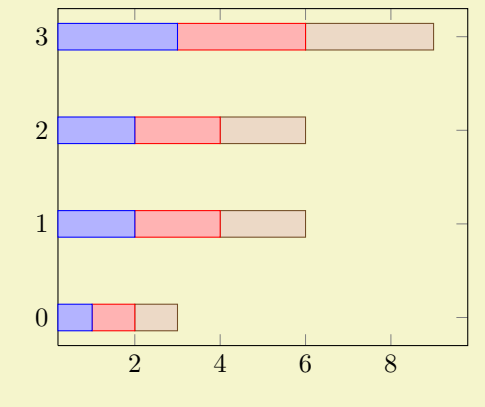

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[ybar stacked]
    \addplot coordinates
        {(0,1) (1,1) (2,3) (3,2) (4,1.5)};
    \addplot coordinates
        {(0,1) (1,1) (2,3) (3,2) (4,1.5)};
    \addplot coordinates
        {(0,1) (1,1) (2,3) (3,2) (4,1.5)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    ybar stacked,
    enlargelimits=0.15,
    legend style={at={(0.5,-0.20)},
      anchor=north,legend columns=-1},
    ylabel={\#participants},
    symbolic x coords={tool1, tool2, tool3, tool4,
        tool5, tool6, tool7},
    xtick=data,
    x tick label style={rotate=45,anchor=east},
    ]
\addplot+[ybar] plot coordinates {(tool1,0) (tool2,2)
  (tool3,2) (tool4,3) (tool5,0) (tool6,2) (tool7,0)};
\addplot+[ybar] plot coordinates {(tool1,0) (tool2,0)
  (tool3,0) (tool4,3) (tool5,1) (tool6,1) (tool7,0)};
\addplot+[ybar] plot coordinates {(tool1,6) (tool2,6)
  (tool3,8) (tool4,2) (tool5,6) (tool6,5) (tool7,6)};
\addplot+[ybar] plot coordinates {(tool1,4) (tool2,2)
  (tool3,0) (tool4,2) (tool5,3) (tool6,2) (tool7,4)};
\legend{never, rarely, sometimes, often}
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[stack plots=x,/tikz/xbar]
    \addplot coordinates
        {(1,0) (2,1) (2,2) (3,3)};
    \addplot coordinates
        {(1,0) (2,1) (2,2) (3,3)};
    \addplot coordinates
        {(1,0) (2,1) (2,2) (3,3)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}

\begin{tikzpicture}
    \begin{axis}[xbar stacked]
    \addplot coordinates
        {(1,0) (2,1) (2,2) (3,3)};
    \addplot coordinates
        {(1,0) (2,1) (2,2) (3,3)};
    \addplot coordinates
        {(1,0) (2,1) (2,2) (3,3)};
    \end{axis}
\end{tikzpicture}
```

The current implementation for `stack plots` does *not* interpolate missing coordinates. That means stacking will fail if the plots have different grids.

`/pgfplots/stack dir=plus|minus` (initially `plus`)

Configures the direction of `stack plots`. The value `plus` will add coordinates of successive plots while `minus` subtracts them.

`/pgfplots/reverse stacked plots=true|false` (initially `true, default true`)

Configures the sequence in which stacked plots are drawn. This is more or less a technical detail which should not be changed in any normal case.

The motivation is as follows: suppose multiple `\addplot` commands are stacked on top of each other and they are processed in the order of appearance. Then, the second plot could easily draw its lines (or fill area) on top of the first one - hiding its marker or line completely. Therefor, PGFPLOTS reverses the sequence of drawing commands.

This has the side-effect that any normal Ti*k*Z-paths inside of an axis will also be processed in reverse sequence.

`/pgfplots/xbar stacked=plus|minus` (style, default `plus`)

A figure-wide style which enables stacked horizontal bars (i.e. `xbar` and `stack plots=x`). It also adjusts the legend and tick appearance and assigns a useful `cycle list`.

`/pgfplots/ybar stacked=plus|minus` (style, default `plus`)

A figure-wide style which enables stacked vertical bars (i.e. `ybar` and `stack plots=y`). It also adjusts the legend and tick appearance and assigns a useful `cycle list`.

`/pgfplots/xbar interval stacked=plus|minus` (style, default `plus`)

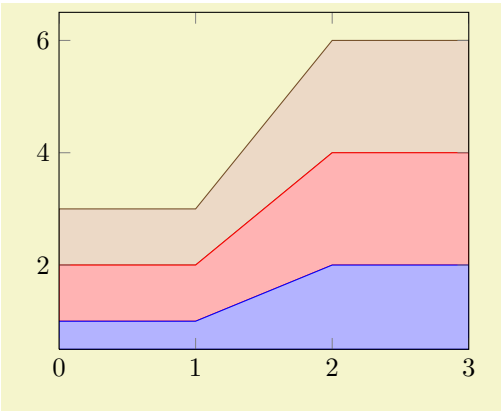
A style similar to `/pgfplots/xbar stacked` for the interval based bar plot variant.

`/pgfplots/ybar interval stacked=plus|minus` (style, default `plus`)

A style similar to `/pgfplots/ybar stacked` for the interval based bar plot variant.

4.4.9 Area Plots

Area plots are a combination of `\closedcycle` and `stack plots`. They can be combined with any other plot type.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        stack plots=y,
        area style,
        enlarge x limits=false]
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \end{axis}
\end{tikzpicture}
```

Area plots may need modified legends, for example using the `area legend` key. Furthermore, one may want to consider the `axis on top` key such that filled areas do not overlap ticks and grid lines.

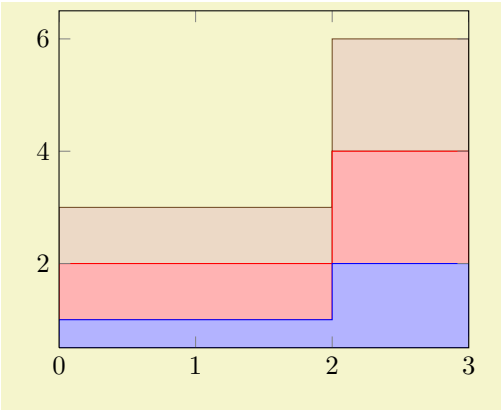
/pgfplots/area style (style, no value)

A style which sets

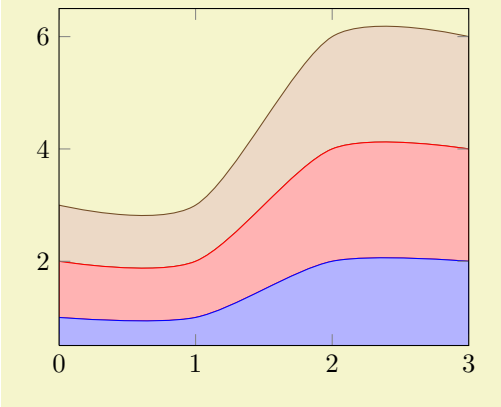
```
\pgfplotsset{
    /pgfplots/area style/.style={%
        area cycle list,
        area legend,
        axis on top,
    }}
```

/pgfplots/area cycle list (style, no value)

A style which installs a `cycle list` suitable for area plots. The initial configuration of this style simply invokes the `bar cycle list` which does also provide filled plot styles.



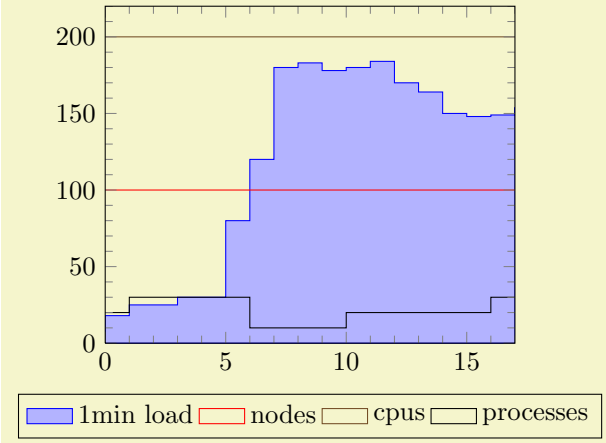

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        const plot,
        stack plots=y,
        area style,
        enlarge x limits=false]
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \end{axis}
\end{tikzpicture}
```



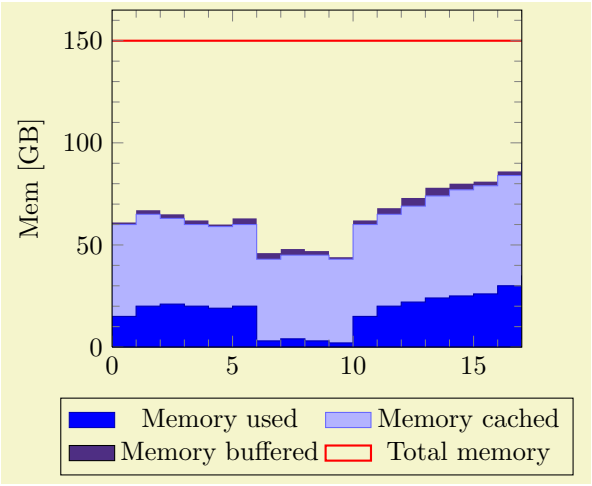

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        smooth,
        stack plots=y,
        area style,
        enlarge x limits=false]
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \addplot coordinates
        {(0,1) (1,1) (2,2) (3,2)}
        \closedcycle;
    \end{axis}
\end{tikzpicture}
```

time	1minload	nodes	cpus	processes	memused	memcached	membuf	memtotal
0	18	100	200	20	15	45	1	150
1	25	100	200	30	20	45	2	150
2	25	100	200	30	21	42	2	150
3	30	100	200	30	20	40	2	150
4	30	100	200	30	19	40	1	150
5	80	100	200	30	20	40	3	150
6	120	100	200	10	3	40	3	150
7	180	100	200	10	4	41	3	150
8	183	100	200	10	3	42	2	150
9	178	100	200	10	2	41	1	150
10	180	100	200	20	15	45	2	150
11	184	100	200	20	20	45	3	150
12	170	100	200	20	22	47	4	150
13	164	100	200	20	24	50	4	150
14	150	100	200	20	25	52	3	150
15	148	100	200	20	26	53	2	150
16	149	100	200	30	30	54	2	150
17	154	100	200	30	35	55	1	150

```
\pgfplotstableread{pgfplots.timeseries.dat}\loadedtable
\pgfplotstabletypeset\loadedtable
```



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotstableread
    {pgfplots.timeseries.dat}
    {\loadedtable}

\begin{tikzpicture}
    \begin{axis}[
        ymin=0,
        minor tick num=4,
        enlarge x limits=false,
        axis on top,
        every axis plot post/.append style=
            {mark=none},
        const plot,
        legend style={
            area legend,
            at={(0.5,-0.15)},
            anchor=north,
            legend columns=-1}]

    \addplot[draw=blue,fill=blue!30!white]
     table[x=time,y=1minload] from \loadedtable
        \closedcycle;
    \addplot table[x=time,y=nodes] from \loadedtable;
    \addplot table[x=time,y=cpus] from \loadedtable;
    \addplot table[x=time,y=processes]
        from \loadedtable;
    \legend{1min load,nodes,cpus,processes}
    \end{axis}
\end{tikzpicture}
```



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotstableread{pgfplots.timeseries.dat}\loadedtable

\begin{tikzpicture}
    \begin{axis}[
        ymin=0,
        minor tick num=4,
        enlarge x limits=false,
        const plot,
        axis on top,
        stack plots=y,
        cycle list={%
            {blue!70!black,fill=blue},%
            {blue!60!white,fill=blue!30!white},%
            {draw=none,fill={rgb:red,138;green,82;blue,232}},%
            {red,thick}%
        },
        ylabel={Mem [GB]},
        legend style={
            area legend,
            at={(0.5,-0.15)},
            anchor=north,
            legend columns=2}]

    \addplot table[x=time,y=memused]      from \loadedtable \closedcycle;
    \addplot table[x=time,y=memcached]    from \loadedtable \closedcycle;
    \addplot table[x=time,y=membuf]       from \loadedtable \closedcycle;
    \addplot+[stack plots=false]
             table[x=time,y=memtotal]     from \loadedtable;
    \legend{Memory used,Memory cached,Memory buffered,Total memory}
    \end{axis}
\end{tikzpicture}
```

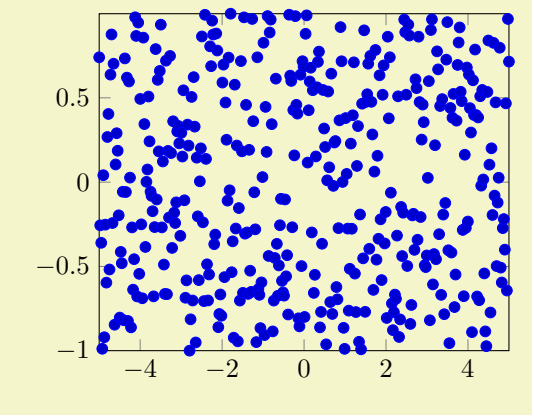
4.4.10 Scatter Plots

The most simple scatter plots produce the same as the line plots above – but they contain only markers. They are enabled using the `only marks` key of Ti*k*Z.

`/tikz/only marks` (no value)

`\addplot+[only marks]`

Draws a simple scatter plot: all markers have the same appearance.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[enlargelimits=false]
    \addplot+[only marks,samples=400]
        {rand};
    \end{axis}
\end{tikzpicture}
```

The `only marks` visualization style simply draws marks at every coordinate. Marks can be set with `mark`=⟨*mark name*⟩ and marker options like size and color can be specified using the `mark options`=⟨*style options*⟩ key (or by modifying the `every mark` style). The available markers along with the accepted style options can be found in Section 4.6 on page 116.

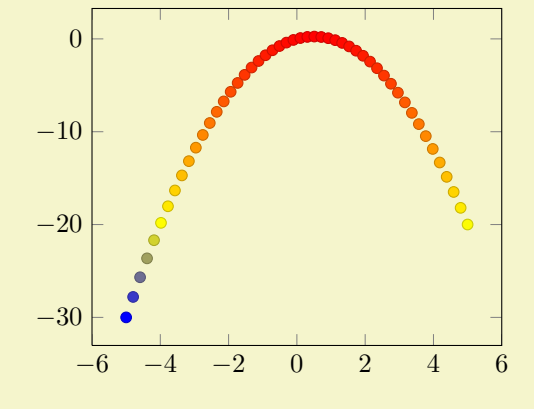
More sophisticated scatter plots change the marker appearance for each data point. An example is that marker colors depend on the magnitude of function values $f(x)$ or other provided coordinates. The term "scatter plot" will be used for this type of plot in the following sections.

Scatter plots require "source" coordinates. These source coordinates can be the y coordinate, or explicitly provided additional values.

/pgfplots/scatter (no value)

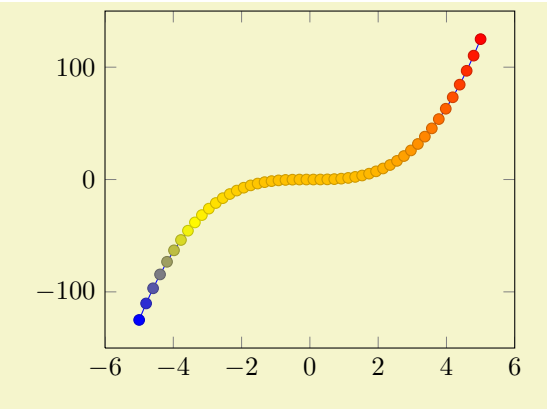
\addplot+[scatter]

Enables marker appearance modifications. The default implementation acquires “source coordinates” for every data point (see scatter src below) and maps them linearly into the current color map. The resulting color is used as draw and fill color of the marker.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot+[scatter,only marks,
         samples=50,scatter src=y]
        {x-x^2};
    \end{axis}
\end{tikzpicture}
```

The key scatter is simply a boolean variable which enables marker modifications. It applies only to markers and it can be combined with any other plot type.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot+[scatter,
         samples=50,scatter src=y]
        {x^3};
    \end{axis}
\end{tikzpicture}
```

Scatter plots can be configured using a set of options. One of them is mandatory, the rest allows fine grained control over marker appearance options.

/pgfplots/scatter src=none|⟨*expression*⟩|x|y|z|f(x)|explicit|explicit symbolic (initially none)

This key is necessary for any scatter plot and it is set to f(x) as soon as scatter is activated and no different choice has been made. It needs to be provided as ⟨*option*⟩ for \addplot to configure the value used to determine marker appearances. Actually, scatter src is nothing but an alias for point meta, so the main documentation for this key is on page 138. However, we summarize the choices here together with scatter plot examples.

Usually, scatter src provides input data (numeric or string type) which is used to determine colors and other style options for markers. The default configuration expects numerical data which is mapped linearly into the current color map.

The value of scatter src determines how to get this data: the choices x, y and z will use either the x, y or z coordinates to determine marker options. Any coordinate filters, logarithms or stacked-plot computations have already been applied to these values (use rawx, rawy and rawz for unprocessed values). The special choice f(x) is the same as y for two dimensional plots and the same as z for three dimensional plots. The choice explicit expects the scatter source data as additional coordinate from the coordinate input streams (see Section 4.2.1 for how to provide input meta data or below for some small examples). They will be treated as numerical data. The choice explicit symbolic also expects scatter source data as additional meta information for each input coordinate, but it treats them as

strings, not as numerical data. Consequently, no arithmetics is performed. It is the task of the scatter plot style to do something with it. See, for example, the `scatter/classes` style below. Finally, it is possible to provide an arbitrary mathematical expression which involves zero, one or more of the values `x` (the current x coordinate), `y` (the current y coordinate) or `z` (the current z coordinate, if any).

If data is read from tables, mathematical expressions might also involve `\thisrow{`⟨*column name*⟩`}` or `\thisrowno{`⟨*column index*⟩`}` to access any of the table cells in the current row.

Here are examples for how to provide data for the choices `explicit` and `explicit symbolic`.

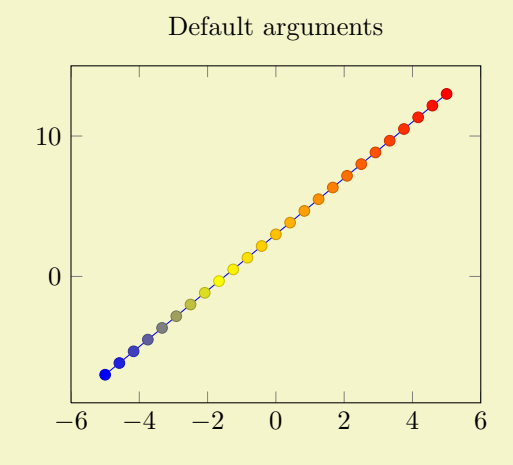
```
\begin{tikzpicture}
    \begin{axis}
        %  provide color data explicitly using [<data>]
        %  behind coordinates:
        \addplot+[scatter,scatter src=explicit]
            coordinates {
                (0,0) [1.0e10]
                (1,2) [1.1e10]
                (2,3) [1.2e10]
                (3,4) [1.3e10]
                %  ...
            };

        %  Assumes a datafile.dat like
        %  xcolname  ycolname    colordata
        %  0         0           0.001
        %  1         2           0.3
        %  2         2.1         0.4
        %  3         3           0.5
        %  ...
        %  the file may have more columns.
        \addplot+[scatter,scatter src=explicit]
            table[x=xcolname,y=ycolname,meta=colordata]
                {datafile.dat};
        %  Same data as last example:
        \addplot+[scatter,scatter src=\thisrow{colordata}+\thisrow{ycolname}]
            table[x=xcolname,y=ycolname]
                {datafile.dat};

        %  Assumes a datafile.dat like
        %  0         0           0.001
        %  1         2           0.3
        %  2         2.1         0.4
        %  3         3           0.5
        %  ...
        %  the first three columns will be used here:
        \addplot+[scatter,scatter src=explicit]
            file {datafile.dat};
    \end{axis}
\end{tikzpicture}
```

Please note that `scatter src`≠`none` results in computational work even if `scatter=false`.

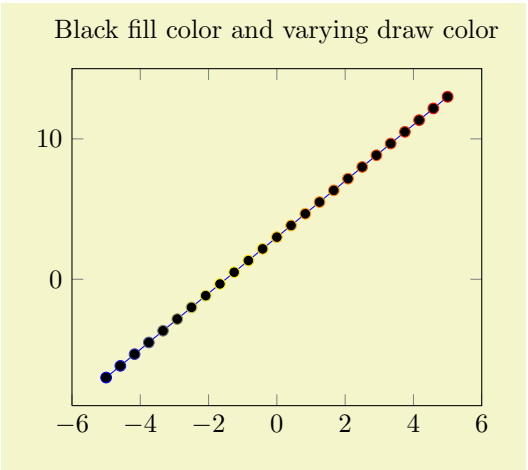
`/pgfplots/scatter/use mapped color=`{⟨*options for each marker*⟩} (style, initially `draw=mapped color!80!black,fill=mapped color`)

This style is installed by default. When active, it recomputes the color `mapped color` for every processed point coordinate by transforming the `scatter src` coordinates into the current color map linearly. Then, it evaluates the options provided as ⟨*options for each marker*⟩ which are expected to depend on `mapped color`.

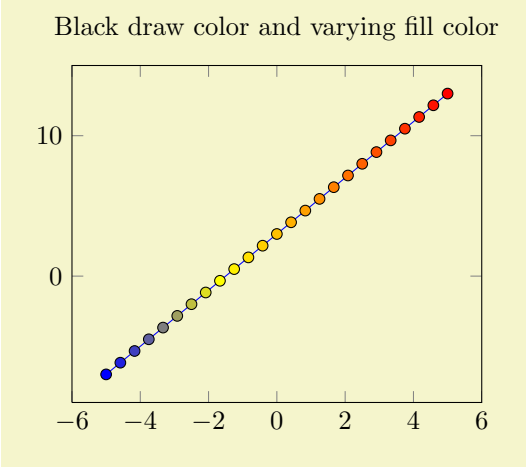
The user interface for color maps is described in Section 4.6.6.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[title=Default arguments]
\addplot+[scatter,scatter src=y]
    {2*x+3};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title=Black fill color and varying draw color,
    scatter/use mapped color=
        {draw=mapped color,fill=black}]
\addplot+[scatter,scatter src=y]
    {2*x+3};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title=Black draw color and varying fill color,
    scatter/use mapped color=
        {draw=black,fill=mapped color}]
\addplot+[scatter,scatter src=y]
    {2*x+3};
\end{axis}
\end{tikzpicture}
```

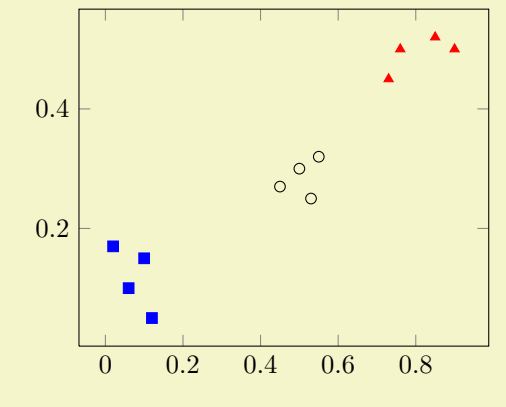
This key is actually a style which redefines `@pre marker code` and `@post marker code` (see below).

Remark: The style `use mapped color` *re*defines `@pre marker code` and `@post marker code`. There is a starred variant `use mapped color*` which *appends* the functionality while keeping the old marker code.

`/pgfplots/scatter/classes={`⟨*styles for each class name*⟩`}`

A scatter plot style which visualizes points using several classes. The style assumes that every point coordinate has a class label attached, that means the choice `scatter src=explicit symbolic` is as-

sumed[17]. A class label can be a number, but it can also be a symbolic constant. Given class labels for every point, ⟨*styles for each class name*⟩ contains a comma-separated list which associates appearance options to each class label.



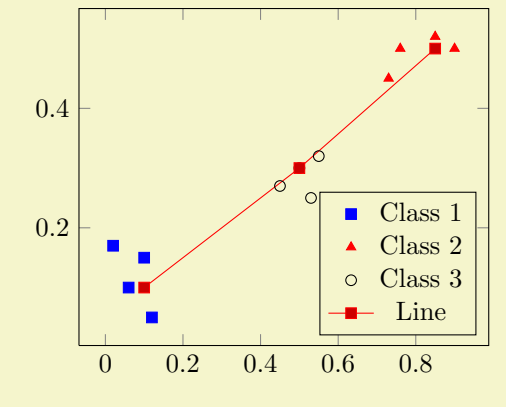

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[scatter/classes={
    a={mark=square*,blue},%
    b={mark=triangle*,red},%
    c={mark=o,draw=black}}]

    %  \addplot[] is better than \addplot+[] here:
    %  it avoids scalings of the cycle list
    \addplot[scatter,only marks,
        scatter src=explicit symbolic]
        coordinates {
            (0.1,0.15)  [a]
            (0.45,0.27) [c]
            (0.02,0.17) [a]
            (0.06,0.1)  [a]
            (0.9,0.5)   [b]
            (0.5,0.3)   [c]
            (0.85,0.52) [b]
            (0.12,0.05) [a]
            (0.73,0.45) [b]
            (0.53,0.25) [c]
            (0.76,0.5)  [b]
            (0.55,0.32) [c]
        };
\end{axis}
\end{tikzpicture}
```

In this example, the coordinate `(0.1,0.15)` has the associated label '`a`' while `(0.45,0.27)` has the label '`c`' (see Section 4.2 for details about specifying point meta data). Now, the argument to `scatter/classes` contains styles for every label – for label '`a`', square markers will be drawn in color blue.

The generation of a legend works as for a normal plot – but `scatter/classes` requires one legend entry for every provided class. It communicates the class labels to the legend automatically. It works as if there had been different `\addplot` commands, one for every class label.

It is also possible to provide `scatter/classes` as argument to a single plot, allowing different scatter plots in one axis.

[17] If `scatter src` is not `explicit symbolic`, we expect a numeric argument which is rounded to the nearest integer. The resulting integer is used a class label. If that fails, the numeric argument is truncated to the nearest integer. If that fails as well, the point has no label.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend pos=south east]
    %  The data file contains:
    %  x     y      label
    %  0.1   0.15   a
    %  0.45  0.27   c
    %  0.02  0.17   a
    %  0.06  0.1    a
    %  0.9   0.5    b
    %  0.5   0.3    c
    %  0.85  0.52   b
    %  0.12  0.05   a
    %  0.73  0.45   b
    %  0.53  0.25   c
    %  0.76  0.5    b
    %  0.55  0.32   c
    \addplot[
        %  clickable coords={\thisrow{label}},
        scatter/classes={
            a={mark=square*,blue},%
            b={mark=triangle*,red},%
            c={mark=o,draw=black,fill=black}%
        },
        scatter,only marks,
        scatter src=explicit symbolic]
    table[x=x,y=y,meta=label]
            {plotdata/scattercl.dat};

    \addplot coordinates
        {(0.1,0.1) (0.5,0.3) (0.85,0.5)};
    \legend{Class 1,Class 2,Class 3,Line}
\end{axis}
\end{tikzpicture}
```

In general, the format of ⟨*styles for each class name*⟩ is a comma separated list of ⟨*label*⟩={⟨*style options*⟩}.

Attention: The keys `every mark` and `mark options` have *no effect* when used inside of ⟨*styles for each class name*⟩! So, instead of assigning `mark options`, you can simply provide the options directly. They apply only to markers anyway.

Remark: To use `\label` and `\ref` in conjunction with `scatter/classes`, you can provide the class labels as optional arguments to `\label` in square brackets:

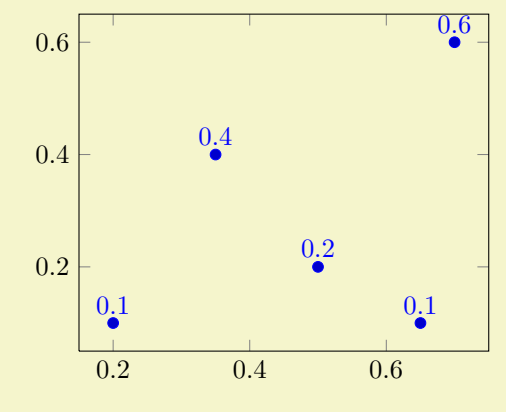
```
\addplot[
    scatter/classes={
        a={mark=square*,blue},%
        b={mark=triangle*,red},%
        c={mark=o,draw=black,fill=black}%
    },
    scatter,only marks,
    scatter src=explicit symbolic]
    %  [and coordinate input here... ]
    ;

\label[a]{label:for:first:class}
\label[b]{label:for:second:class}
\label[c]{label:for:third:class}

...
First class is \ref{label:for:first:class}, second is \ref{label:for:second:class}.
```

Remark: It is possible to click into the plot to display labels with mouse popups, see the `clickable coords` key of the `clickable` library.

Remark: The style scatter/classes *re*defines `@pre marker code` and `@post marker code`. There is a starred variant scatter/classes* which *appends* the functionality while keeping the old marker code.

/pgfplots/nodes near coords={⟨*content*⟩} (default \pgfmathprintnumber\pgfplotspointmeta)
/pgfplots/nodes near coords*={⟨*content*⟩} (default \pgfmathprintnumber\pgfplotspointmeta)

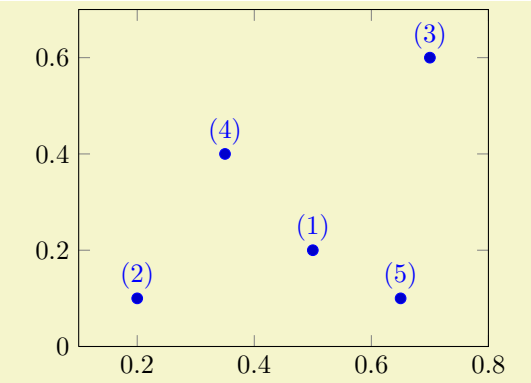
A scatter plot style which places text nodes near every coordinate.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords]
    \addplot+[only marks] coordinates {
        (0.5,0.2) (0.2,0.1) (0.7,0.6)
        (0.35,0.4) (0.65,0.1)};
\end{axis}
\end{tikzpicture}
```

The ⟨*content*⟩ is, if nothing else has been specified, the content of the "point meta", displayed using the default ⟨*content*⟩=\pgfmathprintnumber{\pgfplotspointmeta}. The macro \pgfplotspointmeta contains whatever has been selected by the point meta key, it defaults to the y coordinate for two dimensional plots and the z coordinate for three dimensional plots.

Since point meta=explicit symbolic allows to treat string data, you can provide textual descriptions which will be shown inside of the generated nodes[18]:

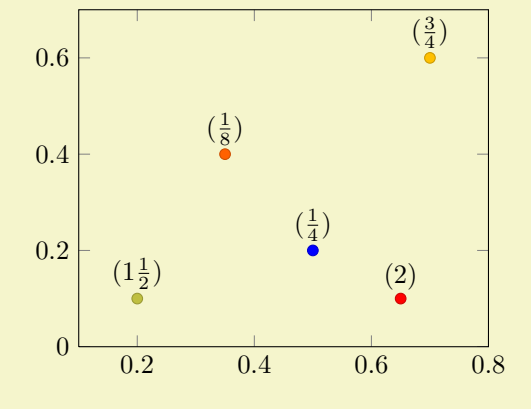



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords,enlargelimits=0.2]
    \addplot+[only marks,
        point meta=explicit symbolic]
    coordinates {
        (0.5,0.2) [(1)]
        (0.2,0.1) [(2)]
        (0.7,0.6) [(3)]
        (0.35,0.4) [(4)]
        (0.65,0.1) [(5)]
    };
\end{axis}
\end{tikzpicture}
```

The square brackets are the way to provide explicit point meta for plot coordinates. Please refer to the documentation of plot file and plot table for how to get point meta from files.

The ⟨*content*⟩ can also depend on something different than \pgfplotspointmeta. But since ⟨*content*⟩ is evaluated during \end{axis}, PGFPLOTS might not be aware of any special information inside of ⟨*content*⟩ – you'll need to communicate it to PGFPLOTS with the visualization depends on key as follows:

[18] In this case, the \pgfmathprintnumber will be skipped automatically.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[enlargelimits=0.2]
        \addplot[
          scatter,mark=*,only marks,
          %  we use 'point meta' as color data...
          point meta=\thisrow{color},
          %  ... therefore, we can't use it as argument for nodes near coords ...
          nodes near coords*={$(\pgfmathprintnumber[frac]\myvalue)$},
          %  ... which requires to define a visualization dependency:
          visualization depends on={\thisrow{myvalue} \as \myvalue},
        ]
        table {
            x      y    color   myvalue
            0.5    0.2  1       0.25
            0.2    0.1  2       1.5
            0.7    0.6  3       0.75
            0.35   0.4  4       0.125
            0.65   0.1  5       2
        };
    \end{axis}
\end{tikzpicture}
```

The example uses a `scatter` plot to get different colors, where the `scatter src` (or, equivalently, `point meta`) is already used to define the markers color. In addition to the colored `scatter` plot, we'd like to add `nodes near coords`, where the displayed nodes should contain `\thisrow{myvalue}`. To do so, we define `scatter,point meta=\thisrow{color}` (just as described in the previous sections). Furthermore, we use nodes near coords* in order to *combine* different `scatter` styles (see below for details). The value for `nodes near coords*` depends on `\thisrow{myvalue}`, but we can't use `\pgfplotspointmeta` (which is already occupied). Thus, we communicate the additional input data by means of `visualization depends on={\thisrow{myvalue} \as \myvalue}`. The statement defines a new macro, `\myvalue`, and assigns the value `\thisrow{myvalue}`. Furthermore, it configures PGFPLOTS to remember this particular macro and its contents until `\end{axis}` (see the documentation for `visualization depends on` for details).

The style `nodes near coords` might be useful for bar plots, see `ybar` for an example of `nodes near coords`.

Remarks and Details:

- `nodes near coords` uses the same options for line styles and colors as the current plot. This may be changed using the style `every node near coord`, see below.
- `nodes near coords` is actually one of the `scatter` plot styles. It redefines `scatter/@pre marker code` to generate several Ti*k*Z `\node` commands.

  In order to use `nodes near coords` together with other `scatter` plot styles (like `scatter/use mapped color` or `scatter/classes`), you may append a star to each of these keys. The variant `nodes near coords*` will *append* code to `scatter/@pre marker code` without overwriting the previous value.
- Consider using `enlargelimits` together with `nodes near coords` if text is clipped away.

- Currently nodes near coords does not work satisfactorily for ybar interval or xbar interval, sorry.

/pgfplots/every node near coord (style, no value)

A style used for every node generated by nodes near coords. It is initially empty.

/pgfplots/nodes near coords align={⟨*alignment method*⟩} (initially auto)

Specifies how to align nodes generated by nodes near coords.

Possible choices for ⟨*alignment method*⟩ are

auto uses horizontal if the x coordinates are shown or vertical in all other cases. This checks the current value of point meta.

horizontal uses left if \pgfplotspointmeta < 0 and right otherwise.

vertical uses below if \pgfplotspointmeta < 0 and above otherwise.

It is also possible to provide any TikZ alignment option such as anchor=north east, below or something like that. It is also allowed to provide multiple options.

/pgfplots/scatter/@pre marker code/.code={⟨...⟩}
/pgfplots/scatter/@post marker code/.code={⟨...⟩}

These two keys constitute the public interface which determines the marker appearance depending on scatter source coordinates.

Redefining them allows fine grained control even over marker types, line styles and colors.

The scatter plot algorithm works as follows:

1. The scatter source coordinates form a data stream whose data limits are computed additionally to the axis limits. This step is skipped for symbolic meta data.
2. Before any markers are drawn, a linear coordinate transformation from these data limits to the interval $[0.0, 1000.0]$ is initialised.
3. Every scatter source coordinate[19] will be transformed linearly and the result is available as macro \pgfplotspointmetatransformed $\in [0.0, 1000.0]$.

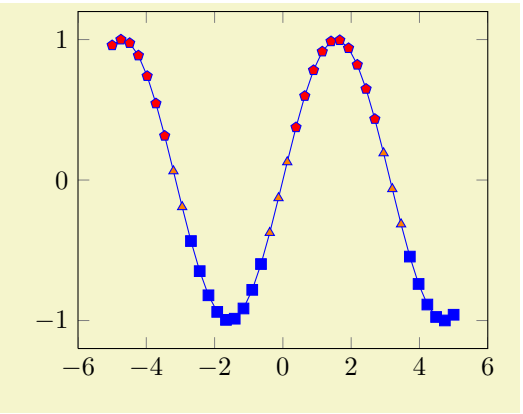
   The decision is thus based on per thousands of the data range. The transformation is skipped for symbolic meta data (and the meta data is simply contained in the mentioned macro).
4. The PGF coordinate system is translated such that (0pt,0pt) is the plot coordinate.
5. The code of scatter/@pre marker code is evaluated (without arguments).
6. The standard code which draws markers is evaluated.
7. The code of scatter/@post marker code is evaluated (without arguments).

The idea is to generate a set of appearance keys which depends on \pgfplotspointmetatransformed. Then, a call to \scope[⟨*generated keys*⟩] as @pre code and the associated \endscope as @post code will draw markers individually using [⟨*generated keys*⟩].

A technical example is shown below. It demonstrates how to write user defined routines, in this case a three–class system[20].

[19] During the evaluation, the public macros \pgfplotspointmeta and \pgfplotspointmetarange indicate the source coordinate and the source coordinate range in the format $a : b$ (for log–axis, they are given in fixed-point representation and for linear axes in floating point).

[20] Please note that you don't need to copy this particular example: the multiple–class example is also available as predefined style scatter/classes.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
%  Low-Level scatter plot interface Example:
%  use three different marker classes
%  0% - 30%   : first class
%  30% - 60%  : second class
%  60% - 100% : third class
\begin{axis}[
scatter/@pre marker code/.code={%
   \ifdim\pgfplotspointmetatransformed pt<300pt
      \def\markopts{mark=square*,fill=blue}%
   \else
      \ifdim\pgfplotspointmetatransformed pt<600pt
         \def\markopts{mark=triangle*,fill=orange}%
      \else
         \def\markopts{mark=pentagon*,fill=red}%
      \fi
   \fi
   \expandafter\scope\expandafter[\markopts]
},%
scatter/@post marker code/.code={%
   \endscope
}]

\addplot+[scatter,scatter src=y,
    samples=40]
   {sin(deg(x))};

\end{axis}
\end{tikzpicture}
```

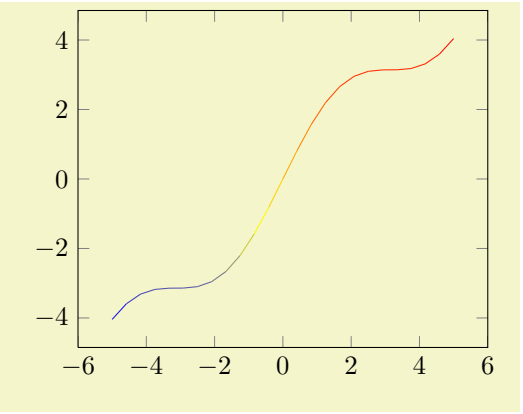
Please note that `\ifdim` compares TEX lengths, so the example employs the suffix `pt` for any number used in this context. That doesn't change the semantics. The two (!) `\expandafter` constructions make sure that `\scope` is invoked with the *content* of `\markopts` instead of the macro name `\markopts`.

4.4.11 1D Colored Mesh Plots

`/pgfplots/mesh` (no value)

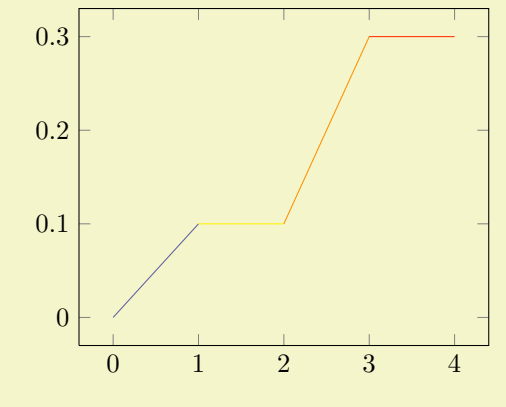
`\addplot+[mesh]`

Uses the current color map to determine colors for each fixed line segment. Each line segment will get the same color.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[mesh] {x+sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

The color data is per default the y value of the plot. It can be reconfigured using the `point meta` key (which is actually the same as `scatter src`). The following example provides the color data explicitly for `plot coordinates`, using the square bracket notation.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[mesh,point meta=explicit]
        coordinates {
            (0,0)   [0]
            (1,0.1) [1]
            (2,0.1) [2]
            (3,0.3) [3]
            (4,0.3) [4]
        };
    \end{axis}
\end{tikzpicture}
```

This one-dimensional `mesh` plot is actually a special case of the twodimensional mesh plots, so more detailed configuration, including how to change the color data, can be found in Section 4.5.5.

4.4.12 Interrupted Plots

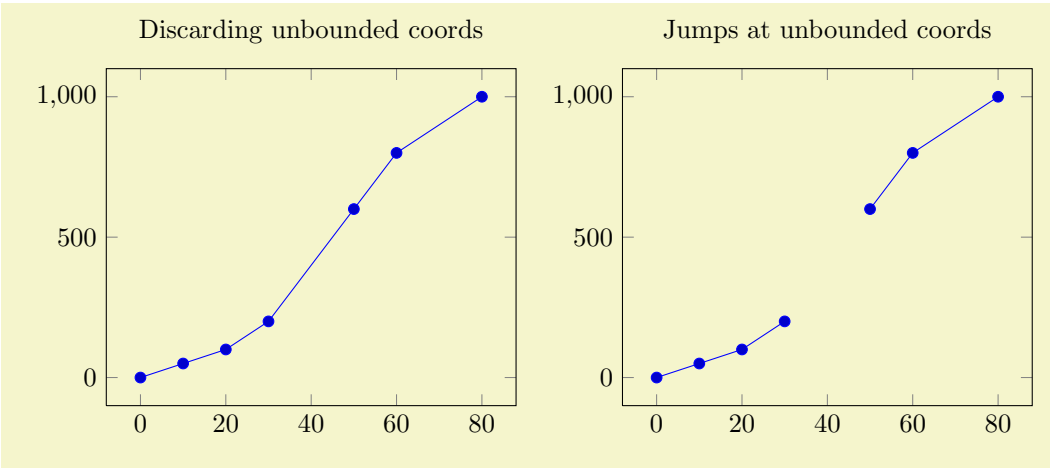
Sometimes it is desirable to draw parts of a single plot separately, without connection between the parts (discontinuities). This can be achieved using the `unbounded coords` key combined with coordinate values `nan`, `inf` or `-inf`.

`/pgfplots/unbounded coords=discard|jump` (initially `discard`)

This key configures what to do if one or more coordinates of a single point are unbounded. Here, unbounded means it is either $\pm\infty$ (`+inf` or `-inf`) or it has the special "not–a–number" value `nan`.

The initial setting `discard` discards the complete point and a warning is issued in the log file[21]. This setting has the same effect as if the unbounded point did not occur: PGFPLOTS will interpolate between the bounded adjacent points.

The alternative `jump` allows interrupted plots: it provides extra checking for these coordinates and does not interpolate over them; only those line segments which are adjacent to unbounded coordinates will be skipped.

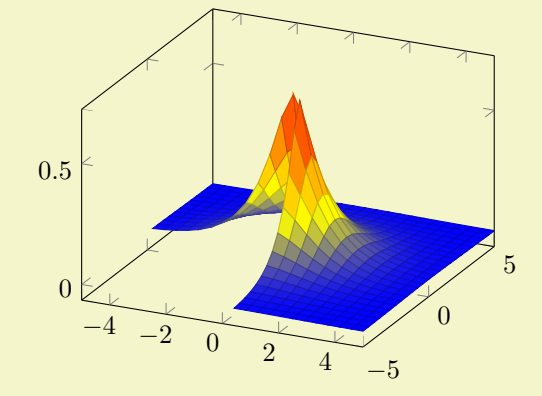


[21] The warning can be disabled with `filter discard warning=false`.

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title=Discarding unbounded coords,
    unbounded coords=discard]

    \addplot coordinates {
        (0,0) (10,50) (20,100) (30,200)
        (40,inf) (50,600) (60,800) (80,1000)
    };
\end{axis}
\end{tikzpicture}
\begin{tikzpicture}
\begin{axis}[
    title=Jumps at unbounded coords,
    unbounded coords=jump]
    \addplot coordinates {
        (0,0) (10,50) (20,100) (30,200)
        (40,inf) (50,600) (60,800) (80,1000)
    };
\end{axis}
\end{tikzpicture}
```

For plot expression and its friends, it is more likely to get very large floating point numbers instead of `inf`. In this case, consider using the `restrict x to domain` key described on page 272.

The `unbounded coords=jump` method does also work for mesh/surface plots: every face adjacent to an unbounded coordinate will be discarded in this case. The following example sets up a (cryptic) coordinate filter which cuts out a quarter of the domain and replaces its values with `nan`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
  unbounded coords=jump,
  %  A technical filter to cut out
  %  the x<0 and y<0 edge.
  filter point/.code={%
    \pgfmathparse
      {\pgfkeysvalueof{/data point/x}<0}%
    \ifpgfmathfloatcomparison
      \pgfmathparse
        {\pgfkeysvalueof{/data point/y}<0}%
      \ifpgfmathfloatcomparison
        \pgfkeyssetvalue{/data point/x}{nan}%
      \fi
    \fi
  },
  ]
  \addplot3[surf] {exp(-sqrt(x^2 + y^2))};
\end{axis}
\end{tikzpicture}
```

More about this coordinate filtering can be found in Section 4.21 “Skipping Or Changing Coordinates – Filters”.

4.4.13 Patch Plots

Patch Plots visualize a sequence of one or more triangles (or other sorts of patches). These triangles can be drawn with a single color (`shader=flat` and `shader=faceted interp`) or with interpolated colors (`shader=interp`).

There are both two- and three-dimensional patch plots, both with the same interface and the same keys. Therefore, the reference documentation for patch plots can be found in Section 4.5.11 together with three–dimensional patch plots.

4.5 Three Dimensional Plot Types

PGFPLOTS provides three dimensional visualizations like scatter, line, mesh or surface plots. This section explains the methods to provide input coordinates and how to use the different plot types.

4.5.1 Before You Start With 3D

Before we delve into the capabilities of PGFPLOTS for three dimensional visualization, let me start with some preliminary remarks. The reason to use PGFPLOTS for three dimensional plots are similar to those of normal, two dimensional plots: the possibility to get consistent fonts and document consistent styles combined with high–quality output.

While this works very nice for (not too complex) two dimensional plots, it requires considerably more effort than non–graphical documents. This is even more so for three dimensional plots. In other words: PGFPLOTS' three dimensional routines are slow. There are reasons for this and some of them may vanish in future versions. But one of these reasons is that TEX has never been designed for complex visualisation techniques. Consider the image externalization routines mentioned in Section 7.1, in particular the `external` library to reduce typesetting time. Besides the speed limitations, three dimensional plots reach memory limits easily. Therefore, the plot complexity of three dimensional plots is limited to relatively coarse resolutions. Section 7.1 also discusses methods to extend the initial TEX memory limits.

Another issue which arises in three dimensional visualization is depth. PGFPLOTS supports z buffering techniques up to a certain extend. It works pretty well for single scatter plots (`z buffer=sort`), mesh or surface plots (`z buffer=auto`) or parametric mesh and surface plots (`z buffer=sort`). However, it can't combine different `\addplot` commands, those will be drawn in the order of appearance. You may encounter the limitations sometimes. Maybe it will be improved in future versions.

If you decide that you need high complexity, speed and 100% reliable z buffers (depth information), you should consider using other visualization tools and return to PGFPLOTS in several years. If you can wait for a complex picture and you don't even see the limitations arising from z buffering limitations, you should use PGFPLOTS. Again, consider using the automatic picture externalization with the `external` library discussed in Section 7.1.

Enough for now, let's continue.

4.5.2 The `\addplot3` Command: Three Dimensional Coordinate Input

`\addplot3[`⟨*options*⟩`]` ⟨*input data*⟩ ⟨*trailing path commands*⟩`;`

The `\addplot3` command is the main interface for any three dimensional plot. It works in the same way as its two dimensional variant `\addplot` which has been described in all detail in Section 4.2 on page 22.

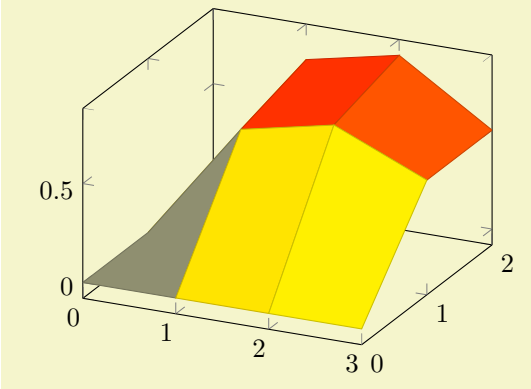
The `\addplot3` command accepts the same input methods as the `\addplot` variant, including expression plotting, coordinates, files and tables. However, a third coordinate is necessary for each of these methods which is usually straight–forward and is explained in all detail in the following.

Furthermore, `\addplot3` has a way to decide whether a *line* visualization or a *mesh* visualization has to be done. The first one is a map from one dimension into $\mathbb{R}^3$ and the latter one a map from two dimensions to $\mathbb{R}^3$. Here, the keys `mesh/rows` and `mesh/cols` are used to define mesh sizes (matrix sizes). Usually, you don't have to care about that because the coordinate input routines already allow either one- or two-dimensional structure.

`\addplot3 coordinates {`⟨*coordinate list*⟩`};`
`\addplot3[`⟨*options*⟩`] coordinates {`⟨*coordinate list*⟩`}` ⟨*trailing path commands*⟩`;`

The `\addplot3 coordinates` method works like its two–dimensional variant, `\addplot coordinates` which is described in all detail on page 24:

A long list of coordinates `(`⟨x⟩`,`⟨y⟩`,`⟨z⟩`)` is expected, separated by white spaces. The input list can be either an unordered series of coordinates, for example for scatter or line plots. It can also have matrix structure, in which case an empty input line (which is equivalent to "`\par`") marks the end of one matrix row. Matrix structure can also be provided if one of `mesh/rows` or `mesh/cols` is provided explicitly.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        %  this yields a 3x4 matrix:
        \addplot3[surf] coordinates {
            (0,0,0) (1,0,0)   (2,0,0)   (3,0,0)

            (0,1,0) (1,1,0.6) (2,1,0.7) (3,1,0.5)

            (0,2,0) (1,2,0.7) (2,2,0.8) (3,2,0.5)
        };
    \end{axis}
\end{tikzpicture}
```

Here, `\addplot3` reads a matrix with three rows and four columns. The empty lines separate one row from the following.

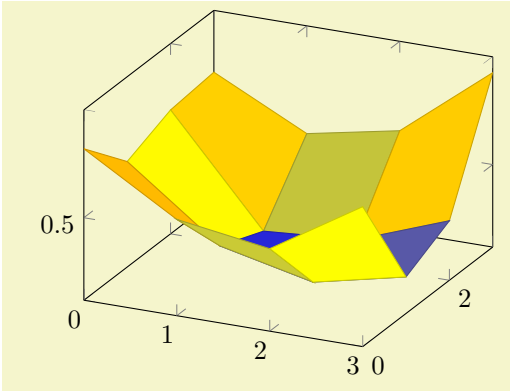
As for the two–dimensional `plot coordinates`, it is possible to provide (constant) mathematical expressions inside of single coordinates. The syntax `(⟨x⟩,⟨y⟩,⟨z⟩) [⟨meta⟩]` can be used just as for two dimensional `plot coordinates` to provide explicit color data; error bars are also supported.

`\addplot3 file {⟨name⟩};`
`\addplot3[⟨options⟩] file {⟨name⟩} ⟨trailing path commands⟩;`

The `\addplot3 file` input method is the same as `\addplot file` – it only expects one more coordinate. Thus, the input file contains x_i in the first column, y_i in the second column and z_i in the third.

A further column is read after z_i if `point meta=explicit` has been requested, see the documentation of `\addplot file` on page 26 for details.

As for `\addplot3 coordinates`, an empty line in the file marks the end of one matrix row.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        %  We have 'plotdata/first3d.dat' with
        % ---------
        %  0 0 0.8
        %  1 0 0.56
        %  2 0 0.5
        %  3 0 0.75
        %
        %  0 1 0.6
        %  1 1 0.3
        %  2 1 0.21
        %  3 1 0.3
        %
        %  0 2 0.68
        %  1 2 0.22
        %  2 2 0.25
        %  3 2 0.4
        %
        %  0 3 0.7
        %  1 3 0.5
        %  2 3 0.58
        %  3 3 0.9
        %  -> yields a 4x4 matrix:
        \addplot3[surf] file {plotdata/first3d.dat};
    \end{axis}
\end{tikzpicture}
```

For matrix data in files, it is important to specify the ordering in which the matrix entries have been written. The default configuration is `mesh/ordering=x varies`, so you need to change it to `mesh/ordering=y varies` in case you have columnwise ordering.

`\addplot3 table [⟨column selection⟩]{⟨file⟩};`
`\addplot3[⟨options⟩] table [⟨column selection⟩]{⟨file⟩} ⟨trailing path commands⟩;`

The `\addplot3 table` input works in the same way as its two dimensional counterpart `\addplot table`. It only expects a column for the z coordinates. Furthermore, it interprets empty input lines as end–

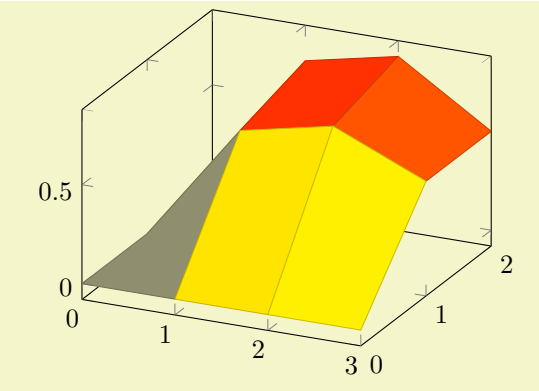
of–row (more generally, end–of–scanline) markers, just as for `plot file`. The remark above about the `mesh/ordering` applies here as well.

`/pgfplots/mesh/rows={`⟨*integer*⟩`}`
`/pgfplots/mesh/cols={`⟨*integer*⟩`}`

For visualization of mesh or surface plots which need some sort of matrix input, the dimensions of the input matrix need to be known in order to visualize the plots correctly. The matrix structure may be known from end–of–row marks (empty lines as general end–of–scanline markers in the input stream) as has been described above.

If the matrix structure is not yet known, it is necessary to provide at least one of `mesh/rows` or `mesh/cols` where `mesh/rows` indicates the number of samples for y coordinates whereas `mesh/cols` is the number of samples used for x coordinates (see also `mesh/ordering`).

Thus, the following example is also a valid method to define an input matrix.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        %  this yields also a 3x4 matrix:
        \addplot3[surf,mesh/rows=3] coordinates {
            (0,0,0) (1,0,0)   (2,0,0)   (3,0,0)
            (0,1,0) (1,1,0.6) (2,1,0.7) (3,1,0.5)
            (0,2,0) (1,2,0.7) (2,2,0.8) (3,2,0.5)
        };
    \end{axis}
\end{tikzpicture}
```

It is enough to supply one of `mesh/rows` or `mesh/cols` – the missing value will be determined automatically.

If you provide one of `mesh/rows` or `mesh/cols`, any end–of–row marker seen inside of input files or coordinate streams will be ignored.

`/pgfplots/mesh/scanline verbose=true|false` (initially `false`)

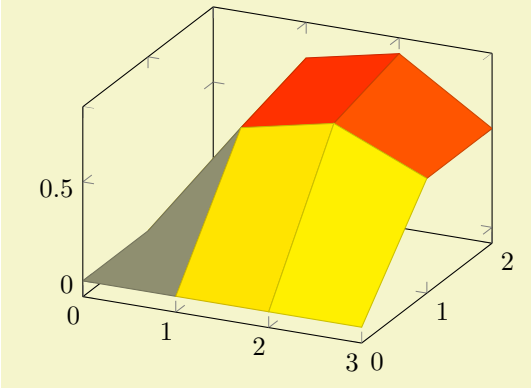
Provides debug messages in the LaTeX output about end–of–scanline markers.

The message will tell whether end–of–scanlines have been found and if they are the same.

`/pgfplots/mesh/ordering=x varies|y varies|rowwise|colwise` (initially `x varies`)

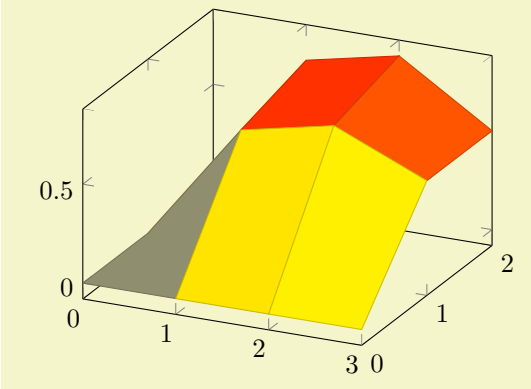
Allows to configure the sequence in which matrices (meshes) are read from `\addplot3 coordinates`, `\addplot3 file` or `\addplot3 table`.

Here, `x varies` means a sequence of points where n=`mesh/cols` successive points have the y coordinate fixed. This is intuitive when you write down a function because x is horizontal and y vertical. Note that in matrix terminology, x refers to *column indices* whereas y refers to *row indices*. Thus, `x varies` is equivalent to `rowwise` ordering in this sense. This is the initial configuration.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[mesh/ordering=x varies]
    %  this yields a 3x4 matrix in 'x varies'
    %  ordering:
    \addplot3[surf] coordinates {
        (0,0,0) (1,0,0)   (2,0,0)   (3,0,0)

        (0,1,0) (1,1,0.6) (2,1,0.7) (3,1,0.5)

        (0,2,0) (1,2,0.7) (2,2,0.8) (3,2,0.5)
    };
\end{axis}
\end{tikzpicture}
```

Consequently, mesh/ordering=y varies provides points such that successive m=mesh/rows points form a column, i.e. the x coordinate is fixed and the y coordinate changes. In this sense, y varies is equivalent to colwise ordering, it is actually a matrix transposition.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[mesh/ordering=y varies]
    %  this yields a 3x4 matrix in colwise ordering:
    \addplot3[surf] coordinates {
        (0,0,0) (0,1,0)   (0,2,0)

        (1,0,0) (1,1,0.6) (1,2,0.7)

        (2,0,0) (2,1,0.7) (2,2,0.8)

        (3,0,0) (3,1,0.5) (3,2,0.5)
    };
\end{axis}
\end{tikzpicture}
```

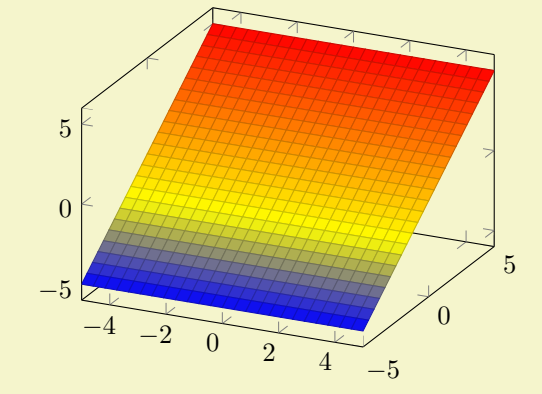
Again, note the subtle difference to the common matrix indexing where a column has the second index fixed. PGFPLOTS refers to the way one would write down a function on a sheet of paper (this is consistent with how Matlab (®) displays discrete functions with matrices).

\addplot3 {⟨*math expression*⟩} ;
\addplot3[⟨*options*⟩] {⟨*math expression*⟩} ⟨*trailing path commands*⟩;

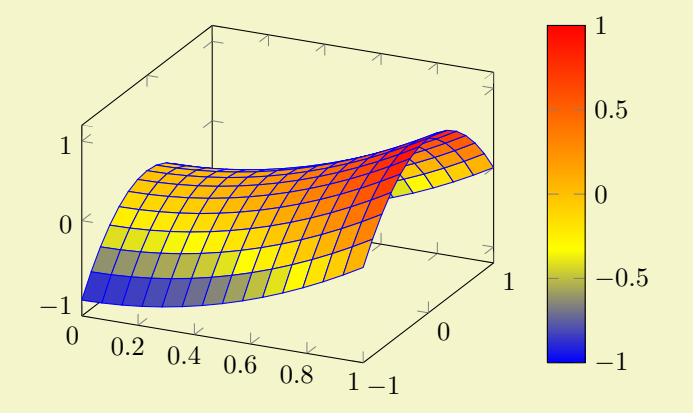
Expression plotting also works in the same way as for two dimensional plots. Now, however, a two dimensional mesh is sampled instead of a single line, which may depend on x and y.

The method \addplot3 {⟨*math expr*⟩} visualizes the function $f(x,y) =$⟨*math expr*⟩ where $f\colon [x_1,x_2]\times[y_1,y_2]\to\mathbb{R}$. The interval $[x_1,x_2]$ is determined using the domain key, for example using domain=0:1. The interval $[y_1,y_2]$ is determined using the y domain key. If y domain is empty, $[y_1,y_2]=[x_1,x_2]$ will be assumed. If y domain=0:0 (or any other interval of length zero), it is assumed that the plot does not depend on y (thus, it is a line plot).

The number of samples in x direction is set using the samples key. The number of samples in y direction is set using the samples y key. If samples y is not set, the same value as for x is used. If samples y ≤ 1, it is assumed that the plot does not depend on y (meaning it is a line plot).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        \addplot3[surf] {y};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar]
        \addplot3
            [surf,faceted color=blue,
             samples=15,
             domain=0:1,y domain=-1:1]
            {x^2 - y^2};
    \end{axis}
\end{tikzpicture}
```

Expression plotting sets `mesh/rows` and `mesh/cols` automatically; these settings don't have any effect for expression plotting.

`\addplot3 expression {`⟨*math expression*⟩`};`
`\addplot3[`⟨*options*⟩`] expression {`⟨*math expression*⟩`}` ⟨*trailing path commands*⟩`;`

The syntax

`\addplot3 {`⟨*math expression*⟩`};`

as short-hand equivalent for

`\addplot3 expression {`⟨*math expression*⟩`};`

`\addplot3 (`⟨*x expression*⟩`,`⟨*y expression*⟩`,`⟨*z expression*⟩`) ;`
`\addplot3[`⟨*options*⟩`] (`⟨*x expression*⟩`,`⟨*y expression*⟩`,`⟨*z expression*⟩`)` ⟨*trailing path commands*⟩`;`

A variant of `\addplot3 expression` which allows to provide different coordinate expressions for the x, y and z coordinates. This can be used to generate parametrized plots.

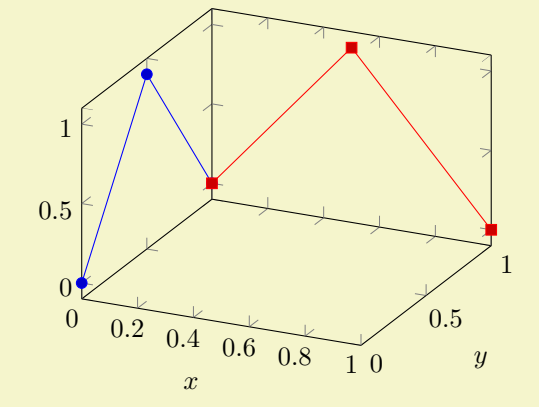
Please note that `\addplot3 (x,y,x^2)` is equivalent to `\addplot3 expression {x^2}`.

Note further that since the complete point expression is surrounded by round braces, round braces inside of ⟨*x expression*⟩, ⟨*y expression*⟩ or ⟨*z expression*⟩ need to be treated specially. Surround the expressions (which contain round braces) with curly braces:

`\addplot3 ({`⟨*x expr*⟩`}, {`⟨*y expr*⟩`}, {`⟨*z expr*⟩`});`

4.5.3 Line Plots

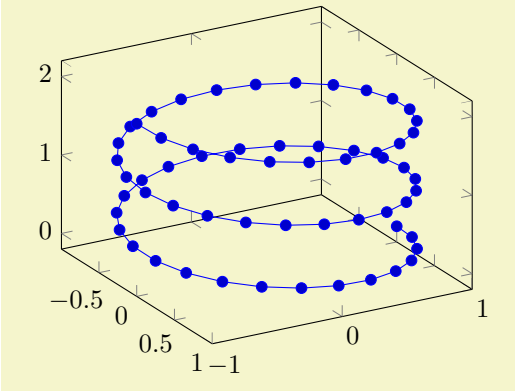
Three dimensional line plots are generated if the input source has no matrix structure. Line plots take the input coordinates and connect them in the order of appearance.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[xlabel=$x$,ylabel=$y$]
    \addplot3 coordinates {(0,0,0) (0,0.5,1) (0,1,0)};
    \addplot3 coordinates {(0,1,0) (0.5,1,1) (1,1,0)};
    \end{axis}
\end{tikzpicture}
```

If there is no value for neither `mesh/rows` nor `mesh/cols` or if one of them is `1`, PGFPLOTS will draw a line plot. This is also the case if there is no end–of–scanline marker (empty line) in the input stream.

For `\addplot3 expression`, this requires to set `samples y=0` to disable the generation of a mesh.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[view={60}{30}]
\addplot3+[domain=0:5*pi,samples=60,samples y=0]
    ({sin(deg(x))},
     {cos(deg(x))},
     {2*x/(5*pi)});
\end{axis}
\end{tikzpicture}
```

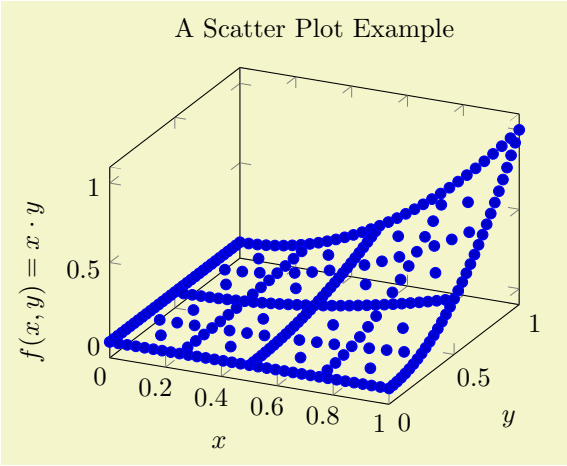
Three dimensional line plots will usually employ lines to connect points (i.e. the initial `sharp plot` handler of Ti*k*Z). The `smooth` method of Ti*k*Z might also prove be an option. Note that no piecewise constant plot, comb or bar plot handler is supported for three dimensional axes.

4.5.4 Scatter Plots

Three dimensional scatter plots have the same interface as for two dimensional scatter plots, so all examples of Section 4.4.10 can be used for the three dimensional case as well. The key features are to use `only marks` and/or `scatter` as plot styles.

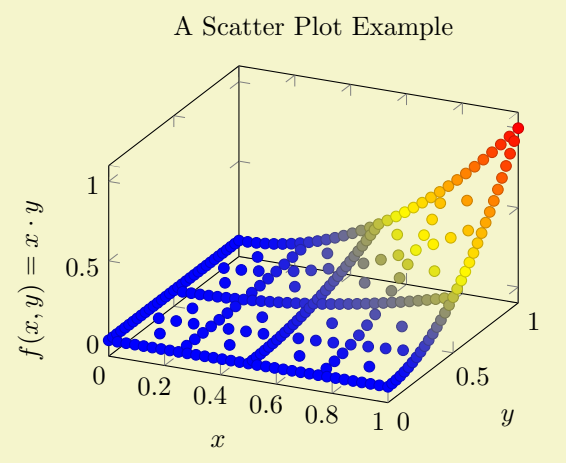
We provide some more examples which are specific for the three dimensional case.

Our first example uses `only marks` to place the current plot `mark` at each input position:



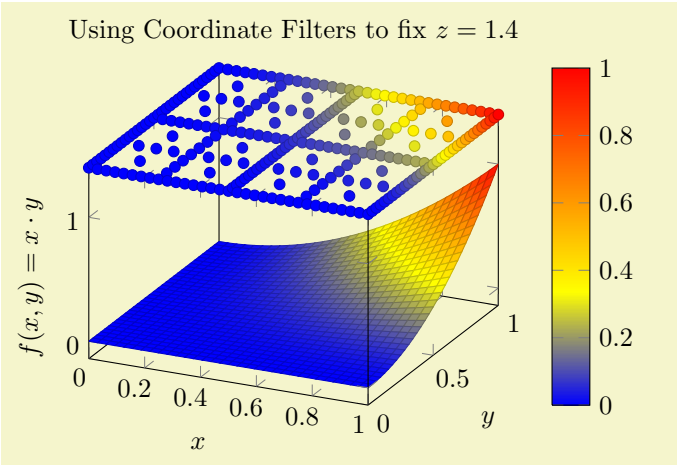

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xlabel=$x$,
        ylabel=$y$,
        zlabel={$f(x,y) = x\cdot y$},
        title=A Scatter Plot Example]
    %  'pgfplotsexample4_grid.dat' contains a
    %  large sequence of input points of the form
    %  x_0   x_1     f(x)
    %  0     0       0
    %  0     0.03125 0
    %  0     0.0625  0
    %  0     0.09375 0
    %  0     0.125   0
    %  0     0.15625 0
    \addplot3+[only marks] table
        {plotdata/pgfplotsexample4_grid.dat};
    \end{axis}
\end{tikzpicture}
```

If we add the key `scatter`, the plot mark will also use the colors of the current `colormap`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xlabel=$x$,
        ylabel=$y$,
        zlabel={$f(x,y) = x\cdot y$},
        title=A Scatter Plot Example]
    \addplot3+[only marks,scatter] table
        {plotdata/pgfplotsexample4_grid.dat};
    \end{axis}
\end{tikzpicture}
```

A more sophisticated example is to draw the approximated function as a `surf` plot (which requires matrix data) and the underlying grid (which is `scatter`ed data) somewhere into the same axis. We choose to place the (x, y) grid points at $z = 1.4$. Furthermore, we want the grid points to be colored according to the value of column `f(x)` in the input table:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        3d box,
        zmax=1.4,
        colorbar,
        xlabel=$x$,
        ylabel=$y$,
        zlabel={$f(x,y) = x\cdot y$},
        title={Using Coordinate Filters to fix $z=1.4$}]
    %  'pgfplotsexample4.dat' contains similar data as in
    %  'pgfplotsexample4_grid.dat', but it uses a uniform
    %  matrix structure (same number of points in every scanline).
    %  See examples above for extracts.
    \addplot3[surf,mesh/ordering=y varies]
        table {plotdata/pgfplotsexample4.dat};
    \addplot3[scatter,scatter src=\thisrow{f(x)},only marks, z filter/.code={\def\pgfmathresult{1.4}}]
        table {plotdata/pgfplotsexample4_grid.dat};
    \end{axis}
\end{tikzpicture}
```

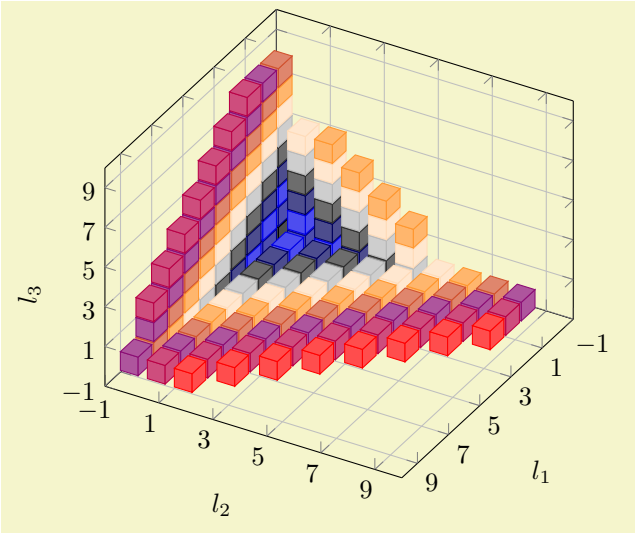
We used `z filter` to fix the z coordinate to 1.4. We could also have used the `table/z expr=1.4` feature

```
    \addplot3[scatter,scatter src=\thisrow{f(x)},only marks]
        table[z expr=1.4] {plotdata/pgfplotsexample4_grid.dat};
```

to get exactly the same effect. Choose whatever you like best. The `z filter` works for every coordinate input routine, the `z expr` feature is only available for `plot table`.

The following example uses `mark=cube*` and `z buffer=sort` to place boxes at each input coordinate. The color for each box is determined by `point meta={x+y+3}`. The remaining keys are just for pretty

printing.



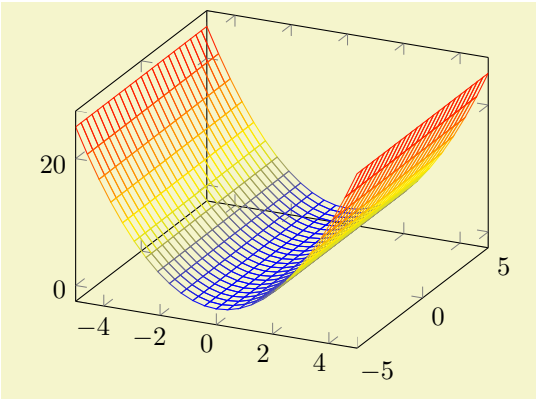

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    view={120}{40},
    width=220pt,
    height=220pt,
    grid=major,
    z buffer=sort,
    xmin=-1,xmax=9,
    ymin=-1,ymax=9,
    zmin=-1,zmax=9,
    enlargelimits=upper,
    xtick={-1,1,...,19},
    ytick={-1,1,...,19},
    ztick={-1,1,...,19},
    xlabel={$l_1$},
    ylabel={$l_2$},
    zlabel={$l_3$},
    point meta={x+y+z+3},
    colormap={summap}{
        color=(black); color=(blue);
        color=(black); color=(white)
        color=(orange) color=(violet)
        color=(red)
    },
    scatter/use mapped color={
        draw=mapped color,fill=mapped color!70},
    ]
    %  'pgfplots_scatter4.dat' contains a large sequence of
    %  the form
    %  l_0   l_1     l_2
    %  1     6       -1
    %  -1    -1      -1
    %  0     -1      -1
    %  -1    0       -1
    %  -1    -1      0
    %  1     -1      -1
    %  0     0       -1
    %  0     -1      0
    \addplot3[only marks,scatter,mark=cube*,mark size=7]
        table {plotdata/pgfplots_scatterdata4.dat};

\end{axis}
\end{tikzpicture}
```

4.5.5 Mesh Plots

/pgfplots/mesh (no value)

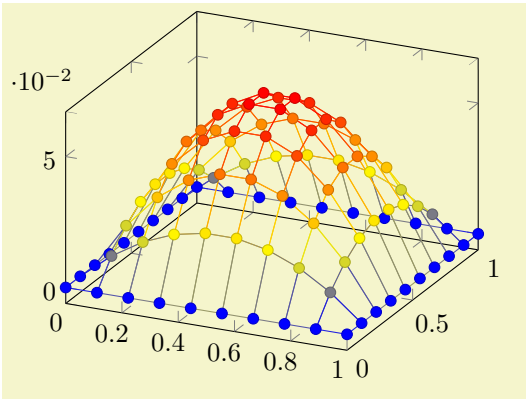
`\addplot+[mesh]`

A mesh plot uses different colors for each mesh segment. The color is determined using a "color coordinate" which is also called "meta data" throughout this document. It is the same data which is used for surface and scatter plots as well, see Section 4.7. In the initial configuration, the "color coordinate" is the z axis (or the y axis for two dimensional plots). This color coordinate is mapped linearly into the current color map to determine the color for each mesh segment. Thus, if the smallest occurring color data is, say, -1 and the largest is 42, points with color data -1 will get the color at the lower end of the color map and points with color data 42 the color of the upper end of the color map.

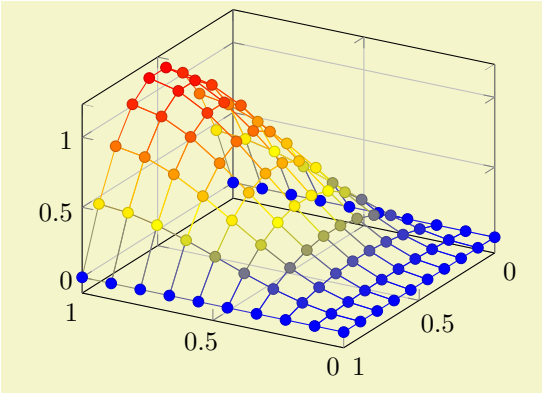



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        \addplot3[mesh] {x^2};
    \end{axis}
\end{tikzpicture}
```

A mesh plot can be combined with markers or with the `scatter` key which also draws markers in different colors.

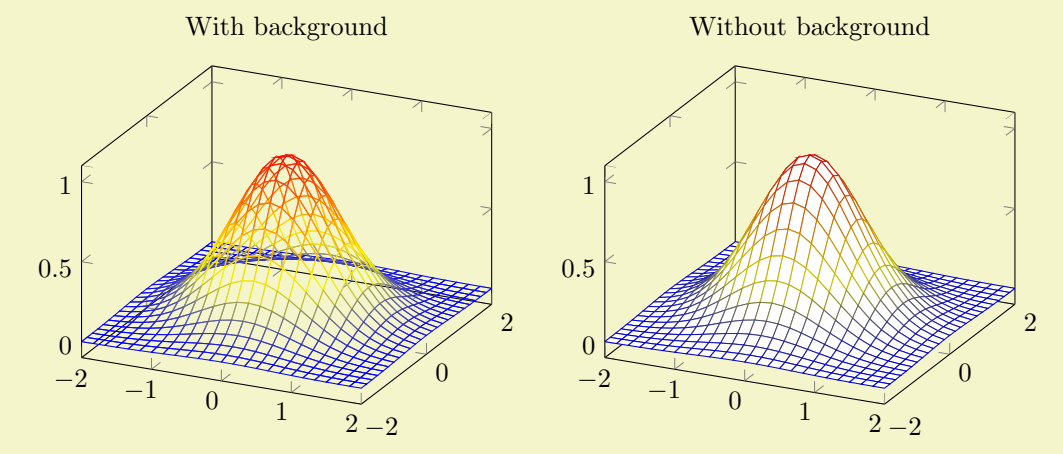



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3+[mesh,scatter,samples=10,domain=0:1]
        {x*(1-x)*y*(1-y)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[grid=major,view={210}{30}]
    \addplot3+[mesh,scatter,samples=10,domain=0:1]
        {5*x*sin(2*deg(x)) * y*(1-y)};
    \end{axis}
\end{tikzpicture}
```

Occasionally, one may want to hide the background mesh segments. This can be realized using the `surf` plot handler (see below) and a specific fill color:



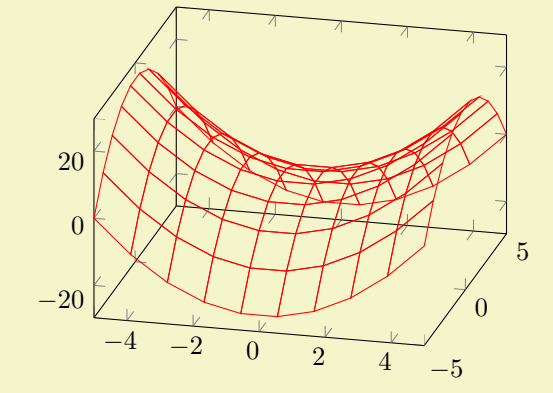

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[title=With background]
    \addplot3[mesh,domain=-2:2] {exp(-x^2-y^2)};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[title=Without background]
    \addplot3[surf,fill=white,domain=-2:2] {exp(-x^2-y^2)};
    \end{axis}
\end{tikzpicture}
```

The fill color needs to be provided explicitly.

Details:

- A mesh plot uses the same implementation as `shader=flat` to get one color for each single segment. Thus, if `shader=flat mean`, the color for a segment is determined using the *mean* of the color data of adjacent vertices. If `shader=flat corner`, the color of a segment is the color of *one* adjacent vertex.
- As soon as `mesh` is activated, `color=mapped color` is installed. This is *necessary* unless one needs a different color – but `mapped color` is the only color which reflects the color data.

  It is possible to use a different color using the `color=`⟨*color name*⟩ as for any other plot.
- It is easily possible to add `mark=`⟨*marker name*⟩ to mesh plots, `scatter` is also possible. Scatter plots will use the same color data as for the mesh.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view/az=14]
    \addplot3[mesh,draw=red,samples=10] {x^2-y^2};
    \end{axis}
\end{tikzpicture}
```

Mesh plots use the `mesh legend` style to typeset legend images.

`/pgfplots/mesh/check=false|warning|error` (initially `error`)

Allows to configure whether an error is generated if `mesh/rows` × `mesh/cols` does not equal the total number of coordinates.

If you know exactly what you are doing, it may be useful to disable the check. If you are unsure, it is best to leave the initial setting.

/pgfplots/z buffer=default|none|auto|sort|reverse x seq|reverse y seq|reverse xy seq (initially default)

This key allows to choose between different z buffering strategies. A z buffer determines which parts of an image should be drawn in front of other parts. Since both, the graphics packages PGF and the final document format .pdf are inherently two dimensional, this work has to be done in TEX. Currently, several (fast) heuristics can be used which work reasonably well for simple mesh- and surface plots. Furthermore, there is a (time consuming) sorting method which also works if the fast heuristics fails.

The z buffering algorithms of PGFPLOTS apply only to a single \addplot command. Different \addplot commands will be drawn on top of each other, in the order of appearance.

The choice default checks if we are currently working with a mesh or surface plot and uses auto in this case. If not, it sets z buffer=none.

The choice none disables z buffering. This is also the case for two dimensional axes which don't need z buffering.

The choice auto is the initial value for any mesh or surface plot: it uses a very fast heuristics to decide how to realize z buffering for mesh and surface plots. The idea is to reverse either the sequence of all x coordinates, or those of all y coordinates, or both. For regular meshes, this suffices to provide z buffering. In other words: the choice auto will use one of the three reverse strategies reverse * seq (or none at all). The choice auto, applied to patch plots, uses z buffer=sort since patch plots have no matrix structure.

The choice sort can be used for scatter, line, mesh, surface and patch plots. It sorts according to the depth of each point (or mesh segment). Sorting in TEX uses a slow algorithm and may require a lot of memory (although it has the expected runtime asymptotics $\mathcal{O}(N \log N)$). The depth of a mesh segment is just *one* number, currently determined as *mean* over the vertex depths. Since z buffer=sort is actually just a more intelligent way of drawing mesh segments on top of each other, it may still fail. Failure can occur if mesh segments are large and overlap at different parts of the segment (see Wikipedia "Painter's algorithm"). If you experience problems of this sort, consider reducing the mesh width (the mesh element size) such that they can be sorted independently (for example automatically using patch refines=2, see the patchplots library).

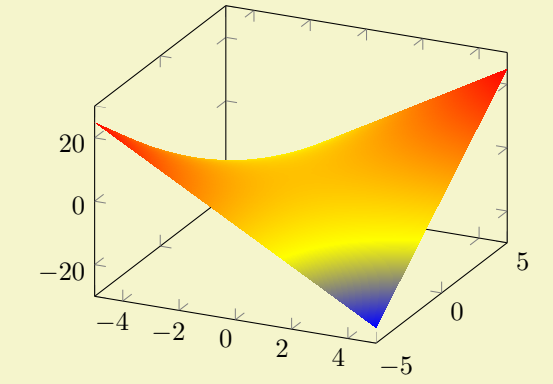
The remaining choices apply only to mesh/surface plots (i.e. for matrix data) and do nothing more then their name indicates: they reverse the coordinate sequences of the input matrix (using quasi linear runtime). They should only be used in conjunction by z buffer=auto.

4.5.6 Surface Plots

/pgfplots/surf (no value)

\addplot+[surf]

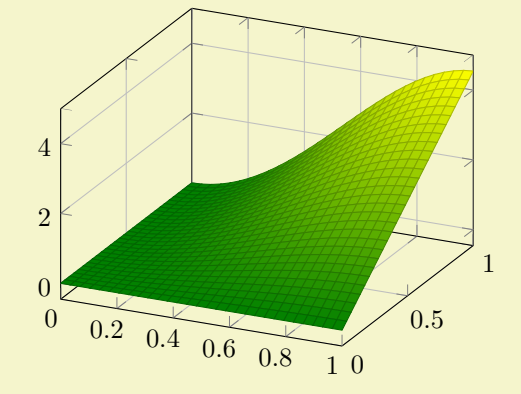
A surface plot visualizes a two dimensional, single patch using different fill colors for each patch segment. Each patch segment is a (pseudo) rectangle, that means input data is given in form of a data matrix as is discussed in the introductory section about three dimensional coordinates, 4.5.2.



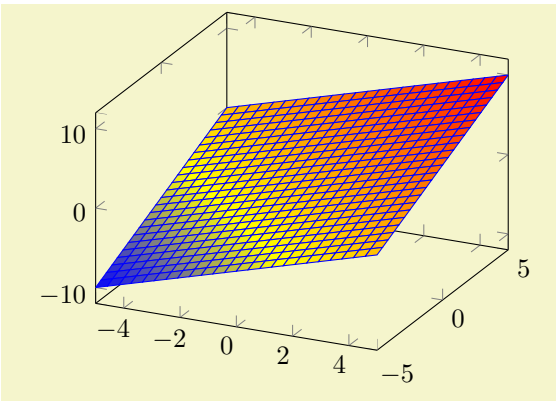

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        \addplot3[surf,shader=interp] {x*y};
    \end{axis}
\end{tikzpicture}
```

The simplest way to generate surface plots is to use the plot expression feature, but – as discussed in Section 4.5.2 – other input methods like `\addplot3 table` or `\addplot3 coordinates` are also possible.

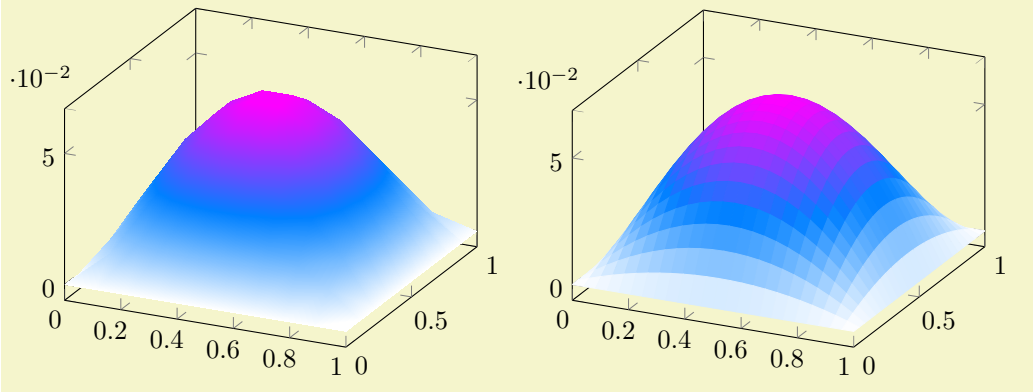
The appearance can be configured using `colormap`s, the value of the `shader`, `faceted color` keys and the current `color` and/or `draw`/`fill` color. As for `mesh` plots, the special `color=mapped color` is installed for the faces. The stroking color for faced plots can be set with `faceted color` (see below for details).



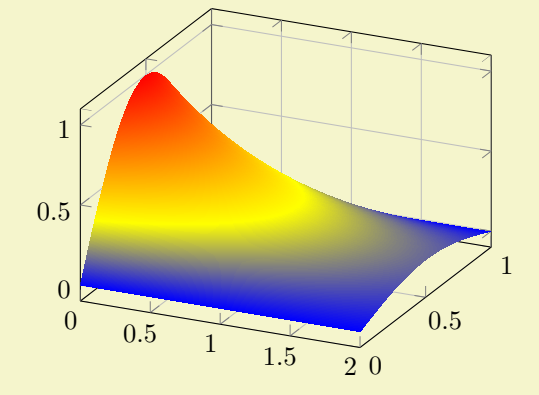

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        grid=major,
        colormap/greenyellow]
    \addplot3[surf,samples=30,domain=0:1]
        {5*x*sin(2*deg(x)) * y};
    \end{axis}
\end{tikzpicture}
```



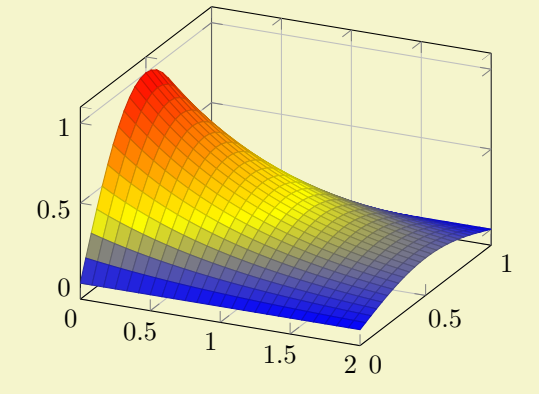

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        \addplot3[surf,faceted color=blue] {x+y};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colormap/cool]
    \addplot3[surf,samples=10,domain=0:1,
        shader=interp]
        {x*(1-x)*y*(1-y)};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[colormap/cool]
    \addplot3[surf,samples=25,domain=0:1,
        shader=flat]
        {x*(1-x)*y*(1-y)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[grid=major]
        \addplot3[surf,shader=interp,
            samples=25,domain=0:2,y domain=0:1]
            {exp(-x) * sin(pi*deg(y))};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[grid=major]
        \addplot3[surf,shader=faceted,
            samples=25,domain=0:2,y domain=0:1]
            {exp(-x) * sin(pi*deg(y))};
    \end{axis}
\end{tikzpicture}
```

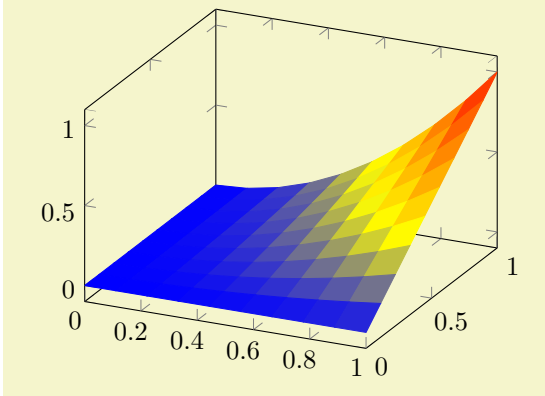
Details about the shading algorithm are provided below in the documentation of `shader`.

Surface plots use the `mesh legend` style to create legend images.

/pgfplots/**shader**=flat|interp|faceted|flat corner|flat mean|faceted interp (initially faceted)

Configures the shader used for surface plots. The shader determines how the color data available at each single vertex is used to fill the surface patch.

The simplest choice is to use one fill color for each segment, the choice `flat`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,shader=flat,
        samples=10,domain=0:1]
        {x^2*y};
    \end{axis}
\end{tikzpicture}
```

There are (currently) two possibilities to determine the single color for every segment:

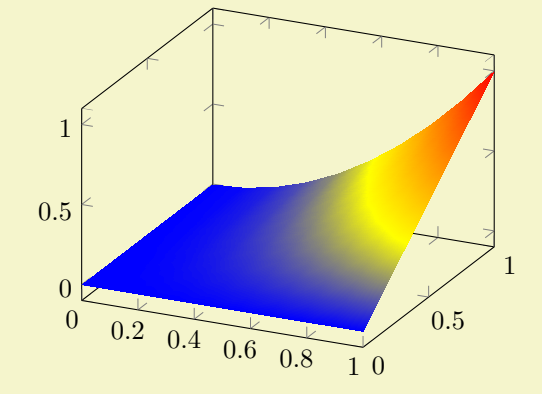
`flat corner` Uses the color data of one vertex to color the segment. It is not defined which vertex is used here[22].

`flat mean` Uses the mean of all four color data values as segment color. This is the initial value as it provides symmetric colors for symmetric functions.

The choice `flat` is actually the same as `flat mean`. Please note that `shader=flat mean` and `shader=flat corner` also influence mesh plots – the choices determine the mesh segment color.

[22] PGFPLOTS just uses the last vertex encountered in its internal processings – but after any z buffer re-orderings.

Another choice is shader=interp which uses Goraud shading (smooth linear interpolation of two triangles approximating rectangles) to fill the segments.

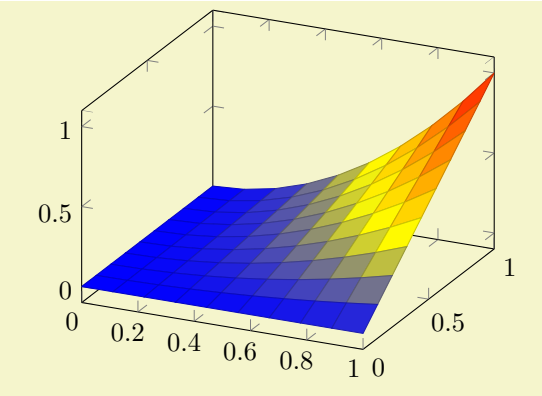



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,shader=interp,
        samples=10,domain=0:1]
        {x^2*y};
    \end{axis}
\end{tikzpicture}
```

The shader=interp employs a low–level shading implementation which is currently (only) available for the postscript driver pgfsys-dvips.def and the pdflatex driver pgfsys-pdftex.def. For other drivers, the choice shader=interp will result in a warning and is equivalent to shader=flat mean. See also below for detail remarks.

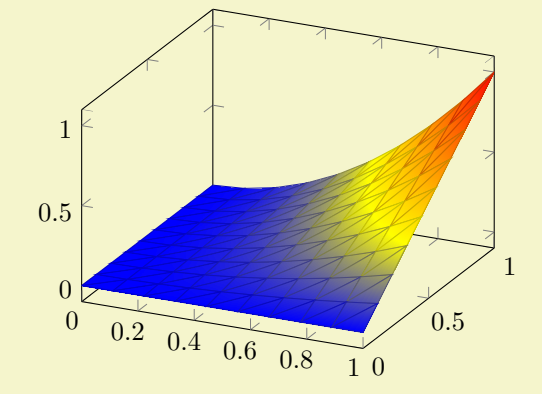
Note that shader=interp,patch type=bilinear allows real bilinear interpolation, see the patchplots library.

The choice shader=faceted uses a constant fill color for every mesh segment (as for flat) and the value of the key /pgfplots/faceted color to draw the connecting mesh elements:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,shader=faceted,
        samples=10,domain=0:1]
        {x^2*y};
    \end{axis}
\end{tikzpicture}
```

The last choice is shader=faceted interp. As the name suggests, it is a mixture of interp and faceted in the sense that each element is shaded using linear triangle interpolation (see also the patchplots library for bilinear interpolation) in the same way as for interp, but additionally, the edges are colored in faceted color:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,shader=faceted interp,
        samples=10,domain=0:1]
        {x^2*y};
    \end{axis}
\end{tikzpicture}
```

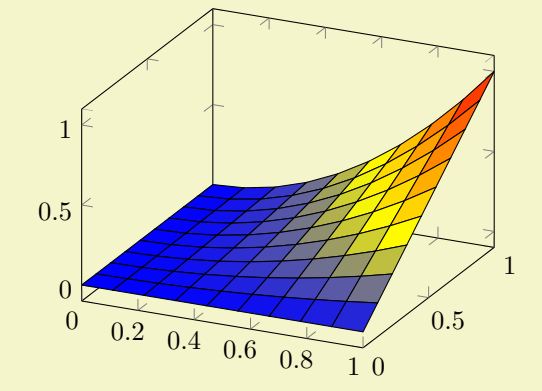
In principle, there is nothing wrong with the idea as such, and it looks quite good – but it enlarges the resulting pdf document considerably and might take a long time to render. It works as follows: for

every mesh element (either triangle for `patch` plots or rectangle for lattice plots), it creates a low level shading. It then fills the single mesh element with that shading, and strokes the edges with `faceted color`. The declaration of that many low level shadings is rather inefficient in terms of pdf objects (large output files) and might render slowly[23]. For orthogonal plots (like `view={0}{90}`), the effect of `faceted interp` can be gained with less cost if one uses two separate `\addplot` commands: one with `surf` and one with `mesh`. Handle this choice with care.

Details:

- All shaders support `z buffer=sort` (starting with version 1.4)
- The choice `shader=faceted` is the same as `shader=flat` – except that it uses a special draw color. So, `shader=faceted` has the same effect as `shader=flat,draw=\pgfkeysvalueof{/pgfplots/faceted color}`.
- The `flat` shader uses the current `draw` and `fill` colors. They are set with `color=mapped color` and can be overruled with `draw=`⟨*draw color*⟩ and `fill=`⟨*fill color*⟩. The `mapped color` always contains the color of the color map.
- You easily add `mark=`⟨*plot mark*⟩ to mesh and/or surface plots or even colored plot marks with `scatter`. The scatter plot feature will use the same color data as for the surface.

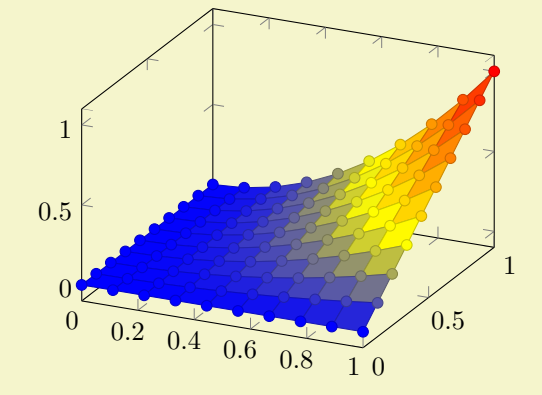
  But: Markers and surfaces do not share the same depth information. They are drawn on top of each other.
- Remarks on `shader=interp`:
  - It uses the current color map in any case, ignoring `draw` and `fill`.
  - For surface plots with lots of points, `shader=interp` produces smaller `pdf` documents, requires less compilation time in TEX and requires less time to display in Acrobat Reader than `shader=flat`.
  - The postscript driver *truncates* coordinates to 24 bit – which might result in a loss of precision (the truncation is not very intelligent). See the `surf shading/precision` key for details. To improve compatibility, this 24 bit truncation algorithm is enabled by default also for pdf documents.
  - The choice `shader=interp` works well with either Acrobat Reader or recent versions of free viewers[24]. However, some free viewers show colors incorrectly (like evince). I hope this message will soon become outdated... if not, provide bug reports to the Linux community to communicate the need to improve support for Type 4 (`patch`) and Type 5 pdf (`surf`) and Type 7 (`patch` and elements of the `patchplots` library) shadings.
  - The `interp` shader yields the same outcome as `faceted interp,faceted color=none`, although `faceted interp` requires much more ressources.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,shader=flat,
        draw=black,
        samples=10,domain=0:1]
        {x^2*y};
    \end{axis}
\end{tikzpicture}
```

[23] My experience is as follows: Acrobat reader can efficiently render huge `interp` shadings. But it is very slow for `faceted interp` shadings. Linux viewers like xpdf are reasonably efficient for `interp` (at least with my bugfixes to libpoppler) and are also fast for `faceted interp` shadings.

[24] The author of this package has submitted bugfixes to xpdf/libpoppler which should be part of the current stable versions of many viewers.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[surf,shader=faceted,
        scatter,mark=*,
        samples=10,domain=0:1]
        {x^2*y};
    \end{axis}
\end{tikzpicture}
```

/pgfplots/faceted color={⟨*color name*⟩} (initially `mapped color!80!black`)

Defines the color to be used for meshes of faceted surface plots.

Set `faceted color=none` to disable edge colors.

/pgfplots/surf shading/precision=pdf|postscript|ps (initially `postscript`)

A key to configure how the low level driver for `shader=interp` writes its data. The choice `pdf` uses 32 bit binary coordinates (which is lossless). The resulting `.pdf` files appear to be correct, but they can't be converted to postscript – the converter software always complains about an error.

The choice `postscript` (or, in short, `ps`) uses 24 bit truncated binary coordinates. This results in both, readable `.ps` and `.pdf` files. However, the truncation is lossy.

If anyone has ideas how to fix this problem: let me know. As far as I know, Postscript should accept 32 bit coordinates, so it might be a mistake in the shading driver.

4.5.7 Contour Plots

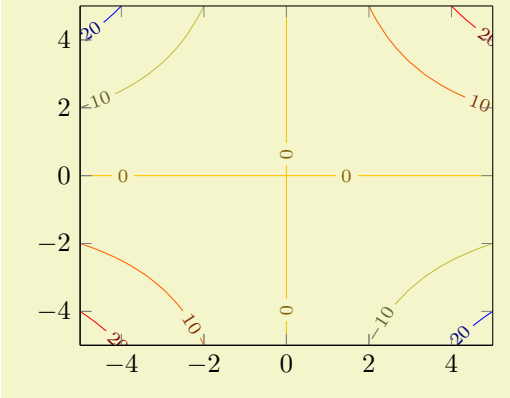
PGFPLOTS supports visualization of contour plots whose coordinates have been computed by *external tools*. The `contour prepared` plot handler coming with PGFPLOTS takes precomputed contour line coordinates and handles their visualization (`contour/draw color`, `contour/labels` etc.). The `contour gnuplot` style takes matrix input in the same format as for `mesh` or `surf` (that includes any of the PGFPLOTS matrix input methods). It then writes the matrix data to a file and invokes `gnuplot` (or other, user customizable external programs) to compute contour coordinates. Finally, the computed contours are visualized with the `contour prepared` algorithm. Thus, external programs need to compute the contour coordinates and PGFPLOTS visualizes the result.

We discuss the high level interface to external programs first and continue with `contour prepared` later-on.

/pgfplots/contour gnuplot={⟨options with '`contour/`' or '`contour external/`' prefix⟩}

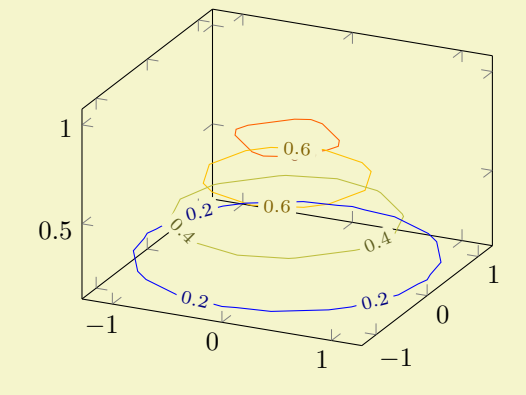
\addplot+[contour gnuplot={⟨options with '`contour/`' or '`contour external/`' prefix⟩}]

This is a high level contour plot interface. It expects matrix data in the same way as two dimensional `surf` or `mesh` plots do. It then computes contours and visualizes them.



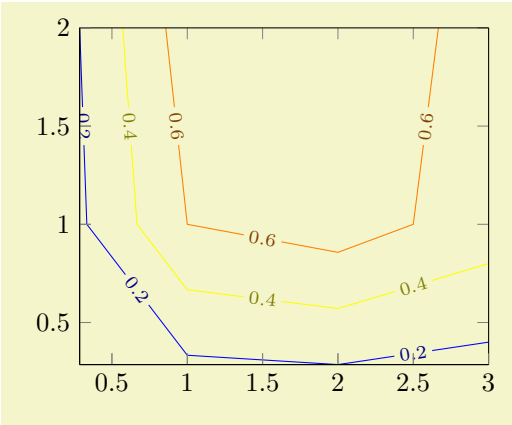

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view={0}{90}]
    \addplot3[contour gnuplot]
        {x*y};
    \end{axis}
\end{tikzpicture}
```

The example uses \addplot3 together with expression plotting, that means the input data is of the form $(x_i, y_i, f(x_i, y_i))$. The view={0}{90} flag means "view from top", otherwise the contour lines would have been drawn as z value:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot3[contour gnuplot]
        {exp(0-x^2-y^2)};
    \end{axis}
\end{tikzpicture}
```

As mentioned, you can use any of the PGFPLOTS input methods as long as it yields matrix output. Thus, we can re-use our introductory example of matrix data, this time with inline data:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}%
\begin{tikzpicture}%
    \begin{axis}[view={0}{90}]%
    \addplot3[contour gnuplot]%
        coordinates {
            (0,0,0) (1,0,0)   (2,0,0)   (3,0,0)

            (0,1,0) (1,1,0.6) (2,1,0.7) (3,1,0.5)

            (0,2,0) (1,2,0.7) (2,2,0.8) (3,2,0.5)
        };
    \end{axis}%
\end{tikzpicture}%
```

What happens behind the scenes is that PGFPLOTS takes the input matrix and writes all encountered coordinates to a temporary file, including the end–of–scanline markers. Then, it generates a small gnuplot script and invokes gnuplot to compute the contour coordinates, writing everything into a temporary output file. Afterwards, it includes gnuplot's output file just as if you'd write \addplot3[contour prepared] file {⟨*temporaryfile*⟩};.

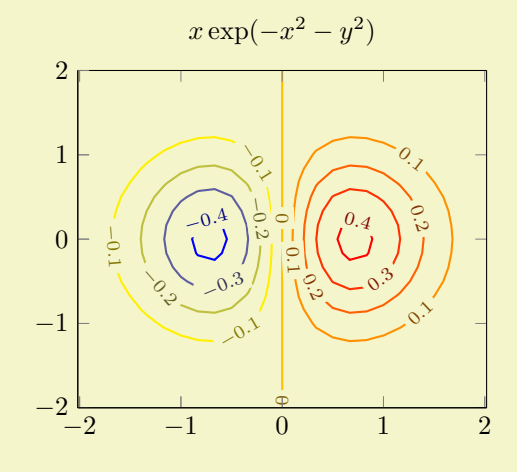
All this invocation of gnuplot, including input/output file management is transparent to the user. It only requires two things: first of all, it requires matrix data as input[25]. Second, it requires you to enable system calls. Consider the documentation for plot gnuplot for how to enable system calls.

There are several fine-tuning parameters of the input/output file management, and it is even possible to invoke different programs than gnuplot (even matlab). These details are discussed at the end of this section, see below at page 107.

/pgfplots/contour/number={⟨*integer*⟩} (initially 5)

Configures the number of contour lines which should be produced by any external contouring algorithm.

[25] Note that contour gnuplot processes the input stream only once. Consequently, the temporary file will contain only information which was available before the first point has been seen. The example above works because it contains empty lines as end-of-scanline markers. If you do not provide such markers, you may need to provide two of the three options mesh/rows, mesh/cols, or mesh/num points.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        title={$x \exp(-x^2-y^2)$},
        domain=-2:2,enlarge x limits,
        view={0}{90},
    ]
   \addplot3[contour gnuplot={number=14},thick]
          {exp(0-x^2-y^2)*x};
    \end{axis}
\end{tikzpicture}
```

It is also possible to change the `/pgf/number format` settings, see the documentation for the `contour/every contour label` style below.

Note that `contour/number` has no effect on `contour prepared`.

`/pgfplots/contour prepared={`⟨options with '`contour/`' prefix⟩`}`

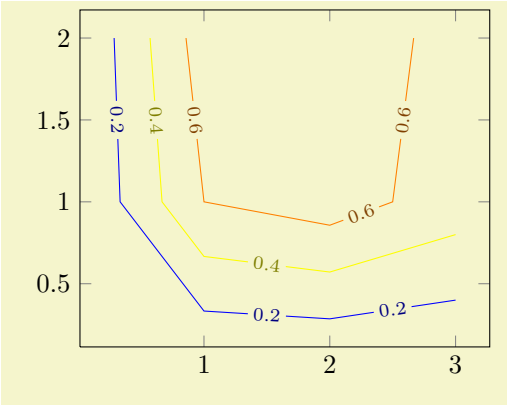
`\addplot+[contour prepared={`⟨options with '`contour/`' prefix⟩`}]`

A plot handler which expects already computed contours on input and visualizes them. It cannot compute contours on its own.

`/pgfplots/contour prepared format=standard|matlab` (initially `standard`)

There are two accepted input formats. The first is a long sequence of coordinates of the form (x, y, z) where all successive coordinates with the same z value make up a contour level (this is only part of complete truth, see below). The end–of–scanline markers (empty lines in the input) mark an interruption in one contour level.

For example, `contour prepared format=standard` could be[26]

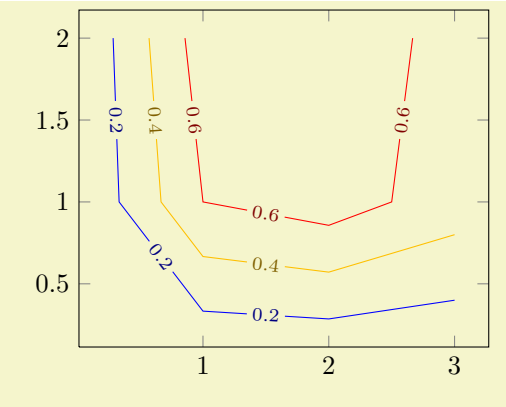



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[contour prepared]
        table {
              2  2  0.8

              0.857143  2  0.6
              1  1  0.6
              2  0.857143  0.6
              2.5  1  0.6
              2.66667  2  0.6

              0.571429  2  0.4
              0.666667  1  0.4
              1  0.666667  0.4
              2  0.571429  0.4
              3  0.8  0.4

              0.285714  2  0.2
              0.333333  1  0.2
              1  0.333333  0.2
              2  0.285714  0.2
              3  0.4  0.2
        };
    \end{axis}
\end{tikzpicture}
```

Note that the empty lines are not necessary in this case: empty lines make only a difference if they occur within the same contour level (i.e. if the same z value appears above and below of them).

[26] This is actually the output from our `\addplot3[contour gnuplot] coordinates` example from above.

The choice `contour prepared format=matlab` expects two–dimensional input data where the contour level and the number of elements of the contour line are provided as x and y coordinates, respectively, of a leading point. Such a format is used by `matlab`'s contour algorithms, i.e. it resembles the output of the matlab commands `data=contour(...)` or `data=contourc(...)`.

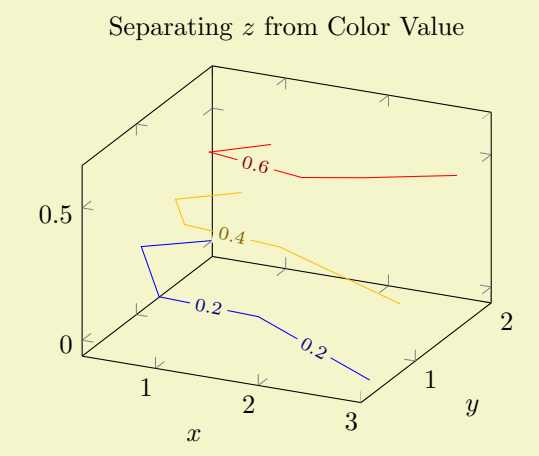



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[contour prepared,
        contour prepared format=matlab]
    table {
%  (0.2,5) ==> contour '0.2' (x), 5 points follow (y):
       2.0000000e-01   5.0000000e+00
       3.0000000e+00   4.0000000e-01
       2.0000000e+00   2.8571429e-01
       1.0000000e+00   3.3333333e-01
       3.3333333e-01   1.0000000e+00
       2.8571429e-01   2.0000000e+00
%  (0.4,5) ==> contour '0.4', consists of 5 points
       4.0000000e-01   5.0000000e+00
       3.0000000e+00   8.0000000e-01
       2.0000000e+00   5.7142857e-01
       1.0000000e+00   6.6666667e-01
       6.6666667e-01   1.0000000e+00
       5.7142857e-01   2.0000000e+00
%  (0.6,6) ==> contour '0.6', has 6 points
       6.0000000e-01   6.0000000e+00
       2.6666667e+00   2.0000000e+00
       2.5000000e+00   1.0000000e+00
       2.0000000e+00   8.5714286e-01
       1.0000000e+00   1.0000000e+00
       1.0000000e+00   1.0000000e+00
       8.5714286e-01   2.0000000e+00
        };
    \end{axis}
\end{tikzpicture}
```

In case you use matlab, you can generate such data with

```
[x,y]=meshgrid(linspace(0,1,15));
data=contour(x,y,x.*y);
data=data';
save 'exporteddata.dat' data -ASCII
```

As already mentioned in the beginning, the z coordinate is not necessarily the coordinate used to delimit contour levels. In fact, the `point meta` data is acquired here, i.e. you are free to use whatever z coordinate you want as long as you have a correct `point meta` value. The example from above could be modified as follows:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        title=Separating $z$ from Color Value,
        xlabel=$x$,
        ylabel=$y$,
    ]
    \addplot3[contour prepared,
        point meta=\thisrow{level}]
        table {
     x         y   z   level
     0.857143  2  0.4 0.6
     1         1  0.6  0.6
     2  0.857143 0.6  0.6
     2.5  1      0.6  0.6
     2.66667  2  0.4  0.6

     0.571429  2  0.2 0.4
     0.666667  1  0.4 0.4
     1  0.666667  0.4 0.4
     2  0.571429  0.4 0.4
     3  0.8       0.2 0.4

     0.285714  2  0   0.2
     0.333333  1  0.2 0.2
     1  0.333333  0.2 0.2
     2  0.285714  0.2 0.2
     3  0.4       0   0.2
        };
    \end{axis}
\end{tikzpicture}
```

The example above uses different z coordinates for each first and each last point on contour lines. The contour lines as such are defined by the `level` column since we wrote `point meta=\thisrow{level}`. Such a feature also allows `contour prepared` for nonstandard axes, compare the examples for the `ternary` lib on page 336.

`/pgfplots/contour/draw color={`⟨*color*⟩`}` (initially `mapped color`)

Defines the draw color for every contour. Note that only `mapped color` actually depends on the contour level.

`/pgfplots/contour/labels={`⟨*true,false*⟩`}` (initially `true`)

Configures whether contour labels shall be drawn or not.

`/pgfplots/contour/label distance={`⟨*dimension*⟩`}` (initially `70pt`)

Configures the distance between adjacend contour labels within the same contour level.

`/pgfplots/contour/every contour label` (style, no value)

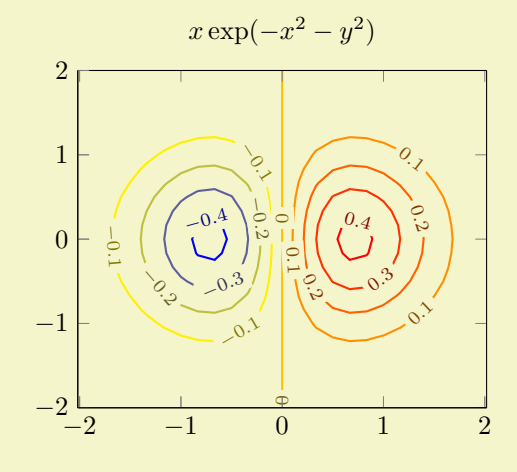
Allows to customize contour labels. The preferred way to change this style is the `contour label style={`⟨*options*⟩`}` method, see below.

The initial value is

```
\pgfplotsset{
    contour/every contour label/.style={
        sloped,
        transform shape,
        inner sep=2pt,
        every node/.style={mapped color!50!black,fill=white},
        /pgf/number format/relative={\pgfplotspointmetarangeexponent},
    }
}
```

Note that `\pgfplotspointmetarangeexponent`$= e$ where $\pm m \cdot 10^e$ is the largest occuring label value (technically, it is the largest occurring value of `point meta`).

The following example modifies the `/pgf/number format` styles for contour labels:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        title={$x \exp(-x^2-y^2)$},
        domain=-2:2,enlarge x limits,
        view={0}{90},
    ]
   \addplot3[
           contour gnuplot={
            scanline marks=required,
               number=14,
            contour label style={
                /pgf/number format/fixed,
                /pgf/number format/precision=1,
            },
        },thick
    ]
          {exp(0-x^2-y^2)*x};
    \end{axis}
\end{tikzpicture}
```

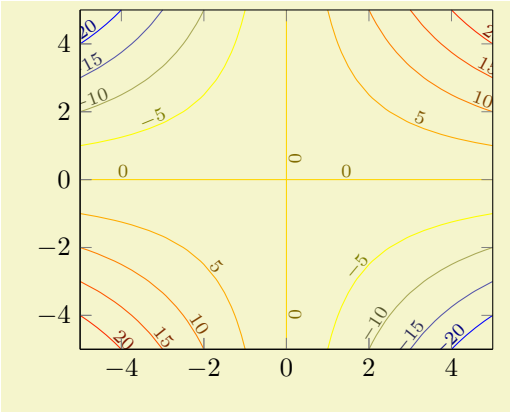
/pgfplots/contour/contour label style={⟨*key-value-list*⟩}

An abbreviation for contour/every contour label/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style contour/every contour label.

/pgfplots/contour/labels over line (style, no value)

A style which changes `every contour label` such that labels are right over the lines, without fill color.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view={0}{90}]
    \addplot3[contour gnuplot={
        labels over line,number=9}]
        {x*y};
    \end{axis}
\end{tikzpicture}
```

/pgfplots/contour/handler (style, no value)

Allows to modify the plot handler which connects the points of a single contour level.

The initial value is

```
\pgfplotsset{contour/handler/.style={/tikz/sharp plot}}
```

but a useful alternative might be the smooth handler.

/pgfplots/contour/label node code/.code={⟨...⟩}

A lowlevel interface to modify how contour labels are placed.

The initial value is

```
\pgfplotsset{
    contour/label node code/.code={\node {\pgfmathprintnumber{#1}};}
}
```

/pgfplots/contour external={⟨options with ‘contour/’ or ‘contour external/’ prefix⟩}

\addplot+[contour external={⟨options with 'contour/' or 'contour external/' prefix⟩}]

This handler constitutes a generic interface to external programs to compute contour lines. The contour gnuplot method is actually a special case of contour external.

/pgfplots/contour external/file={⟨*base file name*⟩} (initially empty)

The initial configuration is to automatically generate a unique file name.

/pgfplots/contour external/scanline marks=false|if in input|required|true (initially if in input)

Controls how contour external writes end-of-scanline markers.

The choice false writes no such markers at all. In this case, script should contain mesh/rows and/or mesh/cols.

The choice if in input generates end-of-scanline markers if they appear in the provided input data (either as empty lines or if the user provided at least two of the three options mesh/rows, mesh/cols, or mesh/num points explicitly).

The choice required works like if in input, but it will fail unless there really was such a marker.

The choice true is an alias for required.

/pgfplots/contour external/script={⟨*Code for external program*⟩} (initially empty)

Provides template code to generate a script for the external program. Inside of ⟨*Code for external program*⟩, the placeholder \infile will expand to the temporary input file and \outfile to the temporary output file. The temporary \infile is a text file containing one point on each line, in the form x y meta meta, separated by tabstops. Whenever a scanline is complete, an empty line is issued (but only if these scanline markers are found in the input stream as well). The complete set of scanlines forms a matrix. There are no additional comments or extra characters in the file. The macro \ordering will expand to 0 if the matrix is stored in mesh/ordering=x varies and \ordering will be 1 for mesh/ordering=y varies.

Inside of ⟨*Code for external program*⟩, you can also use \pgfkeysvalueof{/pgfplots/mesh/rows} and \pgfkeysvalueof{/pgfplots/mesh/cols}; they expand to the matrix' size. Similarly, \pgfkeysvalueof{/pgfplots/mesh/num points} expands to the total number of points.

/pgfplots/contour external/script extension={⟨*extension*⟩} (initially script)

The file name extension for the temporary script.

/pgfplots/contour external/cmd={⟨*system call*⟩} (initially empty)

A template to generate system calls for the external program. Inside of ⟨*system call*⟩, you may use \script as placeholder for the filename which contains the result of contour external/script.

/pgfplots/contour gnuplot (style, no value)

The initial configuration is

```
\pgfplotsset{
    contour gnuplot/.style={
        contour external={
            script={
                unset surface;
                set cntrparam levels \pgfkeysvalueof{/pgfplots/contour/number};
                set contour;
                set table \"\outfile\";
                splot \"\infile\";
            },
            cmd={gnuplot \"\script\"},
            #1,%
        },
    }
}
```

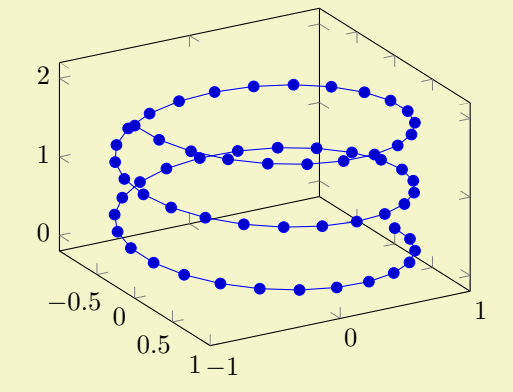
Note that contour gnuplot requires explicit scanline markers in the input stream, and it assumes mesh/ordering=x varies.

Note that contour external lacks the intelligence to detect changes; it will always re-generate the output (unless the -shell-escape feature is not active).

4.5.8 Parameterized Plots

Parameterized plots use the same plot types as documented in the preceding sections: both `mesh` and `surf`ace plots are actually special parametrized plots where x and y are on cartesian grid points.

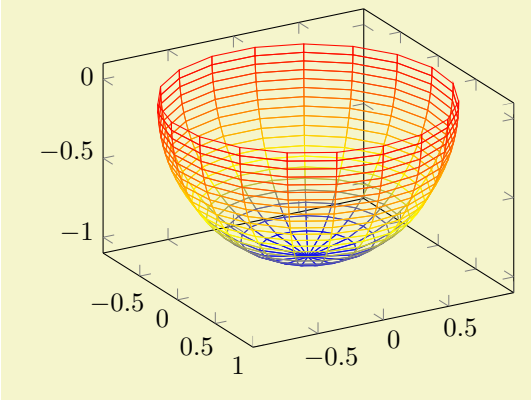
Parameterized plots just need a special way to provide the coordinates:



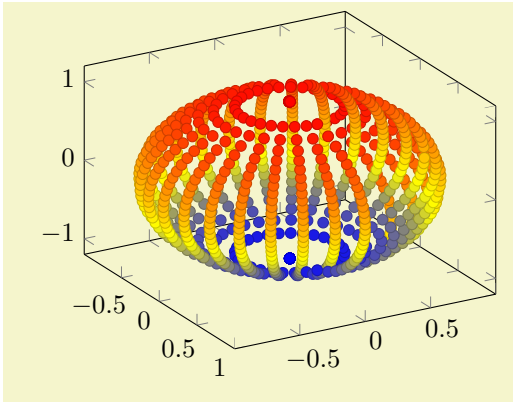

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view={60}{30}]
    \addplot3+[domain=0:5*pi,samples=60,samples y=0]
        ({sin(deg(x))},
         {cos(deg(x))},
         {2*x/(5*pi)});
    \end{axis}
\end{tikzpicture}
```

The preceding example uses `samples y=0` to indicate that a line shall be sampled instead of a matrix. The curly braces are necessary because TEX can't nest round braces. The single expressions here are used to parametrize the helix.

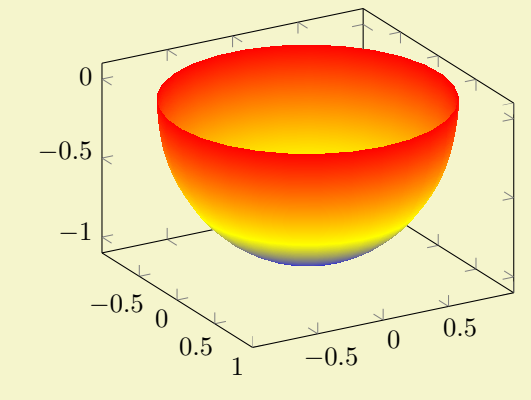
Another example follows. Note that `z buffer=sort` is a necessary method here.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[view={60}{30}]
    \addplot3[mesh,z buffer=sort,
        samples=20,domain=-1:0,y domain=0:2*pi]
        ({sqrt(1-x^2) * cos(deg(y))},
         {sqrt( 1-x^2 ) * sin(deg(y))},
         x);
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[view={60}{30}]
    \addplot3[mesh,z buffer=sort,
        scatter,only marks,scatter src=z,
        samples=30,domain=-1:1,y domain=0:2*pi]
        ({sqrt(1-x^2) * cos(deg(y))},
         {sqrt( 1-x^2 ) * sin(deg(y))},
         x);
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[view={60}{30}]
    \addplot3[surf,shader=interp,z buffer=sort,
        samples=30,domain=-1:0,y domain=0:2*pi]
        ({sqrt(1-x^2) * cos(deg(y))},
         {sqrt( 1-x^2 ) * sin(deg(y))},
         x);
\end{axis}
\end{tikzpicture}
```

4.5.9 3D Quiver Plots (Arrows)

Three dimensional `quiver` plots are possible with the same interface as their two-dimensional counterparts, simply provide the third coordinate using `quiver/w`. Please refer to Section 4.4.7 for details and examples.

4.5.10 About 3D Const Plots and 3D Bar Plots

There are currently *no* equivalents of `const plot` and its variants or the bar plot types like `ybar` for three dimensional axes, sorry.

4.5.11 Patch Plots

`/pgfplots/patch` (no value)

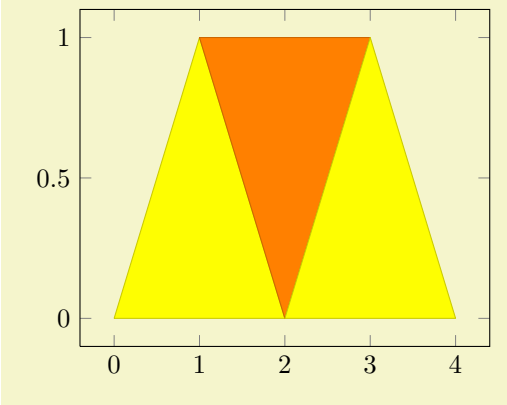
`\addplot+[patch]`

Patch plots are similar to `mesh` and `surf` plots in that they describe a filled area by means of a geometry.

However, `patch` plots are defined by *explicitly* providing the elements of the geometry: they expect a sequence of triangles (or other `patch types`) which make up the mesh.

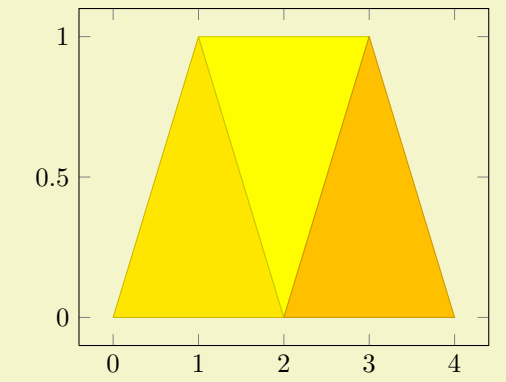
There are two dimensional and three dimensional patch plots, both with the same interfaces which are explained in the following sections.

The standard input format (constituted by `mesh input=patches`) is to provide a sequence of coordinates (either two– or three–dimensional) as usual. Each consecutive set of points makes up a patch element, which is often a triangle:



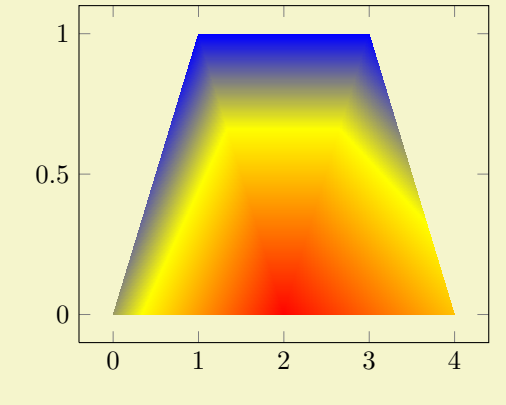

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[patch]
    table {
        x y
        0 0
        1 1
        2 0
%  empty lines do not hurt, they are ignored here:

        1 1
        2 0
        3 1

        2 0
        3 1
        4 0

    };
    \end{axis}
\end{tikzpicture}
```

Patch plots use `point meta` to determine fill colors. In its initial configuration, `point meta` will be set to the y coordinate (or the z coordinate for three dimensional `patch` plots). Set `point meta` somehow to color the patches:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[patch]
    table[point meta=\thisrow{c}] {
        x y c
        0 0 0.2
        1 1 0
        2 0 1

        1 1 0
        2 0 1
        3 1 0

        2 0 1
        3 1 0
        4 0 0.5
    };
    \end{axis}
\end{tikzpicture}
```

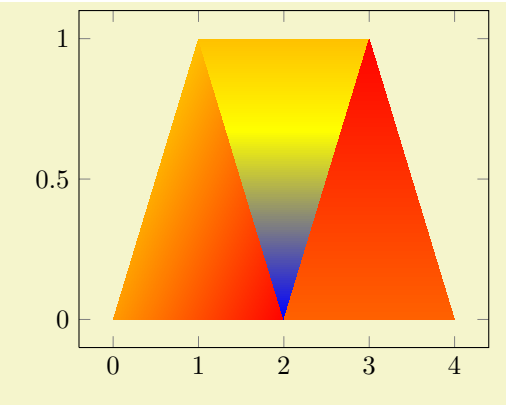
Patch plots make use of the `mesh` configuration, including the `shader`. Thus, the example above uses the initial `shader=faceted` (which uses the *mean* color data to determine a triangle's color and a related stroke color). The `shader=interp` yields the following result:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[patch,shader=interp]
    table[point meta=\thisrow{c}] {
        x y c
        0 0 0.2
        1 1 0
        2 0 1

        1 1 0
        2 0 1
        3 1 0

        2 0 1
        3 1 0
        4 0 0.5
    };
    \end{axis}
\end{tikzpicture}
```

For triangles, `shader=interp` results in linearly interpolated `point meta` values throughout each individual triangle, which are then mapped to the color map (a technique also known as Gouraud shading).

The color data does not need to be continuous, it is associated to triangle vertices. Thus, changing some of the color values allows individually shaded regions:



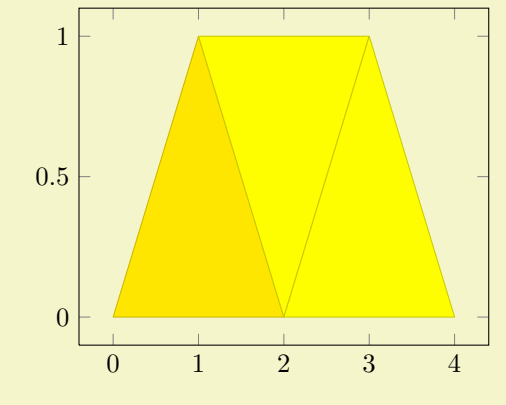

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[patch,shader=interp]
    table[point meta=\thisrow{c}] {
        x y c
        0 0 0.2
        1 1 0
        2 0 1

        1 1 0
        2 0 -1
        3 1 0

        2 0 0.5
        3 1 1
        4 0 0.5
    };
    \end{axis}
\end{tikzpicture}
```

Two dimensional patch plots simply draw triangles in their order of appearance. In three dimensions, single elements are sorted according to their view depth, with foreground elements drawn on top of background elements ("Painter's algorithm", see z buffer=sort).

/pgfplots/patch table={⟨*table file name or inline table*⟩} (initially empty)
/pgfplots/patch table with point meta={⟨*table file name or inline table*⟩} (initially empty)
/pgfplots/patch table with individual point meta={⟨*table file name or inline table*⟩} (initially empty)

Allows to provide patch connectivity data stored in an input table.

A non–empty argument for patch table enables patch input mode. Now, the standard input stream is a long list of vertices which are stored in an array using their \coordindex as key. Each row of ⟨*table file name or inline table*⟩ makes up one patch, defined by indices into the vertex array:



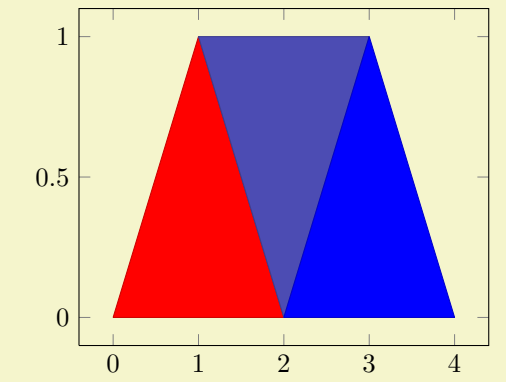

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[patch,table/row sep=\\,patch table={%
        0 1 2\\
        1 2 3\\
        4 3 5\\
    }]
    table[row sep=\\,point meta=\thisrow{c}] {
        x y c  \\
        0 0 0.2\\%  0
        1 1 0  \\%  1
        2 0 1  \\%  2
        3 1 0  \\%  3
        2 0 0.5\\%  4
        4 0 0.5\\%  5
    };
    \end{axis}
\end{tikzpicture}
```

The example consists of *two separate* tables. The patch table argument is a table, provided inline where rows are separated by \\ (which is the purpose of the row sep=\\ key as you guessed[27]). The patch table here declares three triangles: the triangle made up by vertex #0, #1 and #2, the triangle made up by #1, #2 and #3 and finally the one using the vertices #4, #3 and #5. The vertices as such are provided using the standard input methods of PGFPLOTS; in our case using a table as well. The standard input simply provides coordinates (and point meta) which are stored in the vertex array; you could also have used plot coordinates to provide them (or plot expression).

The argument to patch table needs to be a table – either a file name or an inline table as in the example above. The first n columns of this table are assumed to contain indices into the vertex array (which is made up using all vertices of the standard input as explained in the previous paragraph). The entries in this table can be provided in floating point, just make sure they are not rounded. The variable n is the number of vertices required to make up a single patch. For triangular patches, it is $n = 3$, for patch type=bilinear it is $n = 4$ and similar for other choices of patch type.

The alternative patch table with point meta is almost the same as patch table – but it allows to provide (a single) point meta (color data) per patch instead of per vertex. Here, a further column of the argument table is interpreted as color data:

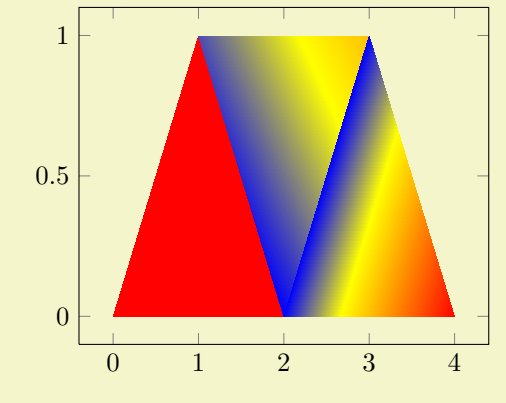
[27] Note that the choice row sep=\\ is much more robust here: newlines would be converted to spaces by TEX before PGFPLOTS had a chance to see them.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    %  this uses per-patch color data:
    \addplot[patch,table/row sep=\\,
    patch table with point meta={%
        0 1 2 100\\
        1 2 3 10\\
        4 3 5 0\\
    }]
    table[row sep=\\] {
        x y \\
        0 0 \\%  0
        1 1 \\%  1
        2 0 \\%  2
        3 1 \\%  3
        2 0 \\%  4
        4 0 \\%  5
    };
    \end{axis}
\end{tikzpicture}
```

The `patch table with point meta` always prefers `point meta` data from the provided table argument. However, it is still supported to write `point meta=\thisrow{`⟨*colname*⟩`}` or similar constructs – but now, ⟨*colname*⟩ refers to the provided table argument. More precisely, `point meta` is evaluated in a context where the patch connectivity has been resolved and the `patch table with point meta` is loaded.

The other alternative `patch table with individual point meta` is very similar, but instead of a flat color per patch, it allows to write one color value for every patch:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    %  this uses n per-patch color values:
    \addplot[patch,shader=interp,
    table/row sep=\\,
    patch table with individual point meta={%
        0 1 2 100 100 100\\%  V_0 V_1 V_2 C_0 C_1 C_2
        1 2 3 10 0 50\\
        4 3 5 0 0 100\\
    }]
    table[row sep=\\] {
        x y \\
        0 0 \\%  0
        1 1 \\%  1
        2 0 \\%  2
        3 1 \\%  3
        2 0 \\%  4
        4 0 \\%  5
    };
    \end{axis}
\end{tikzpicture}
```

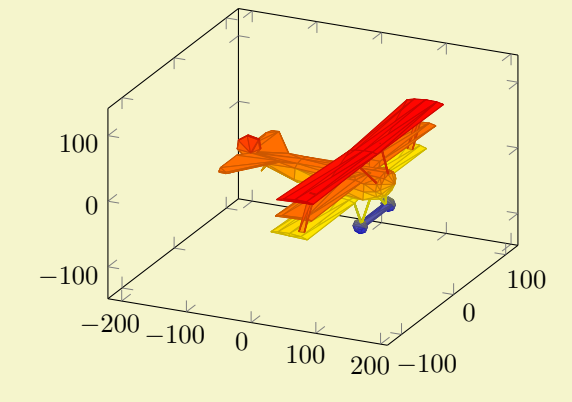
To find the `point meta` data for vertex $\#i, i = 0, 1, 2$, PGFPLOTS searches in column $i + n$ where n is the number of vertices for `patch type` (in our case, $n = 3$).

Technical remark: The key `patch table with individual point meta` automatically installs `point meta=explicit` as well. It might be confusing to override the value of `point meta` here (although it is allowed).

The `patch table` input type allows to reduce the size of geometries since vertices are stored just once. PGFPLOTS unpacks them into memory into the redundant format in order to work with single patch elements[28]. In case you experience TEX memory problems with this connectivity input, consider using the redundant format. It uses other types of memory limits.

A more involved example is shown below; it uses `\addplot3[patch]` to visualize a three dimensional `patch` plot, provided by means of a long sequence of patches:

[28] The reason for such an approach is that TEX doesn't really know what an array is – and according to my experience, arrays implemented by macros tend to blow up TEX's memory limits even faster than the alternative.

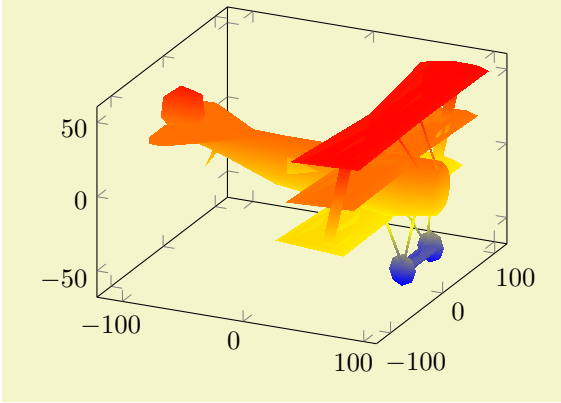



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[axis equal]
%  FokkerDrI_layer_0.patches.dat contains:
%  # each row is one vertex; three consecutive
%  # vertices make one triangle (patch)
%  105.577    -19.7332    2.85249
%  88.9233    -21.1254    13.0359
%  89.2104    -22.1547    1.46467
%  # end of facet 0
%  105.577    -19.7332    2.85249
%  105.577    -17.2161    12.146
%  88.9233    -21.1254    13.0359
%  # end of facet 1
\addplot3[patch]
    file
    {plotdata/FokkerDrI_layer_0.patches.dat};
\end{axis}
\end{tikzpicture}
```

The ordering in which triangles are specified is irrelevant, three–dimensional patch plots use `z buffer=sort` to sort patches according to their depth (defined as mean depth over each vertex), where foreground patches are drawn on top of background patches. This so–called "Painter's algorithm" works well for most meshes. If it fails, consider using `patch refines=1` or `patch refines=2` to split larger elements into small ones automatically.

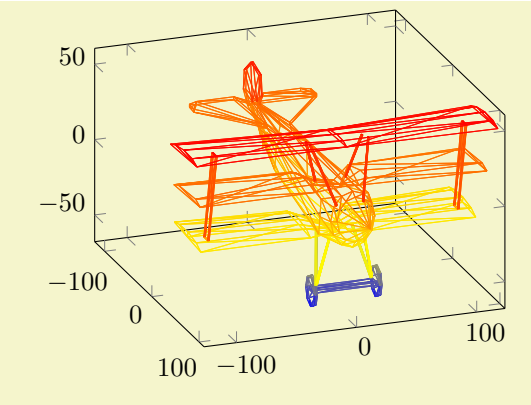
The drawing color associated to single vertices can be changed using the `point meta` key (which is the common method to configure color data in PGFPLOTS). The initial configuration is `point meta=z` for three dimensional `patch` plots, i.e. to use the z coordinate also as color data. Use `point meta=\thisrow{`⟨*colname*⟩`}` in conjunction with `\addplot3[patch] table` to load a selected table column.

Patch plots are (almost) the same as `mesh` or `surf` plots, they only have more freedom in their input format (and a more complicated geometry). Actually, "`patch`" is just a style for `surf,mesh input=patches`. In other words, `patch` *is* the same as `surf`, it even shares the same internal implementation. Thus, most of the keys to configure `mesh` or `surf` plots apply to `patch` as well, especially `shader` and `z buffer`. As already mentioned, `\addplot3[patch]` automatically activates `z buffer=sort` to ensure a good drawing sequence. The `shader` can be used to modify the appearance:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
%  FokkerDrI_layer_0.facetIdx.dat contains:
%  # each row makes up one facet; it
%  # consists of 0-based indices into
%  # the vertex array
%  0    1    2 % triangle of vertices #0,#1 and #2
%  0    3    1 % triangle of vertices #0,#3 and #1
%  3    4    1
%  5    6    7
%  6    8    7
%  8    9    7
%  8    10    9
%  ...
%  while FokkerDrI_layer_0.vertices.dat contains
%  105.577    -19.7332    2.85249    % vertex #0
%  88.9233    -21.1254    13.0359    % vertex #1
%  89.2104    -22.1547    1.46467    % vertex #2
%  105.577    -17.2161    12.146
%  105.577    -10.6054    18.7567
%  105.577    7.98161    18.7567
%  105.577    14.5923    12.146
%  ...
\addplot3[patch,shader=interp,
    patch table=
        {plotdata/FokkerDrI_layer_0.facetIdx.dat}]
    file
    {plotdata/FokkerDrI_layer_0.vertices.dat};
\end{axis}
\end{tikzpicture}
```

See the description of shader=interp for details and remarks. The example above makes use of the alternative syntax to provide a geometry: the patch table input. It allows to provide vertices separate from patch connectivity, where each patch is defined using three indices into the vertex array as discussed above.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[view/h=70]
%  FokkerDrI_layer_0.patches.dat contains:
%  # each row is one vertex; three consecutive
%  # vertices make one triangle (patch)
%  105.577    -19.7332    2.85249
%  88.9233    -21.1254    13.0359
%  89.2104    -22.1547    1.46467
%  # end of facet 0
%  105.577    -19.7332    2.85249
%  105.577    -17.2161    12.146
%  88.9233    -21.1254    13.0359
%  # end of facet 1
\addplot3[patch,mesh]
    file
    {plotdata/FokkerDrI_layer_0.patches.dat};
\end{axis}
\end{tikzpicture}
```

/pgfplots/mesh input=lattice|patches

This key controls how input coordinates are decoded to get patches. It is used only if patch table is empty (patch table has its own way to decode input coordinates). Usually, you won't need to bother with this key as it is set implicitly.

The choice mesh input=lattice is the initial configuration for mesh and surf plots: it expects input in a compact matrix form as described at the beginning of this section starting with page 85 and requires a mesh/ordering and perhaps end–of–scanline markers. It yields patches with exactly four corners and is compatible with patch type=rectangle and patch type=bilinear (the latter requiring to load the patchplots library).

The choice mesh input=patches is implicitly set when you use the patch style (remember that surf is actually some sort of patch plot on its own). It expects the input format as described for patch plots, i.e. n consecutive coordinates make up the vertices of a single patch where n is the expected number of vertices for the configured patch type.

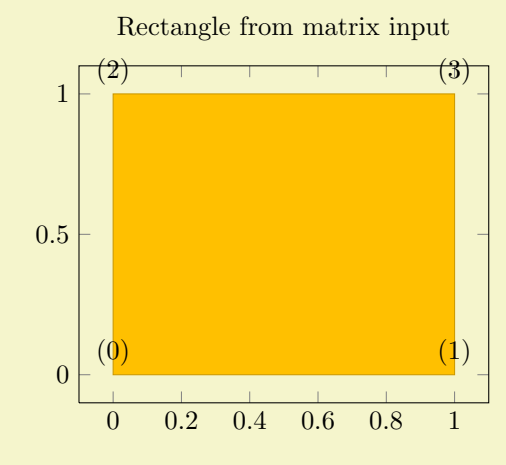
Note that a non–empty patch table implies mesh input=patches.

/pgfplots/patch type=default|rectangle|triangle|line (initially default)

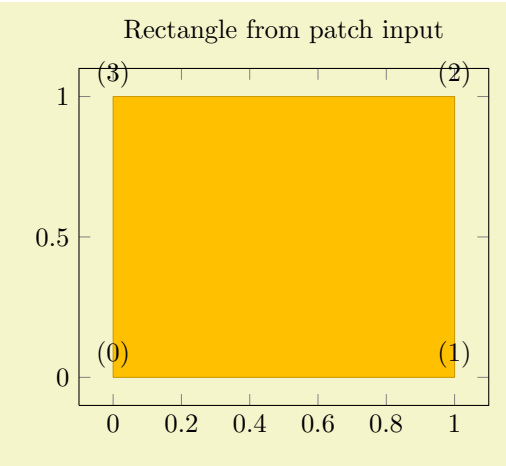
Defines the type of patch.

The initial configuration patch type=default checks the configuration of mesh input: for mesh input=patches, it uses triangle. For mesh input=lattice, it checks if there is just one row or just one col and uses patch type=line in such a case, otherwise it uses patch type=rectangle.

The choice patch type=rectangle expects $n = 4$ vertices. The vertices can be either encoded as a matrix or, using mesh input=patches, in the sequence in which you would connect the vertices:



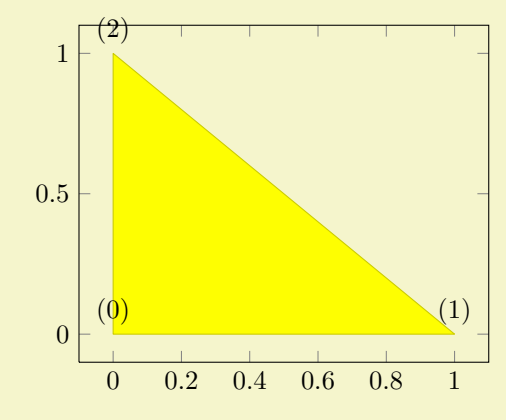

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[nodes near coords={(\coordindex)},
        title=Rectangle from matrix input]
    %  note that surf implies 'patch type=rectangle'
    \addplot[surf,mesh/rows=2,patch type=rectangle]
    coordinates {
        (0,0) (1,0)
        (0,1) (1,1)
    };
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[nodes near coords={(\coordindex)},
        title=Rectangle from patch input]
    \addplot[patch,patch type=rectangle]
    coordinates {
        (0,0) (1,0) (1,1) (0,1)
    };
    \end{axis}
\end{tikzpicture}
```

As for all other `patch type` values, the vertices can be arbitrary two– or three–dimensional points, there may be even two on top of each other (resulting in a triangle). When used together with `shader=interp`, `patch type=rectangle` is visualized using two Gouraud shaded triangles (see below for `triangle`). It is the *most efficient* representation for interpolated shadings together with `mesh input=lattice` since the input lattice is written directly into the pdf. Use `patch type=rectangle` if you want rectangular elements and perhaps "some sort" of smooth shading. Use `patch type=bilinear` of the `patchplots` library in case you need real bilinear shading.

The choice `patch type=triangle` expects $n = 3$ vertices which make up a triangle. The ordering of the vertices is irrelevant:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[nodes near coords={(\coordindex)}]
    \addplot[patch,patch type=triangle]
    coordinates {
        (0,0) (1,0) (0,1)
    };
    \end{axis}
\end{tikzpicture}
```

The use of `shader=interp` is realized by means of linear interpolation of the three color values (specified with the `point meta` key) between the corners; the resulting interpolated `point meta` values are then mapped into the actual `colormap`. This type of interpolation is called Gouraud

shading.

The choice `patch type=line` expects $n = 2$ vertices which make up a line. It is used for one-dimensional `mesh` plots (see Section 4.4.11 for examples).

There are more values for `patch type` like `bilinear`, `triangle quadr`, `biquadratic`, `coons`, `polygon` and `tensor bezier`. Please refer to the separate `patchplots` library in Section 5.6.

`/pgfplots/every patch` (style, no value)

This style will be installed as soon as the `patch` plot handler is activated.

The initial configuration is

```
\pgfplotsset{
    every patch/.style={miter limit=1}
}
```

which improves display of sharp triangle corners significantly (see the Ti*k*Z manual for details about `miter limit` and line join parameters).

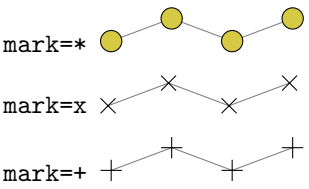
There is much more to say about patch plots, like `patch type` which allows triangles, bilinear elements, quadratic triangles, biquadratic quadrilaterals, coons patches; the `patch refines` key which allows automatic refinement, `patch to triangles` which triangulates higher order elements; how matrix data can be used for rectangular shapes and more. These details are subject of the `patchplots` library in Section 5.6.

4.6 Markers, Linestyles, (Background-) Colors and Colormaps

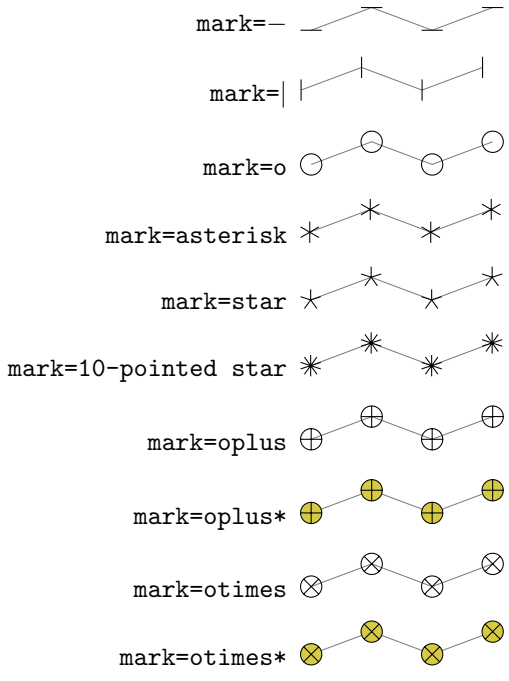
The following options of Ti*k*Z are available to plots.

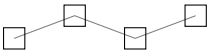
4.6.1 Markers

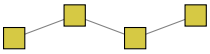
This list is copied from [5, section 29]:

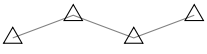


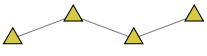
And with `\usetikzlibrary{plotmarks}`:

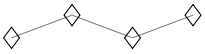


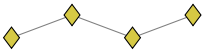
mark=square 

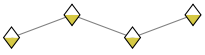
mark=square* 

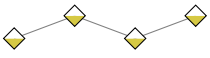
mark=triangle 

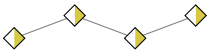
mark=triangle* 

mark=diamond 

mark=diamond* 

mark=halfdiamond* 

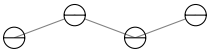
mark=halfsquare* 

mark=halfsquare right* 

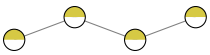
mark=halfsquare left* 

mark=Mercedes star 

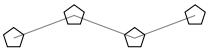
mark=Mercedes star flipped 

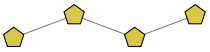
mark=halfcircle 

One half is filled with white (more precisely, with `mark color`).

mark=halfcircle* 

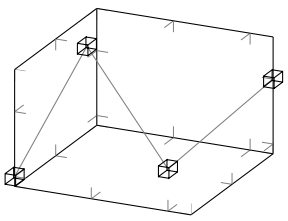
One half is filled with white (more precisely, with `mark color`) and the other half is filled with the actual `fill` color.

mark=pentagon 

mark=pentagon* 

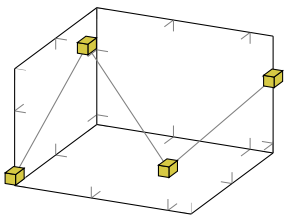
mark=text 

This marker is special as it can be configured freely. The character (or even text) used is configured by a set of variables, see below.

mark=cube 

This marker is only available inside of a PGFPLOTS axis, it draws a cube with axis parallel faces. Its dimensions can be configured separately, see below.

mark=cube*



`User defined` It is possible to define new markers with `\pgfdeclareplotmark`, see below.

All these options have been drawn with the additional options

```
\draw[
    gray,
    thin,
    mark options={%
        scale=2,fill=yellow!80!black,draw=black
    }
]
```

Please see Section 4.6.5 for how to change `draw` and `fill` colors. Note that each of the provided marks can be rotated freely by means of `mark options={rotate=90}` or `every mark/.append style={rotate=90}`.

`/tikz/mark size={`⟨*dimension*⟩`}`

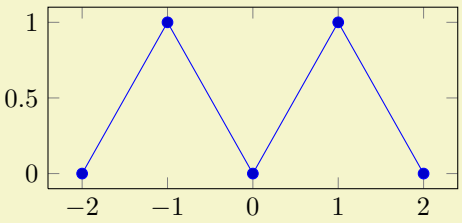
This Ti*k*Z option allows to set marker sizes to ⟨*dimension*⟩. For circular markers, ⟨*dimension*⟩ is the radius, for other plot marks it is about half the width and height.

`/pgfplots/cube/size x={`⟨*dimension*⟩`}` (initially `\pgfplotmarksize`)
`/pgfplots/cube/size y={`⟨*dimension*⟩`}` (initially `\pgfplotmarksize`)
`/pgfplots/cube/size z={`⟨*dimension*⟩`}` (initially `\pgfplotmarksize`)

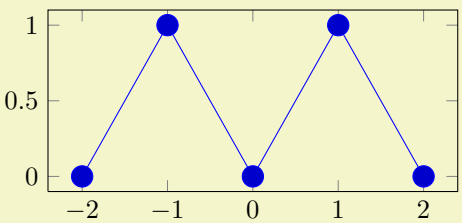
Sets the size for `mark=cube` separately for every axis.

`/tikz/every mark` (no value)

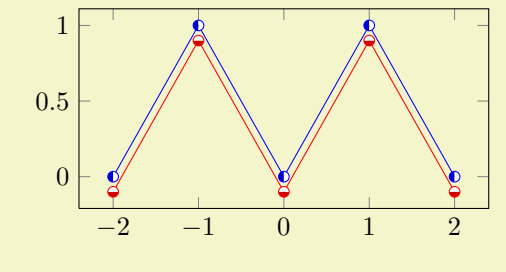
This Ti*k*Z style can be reconfigured to set marker appearance options like colors or transformations like scaling or rotation. PGFPLOTS appends its `cycle list` options to this style.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[y=2cm]
    \addplot coordinates
        {(-2,0) (-1,1) (0,0) (1,1) (2,0)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\tikzset{every mark/.append style={scale=2}}
\begin{tikzpicture}
\begin{axis}[y=2cm]
    \addplot coordinates
        {(-2,0) (-1,1) (0,0) (1,1) (2,0)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[y=2cm]
  \addplot+[
    mark=halfcircle*,
    every mark/.append style={rotate=90}]
  coordinates
    {(-2,0) (-1,1) (0,0) (1,1) (2,0)};

  \addplot+[
    mark=halfcircle*,
    every mark/.append style={rotate=180}]
  coordinates
    {(-2,-0.1) (-1,0.9) (0,-0.1) (1,0.9) (2,-0.1)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/no markers (style, no value)

Disables plot marks.

If this style is provided as argument to a complete axis, it is appended to every axis plot post such that it disables markers even for cycle lists which contain markers.

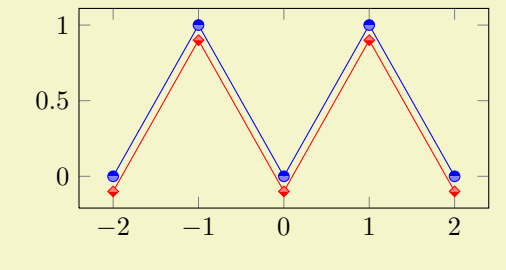
/tikz/mark repeat={⟨*integer*⟩} (initially empty)

Allows to draw only each nth mark where n is provided as ⟨*integer*⟩.

/pgf/mark color={⟨*color*⟩} (initially empty)

Defines the *additional* fill color for the halfcircle, halfcircle*, halfdiamond* and halfsquare* markers. An empty value uses white (which is the initial configuration). The value none disables filling for this part.

These markers have two distinct fill colors, one is determined by fill as for any other marker and the other one is mark color.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[y=2cm]
  \addplot[
    blue,mark color=blue!50!white,
    mark=halfcircle*]
  coordinates
    {(-2,0) (-1,1) (0,0) (1,1) (2,0)};

  \addplot[
    red,mark color=red!50!white,
    mark=halfsquare*]
  coordinates
    {(-2,-0.1) (-1,0.9) (0,-0.1) (1,0.9) (2,-0.1)};
\end{axis}
\end{tikzpicture}
```

Note that this key requires PGF 2.10 or later.

/tikz/mark options={⟨*options*⟩}

Resets every mark to {⟨*options*⟩}.

/pgf/text mark={⟨*text*⟩} (initially p)

Changes the text shown by mark=text.

With /pgf/text mark=m: m m m m

With /pgf/text mark=A: A A A A

There is no limitation about the number of characters or whatever. In fact, any TEX material can be inserted as ⟨*text*⟩, including images.

/pgf/text mark style={⟨*options for* ***mark=text***⟩}

Defines a set of options which control the appearance of mark=text.

If /pgf/text mark as node=false (the default), ⟨*options*⟩ is provided as argument to \pgftext – which provides only some basic keys like left, right, top, bottom, base and rotate.

If /pgf/text mark as node=true, ⟨*options*⟩ is provided as argument to \node. This means you can provide a very powerful set of options including anchor, scale, fill, draw, rounded corners etc.

/pgf/text mark as node=true|false (initially false)

Configures how mark=text will be drawn: either as \node or as \pgftext.

The first choice is highly flexible and possibly slow, the second is very fast and usually enough.

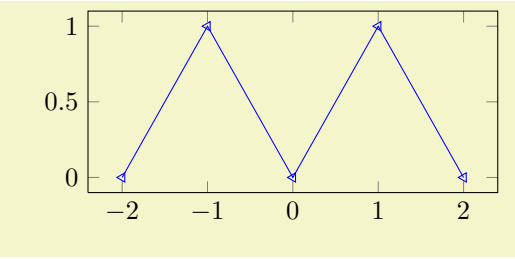
\pgfdeclareplotmark{⟨*plot mark name*⟩}{⟨*code*⟩}

Defines a new marker named ⟨*plot mark name*⟩. Whenever it is used, ⟨*code*⟩ will be invoked. It is supposed to contain (preferrable PGF basic level) drawing commands. During ⟨*code*⟩, the coordinate system's origin denotes the coordinate where the marker shall be placed.

Please refer to [5] section "Mark Plot Handler" for more detailed information.

/pgfplots/every axis plot post (style, initially)

The every axis plot post style can be used to overwrite parts (or all) of the drawing styles which are assigned for plots.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  Overwrite any cycle list:
\pgfplotsset{
  every axis plot post/.append style={
   mark=triangle,
   every mark/.append style={rotate=90}}}
\begin{tikzpicture}
\begin{axis}[y=2cm]
    \addplot coordinates
        {(-2,0) (-1,1) (0,0) (1,1) (2,0)};
\end{axis}
\end{tikzpicture}
```

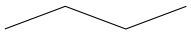
Markers paths are not subjected to clipping as other parts of the figure. Markers are either drawn completely or not at all.

Ti*k*Z offers more options for marker fine tuning, please refer to [5] for details.

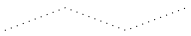
4.6.2 Line Styles

The following line styles are predefined in Ti*k*Z.

/tikz/solid (style, no value)



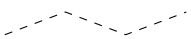
/tikz/dotted (style, no value)



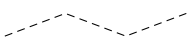
/tikz/densely dotted (style, no value)

/tikz/loosely dotted (style, no value)

/tikz/dashed (style, no value)



/tikz/densely dashed (style, no value)



/tikz/loosely dashed (style, no value)

/tikz/dashdotted (style, no value)

/tikz/densely dashdotted (style, no value)

/tikz/loosely dashdotted (style, no value)

/tikz/dashdotdotted (style, no value)

/tikz/densely dashdotdotted (style, no value)

/tikz/loosely dashdotdotted (style, no value)

since these styles apply to markers as well, you may want to consider using

```
\pgfplotsset{
    every mark/.append style={solid}
}
```

in marker styles.

Besides linestyles, PGF also offers (a lot of) arrow heads. Please refer to [5] for details.

4.6.3 Edges and Their Parameters

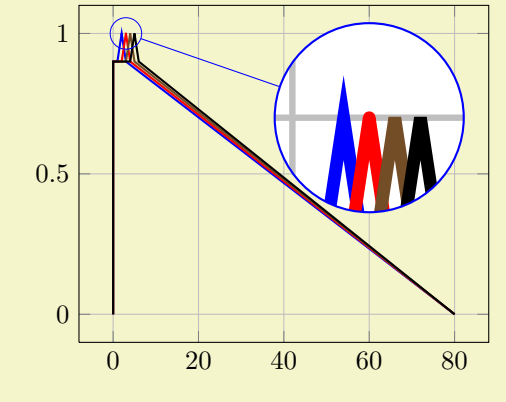
When PGFPLOTS connects points, it relies on PGF drawing parameters to create proper edges (and it only changes them in the `every patch` style).

It might occasionally be necessary to change these parameters:

/tikz/line cap=round|rect|butt (initially `butt`)
/tikz/line join=round|bevel|miter (initially `miter`)
/tikz/miter limit=⟨*factor*⟩ (initially 10)

These keys control how lines are joined at edges. Their description is beyond the scope of this manual, so interested readers should consult [5].

Here is just an example illustrating why it might be of interest to study these parameters:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usetikzlibrary{spy}
\begin{tikzpicture}[spy using outlines=
    {circle, magnification=6, connect spies}]
\begin{axis}[no markers,grid=major,
    every axis plot post/.append style={thick}]
\addplot  coordinates
 {(0, 0.0) (0, 0.9) (1, 0.9) (2, 1) (3, 0.9) (80, 0)};
\addplot +[line join=round] coordinates
 {(0, 0.0) (0, 0.9) (2, 0.9) (3, 1) (4, 0.9) (80, 0)};
\addplot +[line join=bevel] coordinates
 {(0, 0.0) (0, 0.9) (3, 0.9) (4, 1) (5, 0.9) (80, 0)};
\addplot +[miter limit=5] coordinates
 {(0, 0.0) (0, 0.9) (4, 0.9) (5, 1) (6, 0.9) (80, 0)};

  \coordinate (spypoint) at (axis cs:3,1);
  \coordinate (magnifyglass) at (axis cs:60,0.7);
\end{axis}

\spy [blue, size=2.5cm] on (spypoint)
   in node[fill=white] at (magnifyglass);
\end{tikzpicture}
```

4.6.4 Font Size and Line Width

Often, one wants to change line width and font sizes for plots. This can be done using the following options of Ti*k*Z.

/tikz/font={⟨*font name*⟩} (initially \normalfont)

Sets the font which is to be used for text in nodes (like tick labels, legends or descriptions).

A font can be any LaTeX argument like \footnotesize or \small\bfseries[29].

It may be useful to change fonts only for specific axis descriptions, for example using

```
\pgfplotsset{
    tick label style={font=\small},
    label style={font=\small},
    legend style={font=\footnotesize}
}
```

See also the predefined styles normalsize, small and footnotesize in Section 4.8.14.

/tikz/line width={⟨*dimension*⟩} (initially 0.4pt)

Sets the line width. Please note that line widths for tick lines and grid lines are predefined, so it may be necessary to override the styles every tick and every axis grid.

The line width key is changed quite often in Ti*k*Z. You should use

```
\pgfplotsset{every axis/.append style={line width=1pt}}
```

or

```
\pgfplotsset{every axis/.append style={thick}}
```

to change the overall line width. To also adjust ticks and grid lines, one can use

```
\pgfplotsset{every axis/.append style={
    line width=1pt,
    tick style={line width=0.6pt}}}
```

or styles like

```
\pgfplotsset{every axis/.append style={
    thick,
    tick style={semithick}}}
```

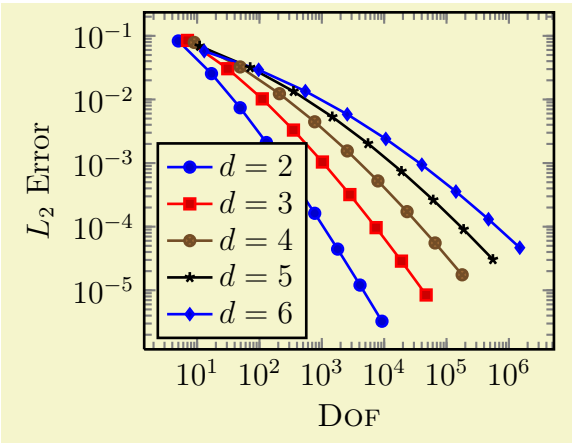
The 'every axis plot' style can be used to change line widths for plots only.

/tikz/thin (no value)
/tikz/ultra thin (no value)
/tikz/very thin (no value)
/tikz/semithick (no value)
/tikz/thick (no value)
/tikz/very thick (no value)
/tikz/ultra thick (no value)

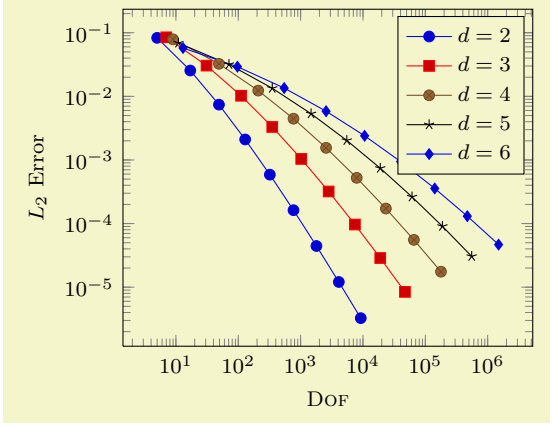
These Ti*k*Z styles provide different predefined line widths.

This example shows the same plots as on page 17 (using \plotcoords as place holder for the commands on page 17), with different line widths and font sizes.

[29] ConTeXt and plain TeX users need to provide other statements, of course.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={
    font=\large,
    line width=1pt,
    tick style={line width=0.8pt}}}
\begin{tikzpicture}
    \begin{loglogaxis}[
        legend style={at={(0.03,0.03)},
            anchor=south west},
        xlabel=\textsc{Dof},
        ylabel=$L_2$ Error
    ]
    %  see above for this macro:
    \plotcoords
    \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
    \end{loglogaxis}
\end{tikzpicture}
```

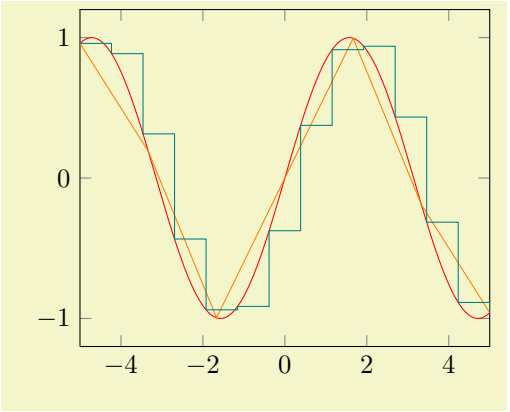



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={
    font=\footnotesize,
    thin,
    tick style={ultra thin}}}
\begin{tikzpicture}
    \begin{loglogaxis}[
        xlabel=\textsc{Dof},
        ylabel=$L_2$ Error
    ]
    %  see above for this macro:
    \plotcoords
    \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
    \end{loglogaxis}
\end{tikzpicture}
```

4.6.5 Colors

PGF uses the color support of `xcolor`. Therefore, the main reference for how to specify colors is the `xcolor` manual [3]. The PGF manual [5] is the reference for how to select colors for specific purposes like drawing, filling, shading, patterns etc. This section contains a short overview over the specification of colors in [3] (which is not limited to PGFPLOTS).

The package `xcolor` defines a set of predefined colors, namely `red`, `green`, `blue`, `cyan`, `magenta`, `yellow`, `black`, `gray`, `white`, `darkgray`, `lightgray`, `brown`, `lime`, `olive`, `orange`, `pink`, `purple`, `teal`, `violet`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[enlarge x limits=false]
    \addplot[red,samples=500] {sin(deg(x))};

    \addplot[orange,samples=7] {sin(deg(x))};

    \addplot[teal,const plot,
        samples=14] {sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

Besides predefined colors, it is possible to *mix* two (or more) colors. For example, `red!30!white` contains 30% of `red` and 70% of `white`. Consequently, one can build `red!70!white` to get 70% red and 30% white or `red!10!white` for 10% red and 90% white. This mixing can be done with any color, for example `red!50!green`, `blue!50!yellow` or `green!60!black`.

A different type of color mixing is supported, which allows to take 100% of *each* component. For example, `rgb,2:red,1;green,1` will add 1/2 part `red` and 1/2 part `green` and we repro-

duced the example from above. Using the denominator 1 instead of 2 leads to rgb,1:red,1;green,1 which uses 1 part red and 1 part green. Many programs allow to select pieces between $0, \ldots, 255$, so a denominator of 255 is useful. Consequently, rgb,255:red,231;green,84;blue,121 uses 231/255 red, 84/255 green and 121/255. This corresponds to the standard RGB color $(231, 84, 121)$. Other examples are rgb,255:red,32;green,127;blue,43, rgb,255:red,178;green,127;blue,43, rgb,255:red,169;green,178;blue,43.

It is also possible to use RGB values, the HSV color model, the CMY (or CMYK) models, or the HTML color syntax directly. However, this requires some more programming. I suppose this is the fastest (and probably the most uncomfortable) method to use colors. For example,

```
\definecolor{color1}{rgb}{1,1,0}
\tikz \fill[color1]
    (0,0) rectangle (1em,0.6em);
```

creates the color with 100% red, 100% green and 0% blue;

```
\definecolor{color1}{cmyk}{0.6,0.9,0.5,0.1}
\tikz \fill[color1]
    (0,0) rectangle (1em,0.6em);
```

creates the color with 60% cyan, 90% magenta, 50% yellow and 10% black;

```
\definecolor{color1}{HTML}{D0B22B}
\tikz \fill[color1]
    (0,0) rectangle (1em,0.6em);
```

creates the color with 208/255 pieces red, 178/255 pieces green and 43 pieces blue, specified in standard HTML notation. Please refer to the xcolor manual [3] for more details and color models.

The xcolor package provides even more methods to combine colors, among them the prefix '-' (minus) which changes the color into its complementary color (-black, -white, -red) or color wheel calculations. Please refer to the xcolor manual [3].

/tikz/color={⟨*a color*⟩}
/tikz/draw={⟨*stroke color*⟩}
/tikz/fill={⟨*fill color*⟩}

These keys are (generally) used to set colors. Use color to set the color for both drawing and filling. Instead of color={⟨*color name*⟩} you can simply write ⟨*color name*⟩. The draw and fill keys only set colors for stroking and filling, respectively.

Use draw=none to disable drawing and fill=none to disable filling[30].

Since these keys belong to Ti*k*Z, the complete documentation can be found in the Ti*k*Z manual [5, Section "Specifying a Color"].

4.6.6 Color Maps

/pgfplots/colormap name={⟨*color map name*⟩} (initially hot)

Changes the current color map to the already defined map named ⟨*color map name*⟩. The predefined color map is

hot 

The definition can be found in the documentation for colormap/hot. This, and further color maps, are described below.

Colormaps can be used, for example, in scatter plots (see Section 4.4.10).

You can use colormap to create new color maps (see below).

/pgfplots/colormap={⟨*name*⟩}{⟨*color specification*⟩}

Defines a new colormap named ⟨*name*⟩ according to ⟨*color specification*⟩ and activates it using colormap name={⟨*name*⟩}.

[30] Up to now, plot marks always have a stroke color (some also have a fill color). This restriction may be lifted in upcoming versions.

The ⟨*color specification*⟩ is a sequence of positions and associated colors where linear interpolation is applied in-between. The syntax is very similar as the one used for PGF shadings described in [5, VIII – Shadings]: it is a semicolon–separated series of

⟨*color type*⟩`(`⟨*offset*⟩`)=(`⟨*color value*⟩`);` :

```
%  possibility 1: like PGF shadings:
rgb(0cm)=(1,0,0); rgb(1cm)=(0,1,0); rgb255(2cm)=(0,0,255); gray(3cm)=(0.3);  color(4cm)=(green)
```

If the distance between successive colors is the same, the ⟨*offset*⟩ can be omitted. The '`;`' separators are not necessary either:

```
%  (simplified) possibility 2: skip ';' and length arguments:
rgb=(1,0,0) rgb=(0,1,0) rgb255=(0,0,255) gray=(0.3) color=(green)
```

It is also possible to provide non-uniform distances between the different colors – if all single positions can be projected onto a uniform grid. PGFPLOTS will perform this interpolation automatically:

```
%  non uniform spacing example: the mesh width is provided as first
%  part of the specification.
\pgfplotsset{colormap={violetnew}
    {[1cm] rgb255(0cm)=(25,25,122) color(1cm)=(white) rgb255(5cm)=(238,140,238)}}
```

In this last example, the mesh width has been provided explicitly and PGFPLOTS interpolates the missing grid points on its own. It is an error if the provided positions are no multiple of the mesh width. The `\pgfplotsset` employs the public user interface to create a new color map named '`violetnew`'.

The single colors can be separated by semicolons '`;`'. The (optional) length describes how much of the bar is occupied by the interval, it is interpreted relative to the complete length. If the length argument is missing, it is taken to be the last specified length plus the last length difference (the first color defaults to `1cm` in this case).

Summary of the expected input format: Each entry in ⟨*color specification*⟩ has the form ⟨*color model*⟩`(`⟨*length*⟩`)=(`⟨*arguments*⟩`)`. Here, the ⟨*length*⟩ argument is optional as discussed above. The entries can be separated by semicolons '`;`' or by white spaces. The leftmost entry *must* have ⟨*length*⟩`=0pt`. As discussed, all entries will be placed on a uniform grid, i.e. the distance between adjacent ⟨*length*⟩ arguments has to be the same (see the previous paragraph for automatic generation of intermediate points). The complete length of a color map is irrelevant: it will be mapped linearly to an internal range anyway (for efficient interpolation). The only requirement is that the left end must be at `0`.

Available choices for ⟨*color model*⟩ are

rgb which expects ⟨*arguments*⟩ of the form `(`⟨*red*⟩`,`⟨*green*⟩`,`⟨*blue*⟩`)` where each component is in the interval $[0, 1]$,

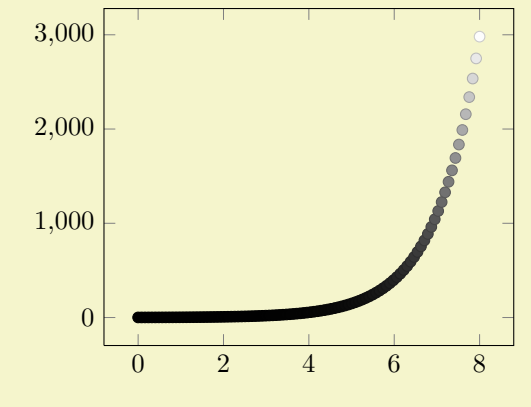
rgb255 which is similar to `rgb` except that each component is expected in the interval `[0,255]`,

gray in which case ⟨*arguments*⟩ is a single number in the interval $[0, 1]$,

color in which case ⟨*arguments*⟩ contains a predefined (named) color like '`red`' or a color expression like '`red!50`',

cmyk which expects ⟨*arguments*⟩ of the form `(`⟨*cyan*⟩`,`⟨*magenta*⟩`,`⟨*yellow*⟩`,`⟨*black*⟩`)` where each component is in the interval $[0, 1]$, and

cmyk255 which is the same as `cmyk` but expects components in the interval $[0, 255]$.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        colormap={bw}{gray(0cm)=(0); gray(1cm)=(1)}]
    \addplot+[scatter,only marks,
         domain=0:8,samples=100]
        {exp(x)};
    \end{axis}
\end{tikzpicture}
```

The color space of a colormap. There are two supported color spaces for a `colormap`: the RGB color space and the CMYK color space. Each access into a `colormap` requires linear interpolation which is performed in its color space. Color spaces make a difference: colors in different color spaces may be represented differently, depending on the output device. Many printers use CMYK for color printing, so providing CMYK colors might improve the printing quality on a color printer. The RGB color space is often used for display devices. The predefined `colormap`s in PGFPLOTS all use RGB.

Whenever a new `colormap` is created, PGFPLOTS determines an associated color space. Then, each color in this specific `colormap` will be represented in its associated color space (converting colors automatically if necessary). Furthermore, every access into the `colormap` will be performed in its associated color space and every returned `mapped color` will be represented with respect to this color space. Furthermore, every shading generated by `shader=interp` will be represented with respect to the `colormap`'s associated color space.

The color space is chosen as follows: in case `colormap default colorspace=auto` (the initial configuration), the color space depends on the *first* encountered color in ⟨*color specification*⟩. For `rgb` or `gray` or `color`, the associated color space will be RGB (as it was in all earlier versions of PGFPLOTS). For `cmyk`, the associated color space will be CMYK. If `colormap default colorspace` is either `rgb` or `cmyk`, this specific color space is used and every color is converted automatically.

`/pgfplots/colormap default colorspace=auto|rgb|cmyk` (initially `auto`)

Allows to set the color space of every *newly created* `colormap`. The choices are explained in the previous paragraph.

It is (not yet) possible to change the color space of an existing `colormap`; re-create it if conversion is required.

The macro `\pgfplotscolormapgetcolorspace{`⟨*name*⟩`}` defines `\pgfplotsretval` to contain the color space of an existing `colormap name`, if you are in doubt.

Available color maps are shown below.

`/pgfplots/colormap/hot` (style, no value)

A style which installs the colormap

```
\pgfplotsset{
    colormap={hot}{color(0cm)=(blue); color(1cm)=(yellow); color(2cm)=(orange); color(3cm)=(red)}
}
```

This is the preconfigured color map.

`/pgfplots/colormap/hot2` (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={hot2}{[1cm]rgb255(0cm)=(0,0,0) rgb255(3cm)=(255,0,0)
        rgb255(6cm)=(255,255,0) rgb255(8cm)=(255,255,255)}
}
```

Note that this particular choice ships directly with PGFPLOTS, you do not need to load the colormaps library for this value.

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/jet (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={jet}{rgb255(0cm)=(0,0,128) rgb255(1cm)=(0,0,255)
        rgb255(3cm)=(0,255,255) rgb255(5cm)=(255,255,0) rgb255(7cm)=(255,0,0) rgb255(8cm)=(128,0,0)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/blackwhite (style, no value)

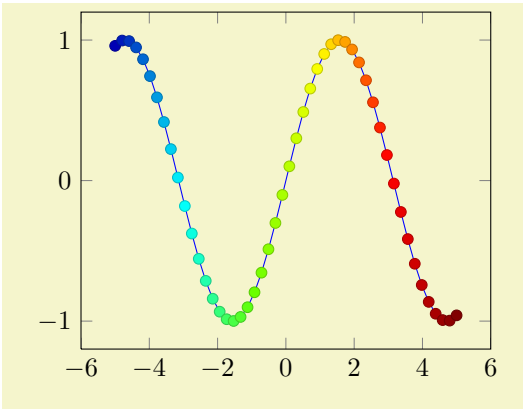
A style which is equivalent to

```
\pgfplotsset{
    colormap={blackwhite}{gray(0cm)=(0); gray(1cm)=(1)}
}
```

/pgfplots/colormap/bluered (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    colormap={bluered}{
        rgb255(0cm)=(0,0,180); rgb255(1cm)=(0,255,255); rgb255(2cm)=(100,255,0);
        rgb255(3cm)=(255,255,0); rgb255(4cm)=(255,0,0); rgb255(5cm)=(128,0,0)}
}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colormap/bluered]
    \addplot+[scatter,
        scatter src=x,samples=50]
        {sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

Remark: The style `bluered` (re-)defines the color map and activates it. TEX will be slightly faster if you call `\pgfplotsset{colormap/bluered}` in the preamble (to create the color map once) and use `colormap name=bluered` whenever you need it. This remark holds for every color map style which follows. But you can simply ignore this remark.

/pgfplots/colormap/cool (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    colormap={cool}{rgb255(0cm)=(255,255,255); rgb255(1cm)=(0,128,255); rgb255(2cm)=(255,0,255)}
}
```

/pgfplots/colormap/greenyellow (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    colormap={greenyellow}{rgb255(0cm)=(0,128,0); rgb255(1cm)=(255,255,0)}
}
```

/pgfplots/colormap/redyellow (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    colormap={redyellow}{rgb255(0cm)=(255,0,0); rgb255(1cm)=(255,255,0)}
}
```

/pgfplots/colormap/violet (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    colormap={violet}{rgb255=(25,25,122) color=(white) rgb255=(238,140,238)}
}
```

\pgfplotscolormaptoshadingspec{⟨*colormap name*⟩}{⟨*right end size*⟩}{⟨*macro*⟩}

A command which converts a colormap into a PGF shading's color specification. It can be used in commands like `\pgfdeclare*shading` (see the PGF manual [5] for details).

The first argument is the name of a (defined) colormap, the second the rightmost dimension of the specification. The result will be stored in ⟨*macro*⟩.

```
%  convert 'hot' -> \result
\pgfplotscolormaptoshadingspec{hot}{8cm}\result
%  define and use a shading in pgf:
\def\tempb{\pgfdeclarehorizontalshading{tempshading}{1cm}}%
%  where '\result' is inserted as last argument:
\expandafter\tempb\expandafter{\result}%
\pgfuseshading{tempshading}%
```

The usage of the result ⟨*macro*⟩ is a little bit low–level.

Attention: PGF shadings are always represented with respect to the RGB color space. Consequently, even CMYK ⟨*colormap name*⟩s will result in an RGB shading specification when using this method[31].

Note that there *more available choices* in the `colormaps` library which needs to be loaded by means of `\usepgfplotslibrary{colormaps}`.

4.6.7 Cycle Lists – Options Controlling Line Styles

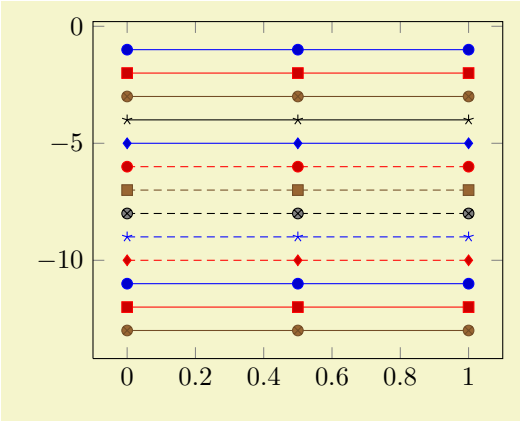
`/pgfplots/cycle list=`{⟨*list*⟩}
`/pgfplots/cycle list name=`{⟨*macro*⟩}

Allows to specify a list of plot specifications which will be used for each `\addplot` command without explicit plot specification. Thus, the currently active `cycle list` will be used if you write either `\addplot+`[⟨*keys*⟩] `...;` or if you *don't* use square brackets as in `\addplot`[⟨*explicit plot specification*⟩] `...;`.

The list element with index i will be chosen where i is the index of the current `\addplot` command (see also the `cycle list shift` key which allows to use $i+n$ instead). This indexing does also include plot commands which don't use the `cycle list`.

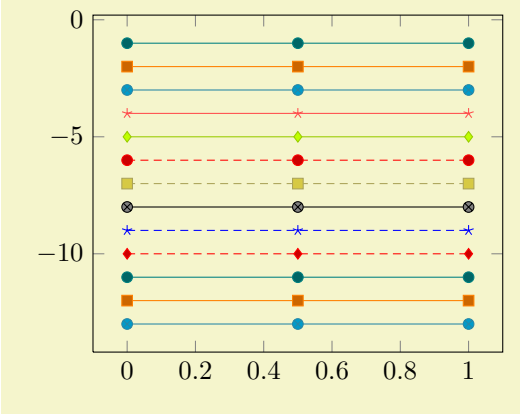
There are several possibilities to change the currently active `cycle list`:

1. Use one of the predefined lists[32],
   - `color` (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=color]
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

   - `exotic` (from top to bottom)

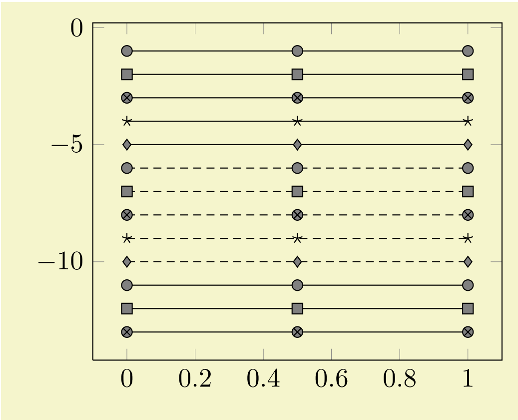



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=exotic]
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

[31] In case PGF should someday support CMYK shadings and you still see this remark, you can define `\def\pgfplotscolormaptoshadingspectorgb{0}`.

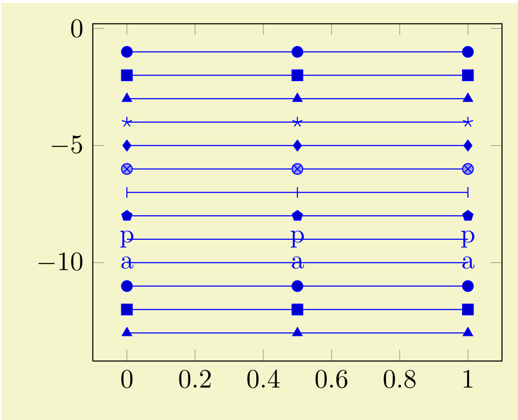
[32] In an early version, these lists were called `\coloredplotspeclist` and `\blackwhiteplotspeclist` which appeared to be unnecessarily long, so they have been renamed. The old names are still accepted, however.

- black white (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=black white]
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

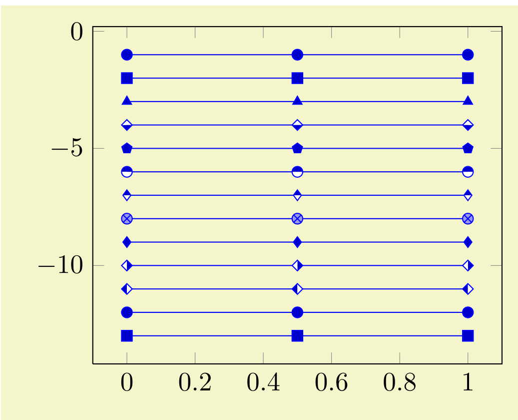
- mark list (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=mark list]
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

The mark list always employs the current color, but it doesn't define one (the \addplot+ statement explicitly sets the current color to blue).
The mark list is especially useful in conjunction with cycle multi list which allows to combine it with other lists (for example linestyles or a list of colors).

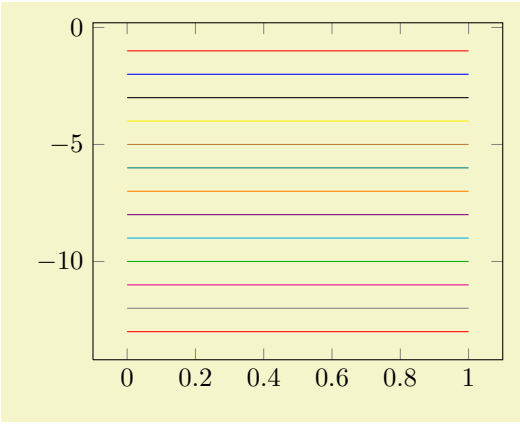
- mark list* A list containing only markers. In contrast to mark list, all these markers are filled. They are defined as (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=mark list*]
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\addplot+[blue] coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

Similar to `mark list`, the `mark list*` always employs the current color, but it doesn't define one (see above for the `\addplot+`).

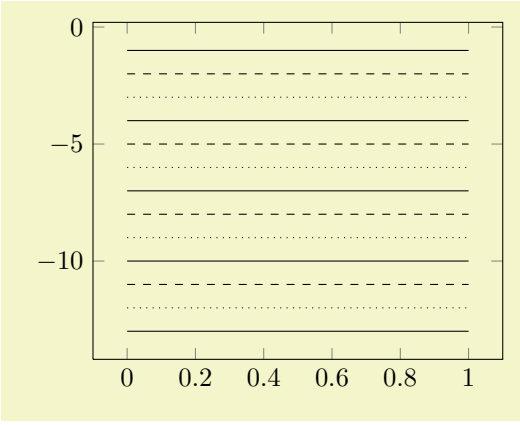
- `color list` (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=color list]
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

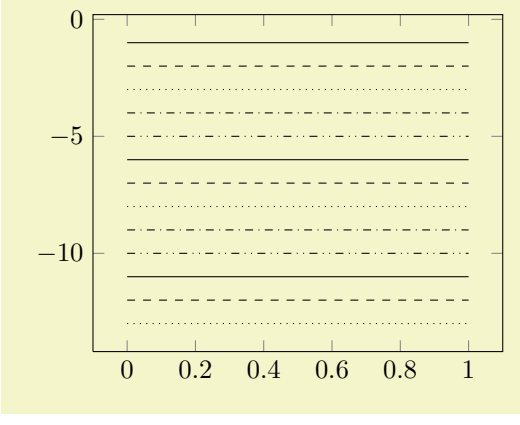
The `cycle list name=color` choice also employs markers whereas `color list` uses *only* colors.

- `linestyles` (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=linestyles]
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

- `linestyles*` contains more dotted line styles than `linestyles` (from top to bottom)




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    stack plots=y,stack dir=minus,
    cycle list name=linestyles*]
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\addplot coordinates {(0,1) (0.5,1) (1,1)};
\end{axis}
\end{tikzpicture}
```

- `auto` The `cycle list name=auto` always denotes the most recently used cycle list activated by `cycle list` or `cycle list name`.

The definitions of all predefined cycle lists follow (see the end of this paragraph for a syntax description).

```
\pgfplotscreateplotcyclelist{color}{%
    blue,every mark/.append style={fill=blue!80!black},mark=*\\%
    red,every mark/.append style={fill=red!80!black},mark=square*\\%
    brown!60!black,every mark/.append style={fill=brown!80!black},mark=otimes*\\%
    black,mark=star\\%
    blue,every mark/.append style={fill=blue!80!black},mark=diamond*\\%
    red,densely dashed,every mark/.append style={solid,fill=red!80!black},mark=*\\%
    brown!60!black,densely dashed,every mark/.append style={
        solid,fill=brown!80!black},mark=square*\\%
    black,densely dashed,every mark/.append style={solid,fill=gray},mark=otimes*\\%
    blue,densely dashed,mark=star,every mark/.append style=solid\\%
    red,densely dashed,every mark/.append style={solid,fill=red!80!black},mark=diamond*\\%
}
```

```
\pgfplotscreateplotcyclelist{black white}{%
    every mark/.append style={fill=gray},mark=*\\%
    every mark/.append style={fill=gray},mark=square*\\%
    every mark/.append style={fill=gray},mark=otimes*\\%
    mark=star\\%
    every mark/.append style={fill=gray},mark=diamond*\\%
    densely dashed,every mark/.append style={solid,fill=gray},mark=*\\%
    densely dashed,every mark/.append style={solid,fill=gray},mark=square*\\%
    densely dashed,every mark/.append style={solid,fill=gray},mark=otimes*\\%
    densely dashed,every mark/.append style={solid},mark=star\\%
    densely dashed,every mark/.append style={solid,fill=gray},mark=diamond*\\%
}
```

```
\pgfplotscreateplotcyclelist{exotic}{%
    teal,every mark/.append style={fill=teal!80!black},mark=*\\%
    orange,every mark/.append style={fill=orange!80!black},mark=square*\\%
    cyan!60!black,every mark/.append style={fill=cyan!80!black},mark=otimes*\\%
    red!70!white,mark=star\\%
    lime!80!black,every mark/.append style={fill=lime},mark=diamond*\\%
    red,densely dashed,every mark/.append style={solid,fill=red!80!black},mark=*\\%
    yellow!60!black,densely dashed,
        every mark/.append style={solid,fill=yellow!80!black},mark=square*\\%
    black,every mark/.append style={solid,fill=gray},mark=otimes*\\%
    blue,densely dashed,mark=star,every mark/.append style=solid\\%
    red,densely dashed,every mark/.append style={solid,fill=red!80!black},mark=diamond*\\%
}
```

```
%  note that "." is the currently defined Tikz color.
\pgfplotscreateplotcyclelist{mark list}{%
    every mark/.append style={solid,fill=.!80!black},mark=*\\%
    every mark/.append style={solid,fill=.!80!black},mark=square*\\%
    every mark/.append style={solid,fill=.!80!black},mark=triangle*\\%
    every mark/.append style={solid},mark=star\\%
    every mark/.append style={solid,fill=.!80!black},mark=diamond*\\%
    every mark/.append style={solid,fill=.!80!black!40},mark=otimes*\\%
    every mark/.append style={solid},mark=|\\%
    every mark/.append style={solid,fill=.!80!black},mark=pentagon*\\%
    every mark/.append style={solid},mark=text,text mark=p\\%
    every mark/.append style={solid},mark=text,text mark=a\\%
}
```

This is not the complete truth: the actual implementation of `mark list` allows to customize the `fill` value:

`/pgfplots/mark list fill={`⟨*color*⟩`}` (initially `.!80!black`)

Allows to customize the fill color for the `mark list` and `mark list*`.

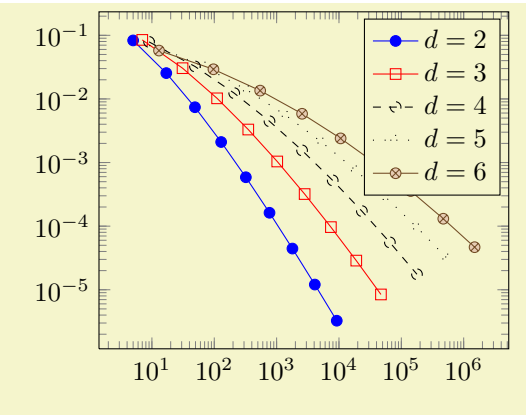
For example, if you have `black` as color, the alternative choice `mark list fill=.!50!white` will produce much better results.

```
%  note that "." is the currently defined Tikz color.
\pgfplotscreateplotcyclelist{mark list*}{%
    every mark/.append style={solid,fill=.!80!black},mark=*\\%
    every mark/.append style={solid,fill=.!80!black},mark=square*\\%
    every mark/.append style={solid,fill=.!80!black},mark=triangle*\\%
    every mark/.append style={solid,fill=.!80!black},mark=halfsquare*\\%
    every mark/.append style={solid,fill=.!80!black},mark=pentagon*\\%
    every mark/.append style={solid,fill=.!80!black},mark=halfcircle*\\%
    every mark/.append style={solid,fill=.!80!black,rotate=180},mark=halfdiamond*\\%
    every mark/.append style={solid,fill=.!80!black!40},mark=otimes*\\%
    every mark/.append style={solid,fill=.!80!black},mark=diamond*\\%
    every mark/.append style={solid,fill=.!80!black},mark=halfsquare right*\\%
    every mark/.append style={solid,fill=.!80!black},mark=halfsquare left*\\%
}
```

```
\pgfplotscreateplotcyclelist{color list}{%
    red,blue,black,yellow,brown,teal,orange,violet,cyan,green!70!black,magenta,gray}
```

```
\pgfplotscreateplotcyclelist{linestyles}{solid,dashed,dotted}
\pgfplotscreateplotcyclelist{linestyles*}{solid,dashed,dotted,dashdotdotted,dashdotdotted}
```

2. The second choice for cycle lists is to provide each entry directly as argument to `cycle list`,




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[cycle list={%
    {blue,mark=*},
    {red,mark=square},
    {dashed,mark=o},
    {loosely dotted,mark=+},
    {brown!60!black,
        mark options={fill=brown!40},
        mark=otimes*}}
]
\plotcoords
\legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
\end{loglogaxis}
\end{tikzpicture}
```

(This example list requires `\usetikzlibrary{plotmarks}`).

The input format is described below in more detail.

3. The last method is to combine 1. and 2.: Define named cycle lists in the preamble and use them with '`cycle list name`':

`\pgfplotscreateplotcyclelist{`⟨*name*⟩`}{`⟨*list*⟩`}`

```
\pgfplotscreateplotcyclelist{mylist}{%
    {blue,mark=*},
    {red,mark=square},
    {dashed,mark=o},
    {loosely dotted,mark=+},
    {brown!60!black,mark options={fill=brown!40},mark=otimes*}}
...
\begin{axis}[cycle list name=mylist]
    ...
\end{axis}
```

The format of ⟨*list*⟩: The argument ⟨*list*⟩ is usually a comma separated list of lists of style keys like colors, line styles, marker types and marker styles. This "comma list of comma lists" structure requires to encapsulate the inner list using curly braces:

```
\pgfplotscreateplotcyclelist{mylist}{%
    {blue,mark=*},
    {red,mark=square},
    {dashed,mark=o},
    {loosely dotted,mark=+},
    {brown!60!black,mark options={fill=brown!40},mark=otimes*}}
```

Alternatively, one can terminate the inner lists (i.e. those for one single plot) with '\\':

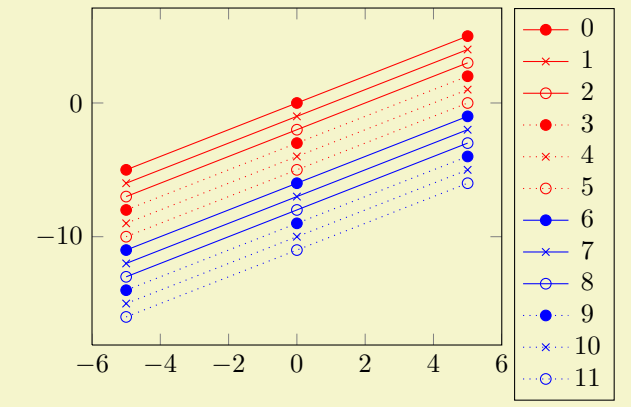
```
\begin{axis}[cycle list={%
    blue,mark=*\\%
    red,mark=square\\%
    dashed,mark=o\\%
    loosely dotted,mark=+\\%
    brown!60!black,mark options={fill=brown!40},mark=otimes*\\%
}
]
...
\end{axis}
```

In this case, the *last* entry also needs a terminating '\\', but one can omit braces around the single entries.

Remark: It is possible to call `\pgfplotsset{cycle list={`⟨*a list*⟩`}}` or `cycle list name` *between* plots. Such a setting remains effective until the end of the current TEX group (that means curly braces). Every `\addplot` command queries the `cycle list` using the plot index; it doesn't hurt if `cycle list`s have changed in the meantime.

`/pgfplots/cycle multi list=`⟨*list 1*⟩`\nextlist`⟨*list 2*⟩`\nextlist`⋯

Allows to supply more than one `cycle list` in a way such that each one contributes to the plot style. This is probably best explained using an example:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    cycle multi list={
      red,blue\nextlist
      solid,{dotted,mark options={solid}}\nextlist
      mark=*,mark=x,mark=o
    },
    samples=3,
    legend entries={0,...,20},
    legend pos=outer north east
]
    \addplot {x};
    \addplot {x-1};
    \addplot {x-2};
    \addplot {x-3};
    \addplot {x-4};
    \addplot {x-5};
    \addplot {x-6};
    \addplot {x-7};
    \addplot {x-8};
    \addplot {x-9};
    \addplot {x-10};
    \addplot {x-11};
\end{axis}
\end{tikzpicture}
```

The provided `cycle multi list` consists of three lists. The style for a single plot is made up using elements of each of the three lists: the first plot has style `red,solid,mark=*`, the second has

`red,solid,mark=x`, the third has `red,solid,mark=o`. The fourth plot restarts the third list and uses the next one of list 2: it has `red,dotted,mark options={solid},mark=*` and so on.

The last list will always be advanced for a new plot. The list before the last (in our case the second list) will be advanced after the last one has been reset. In other words: `cycle multi list` allows a composition of different `cycle list` in a lexicographical way[33].

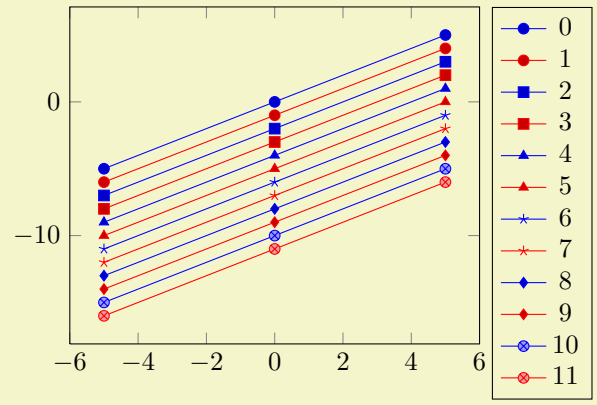
The argument for `cycle multi list` is a sequence of arguments as they would have been provided for `cycle list`, separated by `\nextlist`. In addition to providing a new cycle list, the ⟨*list i*⟩ elements can also denote `cycle list name` values (including the special `auto` cycle list which is the most recently assigned `cycle list` or `cycle list name`). The final `\nextlist` is optional.

The list in our example above could have been written as

```
\begin{axis}[
    cycle multi list={
        red\\blue\\\nextlist
        solid\\dotted,mark options={solid}\\\nextlist
        mark=*\\mark=x\\mark=o\\
    }]
```

as well (note the terminating `\\` commands!).

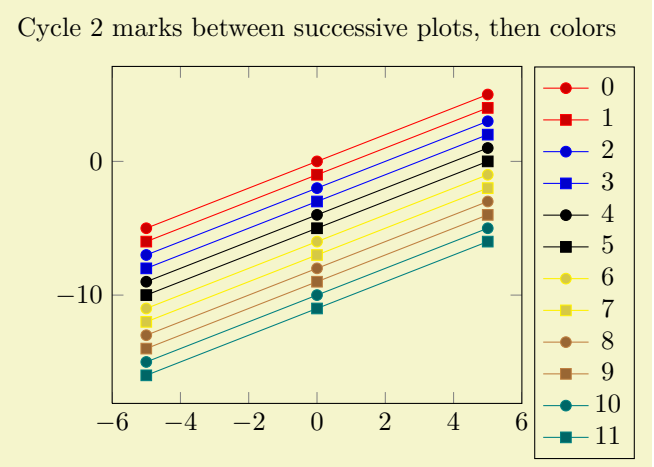
Cycle color between successive plots, then marks




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Cycle color between successive plots, then marks},
    cycle multi list={
        mark list\nextlist
        blue,red%
    },
    samples=3,
    legend entries={0,...,20},
    legend pos=outer north east
]
    \addplot {x};
    \addplot {x-1};
    \addplot {x-2};
    \addplot {x-3};
    \addplot {x-4};
    \addplot {x-5};
    \addplot {x-6};
    \addplot {x-7};
    \addplot {x-8};
    \addplot {x-9};
    \addplot {x-10};
    \addplot {x-11};
\end{axis}
\end{tikzpicture}
```

[33] For those who prefer formulas: The plot with index $0 \le i < N$ will use cycle list offsets $i_0, i_1, \ldots, i_k$, $0 \le i_m < N_m$ where k is the number of arguments provided to `cycle multi list` and N_m is the number of elements in the mth cycle list. The offsets i_m are computed in a loop `{ int tmp=i; for( int m=k-1; m>=0; m=m-1 ) { i_m = tmp%N_m; tmp = tmp/N_m; }}`.

Using Sub–Lists The ⟨*list i*⟩ entry can also contain just the first n elements of an already known cycle list name using the syntax `[`⟨*number*⟩ `of]`⟨*cycle list name*⟩. For example `[2 of]mark list` will use the first 2 elements of `mark list`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Cycle 2 marks between successive plots, then colors},
    cycle multi list={%
        color list\nextlist
        [2 of]mark list
    },
    samples=3,
    legend entries={0,...,20},
    legend pos=outer north east
]
    \addplot {x};
    \addplot {x-1};
    \addplot {x-2};
    \addplot {x-3};
    \addplot {x-4};
    \addplot {x-5};
    \addplot {x-6};
    \addplot {x-7};
    \addplot {x-8};
    \addplot {x-9};
    \addplot {x-10};
    \addplot {x-11};
\end{axis}
\end{tikzpicture}
```

`/pgfplots/cycle list shift=`{⟨*integer*⟩} (initially `empty`)

Allows to *shift* the index into the `cycle list`. If ⟨*integer*⟩ is n, the list element $i+n$ will be taken instead of the ith one. Remember that i is the index of the current `\addplot` command (starting with 0).

Since a `cycle list` is queried *immediately* when `\addplot` (or `\addplot+`) is called, you can adjust the `cycle list shift` for selected plots:

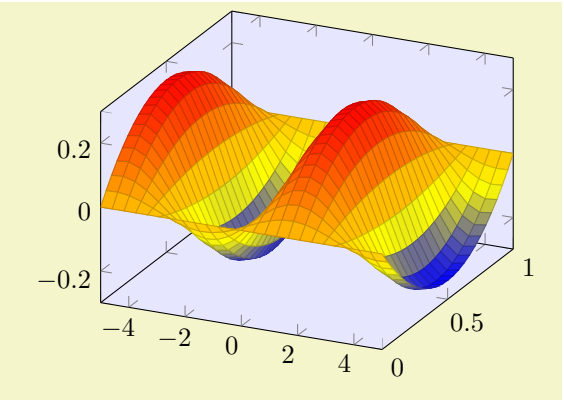
```
\pgfplotsset{cycle list shift=3}
\addplot ....

\pgfplotsset{cycle list shift=-1}
\addplot ....
```

Special case: If the result is negative, $i+n<0$, the list index $-(i+n)$ will be taken. For example, `cycle list shift=-10` and $i<10$ will result in list index $10-i$. Note that you can use `reverse legend` to reverse legends, so this feature is probably never needed.

4.6.8 Axis Background

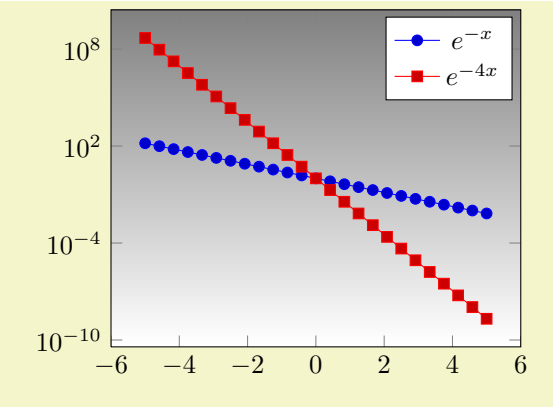
/pgfplots/axis background (initially empty)

This is a style to configure the appearance of the axis as such. It can be defined and/or changed using the axis background/.style={⟨*options*⟩} method. A background path will be generated with ⟨*options*⟩, which may contain fill colors or shadings.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        axis background/.style={fill=blue!10}]

    \addplot3[surf,y domain=0:1]
        {sin(deg(x)) * y*(1-y)};

    \end{axis}
\end{tikzpicture}
```

Please note that legends are filled with white in the default configuration.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{semilogyaxis}[
        axis background/.style={
            shade,top color=gray,bottom color=white},
        legend style={fill=white}]

    \addplot {exp(-x)};
    \addplot {exp(-4*x)};
    \legend{$e^{-x}$,$e^{-4x}$}
    \end{semilogyaxis}
\end{tikzpicture}
```

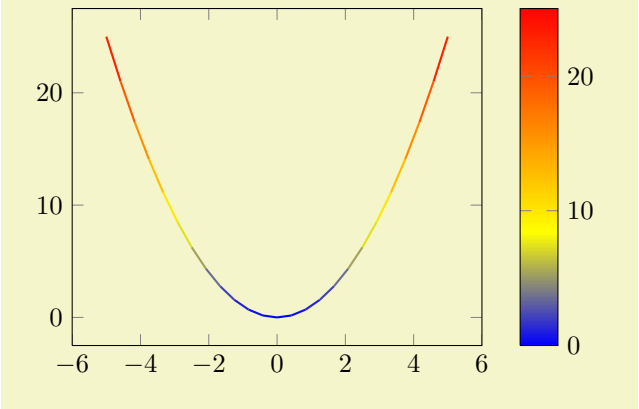
Details about fill and shade can be found in the Ti*k*Z manual, [5].

4.7 Providing Color Data - Point Meta

PGFPLOTS provides features which modify plots depending on a special coordinate, the "point meta data". For example, scatter plots may vary marker colors, size or appearance depending on this special data. Surface and mesh plots are another example: here, the color of a surface patch (or mesh part) depends on "point meta".

The common idea idea is to tell PGFPLOTS how to get this data. It is not necessary to provide data explicitly – in many cases, the data which is used to color surface patches or marker colors is the plot's y or z coordinate. The method used to tell PGFPLOTS where to find "point meta data" is the point meta key.

A further common idea is the use of color maps: if the point meta data is in the interval $[m_{\min}, m_{\max}]$, the point meta coordinate $m = m_{\min}$ will get the lowest color provided by the color map while $m = m_{\max}$ will get the highest color provided by the color map. Any coordinate between this values will be mapped linearly: for example, the mean $m = 1/2(m_{\max} + m_{\min})$ will get the middle color of the color map. This is why "point meta" is sometimes called "color data".




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar]
        \addplot[mesh,point meta=y,thick] {x^2};
    \end{axis}
\end{tikzpicture}
```

/pgfplots/point meta=none|⟨*expression*⟩|x|y|z|f(x)|explicit|explicit symbolic (initially none)

The point meta key tells PGFPLOTS where to get the special point meta data. Please note that point meta and scatter src is actually the same – scatter src is an alias for point meta. Thus, the summary provided for scatter src on page 75 covers the same topics. However, the main reference for point meta is here.

none The initial choice none disables point meta data, resulting in no computational work. Any other choice will activate the computation of upper and lower ranges for point meta data, i.e. the computation of $[m_{\min}, m_{\max}]$.

x The choice x uses the already available x coordinates as point meta data. This does always refer to the *final* x coordinates after any user transformations, logarithms, stacked plot computations etc. have been applied. Consider using rawx if you need the unprocessed coordinate value here.

y

z The choices y and z are similar: they use the y or z coordinates respectively as point meta data. Consequently, these three choices do *not* need any extra data. As for x, there are math constants rawy and rawz which yield the unprocessed y and z value, respectively.

f(x) This will use the last available coordinate, in other words: it is the same as y for two dimensional plots and z for three dimensional ones.

explicit This choice tells PGFPLOTS to expect *numerical* point meta data which is provided explicitly in the coordinate input streams. This data will be transformed linearly into the current color map as it has been motivated above.

How point meta data is provided for plot coordinates, plot table and the other input methods is described in all detail in Section 4.2.1 – but we provide small examples here to summarize the possibilities:

```
%  for 'coordinates':
%  provide color data explicitly using [<data>]
%  behind coordinates:
\addplot+[point meta=explicit]
    coordinates {
        (0,0) [1.0e10]
        (1,2) [1.1e10]
        (2,3) [1.2e10]
        (3,4) [1.3e10]
        %  ...
    };
```

```
%  for 'table':
%  Assumes a datafile.dat like
%  xcolname  ycolname    colordata
%  0         0           0.001
%  1         2           0.3
%  2         2.1         0.4
%  3         3           0.5
%  ...
%  the file may have more columns.
\addplot+[point meta=explicit]
    table[x=xcolname,y=ycolname,meta=colordata]
        {datafile.dat};
%  or, equivalently (perhaps a little bit slower):
\addplot+[point meta=\thisrow{colordata}]
    table[x=xcolname,y=ycolname]
        {datafile.dat};
```

```
%  for 'file':
%  Assumes a datafile.dat like
%  0         0           0.001
%  1         2           0.3
%  2         2.1         0.4
%  3         3           0.5
%  ...
%  the first three columns will be used here as x,y and meta,
%  resp.
\addplot+[point meta=explicit]
    file {datafile.dat};
```

```
%  'table' using expressions which may depend on all
%  columns:
%  Assumes a datafile.dat like
%  xcolname  ycolname    anything    othercol
%  0         0           4           15
%  1         2           5           20
%  2         2.1         8           30
%  3         3           42          40
%  ...
%  the file may have more columns.
\addplot+[point meta={0.5*(\thisrow{anything} + sqrt(\thisrow{othercol}))}]
    table[x=xcolname,y=ycolname]
        {datafile.dat};
```

Thus, there are several methods to provide point meta (color data). The key for the choice `explicit` is that some data is provided explicitly – although `point meta` doesn't know how. The data is expected to be of numerical type and is mapped linearly into the range $[0, 1000]$ (maybe for use in the current color map).

`explicit symbolic` The choice `explicit symbolic` is very similar to `explicit` in that it expects extra data by the coordinate input routines. However, `explicit symbolic` does not necessarily expect numerical data: you can provide any sort of symbols. One might provide a set of styles, one for each class in a scatter plot. This is realised using `scatter/classes`, see page 77. Input data is provided in the same fashion as mentioned above for the choice `explicit`.

Currently, this choice can only be used for scatter plots.

⟨*expression*⟩ This choice allows to compute point meta data using a mathematical expression. The ⟨*expression*⟩ may depend on `x`, `y`, `z` which yield the current x, y or z coordinate, respectively. The coordinates are completely processed (transformations, logs) as mentioned above for the choice `x`. Furthermore, the ⟨*expression*⟩ may depend on commands which are valid during `\addplot` like `\plotnum` or `\coordindex` (see Section 4.24 for details). Computations are performed using the floating point unit of PGF, and all supported arithmetical operations can be used.

In essence, the ⟨*expression*⟩ may depend on everything which is known to all `\addplot` commands: the x, y and (if any) z coordinates. In addition, it may depend upon `rawx`, `rawy` or `rawz`. These three expressions yield the unprocessed x, y or z value as it has been found in the input stream (no logs, no user transformations)[34]. If used together with `plot table`, you may also access other table columns (for example with `\thisrow{`⟨*colname*⟩`}`).

[34] In rare circumstances, it might be interesting to apply a math expression to another source of point meta (one of the other

`TeX code`=⟨*code*⟩ A rather low level choice which allows to provide TEX ⟨*code*⟩ to compute a numerical value. The ⟨*code*⟩ should define the macro `\pgfplotspointmeta`. It is evaluated in a locally scoped environment (it's local variables are freed afterwards). It may depend on the same values as described for ⟨*expression*⟩ above, especially on `\thisrow{`⟨*colname*⟩`}` for table input.

Note that the math parser will be configured to use the `fpu` at this time, so `\pgfmathparse` yields floats.

`TeX code symbolic`=⟨*code*⟩ Just as `TeX code`, you can provide ⟨*code*⟩ which defines the macro `\pgfplotspointmeta`, but the result is not interpreted as a number. It is like the `explicit symbolic` choice.

As already mentioned, a main application of point meta data is to determine (marker/face/edge) colors using a linear map into the range $[0, 1000]$ (maybe for use in the current color map). This map works as follows: it is a function

$$\phi\colon [m_{\min}, m_{\max}] \to [0, 1000]$$

with

$$\phi(m) = \frac{m - m_{\min}}{1000}$$

such that $\phi(m_{\min}) = 0$ and $\phi(m_{\max}) = 1000$. The value 1000 is – per convention – the upper limit of all color maps. Now, if a coordinate (or edge/face) has the point meta data m, its color will be determined using $\phi(m)$: it is the color at $\phi(m)‰$ of the current color map.

This transformation depends on the interval $[m_{\min}, m_{\max}]$ which, in turn, can be modified using the keys `point meta rel`, `point meta min` and `point meta max` described below.

The untransformed point meta data is available in the macro `\pgfplotspointmeta` (only in the correct context, for example the scatter plot styles or the `scatter/@pre marker code` interface). This macro contains a low level floating point number (unless it is non-parsed string data). The transformed data will be available in the macro `\pgfplotspointmetatransformed` and is in fixed point representation. It is expected to be in the range $[0, 1000]$.

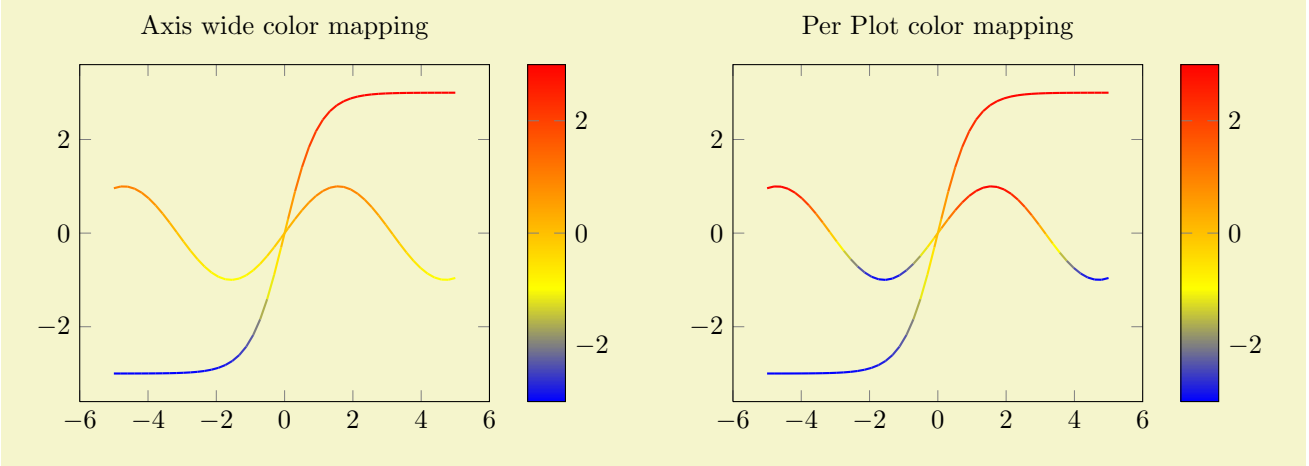
`/pgfplots/set point meta if empty`=`{`⟨*point meta source*⟩`}`

Sets `point meta`=⟨*point meta source*⟩, but only if `point meta=none` currently. This is used for `scatter`, `mesh` and `surf` with `set point meta if empty=f(x)`.

`/pgfplots/point meta rel=axis wide|per plot` (initially `axis wide`)

As already explained in the documentation for `point meta`, one application for point meta data is to determine colors using the current color map and a linear map from point meta data into the current color map. The question is how this linear map is computed.

The key `point meta rel` configures whether the interval of all point meta coordinates, $[m_{\min}, m_{\max}]$ is computed as maximum over all plots in the complete axis (the choice `axis wide`) or only for one particular plot (the choice `per plot`).



choices. To this end, the ⟨*expression*⟩ is checked after the other possible choices have already been evaluated. In other words, the statement `point meta=explicit, point meta=meta*meta+3` will evaluate the expression with `meta` set to whatever data has been provided explicitly.

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
   \begin{axis}[
      title=Axis wide color mapping,
      colorbar,
      samples=50,point meta rel=axis wide,
      point meta=y]

      \addplot[mesh,thick] {sin(deg(x))};
      \addplot[mesh,thick] {3*tanh(x)};
   \end{axis}
\end{tikzpicture}
~
\begin{tikzpicture}
   \begin{axis}[
      title=Per Plot color mapping,
      colorbar,
      samples=50,
      point meta rel=per plot,
      point meta=y]

      \addplot[mesh,thick] {sin(deg(x))};
      \addplot[mesh,thick] {3*tanh(x)};
   \end{axis}
\end{tikzpicture}
```

Note that a `colorbar` will still use the `axis wide` point meta limits. Consider the `colorbar source` key if you want the color data limits of a *particular* plot for your color bar. The `point meta rel` key configures how point meta maps to colors in the `colormap`.

/pgfplots/point meta min={⟨*number*⟩}
/pgfplots/point meta max={⟨*number*⟩}

These keys allow to define the range required for the linear map of point meta data into the range $[0, 1000]$ (for example, for current maps) explicitly. This is necessary if the same mapping shall be used for more than one axis.

Remarks about special cases:

- It is possible to provide limits partially; in this case, only the missing limit will be computed.
- If point meta data falls outside of these limits, the linear transformation is still well defined which is acceptable (unless the interval is of zero length). However, color data can't be outside of these limits, so color bars perform a truncation.
- This key can be provided for single plots as well as for the complete axis (or for both).
- If meta limits are provided for a single plot, these limits may also contribute to the axis wide meta interval.

/pgfplots/colormap access=map|direct (initially `map`)

This key configures how point meta data is used to determine colors from a color map. The initial configuration `map` performs the linear mapping operation explained above. The choice `direct` does not perform any transformation; it takes the point meta as integer indices into the current color map.

Consequently, there is no interpolation between colors in the color map, there will only be as many colors as the color map contains explicitly.

Some more details:

- If there are m colors in the color map and the color data falls outside of $[0, m-1]$, it will be pruned to either the first or the last color.
- If color data is a real number, it will be truncated to the next smaller integer.
- This key does not work for `shader=interp` (note that this shader will always interpolate in the color map).

Attention: This feature is experimental, I did not have time to test it.

4.8 Axis Descriptions

Axis descriptions are labels for x and y axis, titles, legends and the like. Axis descriptions are drawn after the plot is finished and they are not subjected to clipping.

4.8.1 Placement of Axis Descriptions

This section describes how to *modify* the placement of titles, labels, legends and other axis descriptions. It may be skipped at first reading.

There are different methods to place axis descriptions. One of them is to provide coordinates relative to the axis' rectangle such that `(0,0)` is the lower left corner and `(1,1)` is the upper right corner – this is very useful for figure titles or legends. Coordinates of this type, i.e. without unit like `(0,0)` or `(1.03,1)`, are called `axis description cs` (the `cs` stands for "coordinate system"). One other method is of primary interest for axis labels – they should be placed near the tick labels, but it a way that they don't overlap or obscure tick labels. Furthermore, axis labels shall be placed such that they are automatically moved if the axis is rotated (or tick labels are moved to the right side of the figure). There is a special coordinate system to realize these two demands, the `ticklabel cs`.

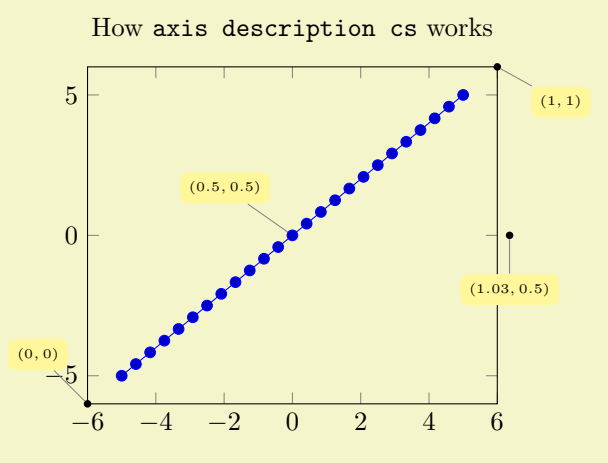
In the following, the two coordinate systems `axis description cs` and `ticklabel cs` are described in more detail. It should be noted that `axis description cs` is used automatically, so it might never be necessary to use it explicitly.

Coordinate system `axis description cs`

A coordinate system which is used to place axis descriptions. Whenever the option '`at={(`$\langle x \rangle$`,`$\langle y \rangle$`)}`' occurs in `label style`, `legend style` or any other axis description, `(`$\langle x \rangle$`,`$\langle y \rangle$`)` is interpreted to be a coordinate in `axis description cs`.

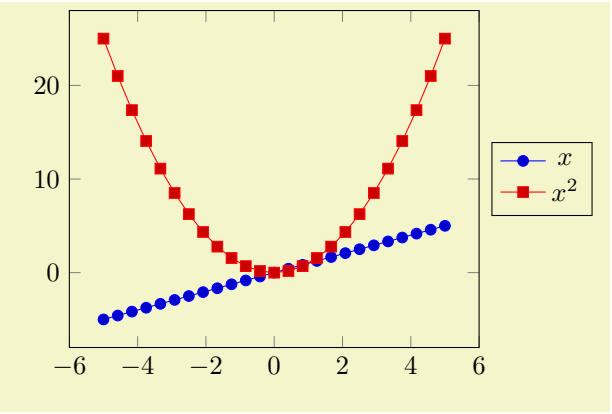
The point $(0, 0)$ is always the lower left corner of the tightest bounding box around the axes (without any descriptions or ticks) while the point $(1, 1)$ is the upper right corner of this bounding box.

In most cases, it is *not* necessary to explicitly write `axis description cs` as it is the default coordinate system for any axis description. An example for how coordinates are placed is shown below.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  [See the TikZ manual if you'd like to learn about nodes and pins]
\begin{tikzpicture}
    \tikzset{
        every pin/.style={fill=yellow!50!white,rectangle,rounded corners=3pt,font=\tiny},
        small dot/.style={fill=black,circle,scale=0.3}
    }
    \begin{axis}[
        clip=false,
        title=How \texttt{axis description cs} works
    ]
    \addplot {x};

    \node[small dot,pin=120:{$(0,0)$}]      at (axis description cs:0,0) {};
    \node[small dot,pin=-30:{$(1,1)$}]      at (axis description cs:1,1) {};
    \node[small dot,pin=-90:{$(1.03,0.5)$}] at (axis description cs:1.03,0.5) {};
    \node[small dot,pin=125:{$(0.5,0.5)$}]  at (axis description cs:0.5,0.5) {};
    \end{axis}
\end{tikzpicture}
```

Axis descriptions are Ti*k*Z nodes, that means all placement and detail options of [5] apply. The point on the node's boundary which is actually shifted to the `at` coordinate needs to be provided with an anchor (cf [5, Nodes and Edges]):



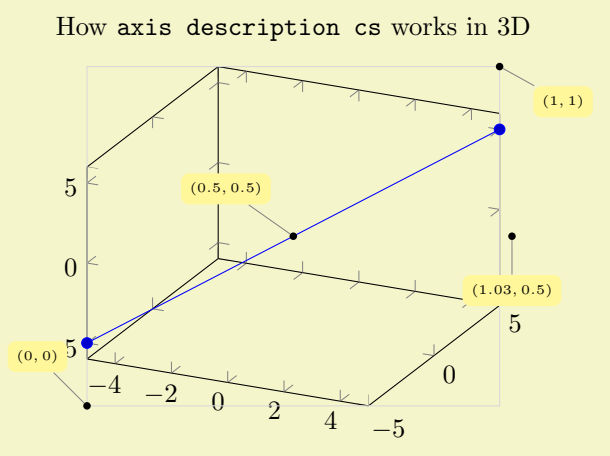

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        legend entries={$x$,$x^2$},
        legend style={
            at={(1.03,0.5)},
            anchor=west
        }
    ]
    \addplot {x};
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

Standard anchors of nodes are `north`, `east`, `south`, `west` and mixed components like `north east`. Please refer to [5] for a complete documentation of anchors.

Remarks:

- Each of the anchors described in Section 4.18 can be described by `axis description cs` as well.
- The `axis description cs` is independent of axis reversals or skewed axes. Only for the default configuration of boxed axes is it the same as `rel axis cs`, i.e. `(0,0)` is the same as the smallest axis coordinate and `(1,1)` is the largest one in case of standard boxed axes[35].
- Even for three dimensional axes, the `axis description cs` is still two-dimensional: it always refers to coordinates relative to the tightest bounding box around the axis (without any descriptions or ticks).

[35] This was different in versions before 1.3: earlier versions did not have the distinction between `axis description cs` and `rel axis cs`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  the same as above for 3D ...
%  [See the TikZ manual if you'd like to learn about nodes and pins]
\begin{tikzpicture}
    \tikzset{
        every pin/.style={fill=yellow!50!white,rectangle,rounded corners=3pt,font=\tiny},
        small dot/.style={fill=black,circle,scale=0.3}
    }
    \begin{axis}[
        clip=false,
        title=How \texttt{axis description cs} works in 3D
    ]
    \addplot3 coordinates {(-5,-5,-5) (5,5,5)};

    \draw[black!15] (axis description cs:0,0) rectangle (axis description cs:1,1);

    \node[small dot,pin=120:{$(0,0)$}]      at (axis description cs:0,0) {};
    \node[small dot,pin=-30:{$(1,1)$}]      at (axis description cs:1,1) {};
    \node[small dot,pin=-90:{$(1.03,0.5)$}] at (axis description cs:1.03,0.5) {};
    \node[small dot,pin=125:{$(0.5,0.5)$}]  at (axis description cs:0.5,0.5) {};
    \end{axis}
\end{tikzpicture}
```

- Since the view does not influence these positions, `axis description cs` might not be a good choice for axis labels in 3D. The `ticklabel cs` is used in this case.

Coordinate system `xticklabel cs`
Coordinate system `yticklabel cs`
Coordinate system `zticklabel cs`
Coordinate system `ticklabel cs`

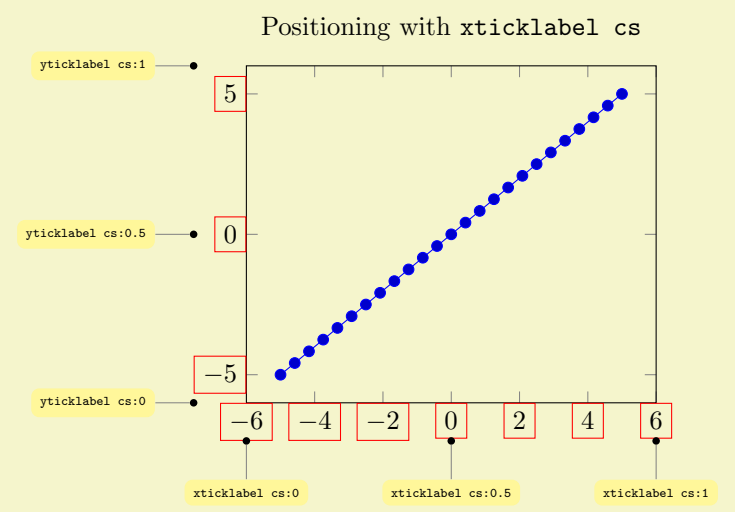
A set of special coordinate systems intended to place axis descriptions (or any other drawing operation) besides tick labels, in a way such that neither tick labels nor the axis as such are obscured.

See also `xlabel near ticks` as one main application of `ticklabel cs`.

The `xticklabel cs` (and its variants) always refer to one, uniquely identified axis: the one which is (or would be) annotated with tick labels.

The `ticklabel cs` (without explicit `x`, `y` or `z`) can only be used in contexts where the axis character is known from context (for example, inside of `xlabel style` – there, the `ticklabel cs` is equivalent to `xticklabel cs`).

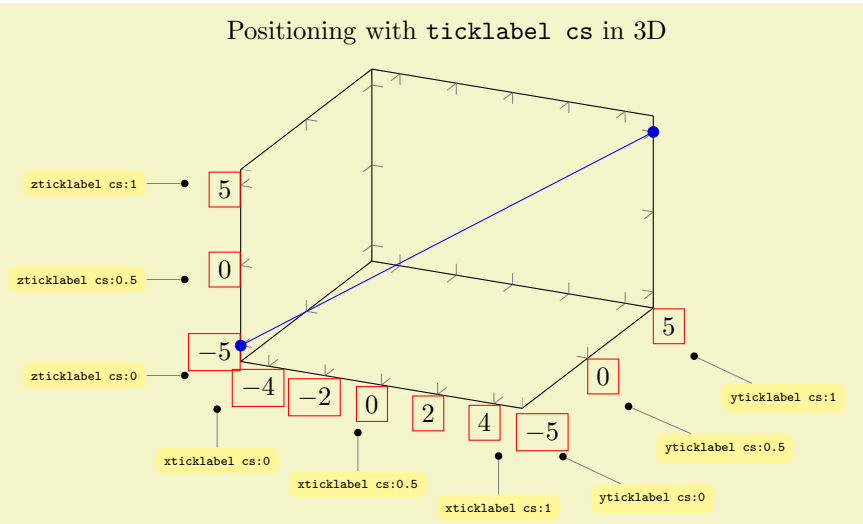
Each of these coordinate systems allows to specify points on a straight line which is placed parallel to an axis containing tick labels, moved away just far enough to avoid overlaps with the tick labels:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\tikzset{
    every pin/.style={fill=yellow!50!white,rectangle,rounded corners=3pt,font=\tiny},
    small dot/.style={fill=black,circle,scale=0.3}
}
\begin{tikzpicture}
\begin{axis}[
    clip=false,
    ticklabel style={draw=red},
    title=Positioning with \texttt{xticklabel cs}]
    \addplot {x};
    \node[small dot,pin=-90:{\texttt{xticklabel cs:0}}]     at (xticklabel cs:0) {};
    \node[small dot,pin=-90:{\texttt{xticklabel cs:0.5}}]   at (xticklabel cs:0.5) {};
    \node[small dot,pin=-90:{\texttt{xticklabel cs:1}}]     at (xticklabel cs:1) {};


    \node[small dot,pin=180:{\texttt{yticklabel cs:0}}]     at (yticklabel cs:0) {};
    \node[small dot,pin=180:{\texttt{yticklabel cs:0.5}}]   at (yticklabel cs:0.5) {};
    \node[small dot,pin=180:{\texttt{yticklabel cs:1}}]     at (yticklabel cs:1) {};
\end{axis}
\end{tikzpicture}
```

The basic idea is to place coordinates on a straight line which is parallel to the axis containing tick labels – but shifted such that the line does not cut through tick labels.

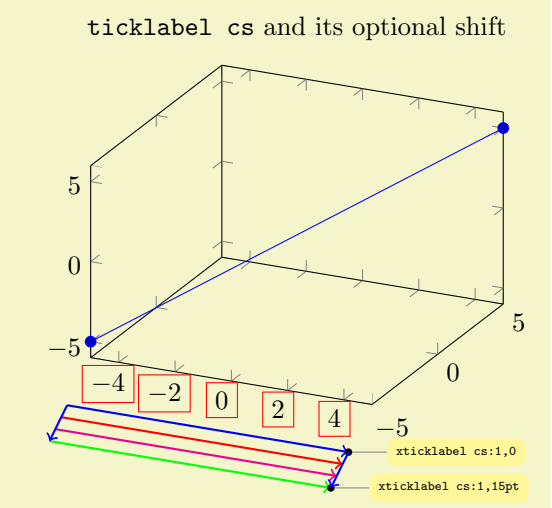
Of course, it is relatively simple to get the same coordinates as in the two dimensional example above with `axis description cs`, except that `ticklabel cs` always respects the tick label sizes appropriately. However, `ticklabel cs` becomes far superior when it comes to three dimensional positioning:



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  the same as above for 3D ...
\begin{tikzpicture}
    \tikzset{
        every pin/.style={fill=yellow!50!white,rectangle,rounded corners=3pt,font=\tiny},
        small dot/.style={fill=black,circle,scale=0.3}
    }
    \begin{axis}[
        ticklabel style={draw=red},
        clip=false,
        title=Positioning with \texttt{ticklabel cs} in 3D
    ]
    \addplot3 coordinates {(-5,-5,-5) (5,5,5)};

    \node[small dot,pin=-90:{\texttt{xticklabel cs:0}}]     at (xticklabel cs:0) {};
    \node[small dot,pin=-90:{\texttt{xticklabel cs:0.5}}]   at (xticklabel cs:0.5) {};
    \node[small dot,pin=-90:{\texttt{xticklabel cs:1}}]     at (xticklabel cs:1) {};

    \node[small dot,pin=-45:{\texttt{yticklabel cs:0}}]     at (yticklabel cs:0) {};
    \node[small dot,pin=-45:{\texttt{yticklabel cs:0.5}}]   at (yticklabel cs:0.5) {};
    \node[small dot,pin=-45:{\texttt{yticklabel cs:1}}]     at (yticklabel cs:1) {};

    \node[small dot,pin=180:{\texttt{zticklabel cs:0}}]     at (zticklabel cs:0) {};
    \node[small dot,pin=180:{\texttt{zticklabel cs:0.5}}]   at (zticklabel cs:0.5) {};
    \node[small dot,pin=180:{\texttt{zticklabel cs:1}}]     at (zticklabel cs:1) {};
    \end{axis}
\end{tikzpicture}
```

The coordinate `ticklabel cs:0` is associated with the lower axis limit while `ticklabel cs:1` is near the upper axis limit. The value `0.5` is in the middle of the axis, any other values (including negative values or values beyond 1) are linearly interpolated inbetween.

The `ticklabel cs` also accepts a second (optional) argument: a shift "away" from the tick labels. The shift points to a vector which is orthogonal to the associated axis, away from the tick labels. A shift of `0pt` is directly at the edge of the tick labels in direction of the normal vector, positive values move the position away and negative closer to the tick labels.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\tikzset{
    every pin/.style={fill=yellow!50!white,rectangle,rounded corners=3pt,font=\tiny},
    small dot/.style={fill=black,circle,scale=0.3}
}
\begin{tikzpicture}
    \begin{axis}[
        xticklabel style={draw=red},
        clip=false,
        title=\texttt{ticklabel cs} and its optional shift
    ]
    \addplot3 coordinates {(-5,-5,-5) (5,5,5)};

    \draw[blue,thick,->]      (xticklabel cs:0,0)     -- (xticklabel cs:1,0);
    \draw[red,thick,->]       (xticklabel cs:0,5pt)   -- (xticklabel cs:1,5pt);
    \draw[magenta,thick,->]   (xticklabel cs:0,10pt)  -- (xticklabel cs:1,10pt);
    \draw[green,thick,->]     (xticklabel cs:0,15pt)  -- (xticklabel cs:1,15pt);
    \node[small dot,pin=0:{\texttt{xticklabel cs:1,0}}]      at (xticklabel cs:1,0) {};
    \node[small dot,pin=0:{\texttt{xticklabel cs:1,15pt}}]   at (xticklabel cs:1,15pt) {};

    \draw[blue,thick,->]      (xticklabel cs:0,0)     -- (xticklabel cs:0,15pt);
    \draw[blue,thick,->]      (xticklabel cs:1,0)     -- (xticklabel cs:1,15pt);
    \end{axis}
\end{tikzpicture}
```

Whenever the `ticklabel cs` is used, the anchor should be set to `anchor=near ticklabel` (see below).

There is one specialty: if you reverse an axis (with `x dir=reverse`), points provided by `ticklabel cs` will be *unaffected* by the axis reversal. This is intented to provide consistent placement even for reversed axes. Use `allow reversal of rel axis cs=false` to disable this feature.

Besides the mentioned positioning methods, there is also the predefined node `current axis`. The anchors of `current axis` can also be used to place descriptions: At the time when axis descriptions are drawn, all anchors which refer to the axis origin (that means the "real" point $(0, 0)$) or any of the axis corners can be referenced using `current axis.`⟨*anchor name*⟩. Please see Section 4.18, Alignment, for further details.

4.8.2 Alignment of Axis Descriptions

This section describes how to modify the default alignment of axis descriptions. It can be skipped at first reading.

The two topics positioning and alignment always work together: *positioning* means to select an appropriate coordinate and *alignment* means to select an anchor inside of the description which will actually be moved to the desired position.

Ti*k*Z uses many anchors to provide alignment; most of them are named like `north`, `north east` etc. These names hold for any axis description as well (as axis descriptions are Ti*k*Z nodes). Readers can learn details about this topic in the Ti*k*Z manual [5] or some more advice in Section 4.18.

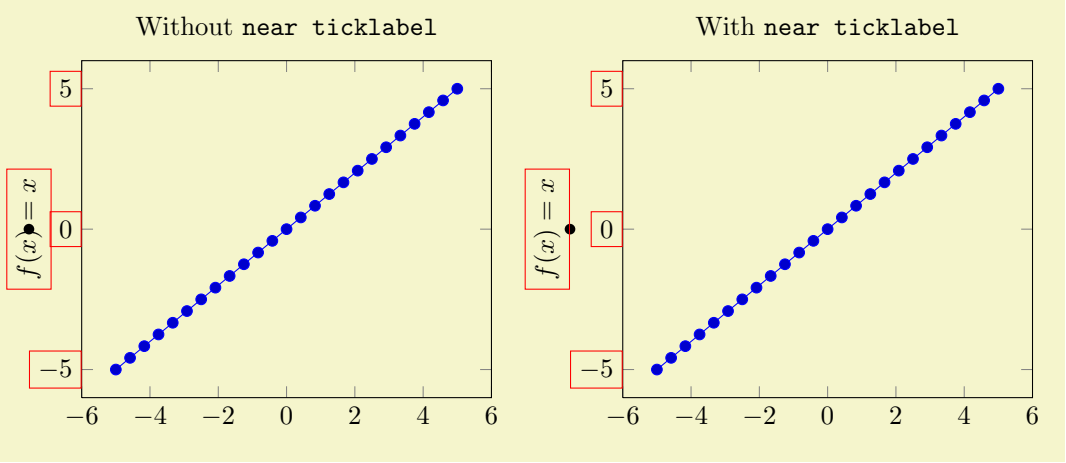
When it comes to axis descriptions, PGFPLOTS offers some specialized anchors and alignment methods which are described below.

Anchor `near xticklabel`
Anchor `near yticklabel`
Anchor `near zticklabel`
Anchor `near ticklabel`

These anchors can be used to align at the part of a node (for example, an axis description) which is *nearest* to the tick labels of a particular axis (or nearest to the position where tick labels would have been drawn if there were any).

These anchors are used for axis labels, especially for three dimensional axes. Furthermore, they are used for every tick label.

Maybe it is best to demonstrate it by example:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        title=Without \texttt{near ticklabel},
        ylabel={$f(x)=x$},
        every axis y label/.style=
            {at={(ticklabel cs:0.5)},rotate=90,anchor=center},
        clip=false,%  to display the \path below
        ylabel style={draw=red},
        yticklabel style={draw=red}
    ]

        \addplot {x};

        %  visualize the position:
        \fill (yticklabel cs:0.5) circle(2pt);
    \end{axis}
\end{tikzpicture}%
~
\begin{tikzpicture}
    \begin{axis}[
        title=With \texttt{near ticklabel},
        ylabel={$f(x)=x$},
        every axis y label/.style=
            {at={(ticklabel cs:0.5)},rotate=90,anchor=near ticklabel},
        clip=false,
        ylabel style={draw=red},
        yticklabel style={draw=red}
    ]

        \addplot {x};
        \fill (yticklabel cs:0.5) circle(2pt);
    \end{axis}
\end{tikzpicture}
```

The motivation is to place nodes such that they are anchored next to the tick label, regardless of the node's rotation or the position of ticks. The special anchor `near ticklabel` is only available for axis labels (as they have a uniquely identified axis, either x, y or z).

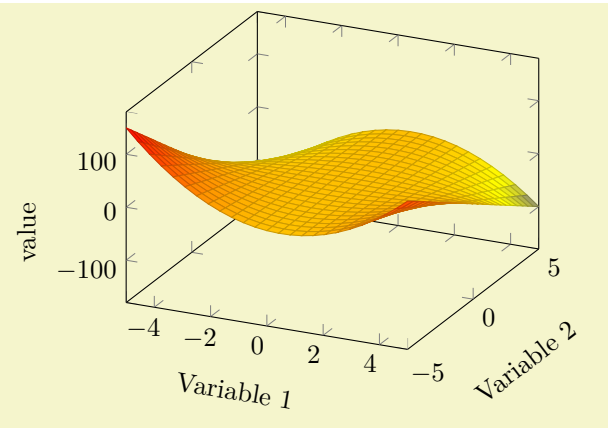
In more detail, the anchor is placed such that first, the node's center is on a line starting in the node's `at` position going in direction of the inwards normal vector of the axis line which contains the tick labels and second, the node does not intrude the axis. This normal vector is the same which is used for the shift argument in `ticklabel cs`: it is orthogonal to the tick label axis. Furthermore, `near ticklabel` inverts the transformation matrix before it computes this intersection point.

The `near ticklabel` anchor and its friends will be added temporarily to any shape used inside of an axis. This includes axis descriptions, but it is not limited to them: it applies to every Ti*k*Z `\node[anchor=near xticklabel] ...` setting.

Note that it is not necessary at all to *have* tick labels in an axis. The anchor will be placed such that it is near the axis on which tick labels *would* be drawn. In fact, every tick label uses `anchor=near ticklabel` as initial configuration.

/tikz/sloped like x axis (no value)
/tikz/sloped like y axis (no value)
/tikz/sloped like z axis (no value)
/tikz/sloped like x axis={⟨*options*⟩}
/tikz/sloped like y axis={⟨*options*⟩}
/tikz/sloped like z axis={⟨*options*⟩}

A key which replaces the rotational / scaling parts of the transformation matrix such that the node is sloped like the provided axis. For two dimensional plots, `sloped like y axis` is effectively the same as `rotate=90`. For a three dimensional axis, this will lead to a larger difference:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xlabel=Variable 1,
        ylabel=Variable 2,
        zlabel=value,
        xlabel style={sloped like x axis},
        ylabel style={sloped}
    ]

    \addplot3[surf] {y*x*(1-x)};
    \end{axis}
\end{tikzpicture}
```

Inside of axis labels, `sloped` is an alias for `sloped like` ⟨*char*⟩ `axis` with the correct ⟨*char*⟩ chosen automatically.

Please note that rotated text might not look very good (neither on screen nor printed).

It is possible to customize `sloped like x axis` by means of the following keys, which need to be provided as ⟨*options*⟩ (simply ignore the lengthy gray key prefixes):

/pgfplots/sloped/allow upside down=true|false (initially `false`)

Use `sloped like x axis=allow upside down` to enable upside down labels.

/pgfplots/sloped/execute for upside down=code (initially `empty`)

Use `sloped like x axis={execute for upside down=\tikzset{anchor=north}}` or something like that to handle upside down text nodes in a customized way (this is used by the `smithchart` library).

/pgfplots/sloped/reset nontranslations=true|false (initially `true`)

Use `sloped like x axis={reset nontranslations=false}` to *append* the transformations to the actual transformation matrix (instead of replacing it).

4.8.3 Labels

/pgfplots/xlabel={⟨*text*⟩}
/pgfplots/ylabel={⟨*text*⟩}
/pgfplots/zlabel={⟨*text*⟩}

These options set axis labels to ⟨*text*⟩ which is any TEX text.

To include special characters, you can use curly braces: "`xlabel={, = characters}`". This is necessary if characters like '`=`' or '`,`' need to be included literally.

Use `xlabel/.add={`⟨*prefix*⟩`}{`⟨*suffix*⟩`}` to modify an already assigned label.

Labels are Ti*k*Z-nodes which are placed with

```
%  for x:
\node
    [style=every axis label,
    style=every axis x label]

%  for y:
\node
    [style=every axis label,
    style=every axis y label]
```

so their position and appearance can be customized.

For example, a multiline `xlabel` can be configured using

```
\begin{axis}[xlabel style={align=right,text width=3cm},xlabel=A quite long label with a line break]
...
\end{axis}
```

See [5] to learn more about `align` and `text width`.

Upgrade notice: Since version 1.3, label placement *can* respect the size of adjacent tick labels. Use `\pgfplotsset{compat=1.3}` (or newer) in the preamble to activate this feature. See `xlabel near ticks` for details.

/pgfplots/xlabel shift={⟨*dimension*⟩} (initially `0pt`)
/pgfplots/ylabel shift={⟨*dimension*⟩} (initially `0pt`)
/pgfplots/zlabel shift={⟨*dimension*⟩} (initially `0pt`)
/pgfplots/label shift={⟨*dimension*⟩}

Shifts labels in direction of the outer normal vector of the axis by an amount of ⟨*dimension*⟩. The `label shift` sets all three label shifts to the same value.

Attention: This does only work if `\pgfplotsset{compat=1.3}` (or newer) has been called (more precisely: if `xlabel near ticks` is active for the respective axis).

/pgfplots/xlabel near ticks (no value)
/pgfplots/ylabel near ticks (no value)
/pgfplots/zlabel near ticks (no value)
/pgfplots/compat=1.3

These keys place axis labels (like `xlabel`) near the tick labels. If tick labels are small, labels will move closer to the axis. If tick labels are large, axis labels will move away from the axis. This is the default for every three dimensional plot, but it *won't* be used initially for two–dimensional plots for backwards compatibility. Take a look at the definition of `near ticklabel` on page 147 for an example.

The definition of these styles is

```
\pgfplotsset{
    /pgfplots/xlabel near ticks/.style={
        /pgfplots/every axis x label/.style={
            at={(ticklabel cs:0.5)},anchor=near ticklabel
        }
    },
    /pgfplots/ylabel near ticks/.style={
        /pgfplots/every axis y label/.style={
            at={(ticklabel cs:0.5)},rotate=90,anchor=near ticklabel
        }
    }
}
```

It is encouraged to write

```
\pgfplotsset{compat=1.3} %  or newer
```

in your preamble to install the styles document-wide – it leads to the best output (it avoids unnecessary space). It is not activated initially for backwards compatibility with older versions which used fixed distances from the tick labels.

/pgfplots/xlabel absolute (no value)
/pgfplots/ylabel absolute (no value)
/pgfplots/zlabel absolute (no value)
/pgfplots/compat=pre 1.3

Installs placement styles for axis labels such that `xlabel` yields a description of absolute, fixed distance to the axis. This is the initial configuration (for backwards compatibility with versions before 1.3). Use `compat=1.3` to get the most recent, more flexible configuration. Take a look at the definition of `near ticklabel` on page 147 for an example.

These styles are defined by

```
\pgfplotsset{
    /pgfplots/xlabel absolute/.style={%
        /pgfplots/every axis x label/.style={at={(0.5,0)},below,yshift=-15pt},%
        /pgfplots/every x tick scale label/.style={
            at={(1,0)},yshift=-2em,left,inner sep=0pt
        },
    },
    /pgfplots/ylabel absolute/.style={%
        /pgfplots/every axis y label/.style={at={(0,0.5)},xshift=-35pt,rotate=90},
        /pgfplots/every y tick scale label/.style={
            at={(0,1)},above right,inner sep=0pt,yshift=0.3em
        },
    }
}
```

There is no predefined absolute placement style for three dimensional axes.

Whenever possible, consider using `/.append style` instead of overwriting the default styles to ensure compatibility with future versions.

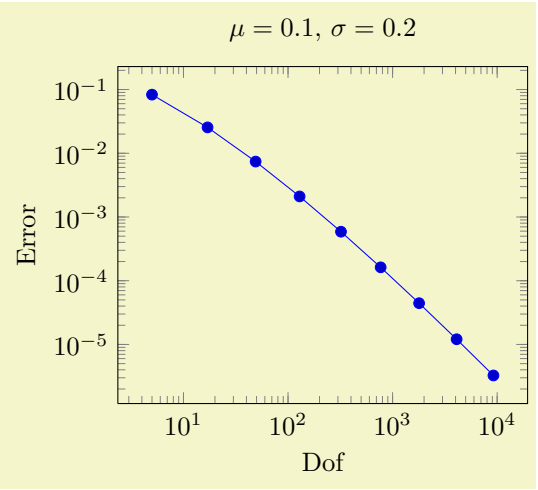
```
\pgfplotsset{every axis label/.append style={...}}
\pgfplotsset{every axis x label/.append style={...}}
\pgfplotsset{every axis y label/.append style={...}}
```

/pgfplots/title={⟨*text*⟩}

Adds a caption to the plot. This will place a Ti*k*Z-node with

```
\node[every axis title] {text};
```

to the current axis.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel=Dof,ylabel=Error,
    title={$\mu=0.1$, $\sigma=0.2$}]

    \addplot coordinates {
        (5,    8.312e-02)
        (17,   2.547e-02)
        (49,   7.407e-03)
        (129,  2.102e-03)
        (321,  5.874e-04)
        (769,  1.623e-04)
        (1793, 4.442e-05)
        (4097, 1.207e-05)
        (9217, 3.261e-06)
    };
\end{loglogaxis}
\end{tikzpicture}%
```

The title's appearance and/or placement can be reconfigured with

```
\pgfplotsset{title style={at={(0.75,1)}}}
%  or, equivalently,
\pgfplotsset{every axis title/.append style={at={(0.75,1)}}}
```

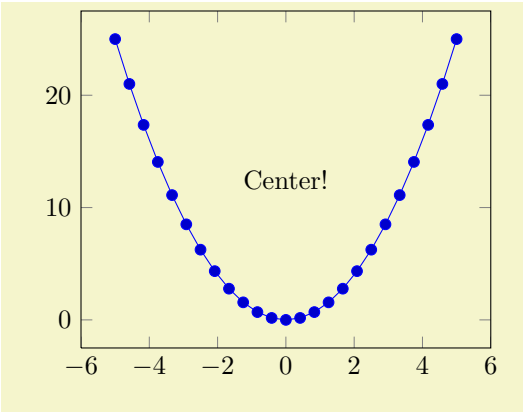
This will place the title at 75% of the x-axis. The coordinate $(0,0)$ is the lower left corner and $(1,1)$ the upper right one (see `axis description cs` for details).

Use `title/.add={`⟨*prefix*⟩`}{`⟨*suffix*⟩`}` to modify an already assigned title.

`/pgfplots/extra description/.code={`⟨`...`⟩`}`

Allows to insert ⟨*commands*⟩ after axis labels, titles and legends have been typeset.

As all other axis descriptions, the code can use $(0,0)$ to access the lower left corner and $(1,1)$ to access the upper right one. It won't be clipped.



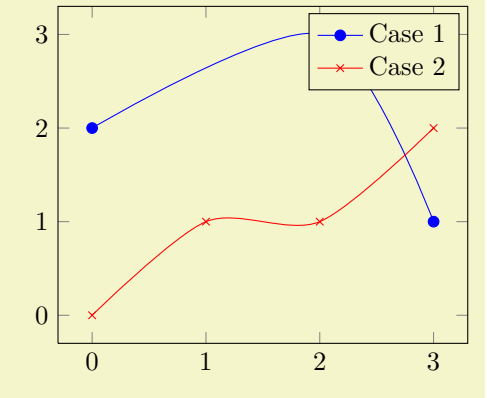

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={
    extra description/.code={
        \node at (0.5,0.5) {Center!};
    }}}
\begin{tikzpicture}
    \begin{axis}
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

4.8.4 Legends

Legends can be generated in two ways: the first is to use `\addlegendentry` or `\legend` inside of an axis. The other method is to use the key `legend entries`.

`\addlegendentry[`⟨*options*⟩`]{`⟨*name*⟩`}`

Adds a single legend entry to the legend list. This will also enable legend drawing.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot[smooth,mark=*,blue] coordinates {
    (0,2)
    (2,3)
    (3,1)
};
\addlegendentry{Case 1}

\addplot[smooth,color=red,mark=x]
    coordinates {
        (0,0)
        (1,1)
        (2,1)
        (3,2)
    };
\addlegendentry{Case 2}
\end{axis}
\end{tikzpicture}
```

It does not matter where `\addlegendentry` commands are placed, only the sequence matters. You will need one `\addlegendentry` for every `\addplot` command (unless you prefer an empty legend).

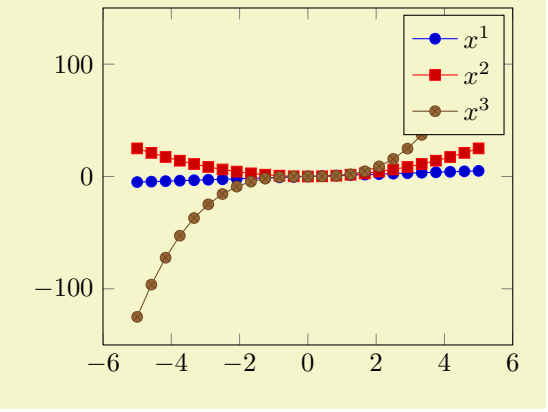
The optional ⟨*options*⟩ affect how the text is drawn; they apply only for this particular description text. For example, `\addlegendentry[red]{Text}` would yield a red legend text. Behind the scenes, the text is placed with `\node[`⟨*options*⟩`] {`⟨*name*⟩`};`, so ⟨*options*⟩ can be any TikZ option which affects nodes.

Using `\addlegendentry` disables the key `legend entries`.

\addlegendentryexpanded[⟨*options*⟩]{⟨*TEX text*⟩}

A variant of \addlegendentry which provides a method to deal with macros inside of ⟨*TEX text*⟩.

Suppose ⟨*TEX text*⟩ contains some sort of parameter which varies *for every plot*. Moreover, you like to use a loop to generate the plots. Then, it is simpler to use \addlegendentryexpanded:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \foreach \p in {1,2,3} {
      \addplot {x^\p};
      \addlegendentryexpanded{$x^\p$}
    }
\end{axis}
\end{tikzpicture}
```

Note that this example wouldn't have worked with \addlegendentry{$x^\p$} because the macro \p is no longer defined when PGFPLOTS attempts to draw the legend.

The invocation \addlegendentryexpanded{$x^\p$} is equivalent to calling \addlegendentry{x^2} if \p expands to 2.

The argument ⟨*TEX text*⟩ is expanded until nothing but un-expandable material remains (i.e. it uses the TEX primitive \edef). Occasionally, ⟨*TEX text*⟩ contains parts which should be expanded (like \p) and other parts which should be left unexpanded (for example \pgfmathprintnumber{\p}). Then, use

```
\noexpand\pgfmathprintnumber{\p}
```

or, equivalently

```
\protect\pgfmathprintnumber{\p}
```

to avoid expansion of the macro which follows the \protect immediately.

\legend{⟨*list*⟩}

You can use \legend{⟨*list*⟩} to assign a complete legend.

```
\legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
```

The argument of \legend is a list of entries, one for each plot.

Two different delimiters are supported:

1. There are comma–separated lists like

   ```
   \legend{$d=2$,$d=3$,$d=4$,$d=5$,$d=6$}
   ```

   These lists are processed using the PGF \foreach command and are quite powerful.

   The \foreach command supports a dots–notation to denote ranges like \legend{1,2,...,5} or even \legend{x^1,$x^...$,x^d}.

   Attention with periods: to avoid confusion with the dots ... notation, you may need to encapsulate a legend entry containing periods by curly braces: \legend{{ML spcm.},{CW spcm.},{ML AC}} (or use the \\ delimiter, see below).

2. It is also possible to delimit the list by '\\'. In this case, the *last element must be terminated* by \\ as well:

   ```
   \legend{$a=1, b=2$\\,$a=2, b=3$\\$a=3, b=5$\\}
   ```

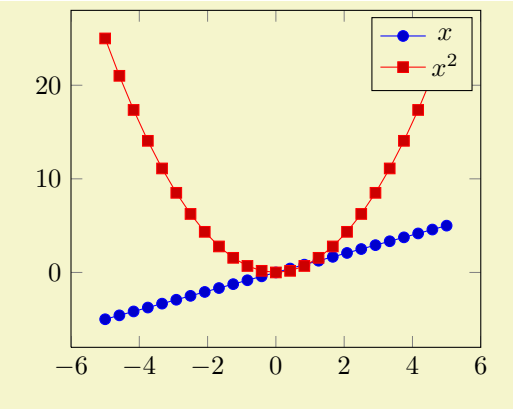
   This syntax simplifies the use of ',' inside of legend entries, but it does not support the dots–notation.

The short marker/line combination shown in legends is acquired from the ⟨*style options*⟩ argument of `\addplot`.

Using `\legend` overwrites any other existing legend entries.

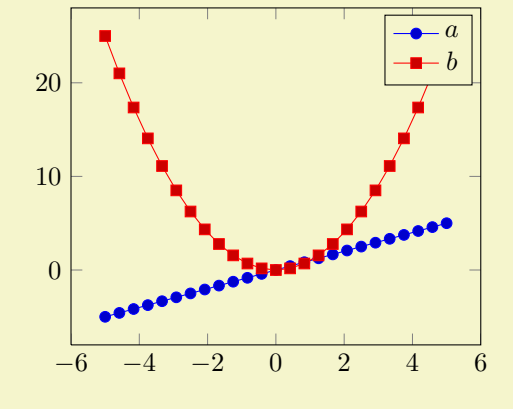
`/pgfplots/legend entries={`⟨*comma separated list*⟩`}`

This key can be used to assign legend entries just like the commands `\addlegendentry` and `\legend`. Again, the positioning is relative to the axis rectangle (unless units like `cm` or `pt` are specified explicitly).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[legend entries={$x$,$x^2$}]
    \addplot {x};
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

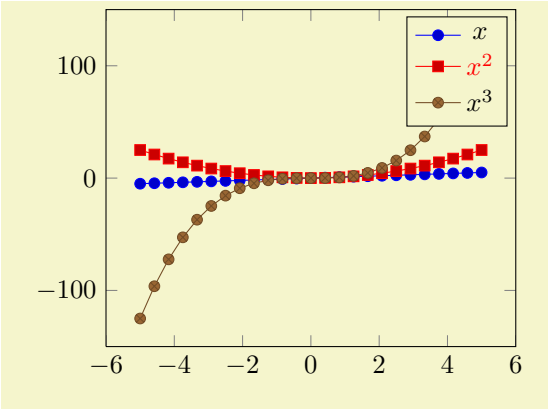
The commands for legend creation take precedence: the key `legend entries` is only considered if there is no legend command in the current axis.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[legend entries={$x$,$x^2$}]
    \addplot {x};
    \addplot {x^2};
    \legend{$a$,$b$}%  overrides the option
    \end{axis}
\end{tikzpicture}
```

Please be careful with whitespaces in ⟨*comma separated list*⟩: they will contribute to legend entries. Consider using '`%`' at the end of each line in multiline arguments (the end of line character is also a whitespace in TEX).

Just as for `\addlegendentry`, it is possible to provide `[`⟨*options*⟩`]` to single descriptions. To do so, place the options in square brackets right before the text:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend entries={$x$,[red]$x^2$,$x^3$}]
    \addplot {x};
    \addplot {x^2};
    \addplot {x^3};
\end{axis}
\end{tikzpicture}
```

If the square brackets contain a comma, you can enclose the complete entry in curly braces like `{[red,font=\Huge]Text}` (or you can use the '\\' delimiters).

4.8.5 Legend Appearance

/pgfplots/every axis legend (style, no value)

The style "every axis legend" determines the legend's position and outer appearance:

```
\pgfplotsset{every axis legend/.append style={
        at={(0,0)},
        anchor=south west}}
```

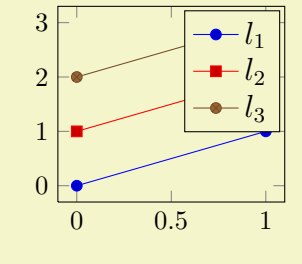
will draw it at the lower left corner of the axis while

```
\pgfplotsset{every axis legend/.append style={
        at={(1,1)},
        anchor=north east}}
```

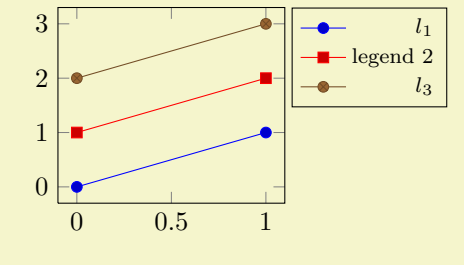
means the upper right corner. The 'anchor' option determines which point *of the legend* will be placed at $(0,0)$ or $(1,1)$.

The legend is a Ti*k*Z-matrix, so one can use any Ti*k*Z option which affects nodes and matrices (see [5, section 13 and 14]). The matrix is created by something like

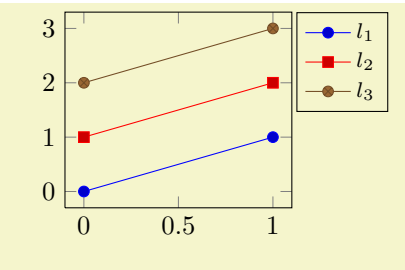
```
\matrix[style=every axis legend] {
    draw plot specification 1 & \node{legend 1}\\
    draw plot specification 2 & \node{legend 2}\\
    ...
};
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    %  this modifies 'every axis legend':
    legend style={font=\large}
]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```

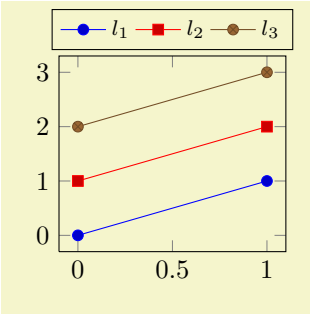



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    %  align right:
    legend style={
        cells={anchor=east},
        legend pos=outer north east,
    }
]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$, legend $2$,$l_3$}
\end{axis}
\end{tikzpicture}
```



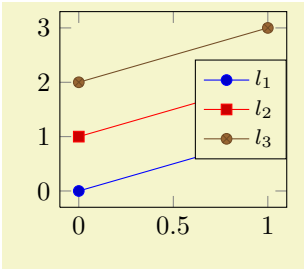

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  similar placement as previous example:
\pgfplotsset{every axis legend/.append style={
        at={(1.02,1)},
        anchor=north west}}
\begin{tikzpicture}
\begin{axis}
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```

Use `legend columns={⟨number⟩}` to configure the number of horizontal legend entries.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\pgfplotsset{every axis legend/.append style={
        at={(0.5,1.03)},
        anchor=south}}
\begin{axis}[legend columns=4]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```

Instead of the `/.append style`, it is possible to use `legend style` as in the following example. It has the same effect.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    legend style={
        at={(1,0.5)},
        anchor=east}]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```

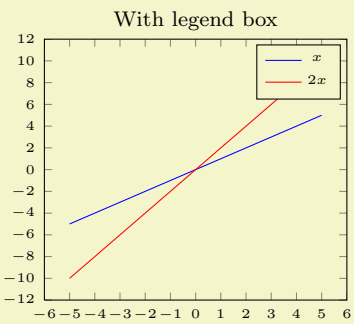
The default `every axis legend` style is

```
\pgfplotsset{every axis legend/.style={
    cells={anchor=center},%  Centered entries
    inner xsep=3pt,inner ysep=2pt,nodes={inner sep=2pt,text depth=0.15em},
    anchor=north east,
    shape=rectangle,
    fill=white,
    draw=black,
    at={(0.98,0.98)}
    }
}
```

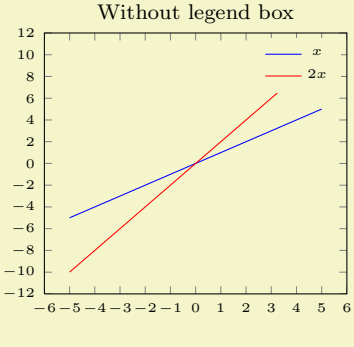
Whenever possible, consider using `/.append style` to keep the default styles active. This ensures compatibility with future versions.

```
\pgfplotsset{every axis legend/.append style={...}}
```

Note that in order to disable drawing of the legend box, you can use `draw=none` as style argument:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[tiny,title=With legend box]
\addplot[blue]{x};
\addplot[red]{2*x};
\legend{$x$,$2x$}
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[tiny,title=Without legend box,
    legend style={draw=none}]
\addplot[blue]{x};
\addplot[red]{2*x};
\legend{$x$,$2x$}
\end{axis}
\end{tikzpicture}
```

/pgfplots/**legend style**={⟨*key-value-list*⟩}

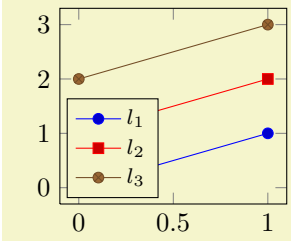
An abbreviation for **every axis legend**/**.append style**={⟨*key-value-list*⟩}.

It appends options to the already existing style **every axis legend**.

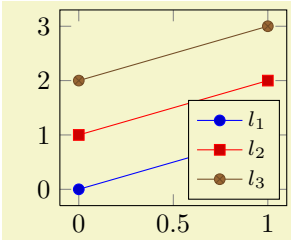
/pgfplots/**legend pos**=south west|south east|north west|north east|outer north east

A style which provides shorthand access to some commonly used legend positions.

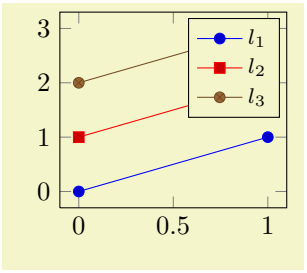
Each of these styles appends **at**={(⟨*x*⟩,⟨*y*⟩)},**anchor**=⟨*name*⟩ values to **every axis legend**.



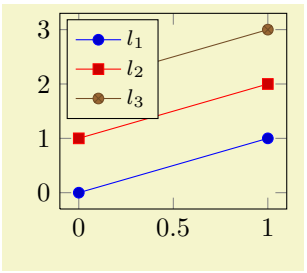

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend pos=south west]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```



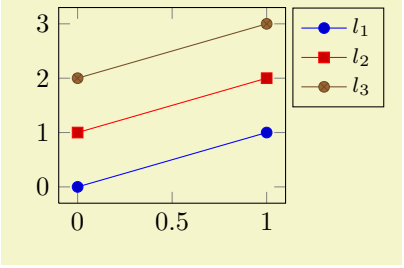

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend pos=south east]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend pos=north east]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```



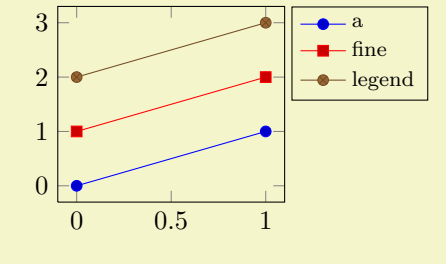

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend pos=north west]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```



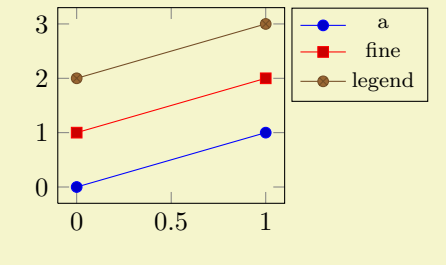

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend pos=outer north east]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{$l_1$,$l_2$,$l_3$}
\end{axis}
\end{tikzpicture}
```

/pgfplots/legend cell align=left|right|center (initially center)

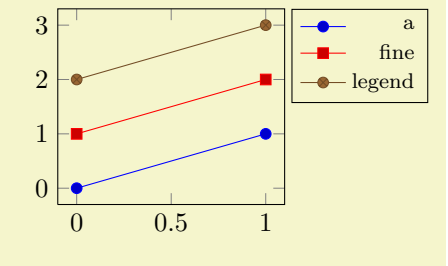
These keys provide horizontal alignment of legend cells.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend cell align=left,
    legend pos=outer north east]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{a,fine,legend}
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend cell align=center,
    legend pos=outer north east]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{a,fine,legend}
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[legend cell align=right,
    legend pos=outer north east]
\addplot coordinates {(0,0) (1,1)};
\addplot coordinates {(0,1) (1,2)};
\addplot coordinates {(0,2) (1,3)};
\legend{a,fine,legend}
\end{axis}
\end{tikzpicture}
```

They are actually just styles for commonly used alignment choices: the choice `left` is equivalent to `legend style={cells={anchor=west}}`; the second choice `right` is equivalent to `legend style={cells={anchor=east}}`, and `center` to `legend style={cells={anchor=center}}`. Using different values allows more control over cell alignment.

`/pgfplots/legend columns={⟨number⟩}` (default `1`)

Allows to configure the maximum number of adjacent legend entries. The default value `1` places legend entries vertically below each other.

Use `legend columns=-1` to draw all entries horizontally.

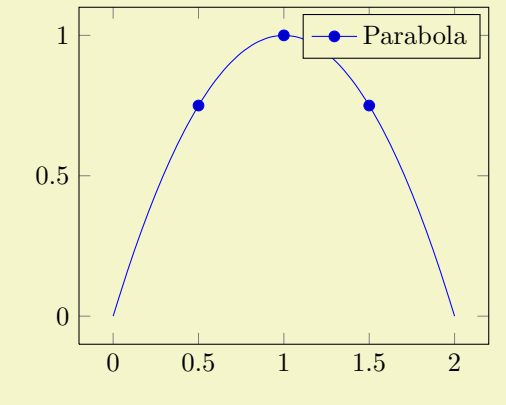
`/pgfplots/legend plot pos=left|right|none` (initially `left`)

Configures where the small line specifications will be drawn: left of the description, right of the description or not at all.

`/pgfplots/every legend image post` (style, no value)

A style which can be used to provide drawing options to every small legend image. These options apply after `current plot style` has been set, allowing users different line styles for legends than for plots.

For example, suppose you have a line plot and you plot selected markers on top of it (in the same color). Then, you may want to draw just a *single* legend entry (which should contain both the line *and* the markers). The following example shows a solution:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[legend image post style={mark=*}]
    \addplot+[only marks,forget plot]
        coordinates {(0.5,0.75) (1,1) (1.5,0.75)};
    \addplot+[mark=none,smooth,domain=0:2]
        {-x*(x-2)};
    \addlegendentry{Parabola}
    \end{axis}
\end{tikzpicture}
```

The example has two `\addplot` commands, one for the line and one for markers. Due to the `forget plot` option, the marker plot (the first one) doesn't advance the `cycle list`. The axis has only one legend entry, and since `legend image post style={mark=*}` has been used, the legend has a plot mark as well. Due to the `forget plot` option, the marker plot will not get a separate legend label.

`/pgfplots/legend image post style={⟨key-value-list⟩}`

An abbreviation for `every legend image post/.append style={⟨key-value-list⟩}`.

It appends options to the already existing style `every legend image post`.

`/pgfplots/legend image code/.code={⟨...⟩}`

Allows to replace the default images which are drawn inside of legends. When this key is evaluated, the current plot specification has already been activated (using `\begin{scope}[current plot style]`)[36], so any drawing operations use the same styles as the `\addplot` command.

The default is the style `line legend`.

Technical note: At the time when legend images are drawn, the style `every axis legend` is in effect – which have unwanted side-effects due to changed parameters (especially those concerning node placement, alignment, and shifting). It might be necessary to reset these parameters manually (PGFPLOTS also attempts to reset the fill color).

[36] This was different in versions before 1.3. The new scope features allow plot styles to change `legend image code`.

/pgfplots/line legend (style, no value)

A style which sets legend image code (back) to its initial value.

Its initial value is

```
\pgfplotsset{
    /pgfplots/line legend/.style={
        legend image code/.code={
            \draw[mark repeat=2,mark phase=2,##1]
                plot coordinates {
                    (0cm,0cm)
                    (0.3cm,0cm)
                    (0.6cm,0cm)
                };%
        }
    }
}
```

The style line legend can also be used to apply a different legend style to one particular plot (see the documentation on area legend for an example).

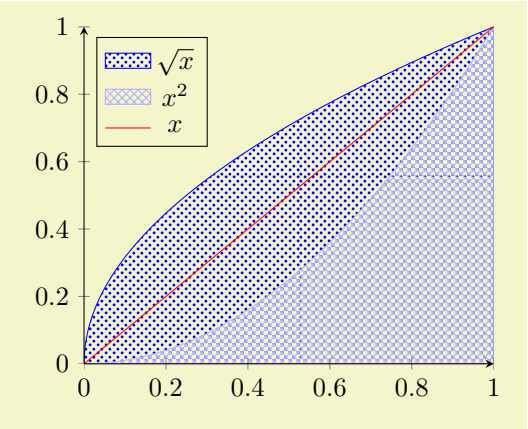
/pgfplots/empty legend (style, no value)

A style which clears legend image code, thereby omitting the legend image.

/pgfplots/area legend (style, no value)

A style which sets legend image code to

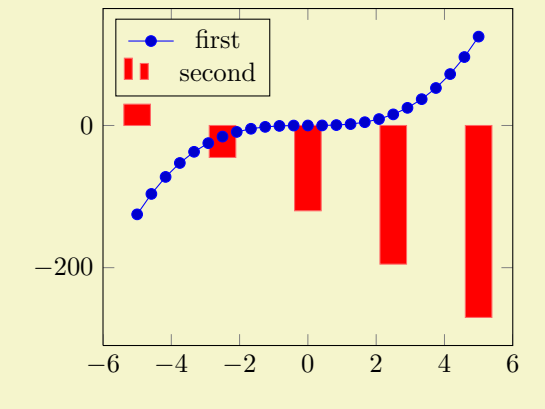
```
\pgfplotsset{
    legend image code/.code={%
        \draw[#1] (0cm,-0.1cm) rectangle (0.6cm,0.1cm);
    }
}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  \usetikzlibrary{patterns}
\begin{tikzpicture}
\begin{axis}[area legend,
    axis x line=bottom,
    axis y line=left,
    domain=0:1,
    legend style={at={(0.03,0.97)},
        anchor=north west},
    axis on top,xmin=0]
\addplot[pattern=crosshatch dots,
    pattern color=blue,draw=blue,
    samples=500]
    {sqrt(x)}    \closedcycle;

\addplot[pattern=crosshatch,
    pattern color=blue!30!white,
    draw=blue!30!white]
    {x^2} \closedcycle;

\addplot[red,line legend] coordinates {(0,0) (1,1)};
\legend{$\sqrt x$,$x^2$,$x$}
\end{axis}
\end{tikzpicture}
```

/pgfplots/xbar legend (no value)
/pgfplots/ybar legend (no value)
/pgfplots/zbar legend (no value)
/pgfplots/xbar interval legend (no value)
/pgfplots/ybar interval legend (no value)
/pgfplots/zbar interval legend (no value)

These style keys redefine legend image code such that legends use xbar, ybar or the xbar interval and ybar interval handlers.



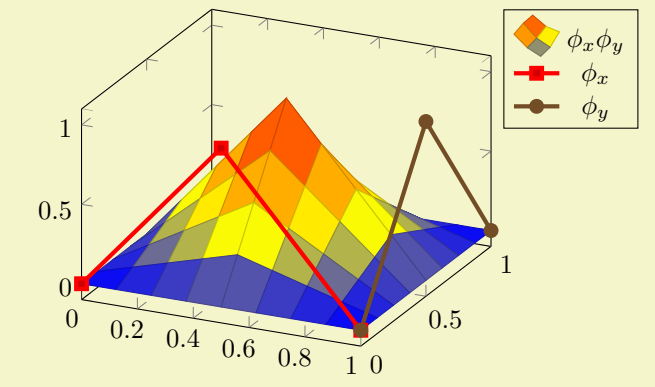

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[legend pos=north west]
    \addplot {x^3};
    \addplot[ybar,fill=red,draw=red!60,
        ybar legend,mark=none,samples=5]
        {-30*(x +4)};
    \legend{first,second}
    \end{axis}
\end{tikzpicture}
```

The initial values for these styles might be interesting if someone wants to modify them. Here they are:

```
\pgfplotsset{
    /pgfplots/xbar legend/.style={
        /pgfplots/legend image code/.code={%
            \draw[##1,/tikz/.cd,bar width=3pt,yshift=-0.2em,bar shift=0pt]
            plot coordinates {(0cm,0.8em) (2*\pgfplotbarwidth,0.6em)};},
    },
    /pgfplots/ybar legend/.style={
        /pgfplots/legend image code/.code={%
            \draw[##1,/tikz/.cd,bar width=3pt,yshift=-0.2em,bar shift=0pt]
            plot coordinates {(0cm,0.8em) (2*\pgfplotbarwidth,0.6em)};},
    },
    /pgfplots/xbar interval legend/.style={%
        /pgfplots/legend image code/.code={%
            \draw[##1,/tikz/.cd,yshift=-0.2em,bar interval width=0.7,bar interval shift=0.5]
            plot coordinates {(0cm,0.8em) (5pt,0.6em) (10pt,0.6em)};},
    },
    /pgfplots/ybar interval legend/.style={
        /pgfplots/legend image code/.code={%
            \draw[##1,/tikz/.cd,yshift=-0.2em,bar interval width=0.7,bar interval shift=0.5]
            plot coordinates {(0cm,0.8em) (5pt,0.6em) (10pt,0.6em)};},
    },
}
```

/pgfplots/mesh legend (no value)

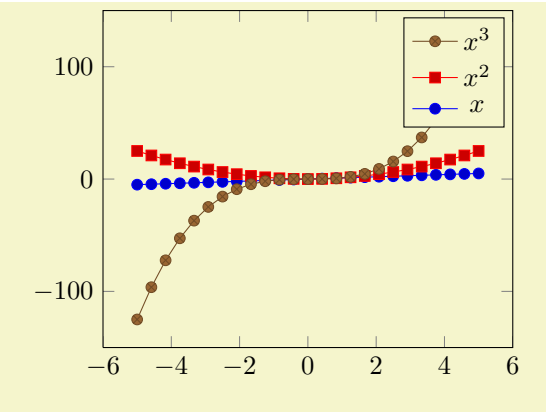
Redefines legend image code such that it is compatible with mesh and surf plot handlers (for three dimensional visualization mainly).



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[legend pos=outer north east]
    \addplot3[surf,samples=9,domain=0:1]
        {(1-abs(2*(x-0.5))) * (1-abs(2*(y-0.5)))};
    \addlegendentry{$\phi_x \phi_y$}

    \addplot3+[ultra thick] coordinates {(0,0,0) (0.5,0,1) (1,0,0)};
    \addlegendentry{$\phi_x $}

    \addplot3+[ultra thick] coordinates {(1,0,0) (1,0.5,1) (1,1,0)};
    \addlegendentry{$\phi_y $}
    \end{axis}
\end{tikzpicture}
```

/pgfplots/reverse legend=true|false (initially false)
/pgfplots/legend reversed=true|false (initially false)

Allows to reverse the order in which the pairs (legend entry, plot style) are drawn.

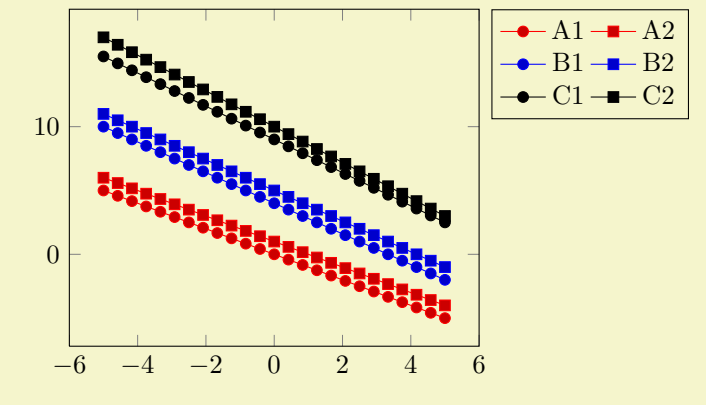



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[reverse legend]
    \addplot {x};
    \addlegendentry{$x$}
    \addplot {x^2};
    \addlegendentry{$x^2$}
    \addplot {x^3};
    \addlegendentry{$x^3$}
    \end{axis}
\end{tikzpicture}
```

/pgfplots/transpose legend=true|false (initially false)
/pgfplots/legend transposed=true|false (initially false)

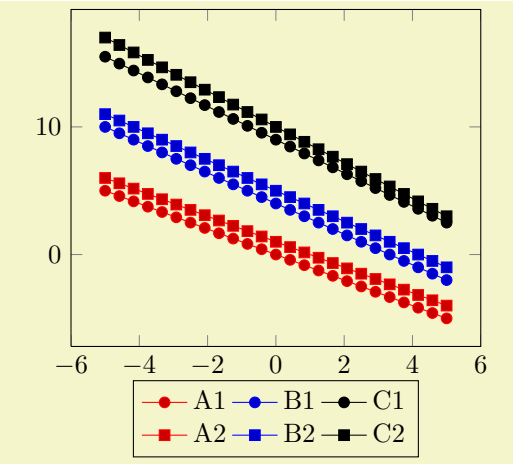
Allows to transpose the order in which the pairs (legend entry, plot style) are drawn.

Consider a set of 3 experiments, each consisting of 2 parameters. We might want to draw them together as in the following example:



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        legend columns=2,
        legend pos=outer north east,
        cycle multi list={%
            color list\nextlist
            [2 of]mark list
        }]
    \addplot {-x};   \addlegendentry{A1}
    \addplot {-x+1}; \addlegendentry{A2}

    \addplot {-1.2*x + 4};  \addlegendentry{B1}
    \addplot {-1.2*x + 5};  \addlegendentry{B2}

    \addplot {-1.3*x + 9};  \addlegendentry{C1}
    \addplot {-1.4*x + 10}; \addlegendentry{C2}
    \end{axis}
\end{tikzpicture}
```

An alternative might be to draw them horizontally – then, we'd like to use `transpose legend` to get a flat legend:



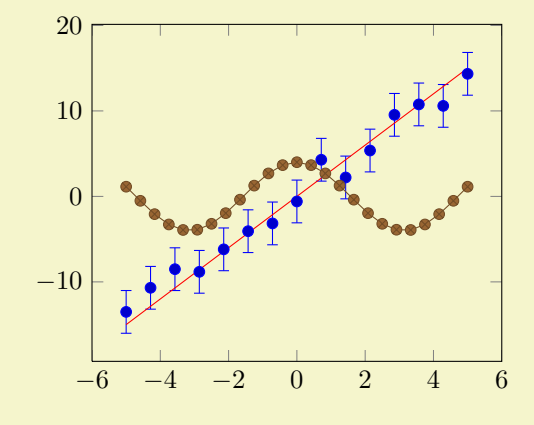

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        transpose legend,
        legend columns=2,
        legend style={at={(0.5,-0.1)},anchor=north},
        cycle multi list={%
            color list\nextlist
            [2 of]mark list
        }]
    \addplot {-x};   \addlegendentry{A1}
    \addplot {-x+1}; \addlegendentry{A2}

    \addplot {-1.2*x + 4};  \addlegendentry{B1}
    \addplot {-1.2*x + 5};  \addlegendentry{B2}

    \addplot {-1.3*x + 9};  \addlegendentry{C1}
    \addplot {-1.4*x + 10}; \addlegendentry{C2}
    \end{axis}
\end{tikzpicture}
```

Thus, `legend columns` defines the *input* columns, before the transposition (in other words, `legend columns` indicates the *rows* of the resulting legend).

Transposing legends has only an effect if `legend columns`> 1. Note that `reverse legend` has higher precedence: it is applied first.

4.8.6 Legends with `\label` and `\ref`

PGFPLOTS offers a `\label` and `\ref` feature for LaTeX to assemble a legend manually, for example as part of the figure caption. These references work as usual LaTeX references: a `\label` remembers where and what needs to be referenced and a `\ref` expands to proper text. In context of plots, a `\label` remembers the plot specification of one plot and a `\ref` expands to the small image which would also be used inside of legends.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}[baseline]
\begin{axis}
    \addplot+[only marks,
            samples=15,
            error bars/y dir=both,
            error bars/y fixed=2.5]
        {3*x+2.5*rand};
    \label{pgfplots:label1}

    \addplot+[mark=none] {3*x};
    \label{pgfplots:label2}

    \addplot {4*cos(deg(x))};
    \label{pgfplots:label3}
\end{axis}
\end{tikzpicture}
```

```
The picture shows the estimations \ref{pgfplots:label1} which are subjected to noise.
It appears the model \ref{pgfplots:label2} fits the data appropriately.
Finally, \ref{pgfplots:label3} is only here to get three examples.
```

The picture shows the estimations • which are subjected to noise. It appears the model —— fits the data appropriately. Finally, —●— is only here to get three examples.

`\label{`⟨*label name*⟩`}`
`\label[`⟨*reference*⟩`]{`⟨*label name*⟩`}`

When used after `\addplot`, this command creates a LaTeX label named ⟨*label name*⟩[37]. If this label is cross-referenced with `\ref{`⟨*label name*⟩`}` somewhere, the associated plot specification will be inserted.

Label3 = —●—; Label2 = ——

```
Label3 = \ref{pgfplots:label3};
Label2 = \ref{pgfplots:label2}
```

The label is assembled using `legend image code` and the plot style of the last plot. Any PGFPLOTS option is expanded until only Ti*k*Z (or PGF) options remain; these options are used to get an independent label.

More precisely, the small image generated by `\ref{`⟨*label name*⟩`}` is

```
\tikz[/pgfplots/every crossref picture] {...}
```

where the contents is determined by `legend image code` and the plot style.

The second syntax, `\label[`⟨*reference*⟩`]{`⟨*label name*⟩`}` allows to label particular pieces of an `\addplot` command. It is (currently) only interesting for `scatter/classes`: there, it allows to reference particular classes of the scatter plot. See page 77 for more details.

Note that `\label` information, even the small Ti*k*Z pictures here, can be combined with the `external` library for image externalization, see Section 7.1 for details (in particular, the `external/mode` key). In other words, references remain valid even if the defining axis has been externalized.

`\ref{`⟨*label name*⟩`}`

Can be used to reference a labeled, single plot. See the example above.

This will also work together with `hyperref` links and `\pageref`[38].

`/pgfplots/refstyle={`⟨*label name*⟩`}`

Can be used to set the *styles* of a labeled, single plot. This allows to write

```
\addplot[/pgfplots/refstyle={pgfplots:label2}]
```

somewhere. Please note that it may be easier to define a style with `.style`.

[37] This feature is *only* available in LaTeX, sorry.
[38] Older versions of PGFPLOTS required the use of `\protect\ref` when used inside of captions or section headings. This is no longer necessary.

/pgfplots/every crossref picture (style, no value)

A style which will be used by the cross-referencing feature for plots. The default is

```
\pgfplotsset{every crossref picture/.style={baseline,yshift=0.3em}}
```

/pgfplots/invoke before crossref tikzpicture={⟨*TEX code*⟩}
/pgfplots/invoke after crossref tikzpicture={⟨*TEX code*⟩}

Code which is invoked just before or just after every cross reference picture. This applies to legend images generated with `\ref`, `legend to name` and `colorbar to name` images.

The initial configuration checks if the `external` library is in effect. If so, it modifies the generated figure names by means of `\tikzappendtofigurename{_crossref}`.

4.8.7 Legends Outside Of an Axis

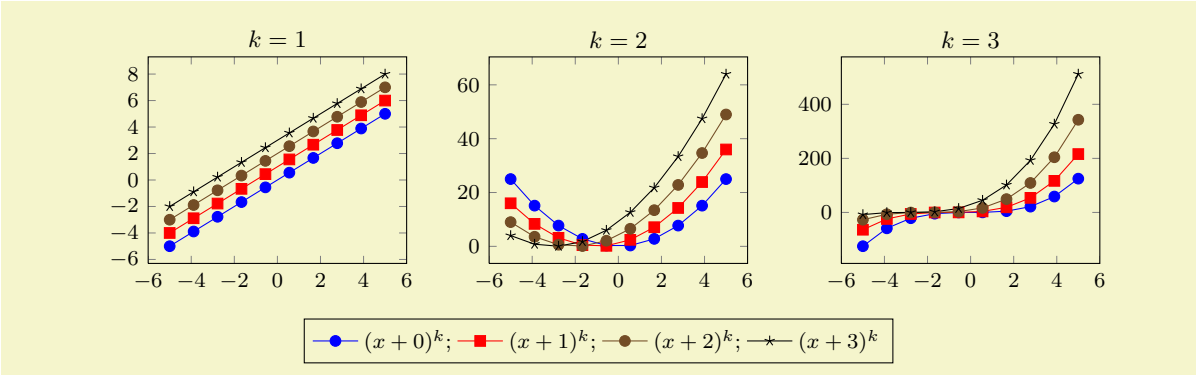
Occasionally, one has multiple adjacent plots, each with the same legend – and just *one* legend suffices. But where shall it be placed? And how? One solution is to use the `overlay` key to exclude the legend from bounding box computations, and place it absolutely such that it fits. Another is the `legend to name` feature:

/pgfplots/legend to name={⟨*name*⟩} (initially `empty`)

Enables a legend export mode: instead of drawing the legend, a self–contained, independent set of drawing commands will be stored using the label ⟨*name*⟩. The definition is done using `\label{`⟨*name*⟩`}`, just like any other LaTeX label. The name can be referenced using

`\ref{`⟨*name*⟩`}`.

Thus, typing `\ref{`⟨*name*⟩`}` somewhere outside of the axis, maybe even outside of any picture, will cause the legend to be drawn.



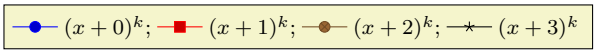
```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{footnotesize,samples=10}
\begin{center}%  note that \centering uses less vspace...
\begin{tikzpicture}
    \begin{axis}[
        legend columns=-1,
        legend entries={$(x+0)^k$;,$(x+1)^k$;,$(x+2)^k$;,$(x+3)^k$},
        legend to name=named,
        title={$k=1$}]
    \addplot {x};
    \addplot {x+1};
    \addplot {x+2};
    \addplot {x+3};
    \end{axis}
\end{tikzpicture}
%
\begin{tikzpicture}
    \begin{axis}[title={$k=2$}]
    \addplot {x^2};
    \addplot {(x+1)^2};
    \addplot {(x+2)^2};
    \addplot {(x+3)^2};
    \end{axis}
\end{tikzpicture}
%
\begin{tikzpicture}
    \begin{axis}[title={$k=3$}]
    \addplot {x^3};
    \addplot {(x+1)^3};
    \addplot {(x+2)^3};
    \addplot {(x+3)^3};
    \end{axis}
\end{tikzpicture}
\\

\ref{named}
\end{center}
```

Note that only the *first* plot has `legend entries`. Thus, its legend will be created as usual, and stored under the name '`named`', but it won't be drawn. The stored legend can then be drawn with `\ref{named}` below the three plots. Since there is no picture in this context, a `\tikz` picture is created and a `\matrix[/pgfplots/every axis legend]` path is drawn inside of it, resulting in the legend as if it had been placed inside of the axis.

The stored legend will contain the currently active values of legend- and plot style related options. This includes `legend image code`, `every axis legend`, and any plot style options (and some more). The algorithm works in the same way as for `\label` and `\ref`, i.e. it keeps any options with `/tikz/` prefix and expands those with `/pgfplots/` prefix.

Note that the legend is drawn with `every axis legend`, even though the placement options might be chosen to fit into an axis. You may want to adjust the style in the same axis in which the stored legend has been defined (the value will be copied and restored as well).

About `\ref{`*name*`}` The `\ref{`⟨*name*⟩`}` command retrieves a stored legend (one defined by `legend to name`) and draws it.

`\ref{named}`: 

If you want the legend to be exported *and* drawn inside of the current axis, consider using `extra description/.append code={\ref{`⟨*name*⟩`}}`.

Note that `\ref` can be combined with the `external` library for image externalization. In other words, the legend will work even if the defining axis has been externalized, see Section 7.1 for details (in particular the `external/mode` key).

Note furthermore that this `.aux` file related stuff is (currently) only supported, if PGFPLOTS is run by means of LaTeX, sorry.

`\pgfplotslegendfromname{`⟨*name*⟩`}`

This command poses an equivalent alternative for `\ref{`⟨*name*⟩`}`: it has essentially the same effect,

but it does not create links when used with the `hyperref` package[39].

`/pgfplots/every legend to name picture` (style, no value)

A style which is installed when `\ref` is used outside of a picture: a new picture will be created with `\tikz[/pgfplots/every legend to name picture]`.

Thus, you can redefine this style to set alignment options (such as `baseline`). For example, the initialization

```
\pgfplotsset{
    legend style={matrix anchor=west,at={(0pt,0pt)}},
    every legend to name picture/.style={baseline},
}
...
```

will cause the legend to be positioned such that its `west` anchor is at `y=0pt`. The `baseline` option will align this point of the legend with the text baseline (please refer to the documentation for `baseline` in Section 4.18 for details).

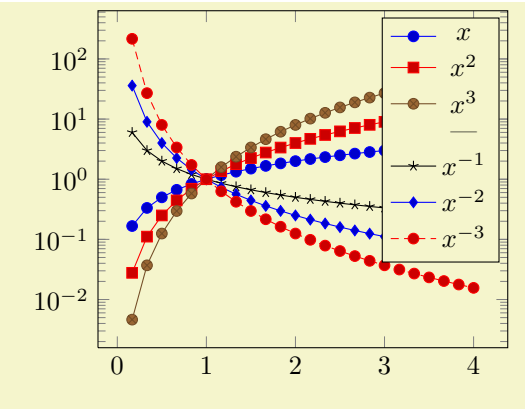
4.8.8 Legends with Customized Texts or Multiple Lines

`\addlegendimage{`⟨*options*⟩`}`

Adds a further legend image for legend creation.

Each `\addplot` command appends its plot style options to a list, and `\addlegendimage` adds ⟨*options*⟩ to the very same list.

Thus, the effect is as if you had provided `\addplot[`⟨*options*⟩`]`, but `\addlegendimage` bypasses all the logic usually associated with a plot. In other words: except for the legend, the state of the axis remains as if the command would not have been issued. Not even the current plot's index is advanced.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}[
    domain=0:4,
]
    \addplot {x};   \addlegendentry{$x$}
    \addplot {x^2}; \addlegendentry{$x^2$}
    \addplot {x^3}; \addlegendentry{$x^3$}
    \addlegendimage{empty legend}
    \addlegendentry{---}
    \addplot {x^(-1)}; \addlegendentry{$x^{-1}$}
    \addplot {x^(-2)}; \addlegendentry{$x^{-2}$}
    \addplot {x^(-3)}; \addlegendentry{$x^{-3}$}
\end{semilogyaxis}
\end{tikzpicture}
```

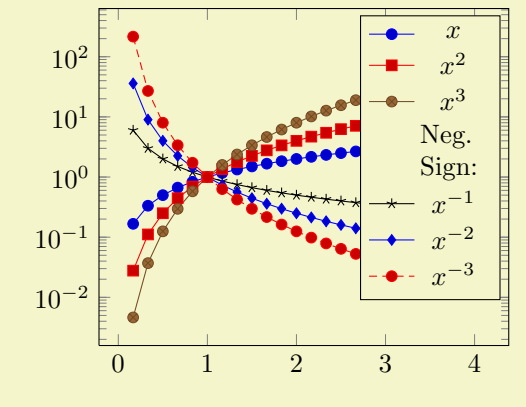
The example above has six plots, each with its legend entry. Furthermore, it has an `\addlegendimage` command and its separate legend entry. We see that `\addlegendimage` needs its own legend entry, but it is detached from the processing of plots as such. In our case, we chose `empty legend` as style for the separator.

Use `\addlegendimage` to provide custom styles into legends, for example to document custom `\draw` commands inside of an axis.

You can call `\label` after `\addlegendimage` just as for a normal style.

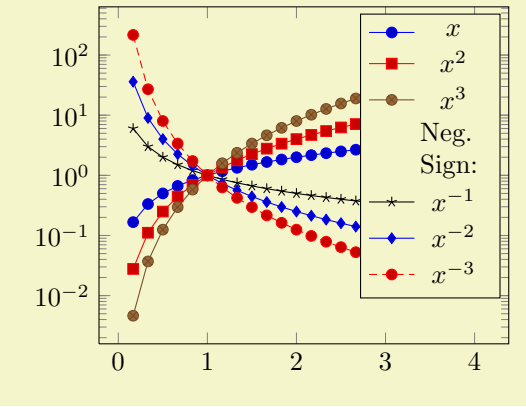
Occasionally, one may want multiple lines for legend entries. That is possible as well using a fixed `text width`:

[39] Since this manual uses colored links, the text in `\ref` would usually be blue. Using `\pgfplotslegendfromname` avoids link text colors in the legend (this has been applied to the manual styles here).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}[
    domain=0:4,
]
    \addplot {x};   \addlegendentry{$x$}
    \addplot {x^2}; \addlegendentry{$x^2$}
    \addplot {x^3}; \addlegendentry{$x^3$}
    \addlegendimage{empty legend}
    \addlegendentry[text width=25pt,text depth=]
        {Neg. Sign:}
    \addplot {x^(-1)}; \addlegendentry{$x^{-1}$}
    \addplot {x^(-2)}; \addlegendentry{$x^{-2}$}
    \addplot {x^(-3)}; \addlegendentry{$x^{-3}$}
\end{semilogyaxis}
\end{tikzpicture}
```

The example provides options for the single multiline element. Note that the initial configuration of `legend style` employs `text depth=0.15em`, which needs to be reset manually to `text depth={}`[40].
There are two approaches with the same effect which are subject of the following example:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}[
    domain=0:4,
    legend entries={%
      $x$,$x^2$,$x^3$,%
      {[text width=25pt,text depth=]Neg. Sign:},%
      $x^{-1}$,$x^{-2}$,$x^{-3}$},
    %  same effect:
    %  legend style={
    %      nodes={text width=25pt,text depth=}}
]
    \addplot {x};
    \addplot {x^2};
    \addplot {x^3};
    \addlegendimage{empty legend}
    \addplot {x^(-1)};
    \addplot {x^(-2)};
    \addplot {x^(-3)};
\end{semilogyaxis}
\end{tikzpicture}
```

Here, the `legend entries` are provided using the single key syntax. Note that the special options are provided as part of the legend entry, using square brackets right before the text as such. The comments indicate that you could also add the `text width` stuff to `legend style`, in which case it would hold for every node.

Note that legend texts are realized using `\node[`⟨*options*⟩`] {`⟨*text*⟩`};`, so anything which produces a valid Ti*k*Z node is permitted (this includes `minipage` or `tabular` environments inside of ⟨*text*⟩).

4.8.9 Axis Lines

An extension by Pascal Wolkotte

By default the axis lines are drawn as a `box`, but it is possible to change the appearance of the x and y axis lines.

`/pgfplots/axis x line=box|top|middle|center|bottom|none` (initially `box`)
`/pgfplots/axis x line*=box|top|middle|center|bottom|none` (initially `box`)
`/pgfplots/axis y line=box|left|middle|center|right|none` (initially `box`)
`/pgfplots/axis y line*=box|left|middle|center|right|none` (initially `box`)
`/pgfplots/axis lines=box|left|middle|center|right|none`
`/pgfplots/axis lines*=box|left|middle|center|right|none`

These keys allow to choose the locations of the axis lines. The last one, `axis lines` sets the same value for every axis.

[40] Perhaps I can reset `text depth` automatically in the future.

Ticks and tick labels are placed according to the chosen value as well. The choice `bottom` will draw the x line at $y = y_{\min}$, `middle` will draw the x line at $y = 0$, and `top` will draw it at $y = y_{\max}$. Finally, `box` is a combination of options `top` and `bottom`. The choice `axis x line=none` is an alias for `hide x axis`. The y- and z variants work in a similar way.

The case `center` is a synonym for `middle`, both draw the line through the respective coordinate 0. If this coordinate is not part of the axis limit, the lower axis limit is chosen instead.

The starred versions `...line*` *only* affect the axis lines, without correcting the positions of axis labels, tick lines or other keys which are (possibly) affected by a changed axis line. The non-starred versions are actually styles which set the starred key *and* some other keys which also affect the figure layout:

- In case `axis x line=box`, the style `every boxed x axis` will be installed immediately.
- In case `axis x line`$\neq$`box`, the style `every non boxed x axis` will be installed immediately. Furthermore, some of these choices will modify axis label positions.

The handling of `axis y line` and `axis z line` is similar. The default styles are defined as

```
\pgfplotsset{
    every non boxed x axis/.style={
        xtick align=center,
        enlarge x limits=false,
        x axis line style={-stealth}
    },
    every boxed x axis/.style={}
}
```

In addition, conditional modifications of axis label styles will be taken. For example, `axis x line=middle` will set

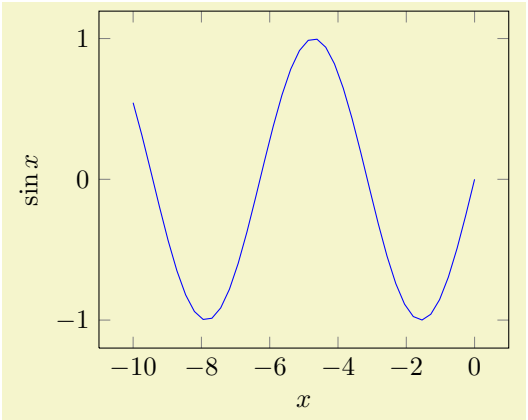
```
\pgfplotsset{every axis x label/.style={at={(current axis.left of origin)},anchor=south west}}
```

if the matching y style has value `axis y line=right` and

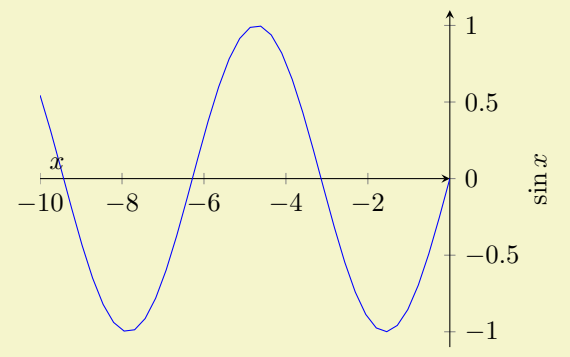
```
\pgfplotsset{every axis x label/.style={at={(current axis.right of origin)},anchor=south east}}
```

if `axis y line`$\neq$`right`.

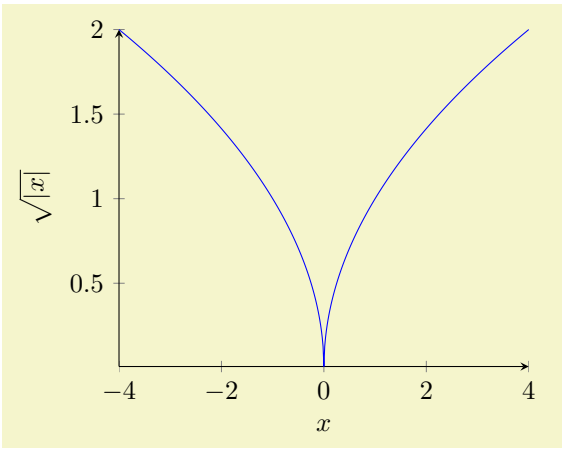
Feel free to overwrite these styles if the default doesn't fit your needs or taste. Again, these styles will *not* be used for `axis line*`.



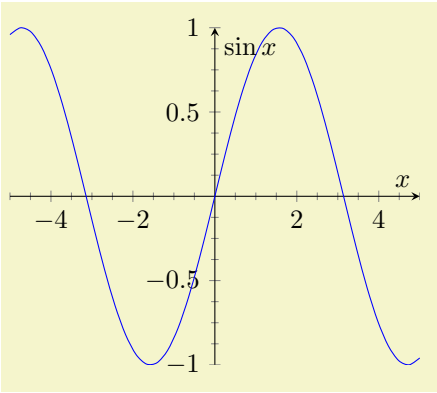

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xlabel=$x$,ylabel=$\sin x$]

    \addplot[blue,mark=none,
         domain=-10:0,samples=40]
        {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```



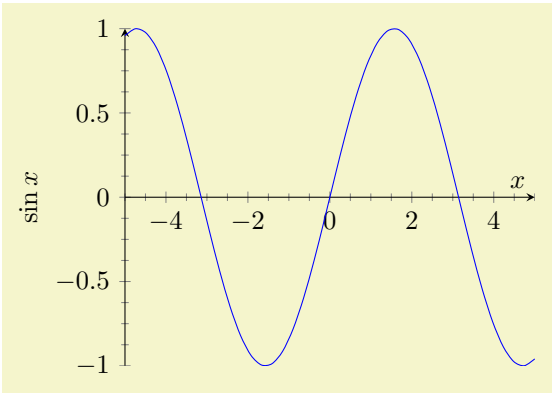

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    axis x line=middle,
    axis y line=right,
    ymax=1.1, ymin=-1.1,
    xlabel=$x$,ylabel=$\sin x$
]
    \addplot[blue,mark=none,
         domain=-10:0,samples=40]
        {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    axis x line=bottom,
    axis y line=left,
    xlabel=$x$,ylabel=$\sqrt{|x|}$
]
\addplot[blue,mark=none,
     domain=-4:4,samples=501]
    {sqrt(abs(x))};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    minor tick num=3,
    axis y line=center,
    axis x line=middle,
    xlabel=$x$,ylabel=$\sin x$
    ]
    \addplot[smooth,blue,mark=none,
         domain=-5:5,samples=40]
        {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    minor tick num=3,
    axis y line=left,
    axis x line=middle,
    xlabel=$x$,ylabel=$\sin x$
    ]
    \addplot[smooth,blue,mark=none,
         domain=-5:5,samples=40]
        {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```

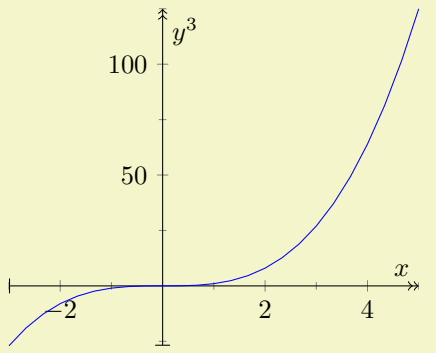
In case `middle`, the style `every inner axis x line` allows to adjust the appearance.

Note that three dimensional axes only support to use the same value for every axis, i.e. three dimensional axes support only the `axis lines` key (or, preferably for 3D axes, the `axis lines*` key – check what looks best). See Section 4.10.4 for examples of three dimensional axis line variations.

/pgfplots/every inner x axis line (no value)
/pgfplots/every inner y axis line (no value)
/pgfplots/every inner z axis line (no value)

A style key which can be redefined to customize the appearance of *inner* axis lines. Inner axis lines are those drawn by the `middle` (or `center`) choice of `axis x line`, see above.

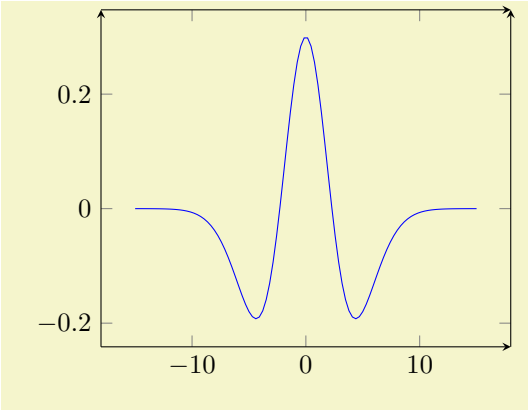
This style affects *only* the line as such.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    minor tick num=1,
    axis x line=middle,
    axis y line=middle,
    every inner x axis line/.append style=
        {|->>},
    every inner y axis line/.append style=
        {|->>},
    xlabel=$x$,ylabel=$y^3$
]
\addplot[blue,domain=-3:5] {x^3};
\end{axis}
\end{tikzpicture}
```

/pgfplots/every outer x axis line (no value)
/pgfplots/every outer y axis line (no value)
/pgfplots/every outer z axis line (no value)

Similar to `every inner x axis line`, this style configures the appearance of all axis lines which are part of the outer box.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    separate axis lines, %  important !
    every outer x axis line/.append style=
        {-stealth},
    every outer y axis line/.append style=
        {-stealth},
]
\addplot[blue,id=DoG,
        samples=100,
        domain=-15:15]
  gnuplot{1.3*exp(-x**2/10) - exp(-x**2/20)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/axis line style={⟨*key-value-list*⟩}

A command which appends ⟨*key-value-list*⟩ to *all* axis line appearance styles.

/pgfplots/inner axis line style={⟨*key-value-list*⟩}

A command which appends ⟨*key-value-list*⟩ to both, `every inner x axis line` and the y variant.

/pgfplots/outer axis line style={⟨*key-value-list*⟩}

A command which appends ⟨*key-value-list*⟩ to both, `every outer x axis line` and the y variant.

/pgfplots/x axis line style={⟨*key-value-list*⟩}
/pgfplots/y axis line style={⟨*key-value-list*⟩}
/pgfplots/z axis line style={⟨*key-value-list*⟩}

A command which appends ⟨*key-value-list*⟩ to all axis lines styles for either x or y axis.

/pgfplots/every boxed x axis (no value)
/pgfplots/every boxed y axis (no value)

/pgfplots/every boxed z axis (no value)

A style which will be installed as soon as axis x line=box (y) is set.

The default is simply empty.

/pgfplots/every non boxed x axis (no value)
/pgfplots/every non boxed y axis (no value)
/pgfplots/every non boxed z axis (no value)

A style which will be installed as soon as axis x line (y) will be set to something different than box.

The default is

```
\pgfplotsset{
    every non boxed x axis/.style={
        xtick align=center,
        enlarge x limits=false,
        x axis line style={-stealth}}}
```

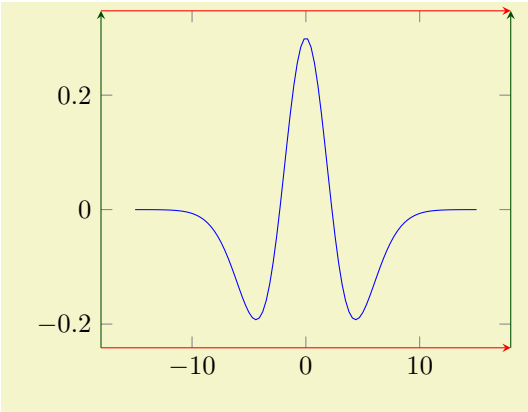
with similar values for the y-variant. Feel free to redefine this style to your needs and taste.

/pgfplots/separate axis lines={⟨*true*,*false*⟩} (default true)

Enables or disables separate path commands for every axis line. This option affects *only* the case if axis lines are drawn as a *box*.

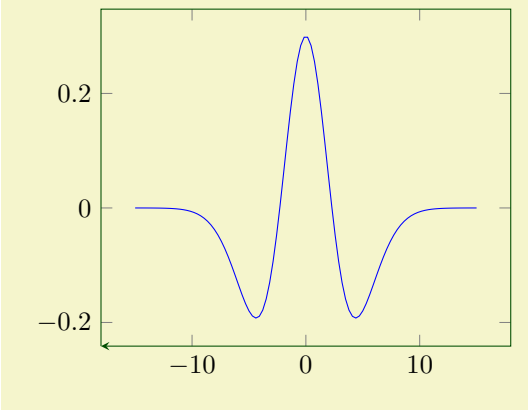
Both cases have their advantages and disadvantages, I fear there is no reasonable default (suggestions are welcome).

The case separate axis lines=true allows to draw arrow heads on each single axis line, but it can't close edges very well – in case of thick lines, unsatisfactory edges occur.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    separate axis lines,
    every outer x axis line/.append style=
        {-stealth,red},
    every outer y axis line/.append style=
        {-stealth,green!30!black},
]
\addplot[blue,
        samples=100,
        domain=-15:15]
    {1.3*exp(0-x^2/10) - exp(0-x^2/20)};
  %  Unfortunately, there is a bug in PGF 2.00
  %  something like exp(-10^2)
  %  must be written as exp(0-10^2) :-(
\end{axis}
\end{tikzpicture}
```

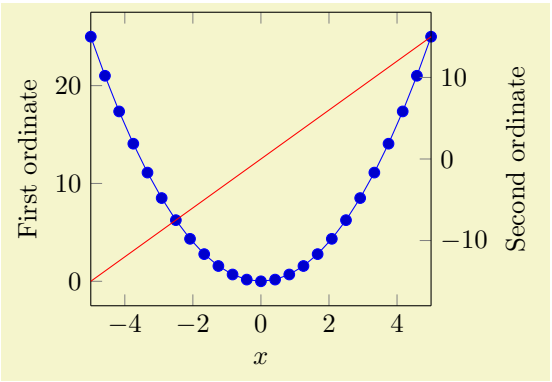
The case separate axis lines=false issues just *one* path for all axis lines. It draws a kind of rectangle, where some parts of the rectangle may be skipped over if they are not wanted. The advantage is that edges are closed properly. The disadvantage is that at most one arrow head is added to the path (and yes, only one drawing color is possible).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    separate axis lines=false,
    every outer x axis line/.append style=
        {-stealth,red},
    every outer y axis line/.append style=
        {-stealth,green!30!black},
]
\addplot[blue,id=DoG,
        samples=100,
        domain=-15:15]
  gnuplot{1.3*exp(-x**2/10) - exp(-x**2/20)};
\end{axis}
\end{tikzpicture}
```

4.8.10 Two Ordinates (y axis) or Multiple Axes

In some applications, more than one y axis is used if the x range is the same. This section demonstrates how to create them. The idea in PGFPLOTS is to draw two axes on top of each other, one with descriptions only on the left and the second with descriptions only on the right:

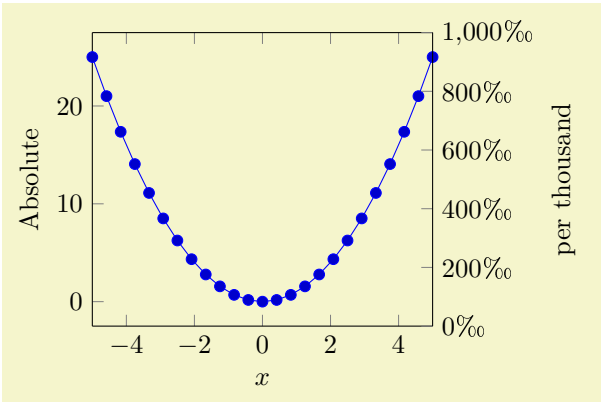



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{axis}[
    scale only axis,
    xmin=-5,xmax=5,
    axis y line*=left,%  the '*' avoids arrow heads
    xlabel=$x$,
    ylabel=First ordinate]
  \addplot {x^2};
  \end{axis}

  \begin{axis}[
    scale only axis,
    xmin=-5,xmax=5,
    axis y line*=right,
    axis x line=none,
    ylabel=Second ordinate]
  \addplot[red] {3*x};
  \end{axis}
\end{tikzpicture}
```

Thus, the two axes are drawn "on top" of each other – one, which contains the x axis and the left y axis, and one which has *only* the right y axis. Since PGFPLOTS does not really know what it's doing here, user attention in the following possibly non-obvious aspects is required:

1. Scaling. You should set `scale only axis` because this forces equal dimensions for both axis, without respecting any labels.
2. Same x limits. You should set those limits explicitly.

You may want to consider different legend styles. It is also possible to use only the axis, without any plots:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  \usepackage{textcomp}
\begin{tikzpicture}
  \begin{axis}[
    scale only axis,
    xmin=-5,xmax=5,
    axis y line*=left,% '*' avoids arrow heads
    xlabel=$x$,
    ylabel=Absolute]
  \addplot {x^2};
  \end{axis}

  \begin{axis}[
    scale only axis,
    xmin=-5,xmax=5,
    ymin=0,ymax=1000,
    yticklabel=
{$\pgfmathprintnumber{\tick}$\textperthousand},
    axis y line*=right,
    axis x line=none,
    ylabel=per thousand]
  \end{axis}
\end{tikzpicture}
```

4.8.11 Axis Discontinuities

An extension by Pascal Wolkotte

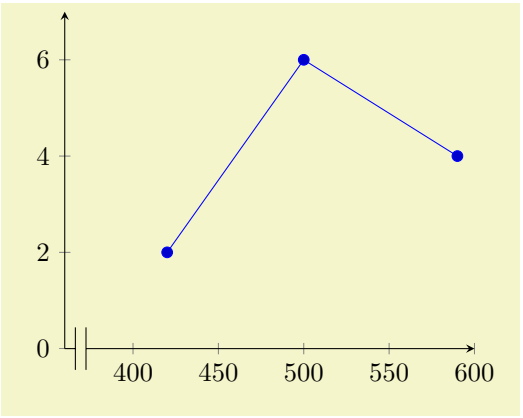
In case the range of either of the axis do not include the zero value, it is possible to visualize this with a discontinuity decoration on the corresponding axis line.

`/pgfplots/axis x discontinuity=crunch|parallel|none` (initially `none`)

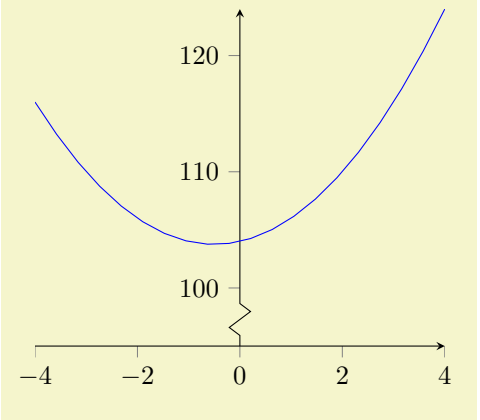
/pgfplots/axis y discontinuity=crunch|parallel|none (initially none)
/pgfplots/axis z discontinuity=crunch|parallel|none (initially none)

Insert a discontinuity decoration on the x (or y, respectively) axis. This is to visualize that the y axis does cross the x axis at its 0 value, because the minimum x axis value is positive or the maximum value is negative.

The description applies to axis y discontinuity and axis z discontinuity as well, simply substitute x by y or z, respectively.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    axis x line=bottom,
    axis x discontinuity=parallel,
    axis y line=left,
    xmin=360, xmax=600,
    ymin=0, ymax=7,
     enlargelimits=false
]
    \addplot coordinates {
        (420,2)
        (500,6)
        (590,4)
    };
\end{axis}
\end{tikzpicture}
```

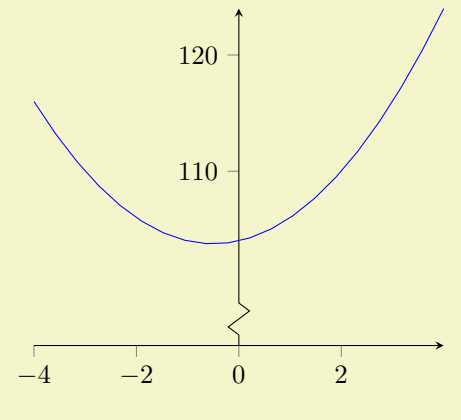



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    axis x line=bottom,
    axis y line=center,
    tick align=outside,
    axis y discontinuity=crunch,
    ymin=95, enlargelimits=false
]
    \addplot[blue,mark=none,
         domain=-4:4,samples=20]
        {x*x+x+104};
\end{axis}
\end{tikzpicture}
```

A problem might occur with the placement of the ticks on the axis. This can be solved by specifying the minimum or maximum axis value for which a tick will be placed.

/pgfplots/xtickmin={⟨*coord*⟩} (default axis limits)
/pgfplots/ytickmin={⟨*coord*⟩} (default axis limits)
/pgfplots/ztickmin={⟨*coord*⟩} (default axis limits)
/pgfplots/xtickmax={⟨*coord*⟩} (default axis limits)
/pgfplots/ytickmax={⟨*coord*⟩} (default axis limits)
/pgfplots/ztickmax={⟨*coord*⟩} (default axis limits)

The options xtickmin, xtickmax and ytickmin, ytickmax allow to define the axis tick limits, i.e. the axis values before respectively after no ticks will be placed. Everything outside of the axis tick limits will be not drawn. Their default values are equal to the axis limits.

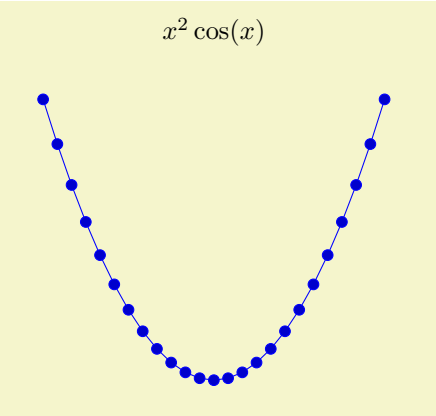



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    axis x line=bottom,
    axis y line=center,
    tick align=outside,
    axis y discontinuity=crunch,
    xtickmax=3,
    ytickmin=110,
    ymin=95, enlargelimits=false
]
    \addplot[blue,mark=none,
         domain=-4:4,samples=20]
        {x*x+x+104};
\end{axis}
\end{tikzpicture}
```

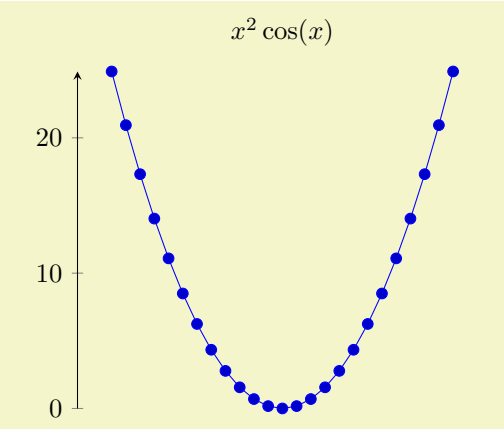
/pgfplots/hide x axis=true|false (initially false)
/pgfplots/hide y axis=true|false (initially false)
/pgfplots/hide z axis=true|false (initially false)
/pgfplots/hide axis=true|false (initially false)

Allows to hide either a selected axis or all of them. No outer rectangle, no tick marks and no labels will be drawn. Only titles and legends will be processed as usual.

Axis scaling and clipping will be done as if you did not use `hide axis`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        hide x axis,
        hide y axis,
        title={$x^2\cos(x)$}]
    \addplot {cos(x)*x^2};
    \end{axis}
\end{tikzpicture}
```



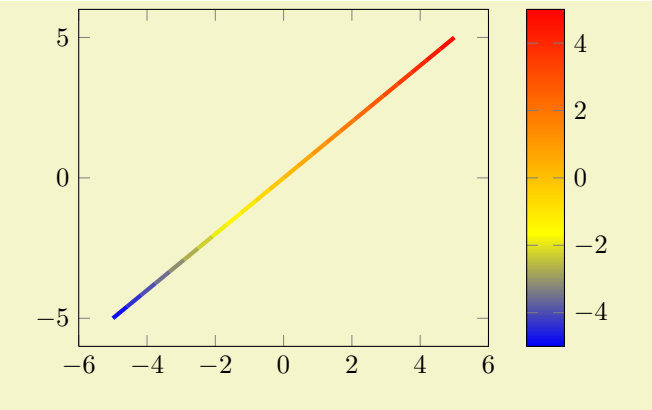

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        hide x axis,
        axis y line=left,
        title={$x^2\cos(x)$}]
    \addplot {cos(x)*x^2};
    \end{axis}
\end{tikzpicture}
```

4.8.12 Color Bars

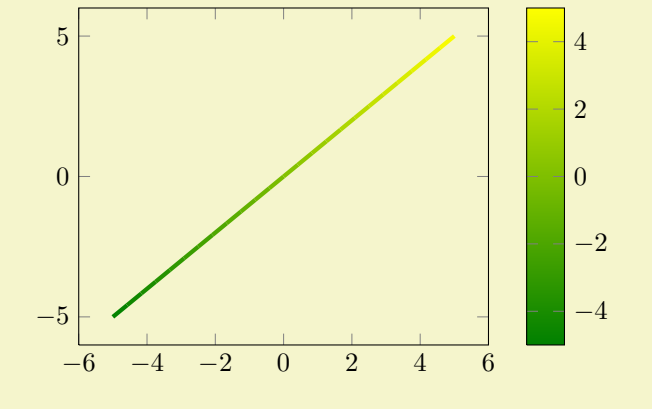
PGFPLOTS supports mesh, surface and scatter plots which can use color maps. While color maps can be chosen as described in Section 4.6.6, they can be visualized using color bars.

/pgfplots/colorbar=true|false (initially false)

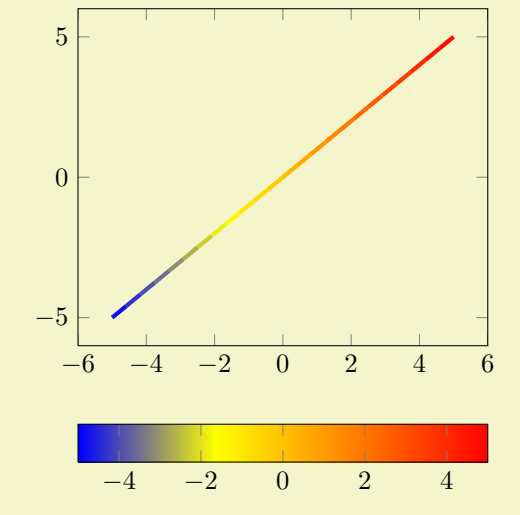
Activates or deactivates color bars.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar]
        \addplot[mesh,ultra thick] {x};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar,colormap/greenyellow]
        \addplot[mesh,ultra thick] {x};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar horizontal]
        \addplot[mesh,ultra thick] {x};
    \end{axis}
\end{tikzpicture}
```

A color bar is only useful for plots with non–zero color data range, more precisely, for which minimum and maximum `point meta` data is available. Usually, this is the case for `scatter`, `mesh` or `surf` (or

similar) plots, but you can also set `point meta min` and `point meta max` manually in order to draw a `colorbar`.

Color bars are just normal axes which are placed right besides their parent axes. The only difference is that they inherit several styles such as line width and fonts and they contain a bar shaded with the color map of the current axis.

Color bars are drawn internally with

```
\axis[every colorbar,colorbar shift,colorbar=false]
    \addplot graphics {};
\endaxis
```

where the placement, alignment, appearance and other options are done by the two styles `every colorbar` and `colorbar shift`. These styles and the possible placement and alignment options are described below.

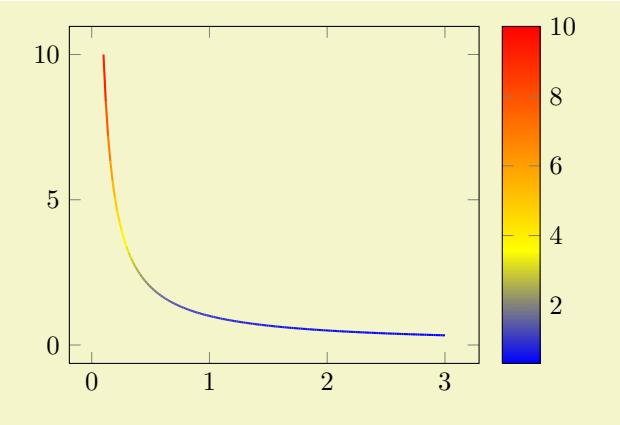
Remarks for special cases:

- Since there is always only one color bar per plot, this color bar uses the axis wide configurations of color map and color data. Consider using `colorbar source` to select color data limits of a particular `\addplot` command instead.
- If someone needs more than one color bar, the draw command above needs to be updated. See the key `colorbar/draw/.code` for this special case.

`/pgfplots/colorbar right` (style, no value)

A style which redefines `every colorbar` and `colorbar shift` such that color bars are placed right of their parent axis.

This is the initial configuration.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar right]
    \addplot[mesh,thick,samples=150,domain=0.1:3]
        {1/x};
    \end{axis}
\end{tikzpicture}
```

The style `colorbar right` is defined as

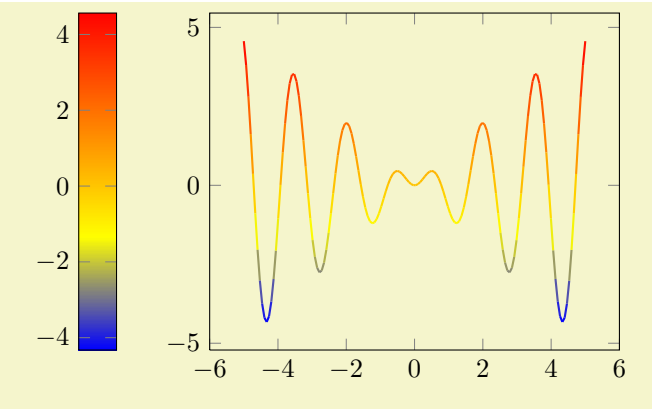
```
\pgfplotsset{
    colorbar right/.style={
        /pgfplots/colorbar=true,
        /pgfplots/colorbar shift/.style={xshift=0.3cm},
        /pgfplots/every colorbar/.style={
            title=,
            xlabel=,
            ylabel=,
            zlabel=,
            legend entries=,
            axis on top,
            at={(parent axis.right of north east)},
            anchor=north west,
            xmin=0,
            xmax=1,
            ymin=\pgfkeysvalueof{/pgfplots/point meta min},
            ymax=\pgfkeysvalueof{/pgfplots/point meta max},
            plot graphics/xmin=0,
            plot graphics/xmax=1,
            plot graphics/ymin=\pgfkeysvalueof{/pgfplots/point meta min},
            plot graphics/ymax=\pgfkeysvalueof{/pgfplots/point meta max},
            enlargelimits=false,
            scale only axis,
            height=\pgfkeysvalueof{/pgfplots/parent axis height},
            x=\pgfkeysvalueof{/pgfplots/colorbar/width},
            yticklabel pos=right,
            xtick=\empty,
            colorbar vertical/lowlevel,
        }
    },
    /pgfplots/colorbar vertical/lowlevel/.style={
        plot graphics/lowlevel draw/.code 2 args={%
            \pgfuseshading{...} %  some advanced basic level shading operations
        }
    },
}
```

Attention: `colorbar right` *re*defines `every colorbar`. That means any user customization must take place *after* `colorbar right`:

```
%  correct:
\begin{axis}[colorbar right, colorbar style={<some customization>}]
%  wrong, colorbar right resets the customization:
\begin{axis}[colorbar style={<some customization>}, colorbar right]
```

`/pgfplots/colorbar left` (style, no value)

A style which re-defines `every colorbar` and `colorbar shift` such that color bars are placed left of their parent axis.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar left]
    \addplot[mesh,thick,samples=150]
        {x*sin(deg(4*x))};
    \end{axis}
\end{tikzpicture}
```

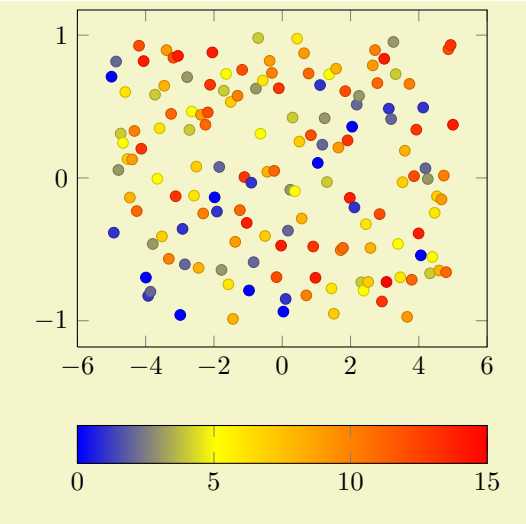
The style `colorbar left` is defined as

```
\pgfplotsset{
    colorbar left/.style={
        /pgfplots/colorbar right,
        /pgfplots/colorbar shift/.style={xshift=-0.3cm},
        /pgfplots/every colorbar/.append style={
            at={(parent axis.left of north west)},
            anchor=north east,
            ytick label pos=left,
        }
    }
}
```

Attention: `colorbar left` *re*defines `every colorbar`. That means any user customization must take place *after* `colorbar left` (see also the documentation for `colorbar right`).

`/pgfplots/colorbar horizontal` (style, no value)

A style which re-defines `every colorbar` and `colorbar shift` such that color bars are placed below their parent axis, with a horizontal bar.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar horizontal]
    \addplot[only marks,scatter,
        scatter src={mod(\coordindex,15)},samples=150]
        {rand};
    \end{axis}
\end{tikzpicture}
```

This style is defined as

```
\pgfplotsset{
    colorbar horizontal/.style={
        /pgfplots/colorbar=true,
        /pgfplots/colorbar shift/.style={yshift=-0.3cm},
        /pgfplots/every colorbar/.style={
            title=,
            xlabel=,
            ylabel=,
            zlabel=,
            legend entries=,
            axis on top,
            at={(parent axis.below south west)},
            anchor=north west,
            ymin=0,
            ymax=1,
            xmin=\pgfkeysvalueof{/pgfplots/point meta min},
            xmax=\pgfkeysvalueof{/pgfplots/point meta max},
            plot graphics/ymin=0,
            plot graphics/ymax=1,
            plot graphics/xmin=\pgfkeysvalueof{/pgfplots/point meta min},
            plot graphics/xmax=\pgfkeysvalueof{/pgfplots/point meta max},
            enlargelimits=false,
            scale only axis,
            width=\pgfkeysvalueof{/pgfplots/parent axis width},
            y=\pgfkeysvalueof{/pgfplots/colorbar/width},
            xticklabel pos=left,
            ytick=\empty,
            colorbar horizontal/lowlevel,
        }%
    },%
    /pgfplots/colorbar horizontal/lowlevel/.style={%
        plot graphics/lowlevel draw/.code 2 args={%
            \pgfuseshading{...} %  some advanced basic level shading operations
        },%
    },%
}
```

Attention: `colorbar horizontal` *re*-defines `every colorbar`. That means any user customization must take place *after* `colorbar horizontal`:

```
%  correct:
\begin{axis}[colorbar horizontal, colorbar style={<some customization>}]
%  wrong, colorbar horizontal resets the customization:
\begin{axis}[colorbar style={<some customization>}, colorbar horizontal]
```

`/pgfplots/every colorbar` (style, no value)

This style governs the placement, alignment and appearance of color bars. Any desired detail changes for color bars can be put into this style. Additionally, there is a style `colorbar shift` which is set after `every colorbar`. The latter style is intended to contain only shift transformations like `xshift` or `yshift` (making it easier to overwrite or deactivate them).

While a color bar is drawn, the predefined node `parent axis` can be used to align at the parent axis.

Predefined node `parent axis`

A node for the parent axis of a color bar. It is only valid for color bars.

Thus,

```
\pgfplotsset{
    colorbar style={
        at={(parent axis.right of north east)},
        anchor=north west,
    },
    colorbar shift/.style={xshift=0.3cm}
}
```

places the colorbar in a way that its top left (north west) corner is aligned right of the top right corner (`right of north east`) of its parent axis. Combining this with the `colorbar shift` is actually the same as the initial setting.

Since color bars depend on some of its parent's properties, these properties are available as values of the following keys:

/pgfplots/point meta min (no value)
/pgfplots/point meta max (no value)

The values of these keys contain the lower and upper bound of the color map, i.e. the lower and upper limit for the color bar.

The value is \pgfkeysvalueof{/pgfplots/point meta min} inside of every colorbar.

The value is usually determined using the axis wide point meta limits, i.e. they are computed as minimum and maximum value over all plots (unless the user provided limits manually). Consider the colorbar source key if you'd like to select point meta limits of one specific \addplot command.

/pgfplots/colorbar source={⟨*true*,*false*⟩} (initially `false`)

Allows to select a specific \addplot command whose point meta limits are taken as upper and lower limit of a colorbar's data range. This affects the tick descriptions of the colorbar. It needs to be provided as argument to \addplot, i.e. using

```
\addplot[...,colorbar source] ...
%  or
\addplot+[colorbar source] ...
```

or as key inside of a cycle list.

Using colorbar source automatically implies point meta rel=per plot for that specific plot.

If there are more than one \addplot commands with colorbar source, the last one is selected.

/pgfplots/parent axis width (no value)
/pgfplots/parent axis height (no value)

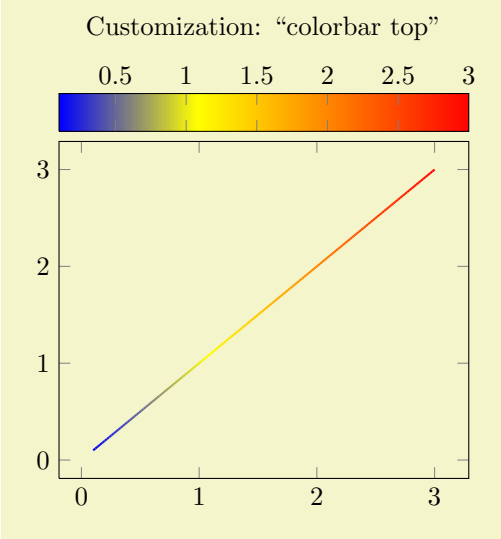
The values of these keys contain the size of the parent axis. They can be used as width and/or height arguments for every colorbar with \pgfkeysvalueof{/pgfplots/parent axis width}.

These values are only valid inside of color bars.

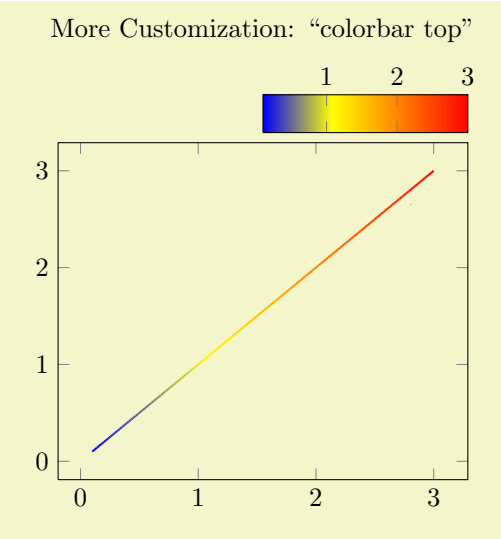
Besides these values, each color bar inherits a list of styles of its parent axis, namely

- every tick,
- every minor tick,
- every major tick,
- every axis grid,
- every minor grid,
- every major grid,
- every tick label.

This can be used to inherit line width and/or fonts.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    colorbar horizontal,
    colorbar style={
      at={(0.5,1.03)},anchor=south,
      xticklabel pos=upper
    },
    title style={yshift=1cm},
    title=Customization: ‘‘colorbar top’’]

    \addplot[mesh,thick,samples=150,domain=0.1:3]
        {x};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    colorbar horizontal,
    colorbar style={
      at={(1,1.03)},anchor=south east,
      width=0.5*
        \pgfkeysvalueof{/pgfplots/parent axis width},
      xticklabel pos=upper,
    },
    title style={yshift=1cm},
    title=More Customization: ‘‘colorbar top’’]

    \addplot[mesh,thick,samples=150,domain=0.1:3]
        {x};
\end{axis}
\end{tikzpicture}
```

Please take a look at the predefined styles `colorbar right`, `colorbar left` and `colorbar horizontal` for more details about configuration possibilities for `every colorbar`.

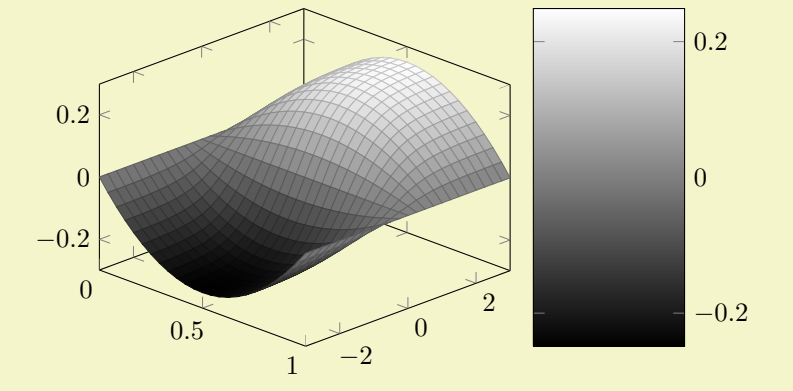
Remark: A color bar is just a normal axis. That means `every colorbar` can contain specifications where to place tick labels, extra ticks, scalings and most other features of a normal axis as well (except nested color bars).

`/pgfplots/colorbar style={`⟨*key-value list*⟩`}`

A shortcut for `every colorbar/.append style={`⟨*key-value list*⟩`}`. It appends options to the colorbar style.

`/pgfplots/colorbar/width={`⟨*dimension*⟩`}` (initially `0.5cm`)

Sets the width of a color bar.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        view/az=45,
        colorbar,
        colorbar/width=2cm,
        colormap/blackwhite]

    \addplot3[surf,domain=0:1,y domain=-3:3] {x*(1-x)*tanh(y)};
    \end{axis}
\end{tikzpicture}
```

For horizontal color bars, this sets the height.

/pgfplots/colorbar shift (style, no value)

This style is installed after every colorbar. It is intended to contain only shift transformations like xshift and/or yshift. The reason to provide two separate styles is to allow easier deactivation of shift transformations.

```
\pgfplotsset{
    colorbar shift/.style={xshift=1cm}
}
```

Predefined node current colorbar axis

A predefined node for the color bar of an axis. After \end{axis}, this node can be used to align further graphical elements at the color bar. Note that current axis refers to the axis as such while current colorbar axis refers to the color bar (which is an axis itself).

/pgfplots/colorbar/draw/.code={⟨...⟩}

This code key belongs to the low level interface of color bars. It is invoked whenever a color bar needs to be drawn. Usually, it won't be necessary to use or modify this key explicitly.

When this key is invoked, the styles inherited from the parent axis are already set and the required variables (see the documentation of every colorbar) are initialized.

This code key can be replaced if one needs more than one color bar (or other wrinkles).

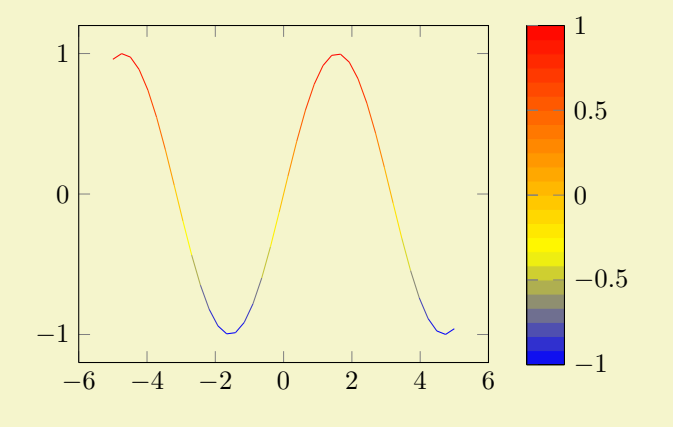
The initial configuration is

```
\pgfplotsset{colorbar/draw/.code={%
    \axis[every colorbar,colorbar shift,colorbar=false]
    \addplot graphics {};
    \endaxis
    }
}
```

Please note that a color bar axis is nothing special as such – it is just a normal axis with one plot graphics command and it is invoked with a special set of options. The only special thing is that a set of styles and some variables are inherited from its parent axis.

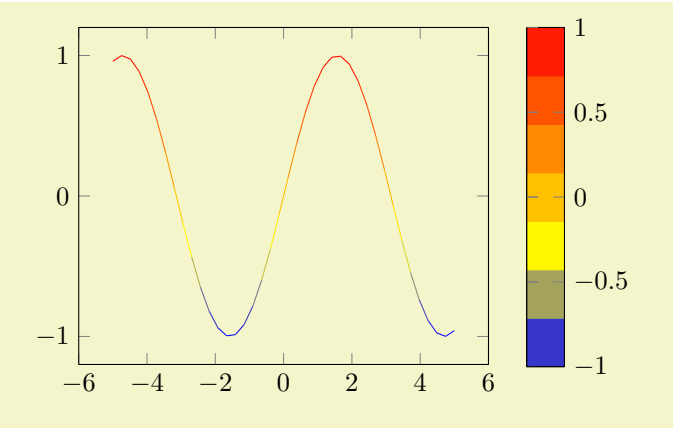
/pgfplots/colorbar sampled={⟨*optional options*⟩} (style, default surf,mark=none,shader=flat)

A style which installs a discretely sampled color bar.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar sampled]
    \addplot[mesh,samples=40] {sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

The style uses `\addplot3[`⟨*options*⟩`]` to draw the `colorbar`, with `domain` set to the color range and the current value of the `samples` key to determine the number of samples. In other words: it uses `plot expression` and a surface plot to visualize the `colorbar`. Use `colorbar style={samples=10}` to change the number of samples.

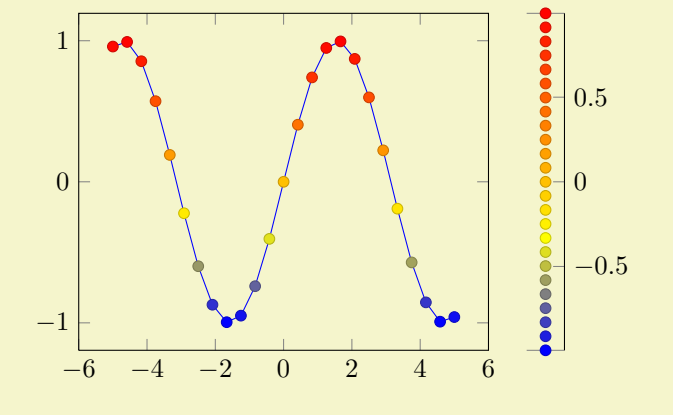



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar sampled,colorbar style={samples=8}]
    \addplot[mesh,samples=40] {sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

The ⟨*options*⟩ can be used to change the `\addplot3` options used for the colorbar visualization. For example, `colorbar sampled={surf,shader=interp}` will use Gouraud shading which has visually the same effect as the standard color bar.

`/pgfplots/colorbar sampled line={`⟨*optional options*⟩`}` (style, default `scatter,only marks`)

A style which draws a discrete colorbar. In contrast to `colorbar sampled`, it visualizes the `colorbar` using a line plot, not a `surf` plot.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[colorbar sampled line]
    \addplot+[scatter] {sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

The initial configuration uses a `scatter` plot to visualize the `colorbar`, it can be changed by specifying ⟨*options*⟩.

Furthermore, the axis appearance is changed using `axis y line*=left|right`, depending on the position of the color bar (or `axis x line*=bottom` for `colorbar horizontal`).

Consider the `tick align=outside` feature if you prefer tick lines outside of the colorbar instead of inside.

`/pgfplots/every colorbar sampled line` (style, no value)

A style which is used by `colorbar sampled line` to change the color of the line without ticks.

It is initially set to `help lines`.

4.8.13 Color Bars Outside Of an Axis

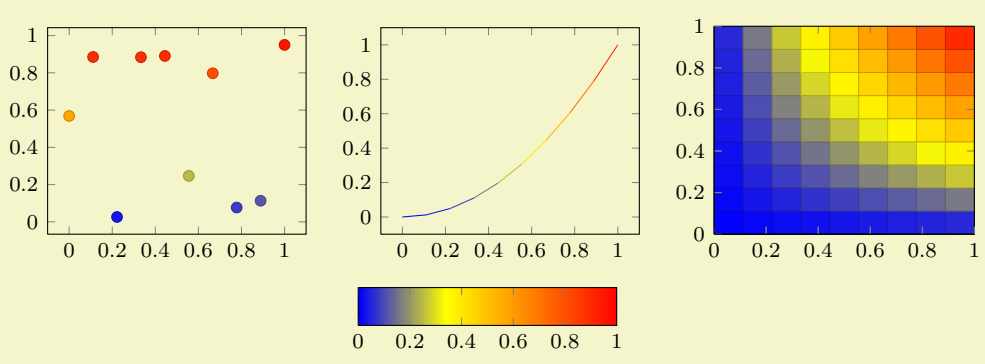
Occasionally, one has multiple adjacent plots, each with the same `colormap` and the same `point meta min` and `point meta max` values and we'd like to show a *single* `colorbar`. PGFPLOTS supports the `colorbar to name` feature which is similar to the related method for legends, `legend to name`:

`/pgfplots/colorbar to name={`⟨*name*⟩`}` (initially `empty`)

Enables to detach a `colorbar` from its parent axis: instead of drawing the `colorbar`, a self–contained, independent set of drawing commands will be stored using the label ⟨*name*⟩. The label is defined using `\label{`⟨*name*⟩`}`, just as for any other LaTeX label. The name can be referenced using

`\ref{`⟨*name*⟩`}`.

Thus, typing `\ref{`⟨*name*⟩`}` somewhere outside of the axis, maybe even outside of any picture, will cause the `colorbar` to be drawn.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{footnotesize,samples=10, domain=0:1,point meta min=0, point meta max=1}
\begin{center}%  note that \centering uses less vspace...
\begin{tikzpicture}
    \begin{axis}[colorbar,colorbar horizontal,colorbar to name={storedcolorbar}]
    \addplot[scatter,only marks,mark=*] {rnd};
    \end{axis}
\end{tikzpicture}
%
\begin{tikzpicture}
    \begin{axis}
    \addplot+[domain=0:1,mark=none,mesh] {x^2};
    \end{axis}
\end{tikzpicture}
%
\begin{tikzpicture}
    \begin{axis}[view={0}{90}]
    \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
\\

\ref{storedcolorbar}
\end{center}
```

The feature works in the same way as described for `legend to name`, please refer to its description on page 165 for the details. We only summarize the differences here.

`\pgfplotscolorbarfromname{`⟨*name*⟩`}`

This command poses an equivalent alternative for `\ref{`⟨*name*⟩`}`: it has essentially the same effect, but it does not create links when used with the `hyperref` package.

`/pgfplots/every colorbar to name picture` (style, no value)

A style which is installed when `\ref` is used outside of a picture: a new picture will be created with `\tikz[/pgfplots/every colorbar to name picture]`.

See also the `every legend to name picture` style.

4.8.14 Scaling Descriptions: Predefined Styles

It is reasonable to change font sizes, marker sizes etc. together with the overall plot size: Large plots should also have larger fonts and small plots should have small fonts and a smaller distance between ticks.

`/tikz/font=`\normalfont|\small|\tiny|...
`/pgfplots/max space between ticks={`⟨*integer*⟩`}`
`/pgfplots/try min ticks={`⟨*integer*⟩`}`
`/tikz/mark size={`⟨*integer*⟩`}`

These keys should be adjusted to the figure's dimensions. Use

```
\pgfplotsset{tick label style={font=\footnotesize},
    label style={font=\small},
    legend style={font=\small}
}
```

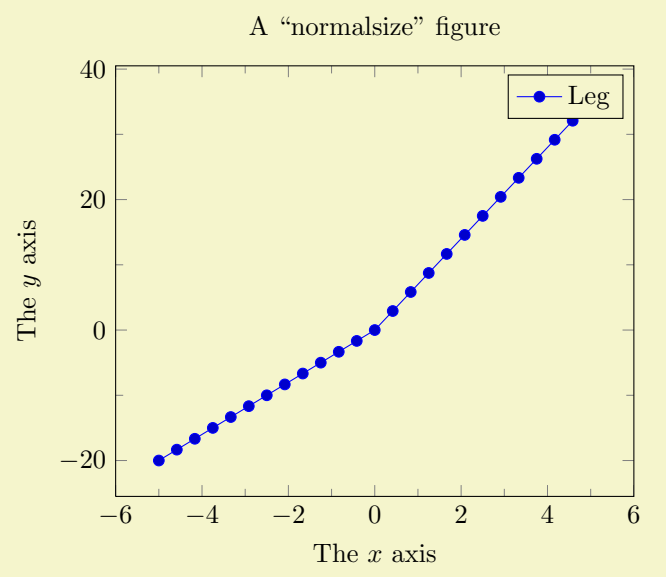
to provide different fonts for different descriptions.

The keys `max space between ticks` and `try min ticks` are described on page 234 and configure the approximate distance and number of successive tick labels (in `pt`). Please omit the `pt` suffix here.

There are a couple of predefined scaling styles which set some of these options:

`/pgfplots/normalsize` (style, no value)

Re-initialises the standard scaling options of PGFPLOTS.



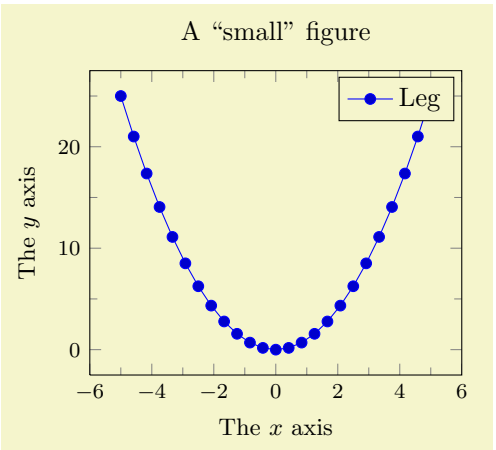

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[normalsize,
        title=A ‘‘normalsize’’ figure,
        xlabel=The $x$ axis,
        ylabel=The $y$ axis,
        minor tick num=1,
        legend entries={Leg}]
        \addplot {max(4*x,7*x)};
    \end{axis}
\end{tikzpicture}
```

The initial setting is

```
\pgfplotsset{
    normalsize/.style={
        /pgfplots/width=240pt,
        /pgfplots/height=207pt,
        /pgfplots/max space between ticks=35
    }
}
```

/pgfplots/small (style, no value)

Redefines several keys such that the axis is “smaller”.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[small,
        title=A ‘‘small’’ figure,
        xlabel=The $x$ axis,
        ylabel=The $y$ axis,
        minor tick num=1,
        legend entries={Leg}]
        \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

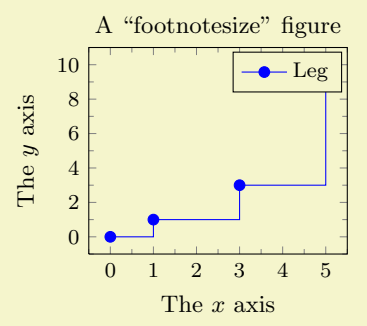
The initial setting is

```
\pgfplotsset{
    small/.style={
        width=6.5cm,
        height=,
        tick label style={font=\footnotesize},
        label style={font=\small},
        max space between ticks=25,
    }
}
```

Feel free to redefine the scaling – the option may still be useful to get more ticks without typing too much. You could, for example, set `small,width=6cm`.

/pgfplots/footnotesize (style, no value)

Redefines several keys such that the axis is even smaller. The tick labels will have `\footnotesize`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[footnotesize,
        title=A ‘‘footnotesize’’ figure,
        xlabel=The $x$ axis,
        ylabel=The $y$ axis,
        minor tick num=1,
        legend entries={Leg}]
        \addplot+[const plot]
            coordinates {
            (0,0) (1,1) (3,3) (5,10)
        };
    \end{axis}
\end{tikzpicture}
```

The initial setting is

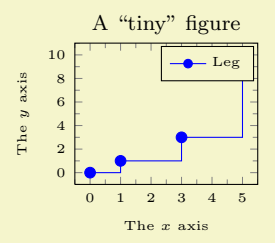
```
\pgfplotsset{
    footnotesize/.style={
        width=5cm,
        height=,
        legend style={font=\footnotesize},
        tick label style={font=\footnotesize},
        label style={font=\small},
        title style={font=\small},
        every axis title shift=0pt,
        max space between ticks=15,
        every mark/.append style={mark size=8},
        major tick length=0.1cm,
        minor tick length=0.066cm,
    },
}
```

As for `small`, it can be convenient to set `footnotesize` and set `width` afterwards.

You will need `compat=1.3` or newer for this to work.

/pgfplots/tiny (style, no value)

Redefines several keys such that the axis is very small. Most descriptions will have `\tiny` as fontsize.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[tiny,
        title=A ‘‘tiny’’ figure,
        xlabel=The $x$ axis,
        ylabel=The $y$ axis,
        minor tick num=1,
        legend entries={Leg}]
        \addplot+[const plot]
            coordinates {
            (0,0) (1,1) (3,3) (5,10)
        };
    \end{axis}
\end{tikzpicture}
```

The initial setting is

```
\pgfplotsset{
    tiny/.style={
        width=4cm,
        height=,
        legend style={font=\tiny},
        tick label style={font=\tiny},
        label style={font=\tiny},
        title style={font=\footnotesize},
        every axis title shift=0pt,
        max space between ticks=12,
        every mark/.append style={mark size=6},
        major tick length=0.1cm,
        minor tick length=0.066cm,
        every legend image post/.append style={scale=0.8},
    },
}
```

As for small, it can be convenient to use tiny,width=4.5cm to adjust the width.

You will need compat=1.3 or newer for this to work.

4.9 Scaling Options

/pgfplots/width={⟨*dimen*⟩}

Sets the width of the final picture to {⟨*dimen*⟩}. If no height is specified, scaling will respect aspect ratios.

Remarks:

- The scaling only affects the width of one unit in x-direction or the height for one unit in y-direction. Axis labels and tick labels won't be resized, but their size is used to determine the axis scaling.
- You can use the scale={⟨*number*⟩} option,

  ```
  \begin{tikzpicture}[scale=2]
  \begin{axis}
  ...
  \end{axis}
  \end{tikzpicture}
  ```

  to scale the complete picture.
- The Ti*k*Z-options x and y which set the unit dimensions in x and y directions can be specified as arguments to \begin{axis}[x=1.5cm,y=2cm] if needed (see below). These settings override the width and height options.
- You can also force a fixed width/height of the axis (without looking at labels) with

  ```
  \begin{tikzpicture}
  \begin{axis}[width=5cm,scale only axis]
      ...
  \end{axis}
  \end{tikzpicture}
  ```

- Please note that up to the writing of this manual, PGFPLOTS only estimates the size needed for axis- and tick labels. It does not include legends which have been placed outside of the axis[41]. This may be fixed in future versions.

  Use the x={⟨*dimension*⟩}, y={⟨*dimension*⟩} and scale only axis options if the scaling happens to be wrong.

/pgfplots/height={⟨*dimen*⟩}

See width.

/pgfplots/scale only axis=true|false (initially false)

If scale only axis is enabled, label, tick and legend dimensions won't influence the size of the axis rectangle, that means width and height apply only to the axis rectangle.

[41] I.e. the 'width' option will not work as expected, but the bounding box is still ok.

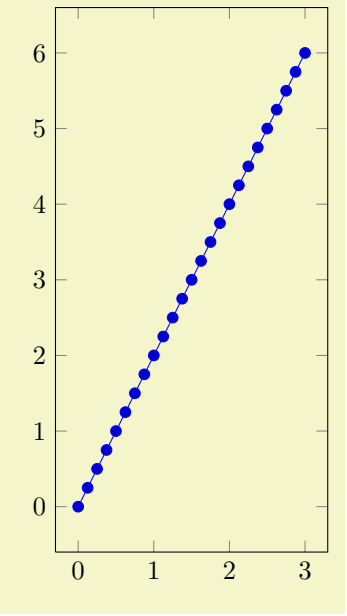
If `scale only axis=false` (the default), PGFPLOTS will try to produce the desired width *including* labels, titles and ticks.

`/pgfplots/x={`⟨*dimen*⟩`}`
`/pgfplots/y={`⟨*dimen*⟩`}`
`/pgfplots/z={`⟨*dimen*⟩`}`
`/pgfplots/x={(`⟨*x*⟩`,`⟨*y*⟩`)}`
`/pgfplots/y={(`⟨*x*⟩`,`⟨*y*⟩`)}`
`/pgfplots/z={(`⟨*x*⟩`,`⟨*y*⟩`)}`

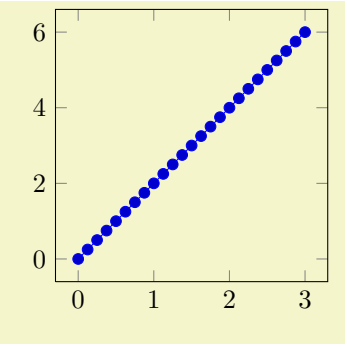
Sets the unit vectors for x (or y). Every logical plot coordinate (x, y) is drawn at the position

$$x \cdot \begin{bmatrix} e_{xx} \\ e_{xy} \end{bmatrix} + y \cdot \begin{bmatrix} e_{yx} \\ e_{yy} \end{bmatrix}.$$

The unit vectors e_x and e_y determine the paper position in the current (always two dimensional) image. The key `x={`⟨*dimen*⟩`}` simply sets $e_x = (\langle dimen \rangle, 0)^T$ while `y={`⟨*dimen*⟩`}` sets $e_y = (0, \langle dimen \rangle)^T$. Here, `{`⟨*dimen*⟩`}` is any TEX size like `1mm`, `2cm` or `5pt`. Note that you should not use negative values for ⟨*dimen*⟩ (consider using `x dir` and its variants to reverse axis directions).



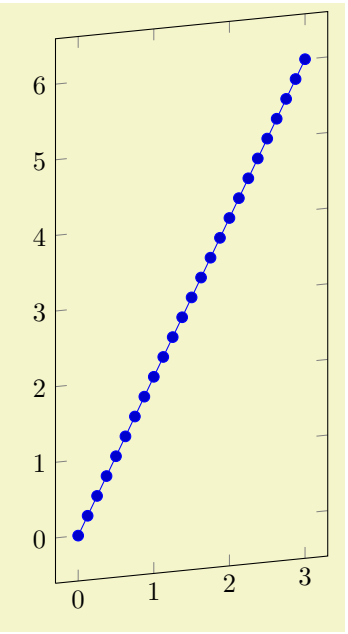

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[x=1cm,y=1cm]
\addplot expression[domain=0:3] {2*x};
\end{axis}
\end{tikzpicture}
```



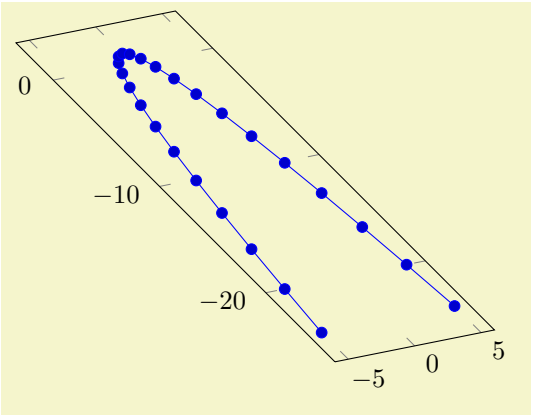

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[x=1cm,y=0.5cm,y dir=reverse]
\addplot expression[domain=0:3] {2*x};
\end{axis}
\end{tikzpicture}
```

The second syntax, `x={(`⟨*x*⟩`,`⟨*y*⟩`)}` sets $e_x = (\langle x \rangle, \langle y \rangle)^T$ explicitly[42]; the corresponding `y` key works similarly. This allows to define skewed or rotated axes.

[42] Please note that you need extra curly braces around the vector. Otherwise, the comma will be interpreted as separator for the next key-value pair.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[x={(1cm,0.1cm)},y=1cm]
\addplot expression[domain=0:3] {2*x};
\end{axis}
\end{tikzpicture}
```

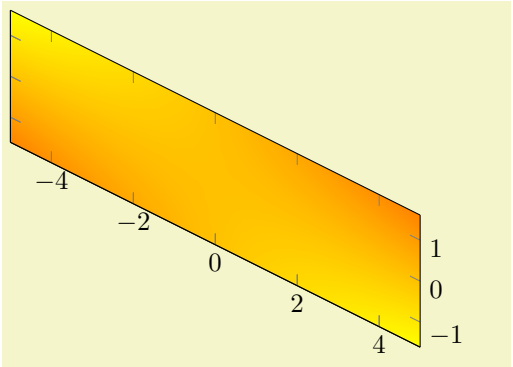



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
        x={(5pt,1pt)},
        y={(-4pt,4pt)}]
\addplot {1-x^2};
\end{axis}
\end{tikzpicture}
```

Setting x explicitly overrides the `width` option. Setting y explicitly overrides the `height` option. Setting `x` and/or `y` for logarithmic axis will set the dimension used for $1 \cdot e \approx 2.71828$.
Please note that it is *not* possible to specify `x` as argument to `tikzpicture`. The option

```
\begin{tikzpicture}[x=1.5cm]
\begin{axis}
    ...
\end{axis}
\end{tikzpicture}
```

won't have any effect because an axis rescales its coordinates (see the `width` option).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    x={(1cm,-0.5cm)},
    y=1cm,
    z=0cm,
    axis on top,
    scale mode=scale uniformly,
    ]
    \addplot3[surf,shader=interp] {x*y};
\end{axis}
\end{tikzpicture}
```

Explicit units for 3D axes: As of version 1.5, it is also possible to supply unit vectors to three-dimensional axes. In this case, the following extra assumptions need to be satisfied:

1. If you want to control three-dimensional units, you need to provide *all* of `x`, `y`, and `z` keys. For two–dimensional axes, it is also supported to supply just one of `x` or `y`.
2. Any provided three-dimensional unit vectors are assumed to form a *right–handed coordinate system*. In other words: take your right hand, let the thumb point into the `x` direction, the index finger in `y` direction and the middle finger in `z` direction. If that is impossible, the PGFPLOTS output will be wrong. The reason for this assumption is that PGFPLOTS needs to compute the view direction out of the provided units (see below).

   Consider using `x dir=reverse` or its variants in case you want to reverse directions.
3. For three-dimensional axes, PGFPLOTS computes a view direction out of the provided unit vectors. The view direction is required to realize the `z buffer` feature (i.e. to decide about depths)[43].

This feature is used to realize the `\addplot3 graphics` feature, compare the examples in Section 4.2.8 on page 43.

Limitations: Unfortunately, skewed axes are **not available for bar plots**.

`/pgfplots/scale mode=auto|none|stretch to fill|scale uniformly` (initially `auto`)

Specifies how `width`, `height`, and the three unit vector keys `x`, `y`, and `z` are combined to produce the final image.

The initial choice `auto` chooses one of the other possible choices depending on the actual context. More precisely, it is the same as `none` if at least one of `x`, `y`, or `z` is provided (meaning that these keys are the final unit size). If no unit is provided, it defaults to `stretch to fill`.

The choice `none` does not apply any rescaling at all. Use this if prescribed lengths of `x`, `y` (and perhaps `z`) should be used. In other words: it ignores `width` and `height`. In this case, you may want to set `x post scale` and its variants to rescale units manually.

The choice `stretch to fill` takes `x`, `y`, and `z` as they have been found (either from user input or from some automatic processing) and rescales the axis with two *separate* scales: one which results in the proper `width` and one which results in the proper `height`. As a consequence, the unit vectors are modified and distorted such that the final image fits into the prescribed dimensions. This is usually what one expects unless one provides unit directions explicitly.

The choice `scale uniformly` is similar – but it applies only *one* scaling factor to fit into the prescribed dimensions. This scaling factor will be the same for both, `width` and `height`. Naturally, this will result in unsatisfactory results because either the final width or the final height will not be met. Therefore, this choice will enlarge or shrink limits to get the desired dimensions. Thus, the unit vectors have exactly the same size *relations and angles* as they had before the scaling; only their magnitude is changed uniformly. But due to the given constraints, parts of the image may be empty or may no longer be visible. In case the outcome needs manual improvements, you can improve the visibility by modifying `width` and/or `height` (more freedom is not supported for this choice). Note that PGFPLOTS implements this rescaling **if and only if** $e_{yx} = 0$ (for two–dimensional axes) or $e_{zx} = 0$ (for three-dimensional axes). In other words: for two–dimensional axes, the `y` vector has to be parallel to the canvas y direction and for three-dimensional axes, the same holds for the `z` vector. The idea is to rescale only the vertical part of one unit vector and to change the limits of that respective axis simultaneously. The `scale uniformly` choice is used to realize the `\addplot3 graphics` feature, see the documentation in Section 4.2.8 on page 43 for its examples.

`/pgfplots/xmode=normal|linear|log` (initially `normal`)
`/pgfplots/ymode=normal|linear|log` (initially `normal`)
`/pgfplots/zmode=normal|linear|log` (initially `normal`)

Allows to choose between linear (=normal) or logarithmic axis scaling or logplots for each x, y, z-combination.

Logarithmic plots use the current setting of `log basis x` and its variants to determine the basis (default is e).

[43] PGFPLOTS provides a debug option called `view dir={⟨x⟩}{⟨y⟩}{⟨z⟩}` to override the view direction, should that ever be interesting.

/pgfplots/x dir=normal|reverse (initially normal)
/pgfplots/y dir=normal|reverse (initially normal)
/pgfplots/z dir=normal|reverse (initially normal)

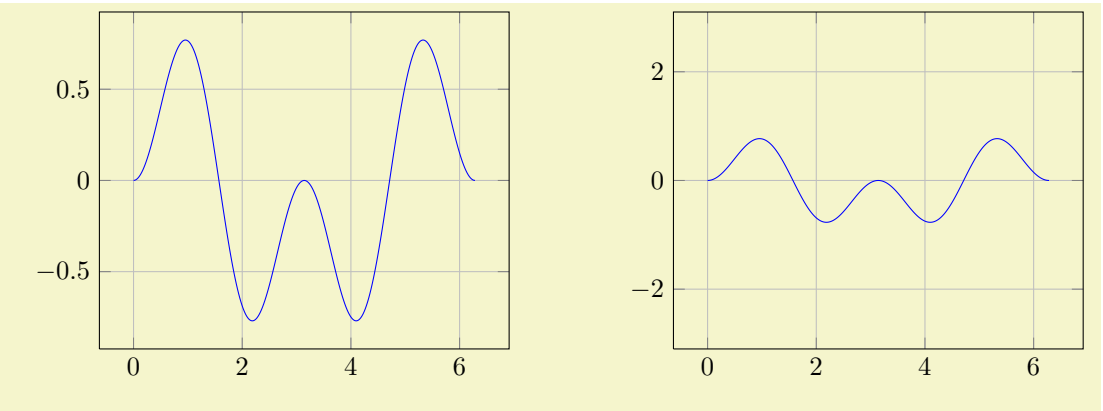
Allows to reverse axis directions such that values are given in decreasing order.

This key is documented in all detail on page 214.

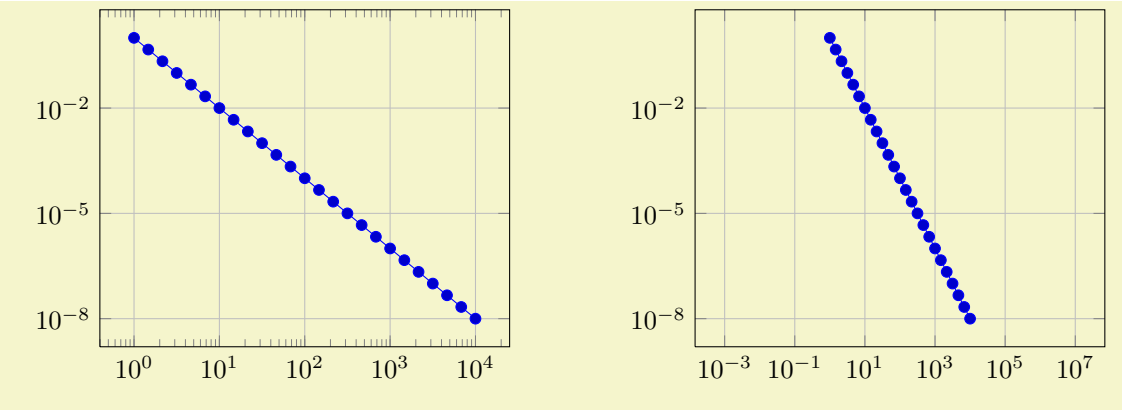
/pgfplots/axis equal={⟨*true*,*false*⟩} (initially false)

Each unit vector is set to the same length while the axis dimensions stay constant. Afterwards, the size ratios for each unit in x and y will be the same.

Axis limits will be enlarged to compensate for the scaling effect.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[axis equal=false,grid=major]
        \addplot[blue] expression[domain=0:2*pi,samples=300] {sin(deg(x))*sin(2*deg(x))};
    \end{axis}
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
    \begin{axis}[axis equal=true,grid=major]
        \addplot[blue] expression[domain=0:2*pi,samples=300] {sin(deg(x))*sin(2*deg(x))};
    \end{axis}
\end{tikzpicture}
```

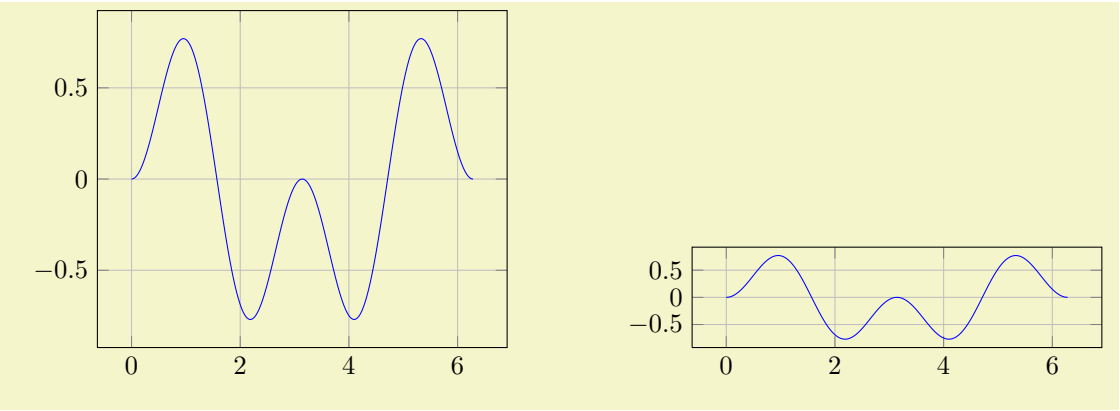



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[axis equal=false,grid=major]
        \addplot expression[domain=1:10000] {x^-2};
    \end{loglogaxis}
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
    \begin{loglogaxis}[axis equal=true,grid=major]
        \addplot expression[domain=1:10000] {x^-2};
    \end{loglogaxis}
\end{tikzpicture}
```

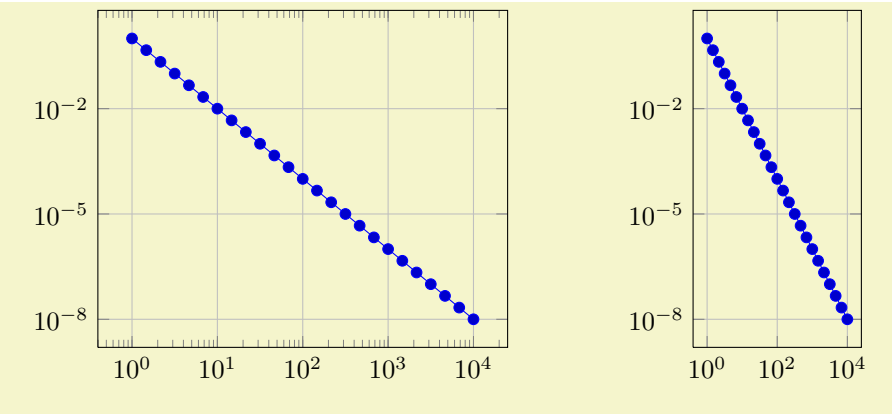
The configuration `axis equal=true` is actually just a style which sets `unit vector ratio=1 1 1,unit rescale keep size=true`.

`/pgfplots/axis equal image={`⟨*true,false*⟩`}` (initially `false`)

Similar to `axis equal`, but the axis limits will stay constant as well (leading to smaller images).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[axis equal image=false,grid=major]
        \addplot[blue] expression[domain=0:2*pi,samples=300] {sin(deg(x))*sin(2*deg(x))};
    \end{axis}
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
    \begin{axis}[axis equal image=true,grid=major]
        \addplot[blue] expression[domain=0:2*pi,samples=300] {sin(deg(x))*sin(2*deg(x))};
    \end{axis}
\end{tikzpicture}
```



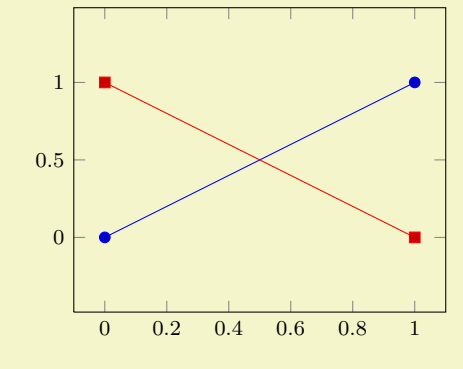

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[axis equal image=false,grid=major]
        \addplot expression[domain=1:10000] {x^-2};
    \end{loglogaxis}
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
    \begin{loglogaxis}[axis equal image=true,grid=major]
        \addplot expression[domain=1:10000] {x^-2};
    \end{loglogaxis}
\end{tikzpicture}
```

The configuration `axis equal image=true` is actually just a style which sets `unit vector ratio=1 1 1,unit rescale keep size=false`.

`/pgfplots/unit vector ratio={`⟨*rx ry rz*⟩`}` (initially `empty`)

Allows to provide custom unit vector ratios.

The key allows to tell PGFPLOTS that, for example, one unit in x direction should be twice as long as one unit in y direction:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[unit vector ratio=2 1,small]
        \addplot coordinates {(0,0) (1,1)};
        \addplot table[row sep=\\,col sep=&] {
            x & y \\
            0 & 1 \\
            1 & 0 \\
        };
    \end{axis}
\end{tikzpicture}
```

Providing `unit vector ratio=2 1` means that $\frac{e_x}{e_y} = 2$ where each coordinate (x, y) is placed at $xe_x + ye_y \in \mathbb{R}^2$ (see the documentation for `x` and `y` options). Note that `axis equal` is nothing but `unit vector ratio=1 1 1`.

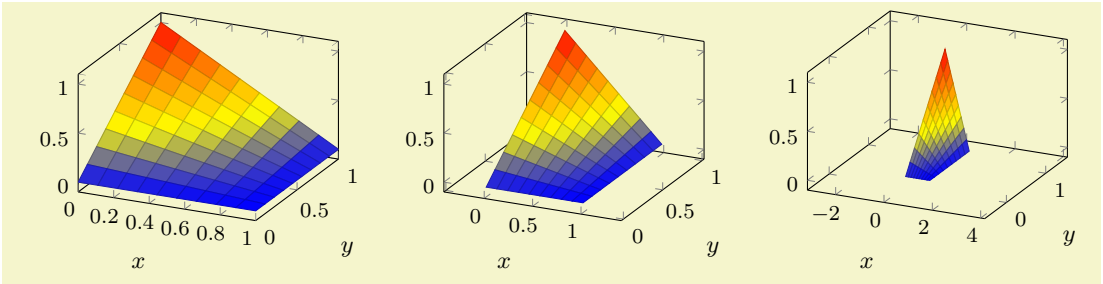
The arguments $\langle rx \rangle$, $\langle ry \rangle$, and $\langle rz \rangle$ are ratios for x, y and z vectors, respectively. For two–dimensional axes, only $\langle rx \rangle$ and $\langle ry \rangle$ are considered; they are provided relative to the y axis. In other words: the x unit vector will be $\langle rx \rangle$ / $\langle ry \rangle$ times longer than the y unit vector. For three-dimensional axes, all three arguments can be provided; they are interpreted relative to the z unit vector. Thus, a three dimensional axis with `unit vector ratio=1 2 4` will have an x unit which is $1/4$ the length of the z unit, and a y unit which is $2/4$ the length of the z unit.

Trailing values of `1` can be omitted, i.e. `unit vector ratio=2 1` is the same as `unit vector ratio=2`; and `unit vector ratio=3 2 1` is the same as `unit vector ratio=3 2`. An empty value `unit vector ratio={}` disables unit vector rescaling.

`/pgfplots/unit vector ratio*={`$\langle rx\ ry\ rz \rangle$`}`
`/pgfplots/unit rescale keep size={`$\langle true,false \rangle$`}` (initially `true`)

In the default configuration, PGFPLOTS maintains the original axis dimensions even though `unit vector ratio` involves different scalings.

It does so by enlarging the limits.

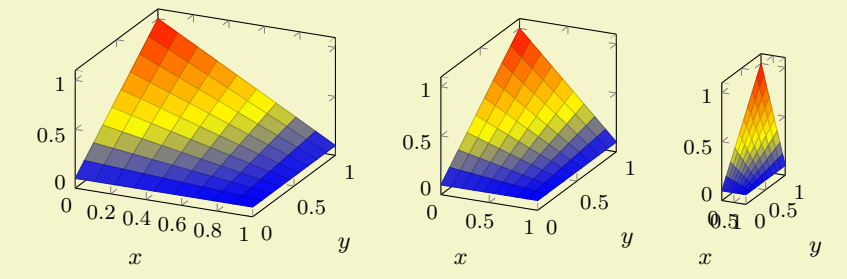



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[footnotesize,xlabel=$x$,ylabel=$y$,unit vector ratio=]
    \addplot3[surf,samples=10,domain=0:1] {(1-x)*y};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[footnotesize,xlabel=$x$,ylabel=$y$,unit vector ratio=1 1 1]
    \addplot3[surf,samples=10,domain=0:1] {(1-x)*y};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[footnotesize,xlabel=$x$,ylabel=$y$,unit vector ratio=0.25 0.5]
    \addplot3[surf,samples=10,domain=0:1] {(1-x)*y};
    \end{axis}
\end{tikzpicture}
```

The example above has the same plot, with three different unit ratios. The first has no limitations

(it is the default configuration). The second uses the same length for each unit vector and enlarges the limits in order to maintain the same dimensions. The third example has an x unit which is $1/4$ the length of a z unit, and an y unit which is $1/2$ the length of a z unit.

The `unit rescale keep size=false` key, or, equivalently, `unit vector ratio*=...`, does not enlarge limits:



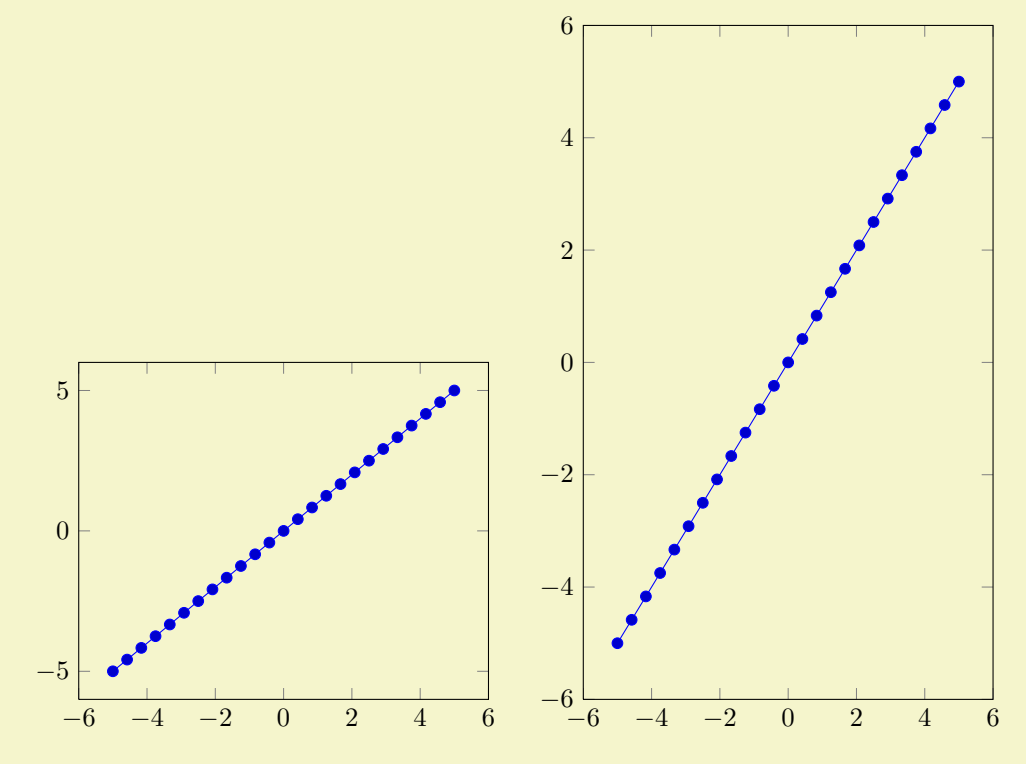

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[footnotesize,xlabel=$x$,ylabel=$y$,unit vector ratio=]
    \addplot3[surf,samples=10,domain=0:1] {(1-x)*y};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[footnotesize,xlabel=$x$,ylabel=$y$,
        unit rescale keep size=false,
        unit vector ratio=1 1 1]
    \addplot3[surf,samples=10,domain=0:1] {(1-x)*y};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[footnotesize,xlabel=$x$,ylabel=$y$,
        unit vector ratio*=0.25 0.5, %  the '*' implies 'unit rescale keep size=false'
    ]
    \addplot3[surf,samples=10,domain=0:1] {(1-x)*y};
    \end{axis}
\end{tikzpicture}
```

`/pgfplots/x post scale={`⟨*scale*⟩`}` (initially `empty`)
`/pgfplots/y post scale={`⟨*scale*⟩`}` (initially `empty`)
`/pgfplots/z post scale={`⟨*scale*⟩`}` (initially `empty`)
`/pgfplots/scale={`⟨*scale*⟩`}` (initially `empty`)

Lets PGFPLOTS compute the axis scaling based on `width`, `height`, `view`, `plot box ratio`, `axis equal` or explicit unit vectors with `x`, `y`, `z` and *rescales* the resulting vector(s) according to ⟨*scale*⟩.

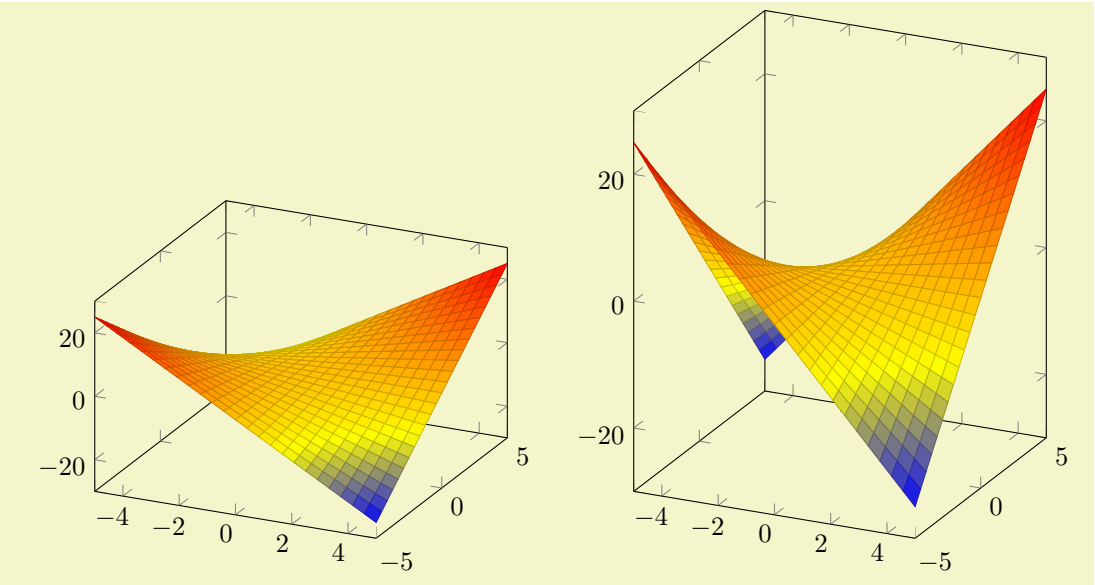
The `scale` key sets all three keys to the same ⟨*uniform scale*⟩ value. This is effectively the same as if you rescale the complete axis (without changing sizes of descriptions).

The other keys allow individually rescaled axes.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[y post scale=1]
    \addplot {x};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[y post scale=2]
    \addplot {x};
    \end{axis}
\end{tikzpicture}
```

Thus, the axis becomes *larger*. This overrules any previous scaling.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[z post scale=1]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
\begin{tikzpicture}
    \begin{axis}[z post scale=2]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```

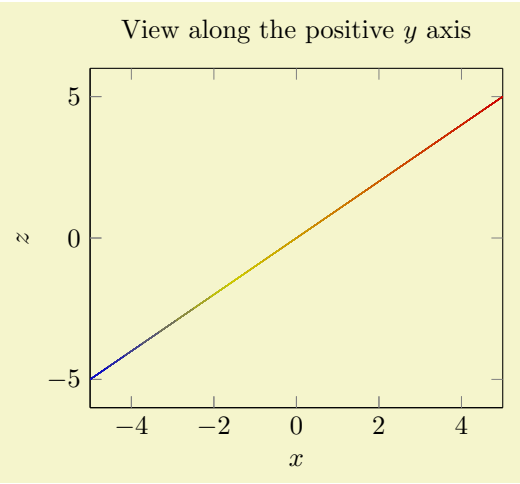
4.10 3D Axis Configuration

This section described keys which are used to configure the appearance of three dimensional figures. Some of them apply for two–dimensional plots as special case as well, and they will also be discussed in the respective sections of this manual.

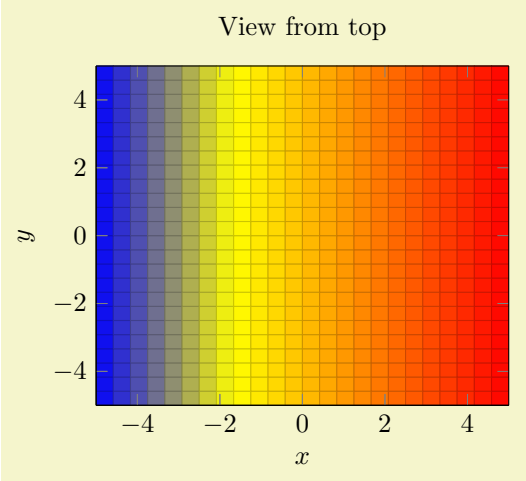
4.10.1 View Configuration

/pgfplots/view={⟨*azimuth*⟩}{⟨*elevation*⟩} (initially {25}{30})

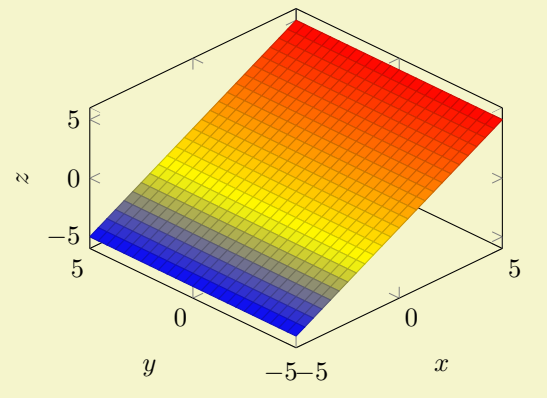
Changes both view angles of a 3D axis. The azimuth (first argument) is the horizontal angle which is rotated around the z axis. For a 3D plot, the z axis always points to the top. The elevation (second argument) is the vertical rotation around the (rotated) x axis. Positive elevation values indicate a view from above, negative a view from below. All values are measured in degree.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view={0}{0},
        xlabel=$x$,
        zlabel=$z$,
        title=View along the positive $y$ axis]
        \addplot3[surf] {x};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view={0}{90},
        xlabel=$x$,
        ylabel=$y$,
        title=View from top]
        \addplot3[surf] {x};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view={-45}{45},
        xlabel=$x$,ylabel=$y$,zlabel=$z$]
        \addplot3[surf] {x};
    \end{axis}
\end{tikzpicture}
```

The `view` is computed as follows. The view is defined by two rotations: the first rotation uses the ⟨*azimuth*⟩ angle to rotate around the z axis. Afterwards, the view is rotated ⟨*elevation*⟩ degrees around the *rotated* x axis (more precisely, it is rotated $-\langle elevation\rangle$ degrees). The resulting transformed x–z–plane is the viewport, i.e. the view direction is always the transformed positive y axis.

The `view` argument is compatible with the argument of the Matlab (®) view command, i.e. you can use

```
[h,v] = view
```

in matlab and pack the resulting arguments into PGFPLOTS[44].

If you work with `gnuplot`, you can convert the view arguments as follows: the `gnuplot` command

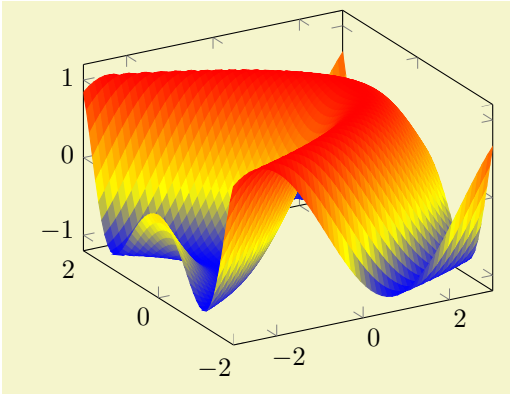
```
set view v,h
```

is *equivalent* to `view={h}{90-v}`. For example, the default `gnuplot` configuration `set view 60,60` is equivalent to `view={60}{30}` in PGFPLOTS.

The `view` is (currently) always an orthogonal projection, no perspective is possible, yet.

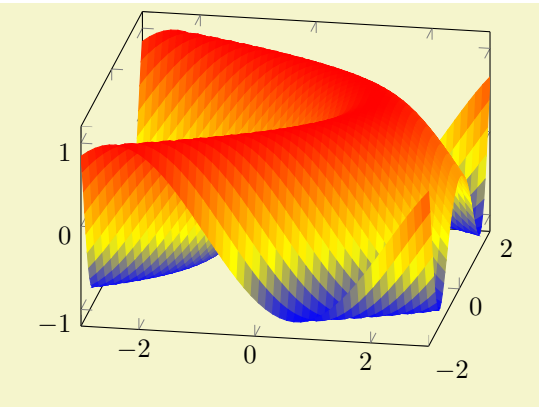
/pgfplots/view/az={⟨*azimuth*⟩}
/pgfplots/view/h={⟨*azimuth*⟩} (initially 25)

Changes only the azimuth view angle, i.e. the horizontal (first) view angle which is rotated around the z axis.

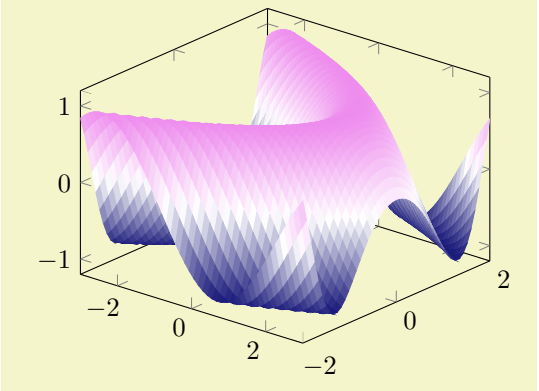



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view/h=-30]
    \addplot3[
        surf,
        % shader=interp,
        shader=flat,
        samples=50,
        domain=-3:3,y domain=-2:2]
        {sin(deg(x+y^2))};
    \end{axis}
\end{tikzpicture}
```

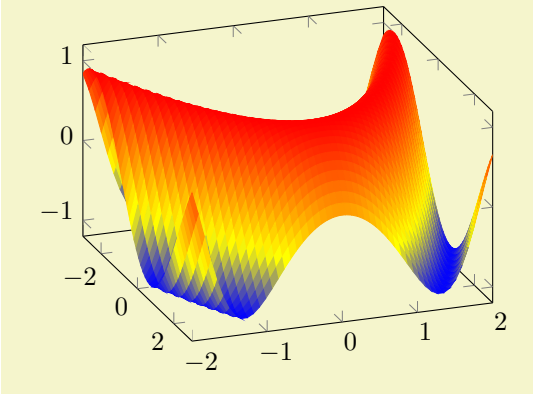
[44] In case it does not work, try `h` and `-v` in PGFPLOTS.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view/h=10]
    \addplot3[
        surf,
        % shader=interp,
        shader=flat,
        samples=50,
        domain=-3:3,y domain=-2:2]
        {sin(deg(x+y^2))};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view/h=40,colormap/violet]
    \addplot3[
        surf,
        % shader=interp,
        shader=flat,
        samples=50,
        domain=-3:3,y domain=-2:2]
        {sin(deg(x+y^2))};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view/h=70]
    \addplot3[
        surf,
        % shader=interp,
        shader=flat,
        samples=50,
        domain=-3:3,y domain=-2:2]
        {sin(deg(x+y^2))};
    \end{axis}
\end{tikzpicture}
```

/pgfplots/view/el={⟨*elevation*⟩}
/pgfplots/view/v={⟨*elevation*⟩} (initially 30)

Changes only the vertical elevation, i.e. the second argument to `view`. Positive values view from above, negative values from below.

4.10.2 Styles Used Only For 3D Axes

/pgfplots/every 3d description (style, no value)

This style allows to change the appearance of *descriptions* for three dimensional axes. Naturally, a three dimensional axis will display axis labels for x and y differently than a two dimensional axis (for example, the y axis label won't be rotated by 90 degrees). The `every 3d description` style installs the necessary display options for three dimensional axis descriptions.

The initial value is:

```
\pgfkeys{
    /pgfplots/every 3d description/.style={
        %  Only these description styles can be changed here:
        every axis x label/.style={at={(ticklabel cs:0.5)},
            anchor=near ticklabel},
        every axis y label/.style={at={(ticklabel cs:0.5)},
            anchor=near ticklabel},
        every x tick scale label/.style={
            at={(xticklabel cs:0.95,5pt)},
            anchor=near xticklabel,inner sep=0pt},
        every y tick scale label/.style={
            at={(yticklabel cs:0.95,5pt)},
            anchor=near yticklabel,inner sep=0pt},
        try min ticks=3,
    }%
}
```

As the name suggests, `every 3d description` can only be used to set styles for axis labels, tick labels and titles. It has *not* been designed to reset other styles, you will need to change these options either for each axis separately or by means of user defined styles. The reason for this limitation is: other options can (and, in many cases, needs to) be set before the axis is processed. However, the decision whether we have a two dimensional or a three dimensional axis has to be postponed until the processing is more or less complete – so only some remaining keys can be set.

`/pgfplots/every 3d view {⟨h⟩}{⟨v⟩}` (style, no value)

A style which can be used for fine-tuning of the output for specific views.

This style will be installed right after `every 3d description`, but before other axis description related keys are set (in other words: it has higher precedence than `every 3d description`, but lower precedence than keys provided to the axis directly).

One example is preconfigured for `view={0}{90}` (from top):

```
\pgfplotsset{
    /pgfplots/every 3d view {0}{90}/.style={
        xlabel near ticks,
        ylabel near ticks,
        axis on top=true
    }
}
```

4.10.3 Appearance Of The 3D Box

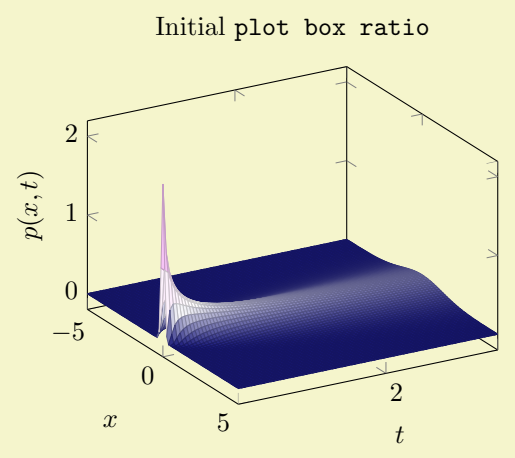
`/pgfplots/plot box ratio={⟨⟨x stretch⟩ ⟨y stretch⟩ ⟨z stretch⟩⟩}` (initially `1 1 1`)

Allows to customize the aspect ratio between the three different axes in a three dimensional plot.

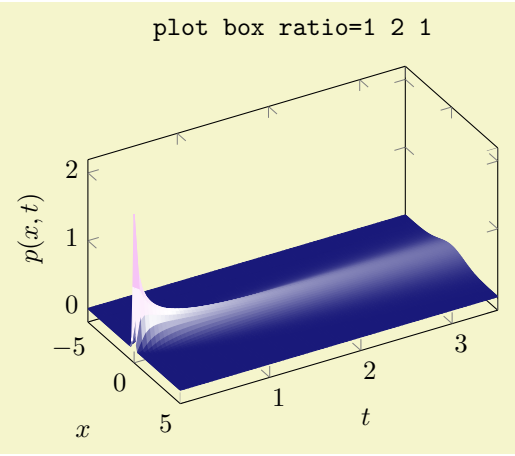
Note that this key is different from the related `unit vector ratio`: the plot box is only useful for three dimensional axes, and it will usually distort the unit vector ratios. If you want equal unit ratios, consider using `unit vector ratio`.

The `plot box ratio` is applied before any rotations and stretch–to–fill routines have been invoked. Thus, the initial setting[45] `1 1 1` makes all axes equally long before the stretch–to–fill routine is applied.

[45] Note that you can also use the syntax `{1}{1}{1}` instead of space-separation.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    view/h=60,
    plot box ratio=1 1 1,
    colormap={violet}{[1cm] rgb255(0cm)=(25,25,122)
        color(1cm)=(white) rgb255(5cm)=(238,140,238)},
    xlabel=$x$,
    ylabel=$t$,
    zlabel={$p(x,t)$},
    shader=faceted,
    title=Initial \texttt{plot box ratio},
]
    \addplot3[surf,y domain=0.02:3.5,samples=81]
        {1/(2*sqrt(pi*y)) * exp(0-x^2/y)};
    %  the '0' is a work-around for a bug in PGF 2.00
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    view/h=60,
    plot box ratio=1 2 1,
    colormap={violet}{[1cm] rgb255(0cm)=(25,25,122)
        color(1cm)=(white) rgb255(5cm)=(238,140,238)},
    xlabel=$x$,
    ylabel=$t$,
    zlabel={$p(x,t)$},
    shader=flat,
    title=\texttt{plot box ratio=1 2 1},
]
    \addplot3[surf,y domain=0.02:3.5,samples=81]
        {1/(2*sqrt(pi*y)) * exp(0-x^2/y)};
    %  the '0' is a work-around for a bug in PGF 2.00
\end{axis}
\end{tikzpicture}
```

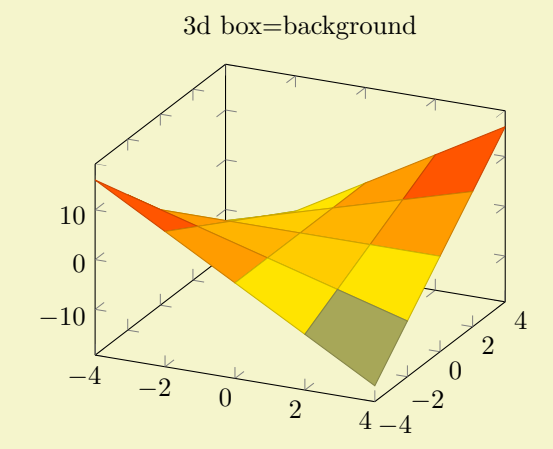
This key applies only to three dimensional axes. After the scaling, the axes will be stretched to fill the `width` and `height` for this plot. Thus, the effects of `plot box ratio` might be undone by this stretching for particular views.

`/pgfplots/3d box=background|complete|complete*` (initially `background`)

Allows to configure the appearance of boxed three dimensional axes.

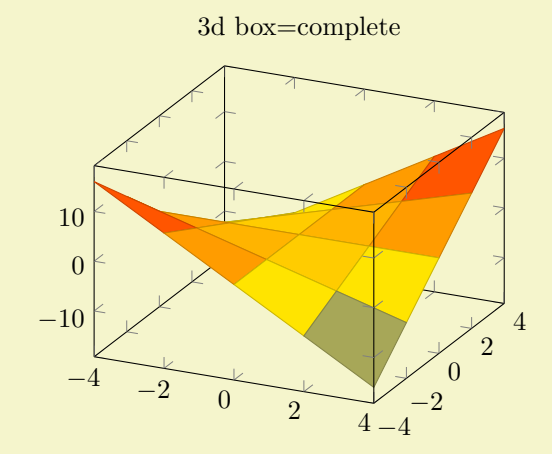
Type only `3d box` (without value) as alias for `3d box=complete`.

The choice `background` is the initial setting, it does not draw axis lines (and grid lines) which are in the foreground.



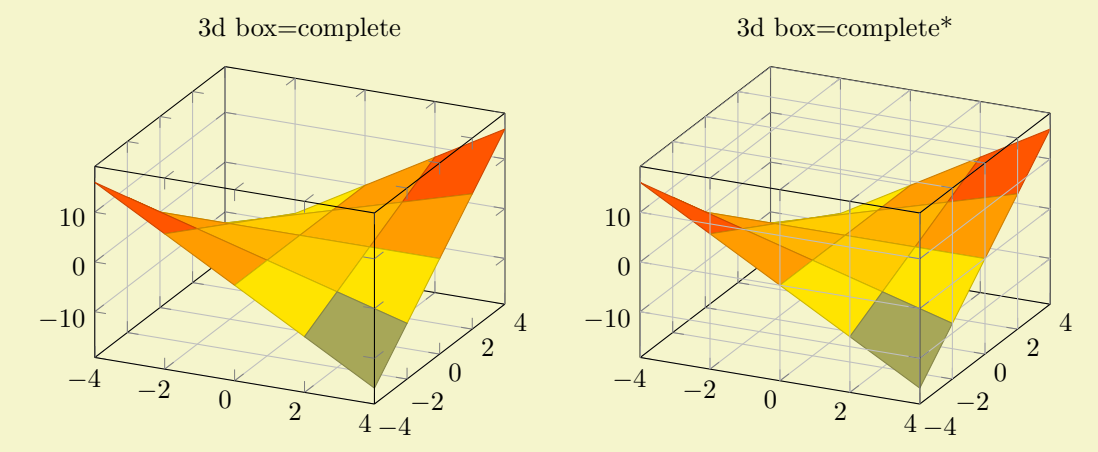

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        3d box=background,
        %  pretty printing, but irrelevant:
        title={3d box=background},
        samples=5,
        domain=-4:4,
        xtick=data,
        ytick=data,
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```

The choice `complete` also draws axis lines and tick lines in the foreground, but it doesn't draw grid lines in the foreground. The result yields a complete box:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        3d box,%  same as 3d box=complete
        %  pretty printing, but irrelevant:
        title={3d box=complete},
        samples=5,
        domain=-4:4,
        xtick=data,
        ytick=data,
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```

Finally, the choice `complete*` is the same as `complete`, but it also draws grid lines in the foreground.



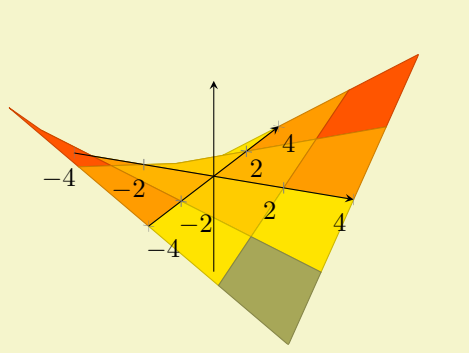

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        3d box=complete,
        grid=major,
        title={3d box=complete},
        samples=5, domain=-4:4,
        xtick=data, ytick=data,
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}%
~
\begin{tikzpicture}
    \begin{axis}[
        3d box=complete*,
        grid=major,
        title={3d box=complete*},
        samples=5, domain=-4:4,
        xtick=data, ytick=data,
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```

Before any foreground parts are actually processed, the style `every 3d box foreground` will be installed. This allows to change the appearance of foreground axis components like `tick style` or `axis line style` separately from the background components.

Note that `3d box=complete` is *only* available for boxed axes, i.e. together with `axis lines=box`. It is an error to use a different combination.

4.10.4 Axis Line Variants

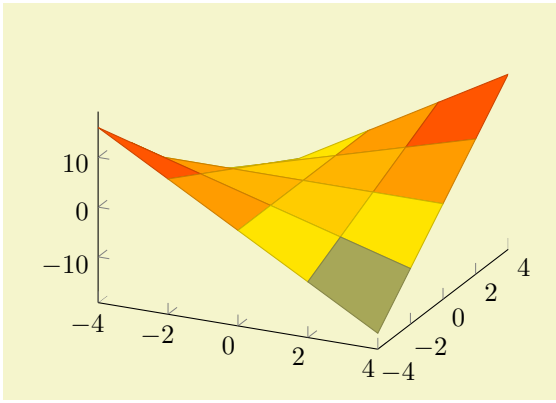
Three dimensional axes also benefit from the `axis lines=box` or `axis lines=center` styles discussed in Section 4.8.9. The choice `axis lines=box` is standard, it draws a box (probably affected by the `3d box=complete` key). The choice `axis lines=center` draws all three axes such that they pass through the origin. It might be necessary to combine this key with `axis on top` as there is no depth information.



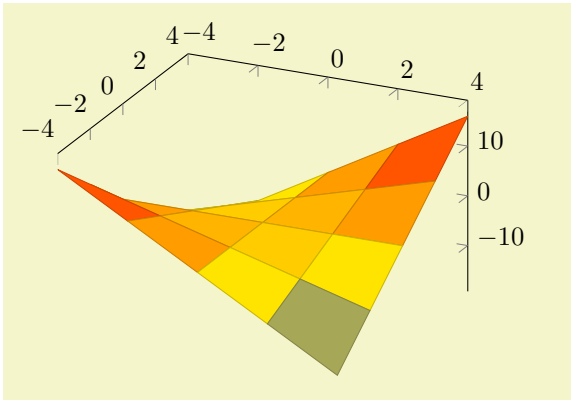

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        axis lines=center,
        axis on top,
        samples=5, domain=-4:4,
        xtick=data, ytick=data,
        ztick=\empty, %  no z ticks here
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```

The remaining choices `axis lines*=left` and `axis lines*=right` select different sets of axes in a way such that tick labels and axis label won't disturb the plot's content. The '`*`' suppresses the use of special styles which are mainly adequate for two-dimensional axes, see the documentation of `axis lines`. Such a set of axes is always on the boundary of the two-dimensional projection.

The choice `axis lines*=left` chooses a set of axes which are on the left (or bottom, respectively) whereas the choice `axis lines*=right` chooses a set of axes which are on the right (or top, respectively):




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        axis lines*=left,
        samples=5, domain=-4:4,
        xtick=data, ytick=data,
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        axis lines*=right,
        samples=5, domain=-4:4,
        xtick=data, ytick=data,
    ]
        \addplot3[surf] {x*y};
    \end{axis}
\end{tikzpicture}
```

It is not possible to mix different styles like `axis x line=center,axis z line=top`.

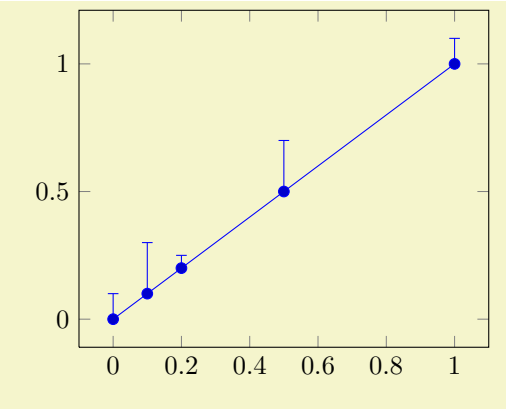
4.11 Error Bars

PGFPLOTS supports error bars for normal and logarithmic plots.

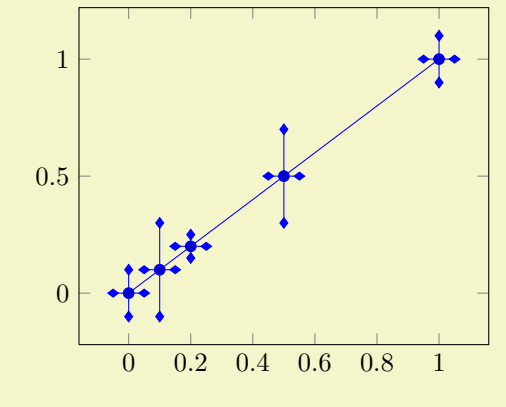
Error bars are enabled for each plot separately, using ⟨*options*⟩ after `\addplot`:

```
\addplot+[error bars/.cd,x dir=both,y dir=both] ...
```

Error bars inherit all drawing options of the associated plot, but they use their own marker and style arguments additionally.

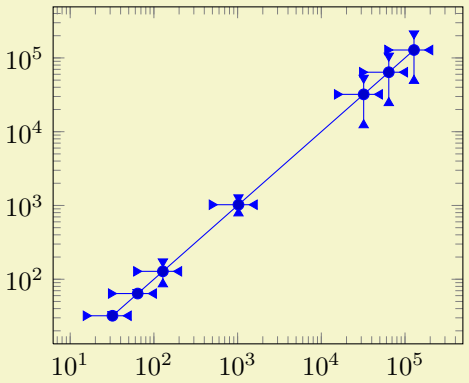



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[error bars/.cd,
    y dir=plus,y explicit]
coordinates {
    (0,0)     += (0.5,0.1)
    (0.1,0.1) += (0.05,0.2)
    (0.2,0.2) += (0,0.05)
    (0.5,0.5) += (0.1,0.2)
    (1,1)     += (0.3,0.1)};
\end{axis}
\end{tikzpicture}
```

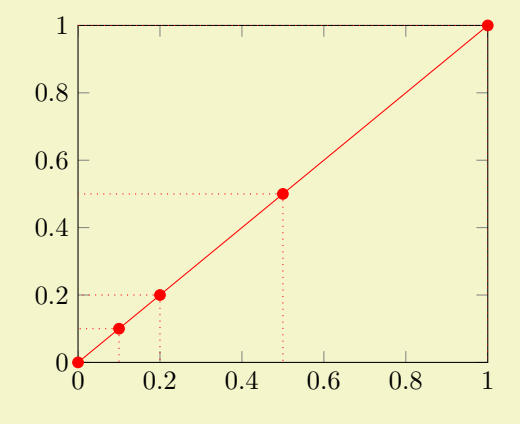



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot+[error bars/.cd,
    y dir=both,y explicit,
    x dir=both,x fixed=0.05,
    error mark=diamond*]
coordinates {
    (0,0)     += (0.5,0.1)
    (0.1,0.1) += (0.05,0.2)
    (0.2,0.2) += (0,0.05)
    (0.5,0.5) += (0.1,0.2)
    (1,1)     += (0.3,0.1)};
\end{axis}
\end{tikzpicture}
```

x	y	errorx	errory
32	32	0	0
64	64	0	0
128	128	0	0.3
1,024	1,024	0	0.2
32,068	32,068	0	0.6
64,000	64,000	0	0.6
$1.28 \cdot 10^5$	$1.28 \cdot 10^5$	0	0.6




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotstabletypeset{pgfplots.testtable2.dat}

\begin{tikzpicture}
\begin{loglogaxis}
\addplot+[error bars/.cd,
    x dir=both,x fixed relative=0.5,
    y dir=both,y explicit relative,
    error mark=triangle*]
    table[x=x,y=y,y error=errory]
    {pgfplots.testtable2.dat};
\end{loglogaxis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[enlargelimits=false]
\addplot[red,mark=*]
    plot[error bars/.cd,
    y dir=minus,y fixed relative=1,
    x dir=minus,x fixed relative=1,
    error mark=none,
    error bar style={dotted}]
coordinates
    {(0,0) (0.1,0.1) (0.2,0.2)
     (0.5,0.5) (1,1)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/error bars/x dir=none|plus|minus|both (initially none)
/pgfplots/error bars/y dir=none|plus|minus|both (initially none)
/pgfplots/error bars/z dir=none|plus|minus|both (initially none)

Draws either no error bars at all, only marks at $x + \epsilon_x$, only marks at $x - \epsilon_x$ or marks at both, $x + \epsilon_x$ and $x - \epsilon_x$. The x-error ϵ_x is acquired using one of the following options.

The same holds for the `y dir` option.

/pgfplots/error bars/x fixed={⟨*value*⟩} (initially 0)
/pgfplots/error bars/y fixed={⟨*value*⟩} (initially 0)
/pgfplots/error bars/z fixed={⟨*value*⟩} (initially 0)

Provides a common, absolute error $\epsilon_x = \langle value \rangle$ for all input coordinates.

For linear x axes, the error mark is drawn at $x \pm \epsilon_x$ while for logarithmic x axes, it is drawn at $\log(x \pm \epsilon_x)$. Computations are performed in PGF's floating point arithmetics.

/pgfplots/error bars/x fixed relative={⟨*percent*⟩} (initially 0)
/pgfplots/error bars/y fixed relative={⟨*percent*⟩} (initially 0)
/pgfplots/error bars/z fixed relative={⟨*percent*⟩} (initially 0)

Provides a common, relative error $\epsilon_x = \langle percent \rangle \cdot x$ for all input coordinates. The argument $\langle percent \rangle$ is thus given relatively to input x coordinates such that $\langle percent \rangle = 1$ means 100%.

Error marks are thus placed at $x \cdot (1 \pm \epsilon_x)$ for linear axes and at $\log(x \cdot (1 \pm \epsilon_x))$ for logarithmic axes. Computations are performed in floating point for linear axis and using the identity $\log(x \cdot (1 \pm \epsilon_x)) = \log(x) + \log(1 \pm \epsilon_x)$ for logarithmic scales.

/pgfplots/error bars/x explicit (no value)
/pgfplots/error bars/y explicit (no value)
/pgfplots/error bars/z explicit (no value)

Configures the error bar algorithm to draw x-error bars at any input coordinate for which user-specified errors are available. Each error is interpreted as absolute error, see `x fixed` for details.

The different input formats of errors are described in Section 4.11.1.

/pgfplots/error bars/x explicit relative (no value)
/pgfplots/error bars/y explicit relative (no value)
/pgfplots/error bars/z explicit relative (no value)

Configures the error bar algorithm to draw x-error bars at any input coordinate for which user-specified errors are available. Each error is interpreted as relative error, that means error marks are placed at $x(1 \pm \langle value \rangle(x))$ (works as for `error bars/x fixed relative`).

/pgfplots/error bars/error mark=⟨*marker*⟩

Sets an error marker for any error bar. {⟨*marker*⟩} is expected to be a valid plot mark, see Section 4.6.

/pgfplots/error bars/error mark options={⟨*key-value-list*⟩}

Sets a key-value list of options for any error mark. This option works similary to the Ti*k*Z '`mark options`' key.

/pgfplots/error bars/error bar style={⟨*key-value-list*⟩}

Appends the argument to '/pgfplots/every error bar' which is installed at the beginning of every error bar.

/pgfplots/error bars/draw error bar/.code 2 args={⟨...⟩}

Allows to change the default drawing commands for error bars. The two arguments are

- the source point, (x, y) and
- the target point, $(\tilde{x}, \tilde{y})$.

Both are determined by PGFPLOTS according to the options described above. The default code is

```
\pgfplotsset{
    /pgfplots/error bars/draw error bar/.code 2 args={%
        \pgfkeysgetvalue{/pgfplots/error bars/error mark}%
            {\pgfplotserrorbarsmark}%
        \pgfkeysgetvalue{/pgfplots/error bars/error mark options}%
            {\pgfplotserrorbarsmarkopts}%
        \draw #1 -- #2 node[pos=1,sloped,allow upside down] {%
            \expandafter\tikz\expandafter[\pgfplotserrorbarsmarkopts]{%
                \expandafter\pgfuseplotmark\expandafter{\pgfplotserrorbarsmark}%
                \pgfusepath{stroke}}%
        };
    }
}
```

4.11.1 Input Formats of Error Coordinates

Error bars with explicit error estimations for single data points require some sort of input format. This applies to 'error bars/⟨*[xy]*⟩ explicit' and 'error bars/⟨*[xy]*⟩ explicit relative'.

Error bar coordinates can be read from 'plot coordinates' or from 'plot table'. The inline plot coordinates format is

```
\addplot coordinates {
    (1,2) +- (0.4,0.2)
    (2,4) +- (1,0)
    (3,5)
    (4,6) +- (0.3,0.001)
}
```

where $(1, 2) \pm (0.4, 0.2)$ is the first coordinate, $(2, 4) \pm (1, 0)$ the second and so forth. The point $(3, 5)$ has no error coordinate.

The 'plot table' format is

```
\addplot table[x error=COLNAME,y error=COLNAME]
```

or

```
\addplot table[x error index=COLINDEX,y error index=COLINDEX]
```

These options are used like the 'x' and 'x index' options.

You can supply error coordinates even if they are not used at all; they will be ignored silently in this case.

4.12 Number Formatting Options

PGFPLOTS typesets tick labels rounded to given precision and in configurable number formats. The command to do so is \pgfmathprintnumber; it uses the current set of number formatting options. In addition, PGFPLOTS might prepare tick numbers before they are handed over to \pgfmathprintnumber.

The options related to number printing as such are described in all detail in the manual for PGFPLOTSTABLE, which comes with PGFPLOTS. This section contains the reference for everything which is specific to an axis, and only a brief survey over the number formatting options as such.

4.12.1 Frequently Used Number Printing Settings

This section provides a brief survey about the most frequently used aspects of number formatting in PGFPLOTS.

1. PGFPLOTS computes common tick scaling factors like $\cdot 10^2$ and produces only integers as tick labels.

   In order to get numbers like 0.001 as tick labels instead of 1 with a separate label $\cdot 10^{-3}$, you can use `scaled ticks=false` in your axis. See the description of `scaled ticks` for details.

2. In order to customize the way numbers are rounded and/or displayed, use something like `xticklabel style={/pgf/number format/.cd,fixed,precision=5}`.

   Here is a short list of possibilities:

123.46	`\pgfmathprintnumber{123.456789}`
12,345.68	`\pgfmathprintnumber{12345.6789}`
12,345.6789	`\pgfmathprintnumber [fixed,precision=5]{12345.6789}`
12,345.67890	`\pgfmathprintnumber [fixed,fixed zerofill,precision=5]{12345.6789}`
12.345,67890	`\pgfmathprintnumber [fixed,fixed zerofill,precision=5,use comma] {12345.6789}`
$1.23 \cdot 10^4$	`\pgfmathprintnumber [sci]{12345.6789}`
$1.23457 \cdot 10^4$	`\pgfmathprintnumber [sci,sci zerofill,precision=5]{12345.6789}`
1.23×10^1	`\pgfmathprintnumber [sci,sci generic= {mantissa sep=\times,exponent={10^{#1}}}] {12.345}`
$\frac{1}{3}$; $\frac{1}{2}$	`\pgfmathprintnumber[frac]{0.333333333333333}; \pgfmathprintnumber[frac]{0.5}`
+2	`\pgfmathprintnumber[print sign]{2}`
1 000 000.123456	`\pgfmathprintnumber [1000 sep={\,},fixed,precision=6]{1000000.123456}`
1 000 000.123 456	`\pgfmathprintnumber [1000 sep={\,},fixed,precision=6,1000 sep in fractionals]{1000000.123456}`

   Each of these keys requires the prefix '`/pgf/number format/`' when used inside of a PGFPLOTS style (try `/pgf/number format/.cd,`⟨*number formatting keys*⟩ to use the same prefix for many ⟨*number formatting keys*⟩).

   The number formatting uses `\pgfmathprintnumber`, a PGF command to typeset numbers. A full reference of all supported options is shipped with PGFPLOTS: it is documented in the reference manual for PgfplotsTable, Section 'Number Formatting Options'. The same reference can be found in the documentation for PGF.

   Note that the number printer knows *nothing* about PGFPLOTS. In particular, it is not responsible for logs and their representation.

3. For a logarithmic axis, one may want to modify the number formatting style for the *exponent only*. In this case, redefine the style `log plot exponent style` (its documentation contains a couple of examples).

4. In order to get `fixed` point tick labels on a logarithmic axis, you can use `log ticks with fixed point` (see below).

4.12.2 PGFPlots-specific Number Formatting

This section contains fine–tuning options to change number formatting aspects – but only things which are specific to PGFPLOTS like peculiarities of tick labels on logarithmic axes. Consider browsing Section 4.12.1 first to see if you need this section.

`\pgfmathprintnumber{`⟨*x*⟩`}`

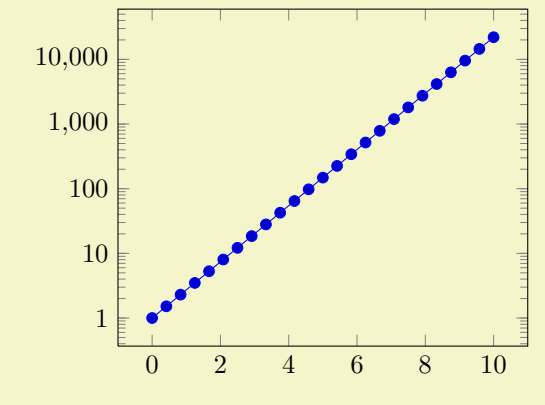
Generates pretty-printed output for the (real) number ⟨*x*⟩. The input number ⟨*x*⟩ is parsed using `\pgfmathfloatparsenumber` which allows arbitrary precision.

Numbers are typeset in math mode using the current set of number printing options, see below. Optional arguments can also be provided using `\pgfmathprintnumber[`⟨*options*⟩`]{`⟨*x*⟩`}`.

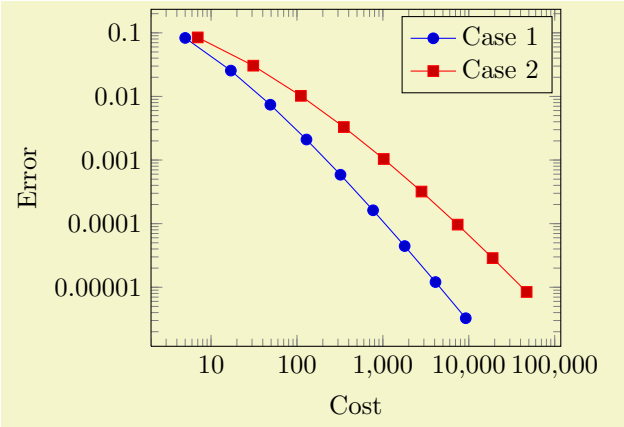
Please refer to the manual of PGFPLOTSTABLE (shipped with this package) for details about options related to number-printing.

`/pgfplots/log ticks with fixed point` (style, no value)

Reconfigures PGFPLOTS to display tick labels of logarithmic axes using *fixed point* numbers instead of the exponential style.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}[log ticks with fixed point]
    \addplot+[domain=0:10] {exp(x)};
\end{semilogyaxis}
\end{tikzpicture}
```

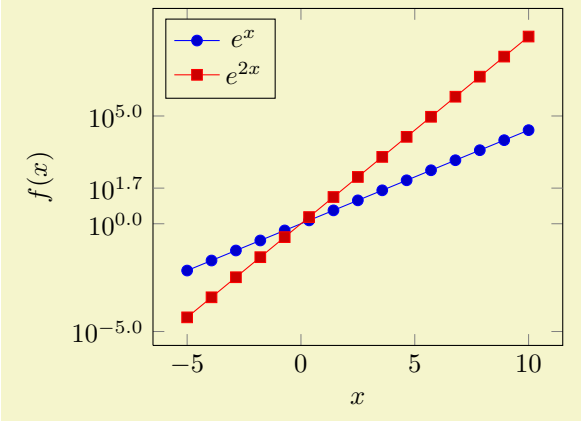


```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    log ticks with fixed point,
    xlabel=Cost,ylabel=Error]
\addplot coordinates {
    (5,     8.31160034e-02)
    (17,    2.54685628e-02)
    (49,    7.40715288e-03)
    (129,   2.10192154e-03)
    (321,   5.87352989e-04)
    (769,   1.62269942e-04)
    (1793,  4.44248889e-05)
    (4097,  1.20714122e-05)
    (9217,  3.26101452e-06)
};
\addplot coordinates {
    (7,     8.47178381e-02)
    (31,    3.04409349e-02)
    (111,   1.02214539e-02)
    (351,   3.30346265e-03)
    (1023,  1.03886535e-03)
    (2815,  3.19646457e-04)
    (7423,  9.65789766e-05)
    (18943, 2.87339125e-05)
    (47103, 8.43749881e-06)
};
\legend{Case 1,Case 2}
\end{loglogaxis}
\end{tikzpicture}
```

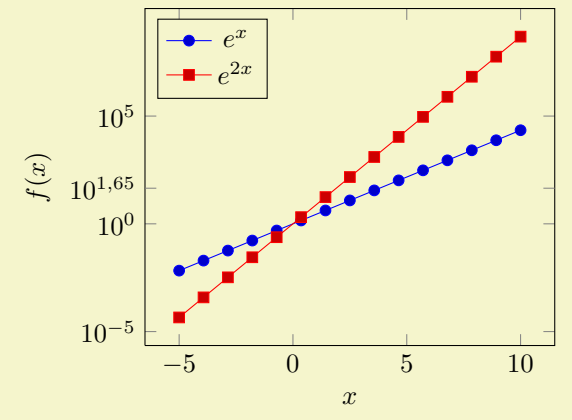
The style replaces `log number format basis`.

`/pgfplots/log plot exponent style={`⟨*key-value-list*⟩`}`

Allows to configure the number format of log plot exponents. This style is installed just before '`log number format basis`' will be invoked. Please note that this style will be installed within the default code for '`log number format code`'.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{
    samples=15,
    width=7cm,
    xlabel=$x$,
    ylabel=$f(x)$,
    extra y ticks={45},
    legend style={at={(0.03,0.97)},
        anchor=north west}}

\begin{tikzpicture}
\begin{semilogyaxis}[
    log plot exponent style/.style={
        /pgf/number format/fixed zerofill,
        /pgf/number format/precision=1},
    domain=-5:10]

    \addplot {exp(x)};
    \addplot {exp(2*x)};

    \legend{$e^x$,$e^{2x}$}
\end{semilogyaxis}
\end{tikzpicture}
```



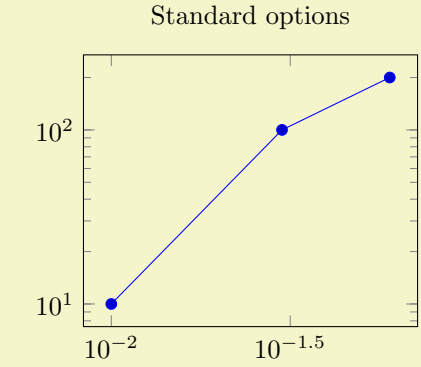

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{
    samples=15,
    width=7cm,
    xlabel=$x$,
    ylabel=$f(x)$,
    extra y ticks={45},
    legend style={at={(0.03,0.97)},
        anchor=north west}}

\begin{tikzpicture}
\begin{semilogyaxis}[
    log plot exponent style/.style={
        /pgf/number format/fixed,
        /pgf/number format/use comma,
        /pgf/number format/precision=2},
    domain=-5:10]

    \addplot {exp(x)};
    \addplot {exp(2*x)};

    \legend{$e^x$,$e^{2x}$}
\end{semilogyaxis}
\end{tikzpicture}
```

/pgfplots/log identify minor tick positions=true|false (initially false)

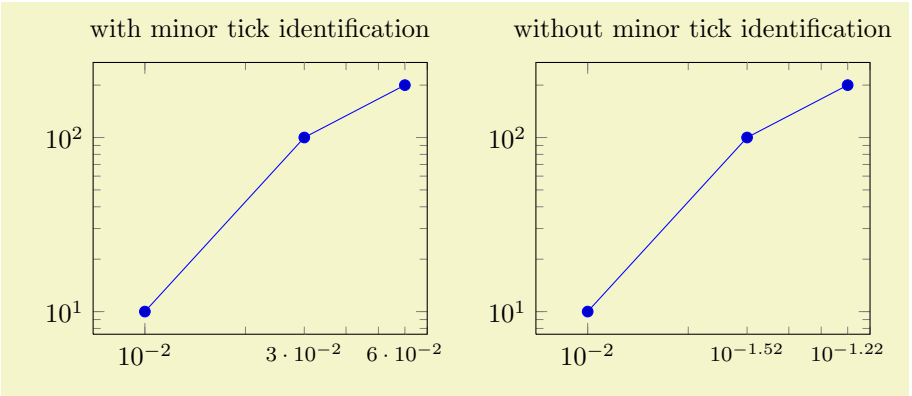
Set this to true if you want to identify log–plot tick labels at positions

$$i \cdot 10^j$$

with $i \in \{2, 3, 4, 5, 6, 7, 8, 9\}$, $j \in \mathbb{Z}$. This may be valuable in conjunction with the 'extra x ticks' and 'extra y ticks' options.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}%
\begin{loglogaxis}
    [title=Standard options,
    width=6cm]
\addplot coordinates {
    (1e-2,10)
    (3e-2,100)
    (6e-2,200)
};
\end{loglogaxis}
\end{tikzpicture}%
```



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={%
    width=6cm,
    xmin=7e-3,xmax=7e-2,
    extra x ticks={3e-2,6e-2},
    extra x tick style={major tick length=0pt,font=\footnotesize}
}}%

\begin{tikzpicture}%
    \begin{loglogaxis}[
        xtick={1e-2},
        title=with minor tick identification,
        extra x tick style={
            log identify minor tick positions=true}]
    \addplot coordinates {
        (1e-2,10)
        (3e-2,100)
        (6e-2,200)
    };
    \end{loglogaxis}
\end{tikzpicture}%

\begin{tikzpicture}%
    \begin{loglogaxis}[
        xtick={1e-2},
        title=without minor tick identification,
        extra x tick style={
            log identify minor tick positions=false}]
    \addplot coordinates {
        (1e-2,10)
        (3e-2,100)
        (6e-2,200)
    };
    \end{loglogaxis}%
\end{tikzpicture}%
```

This key is set by the default styles for extra ticks.

/pgfplots/log number format code/.code={⟨...⟩}

Provides TEX-code to generate log plot tick labels. Argument '#1' is the (natural) logarithm of the tick position. The default implementation invokes log base 10 number format code after it changed the log basis to 10. It also checks the other log plot options.

This key will have a different meaning when the log basis has been chosen explicitly, see the log basis x key.

/pgfplots/log base 10 number format code/.code={⟨...⟩}

Allows to change the overall appearance of base 10 log plot tick labels. The default implementation invokes log number format basis={10}{#1}.

Use log plot exponent style if you only want to change number formatting options for the exponent.

/pgfplots/log number format basis/.code={⟨...⟩}

Typesets a logarithmic tick. The first supplied argument is the log basis, the second the exponent. The initial configuration is

```
\pgfplotsset{
    /pgfplots/log number format basis/.code 2 args={$#1^{\pgfmathprintnumber{#2}}$}
}
```

Use log plot exponent style if you only want to change number formatting options for the exponent.

4.13 Specifying the Plotted Range

/pgfplots/xmin={⟨*coord*⟩}
/pgfplots/ymin={⟨*coord*⟩}
/pgfplots/zmin={⟨*coord*⟩}
/pgfplots/xmax={⟨*coord*⟩}

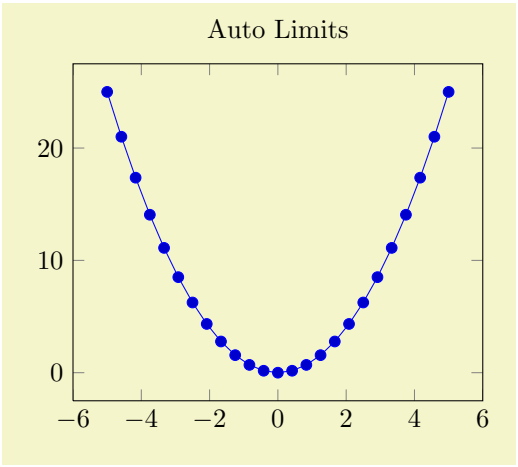
/pgfplots/ymax={⟨*coord*⟩}
/pgfplots/zmax={⟨*coord*⟩}
/pgfplots/min={⟨*coord*⟩}
/pgfplots/max={⟨*coord*⟩}

These options allow to define the axis limits, i.e. the lower left and the upper right corner. Everything outside of the axis limits will be clipped away.

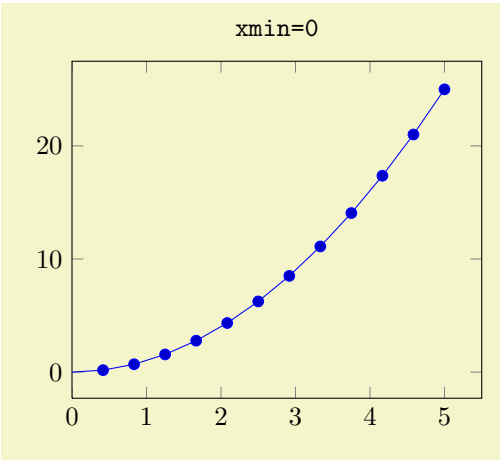
Each of these keys is optional, and missing limits will be determined automatically from input data. Here, the `min` and `max` keys set limits for x, y and z to the same ⟨*coord*⟩.

If x-limits have been specified explicitly and y-limits are computed automatically, the automatic computation of y-limits will only considers points which fall into the specified x-range (and vice–versa). The same holds true if, for example, only `xmin` has been provided explicitly: in that case, `xmax` will be updated only for points for which $x \geq$ `xmin` holds. This feature can be disabled using `clip limits=false`.

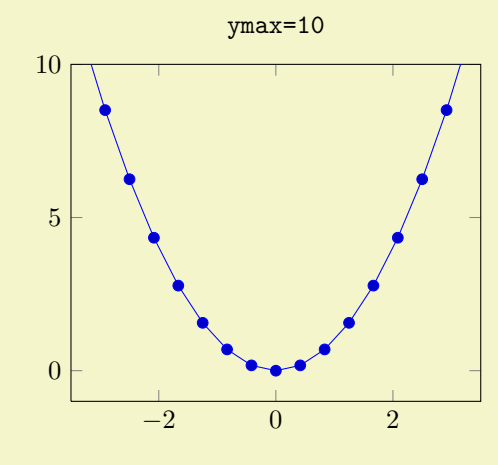
Axis limits can be increased automatically using the `enlargelimits` option.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[title=Auto Limits]
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[title={\texttt{xmin=0}},xmin=0]
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

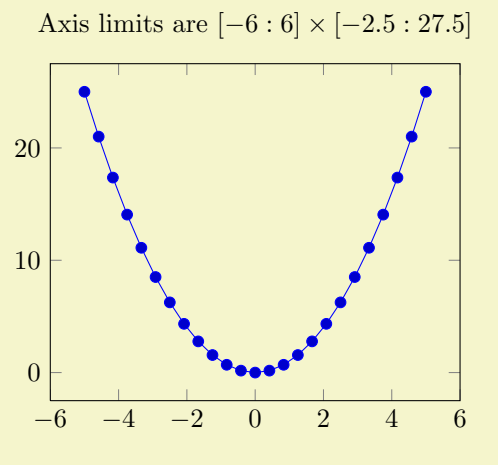



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[title={\texttt{ymax=10}},ymax=10]
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

Note that even if you provide `ymax=10`, data points with $y > 10$ will still be visualized – producing a line which leaves the plotted range.

See also the `restrict x to domain` and `restrict x to domain*` keys – they allow to discard or clip input coordinates which are outside of some domain, respectively.

During the visualization phase, i.e. during `\end{axis}`, these keys will be set to the final axis limits. You can access the values by means of `\pgfkeysvalueof{/pgfplots/xmin}`, for example:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
 %  Show (automatically) computed limits:
 title={
  Axis limits are
  $
 [\pgfmathprintnumber{\pgfkeysvalueof{/pgfplots/xmin}}
 :\pgfmathprintnumber{\pgfkeysvalueof{/pgfplots/xmax}}
 ]  \times
 [\pgfmathprintnumber{\pgfkeysvalueof{/pgfplots/ymin}}
 :\pgfmathprintnumber{\pgfkeysvalueof{/pgfplots/ymax}}
 ]$ },
]
    \addplot {x^2};
\end{axis}
\end{tikzpicture}
```

This access is possible inside of any axis description (like `xlabel`, `title`, `legend entries` etc.) or any annotation (i.e. inside of `\node`, `\draw` or `\path` and coordinates in `(axis cs:`$\langle x\rangle$`,`$\langle y\rangle$`)`), but not inside of `\addplot` (limits may not be complete at this stage).

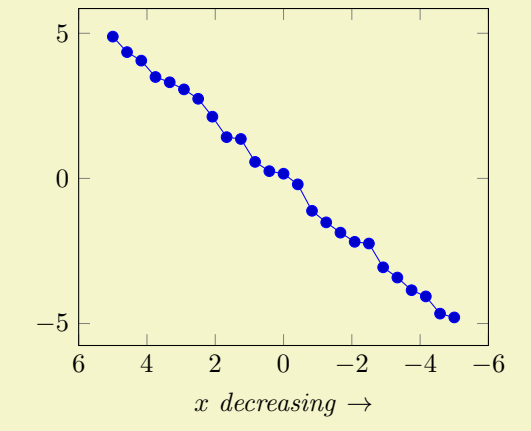
`/pgfplots/xmode=normal|linear|log` (initially `normal`)
`/pgfplots/ymode=normal|linear|log` (initially `normal`)
`/pgfplots/zmode=normal|linear|log` (initially `normal`)

Allows to choose between linear (=normal) or logarithmic axis scaling or logplots for each x, y, z-combination.

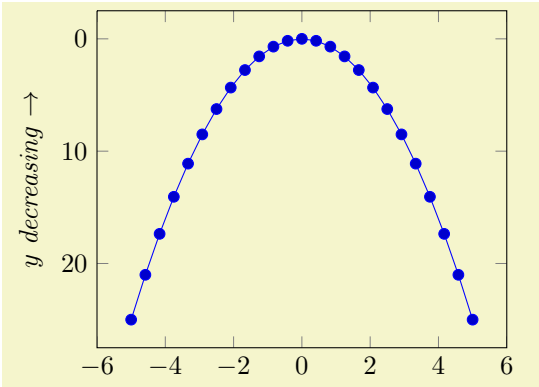
Logarithmic plots use the current setting of `log basis x` and its variants to determine the basis (default is e).

`/pgfplots/x dir=normal|reverse` (initially `normal`)
`/pgfplots/y dir=normal|reverse` (initially `normal`)
`/pgfplots/z dir=normal|reverse` (initially `normal`)

Allows to reverse axis directions such that values are given in decreasing order.

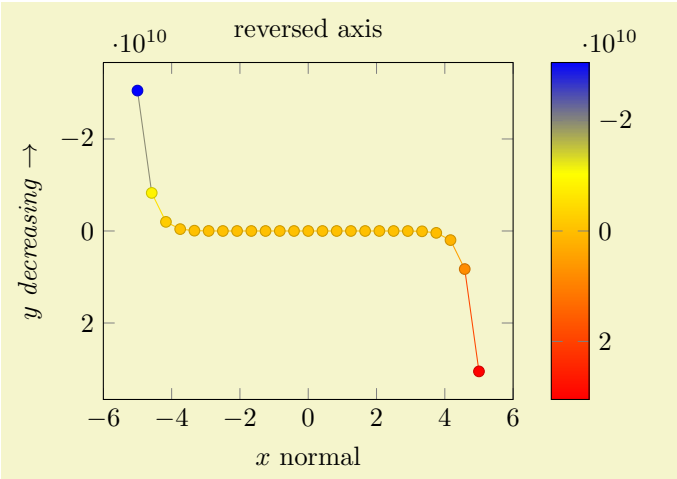



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xlabel=$x$ \emph{decreasing} $\to$,
    x dir=reverse]
    \addplot {x+rand*0.3};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    ylabel=$y$ \emph{decreasing} $\to$,
    y dir=reverse]
    \addplot {x^2};
\end{axis}
\end{tikzpicture}
```

Note that axis descriptions and relative positioning macros will stay at the same place as they would for non–reversed axes.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    ylabel=$y$ \emph{decreasing} $\to$,
    xlabel=$x$ normal,
    title=reversed axis,
    y dir=reverse,
    colorbar,
    colorbar style={y dir=reverse}]
    \addplot+[mesh,scatter] {x^15};
\end{axis}
\end{tikzpicture}
```

Note that colorbars won't be reversed automatically, you will have to reverse the sequence of color bars manually in case this is required as in the preceding example.

/pgfplots/clip limits=true|false (initially true)

Configures what to do if some, but not all axis limits have been specified explicitly. In case clip limits=true, the automatic limit computation will *only* consider points which do not contradict the explicitly set limits.

This option has nothing to do with path clipping, it only affects how the axis limits are computed.

/pgfplots/enlarge x limits=auto|true|false|upper|lower|⟨*val*⟩|value=⟨*val*⟩|abs value=⟨*val*⟩|abs=⟨*val*⟩|rel=⟨*val*⟩ (initially auto)
/pgfplots/enlarge y limits=auto|true|false|upper|lower|⟨*val*⟩|value=⟨*val*⟩|abs value=⟨*val*⟩|abs=⟨*val*⟩|rel=⟨*val*⟩ (initially auto)
/pgfplots/enlarge z limits=auto|true|false|upper|lower|⟨*val*⟩|value=⟨*val*⟩|abs value=⟨*val*⟩|abs=⟨*val*⟩|rel=⟨*val*⟩ (initially auto)
/pgfplots/enlargelimits=⟨*common value*⟩

Enlarges the axis size for one axis (or all of them for enlargelimits) somewhat if enabled.

You can set xmin, xmax and ymin, ymax to the minimum/maximum values of your data and enlarge x limits will enlarge the canvas such that the axis doesn't touch the plots.

- The value true enlarges the lower and upper limit.
- The value false uses tight axis limits as specified by the user (or read from input coordinates).
- The value auto will enlarge limits only for axis for which axis limits have been determined automatically.

  For three–dimensional figures, the auto mechanism applies only for the z axis. The x and y axis won't be enlarged.
- The value upper enlarges only the upper axis limit while lower enlarges only the lower axis limit.
- Values like 'enlarge x limits=0.1' will enlarge lower and upper axis limit relatively (in this example, 10% of the axis limits will be added on both sides).
- It is also possible to change just the relative threshold using the value={⟨*val*⟩} key. It can be combined with any of the other possible values. For example,

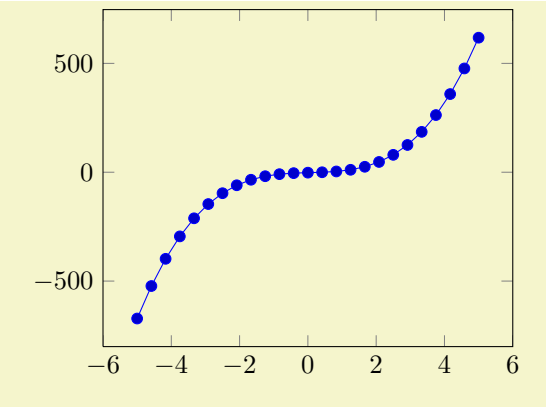
  \pgfplotsset{enlarge x limits={value=0.2,upper}}

  will enlarge (only) the upper axis limit by 20% of the axis range. Another example is

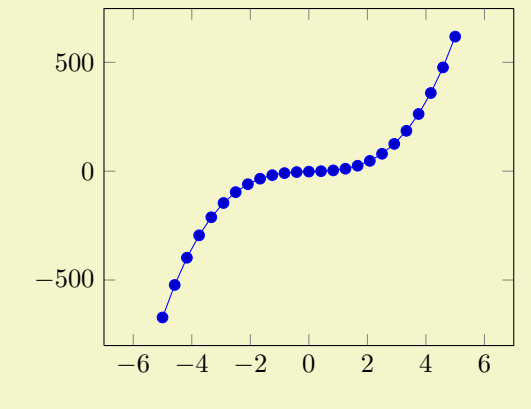
  \pgfplotsset{enlarge x limits={value=0.2,auto}}

  which changes the default threshold of the auto value to 20%.
- While value uses relative thresholds, abs value is used in the same way with absolute values.

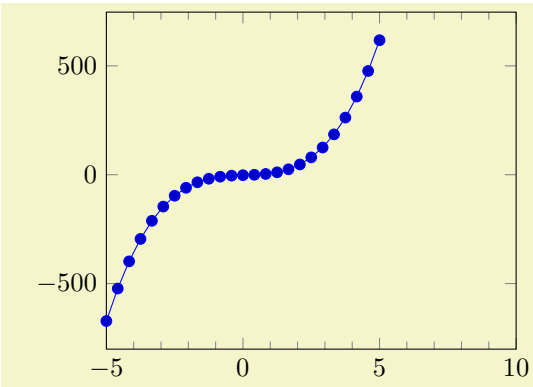
  Attention: abs value is applied *multiplicative* for logarithmic axes! That means abs value=10 for a logarithmic axis adds $\log 10$ to upper and/or lower axis limits.
- Finally, abs={⟨*value*⟩} is the same as true,abs value={⟨*value*⟩} and rel={⟨*value*⟩} is the same as true,value={⟨*value*⟩}.



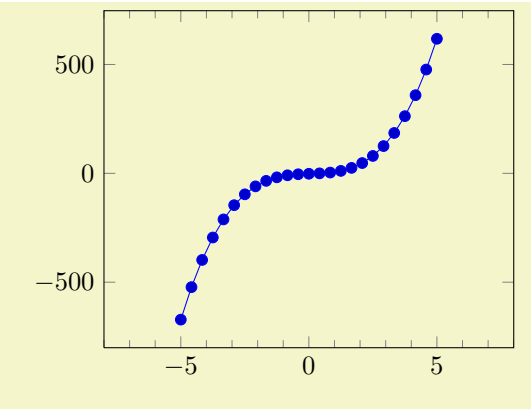

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
        \addplot {5 * x^3 - x^2 + 4*x -2};
    \end{axis}
\end{tikzpicture}
```



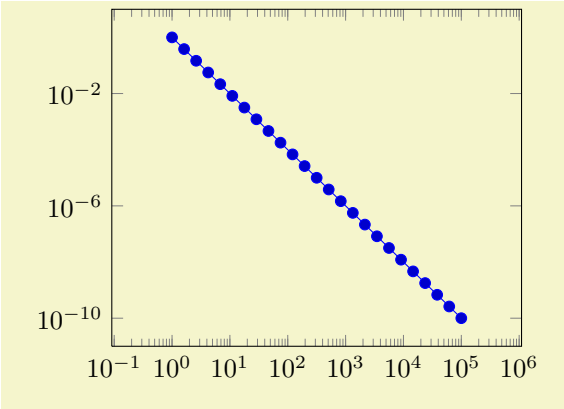

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[enlarge x limits=0.2]
        \addplot {5 * x^3 - x^2 + 4*x -2};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor x tick num=4,
        enlarge x limits={rel=0.5,upper}
    ]
        \addplot {5 * x^3 - x^2 + 4*x -2};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor x tick num=4,
        enlarge x limits={abs=3}
    ]
        \addplot {5 * x^3 - x^2 + 4*x -2};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[enlarge x limits={abs=11}]
        \addplot+[domain=1:100000] {x^-2};
    \end{loglogaxis}
\end{tikzpicture}
```

/pgfplots/log origin x=0|infty (initially infty)

/pgfplots/log origin y=0|infty (initially infty)
/pgfplots/log origin z=0|infty (initially infty)
/pgfplots/log origin=0|infty (initially infty)

Allows to choose which coordinate is the logical "origin" of a logarithmic plot (either for a particular axis or for all of them).

The choice log origin=infty is probably useful for stacked plots: it defines the "origin" in log–coordinates to be $-\infty$. To be compatibly with older versions, this is the default.

The choice log origin=0 defines the logarithmic origin to be the natural choice $\log(1) = 0$. This is particularly useful for ycomb plots.

/pgfplots/update limits=true|false (initially true)

Can be used to interrupt updates of the data limits (for example, for single \addplot commands).

This has the same effect as \pgfplotsinterruptdatabb ... \endpgfplotsinterruptdatabb.

\begin{pgfplotsinterruptdatabb}
⟨*environment contents*⟩
\end{pgfplotsinterruptdatabb}

Everything in ⟨*environment contents*⟩ will not contribute to the data bounding box.

The same effect can be achieved with update limits=false inside curly braces.

4.14 Tick Options

4.14.1 Tick Coordinates and Label Texts

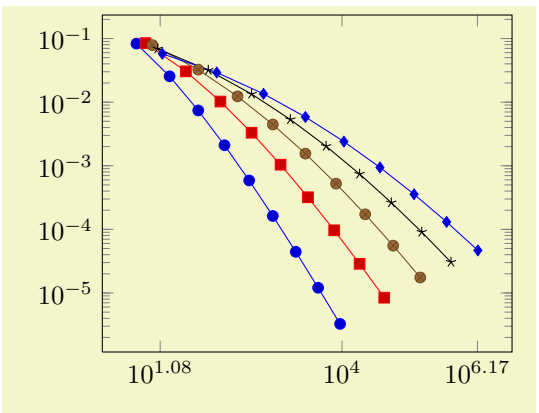
/pgfplots/xtick=\empty|data|{⟨*coordinate list*⟩} (initially {⟨⟩})
/pgfplots/ytick=\empty|data|{⟨*coordinate list*⟩} (initially {⟨⟩})
/pgfplots/ztick=\empty|data|{⟨*coordinate list*⟩} (initially {⟨⟩})

These options assign a list of *positions* where ticks shall be placed. The argument is either the empty string (which is the initial value), the command \empty, the special string 'data' or a list of coordinates. The initial configuration of an empty string means to generate these positions automatically. The choice \empty will result in no tick at all. The special value 'data' will produce tick marks at every coordinate of the first plot. Otherwise, tick marks will be placed at every coordinate in ⟨*coordinate list*⟩.

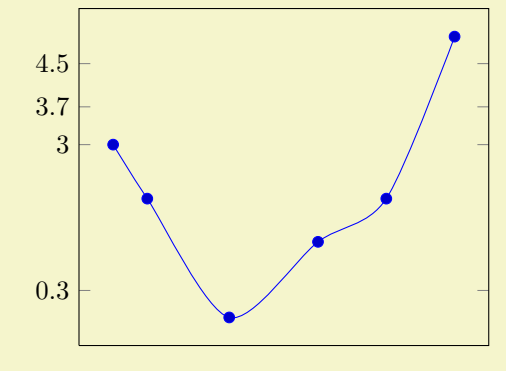
The ⟨*coordinate list*⟩ will be used inside of a \foreach \x in {⟨*coordinate list*⟩} statement. The format is as follows:

- {0,1,2,5,8,1e1,1.5e1} (a series of coordinates),
- {0,...,5} (the same as {0,1,2,3,4,5}),
- {0,2,...,10} (the same as {0,2,4,6,8,10}),
- {9,...,3.5} (the same as {9, 8, 7, 6, 5, 4}),
- See [5, Section 34] for a more detailed definition of the options.
- Please be careful with white spaces inside of ⟨*coordinate list*⟩ (at least around the dots).

For logplots, PGFPLOTS will apply $\log(\cdot)$ to each element in '⟨*coordinate list*⟩' (similarly, any custom transformations are applied to the argument list).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[xtick={12,9897,1468864}]
    %  see above for this macro:
    \plotcoords
    \end{loglogaxis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xtick=\empty,
    ytick={-2,0.3,3,3.7,4.5}]
\addplot+[smooth] coordinates {
    (-2,3) (-1.5,2) (-0.3,-0.2)
    (1,1.2) (2,2) (3,5)};
\end{axis}
\end{tikzpicture}
```

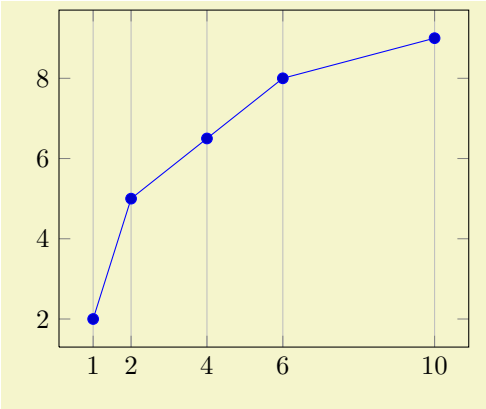
Attention: You can't use the '`...`' syntax if the elements are too large for TEX! For example, '`xtick=1.5e5,2e7,3e8`' will work (because the elements are interpreted as strings, not as numbers), but '`xtick=1.5,3e5,...,1e10`' will fail because it involves real number arithmetics beyond TEX's capacities.

The default choice for tick *positions* in normal plots is to place a tick at each coordinate $i \cdot h$. The step size h depends on the axis scaling and the axis limits. It is chosen from a list of "feasible" step sizes such that neither too much nor too few ticks will be generated. The default for logplots is to place ticks at positions 10^i in the axis' range. The positions depend on the axis scaling and the dimensions of the picture. If log plots contain just one (or two) positions 10^i in their limits, ticks will be placed at positions $10^{i \cdot h}$ with "feasible" step sizes h as in the case of linear axis.

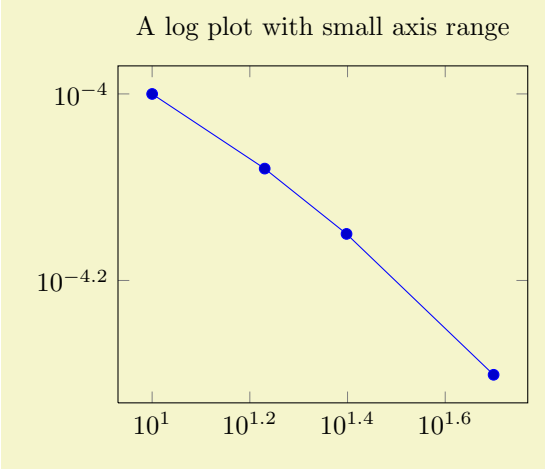
The tick *appearance* can be (re)configured with

```
\pgfplotsset{tick style={very thin,gray}}%  modifies the style 'every tick'
\pgfplotsset{minor tick style={black}}   %  modifies the style 'every minor tick'
```

These style commands can be used at any time. The tick line width can be configured with '`major tick length`' and '`minor tick length`'.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[xtick=data,xmajorgrids]
    \addplot coordinates {
        (1,2)
        (2,5)
        (4,6.5)
        (6,8)
        (10,9)
    };
\end{axis}
\end{tikzpicture}
```

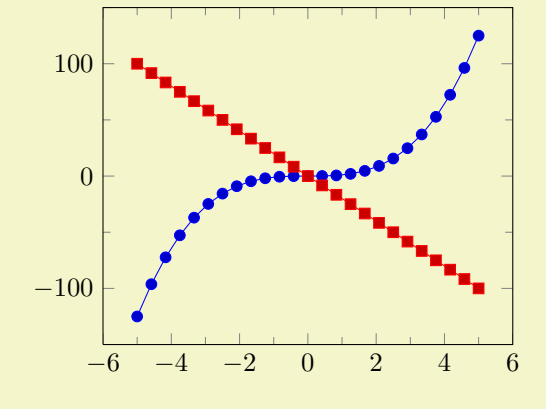



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    title=A log plot with small axis range]

    \addplot coordinates {
        (10,1e-4)
        (17,8.3176e-05)
        (25,7.0794e-05)
        (50,5e-5)
    };
\end{loglogaxis}
\end{tikzpicture}
```

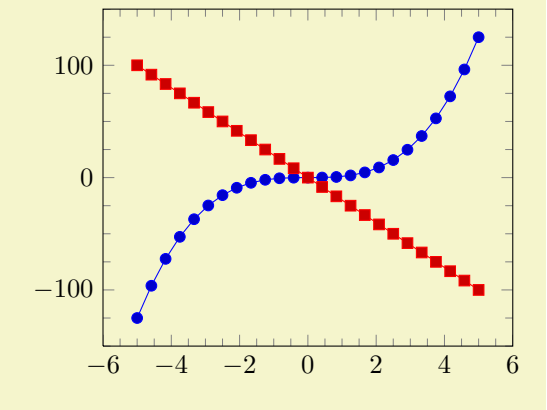
/pgfplots/minor x tick num={⟨*number*⟩} (initially 0)
/pgfplots/minor y tick num={⟨*number*⟩} (initially 0)
/pgfplots/minor z tick num={⟨*number*⟩} (initially 0)
/pgfplots/minor tick num={⟨*number*⟩}

Sets the number of minor tick lines used either for single axes or for all of them.

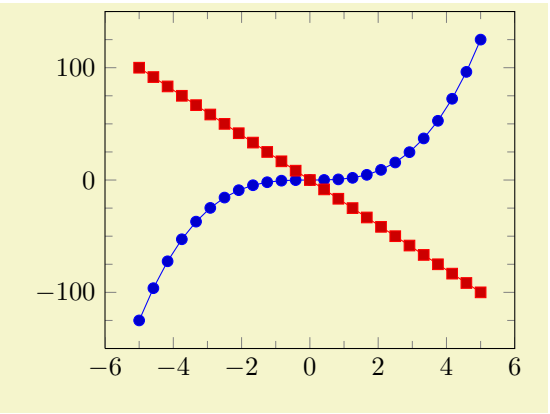
Minor ticks will be disabled if the major ticks don't have the same distance and they are currently only available for linear axes (not for logarithmic ones).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor tick num=1]
    \addplot {x^3};
    \addplot {-20*x};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor tick num=3]
    \addplot {x^3};
    \addplot {-20*x};
    \end{axis}
\end{tikzpicture}
```

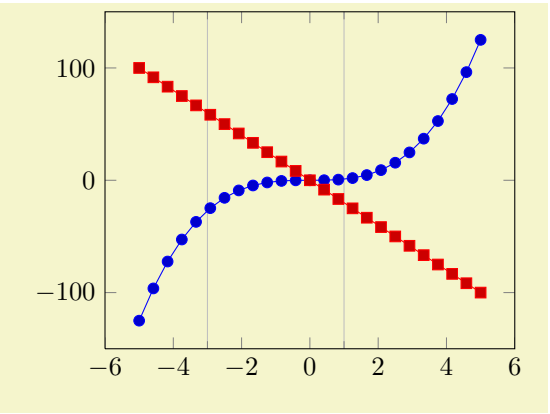



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor x tick num=1,
                 minor y tick num=3]
    \addplot {x^3};
    \addplot {-20*x};
    \end{axis}
\end{tikzpicture}
```

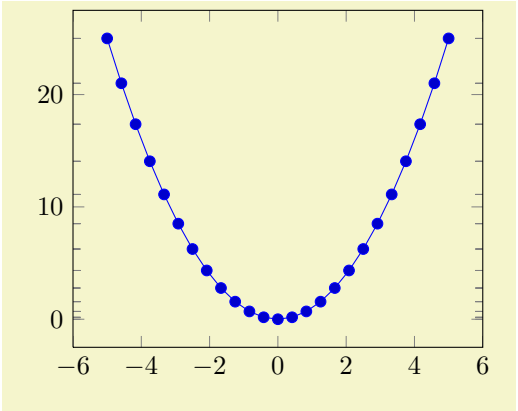
/pgfplots/minor xtick=data|{⟨*coordinate list*⟩} (initially `empty`)
/pgfplots/minor ytick=data|{⟨*coordinate list*⟩} (initially `empty`)
/pgfplots/minor ztick=data|{⟨*coordinate list*⟩} (initially `empty`)
/pgfplots/minor tick=data|{⟨*coordinate list*⟩}

Allows to provide a list of minor tick positions manually. The syntax is almost the same as for `xtick` or `ytick`: simply provide either a comma–separated list of tick positions or the special value `data`. An empty argument argument disables the `minor tick` feature (in contrast to `xtick` where the special value `\empty` clears the list and an empty argument causes PGFPLOTS to compute a default tick list).

In contrast to `minor x tick num`, this key allows to provide *non–uniform* minor tick positions.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor xtick={-3,1},grid=minor]
    \addplot {x^3};
    \addplot {-20*x};
    \end{axis}
\end{tikzpicture}
```

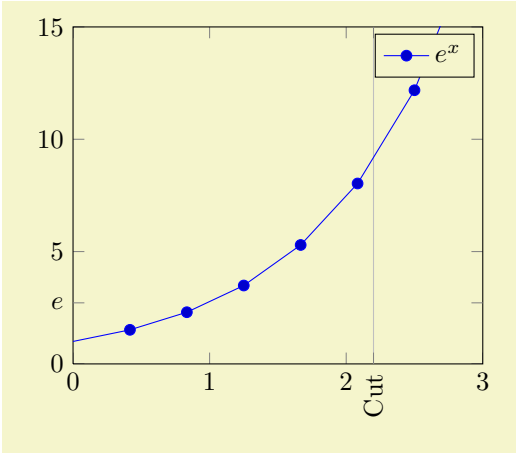



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[minor ytick=data]
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

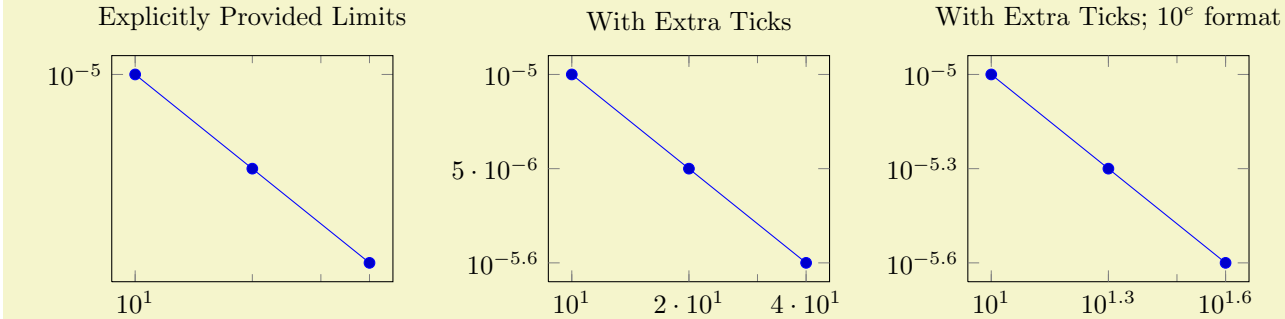
This key has precedence over `minor x tick num` and its variants; if both of them are given, `minor xtick` is preferred and `minor x tick num` is ignored.

/pgfplots/extra x ticks={⟨*coordinate list*⟩}
/pgfplots/extra y ticks={⟨*coordinate list*⟩}
/pgfplots/extra z ticks={⟨*coordinate list*⟩}

Adds *additional* tick positions and tick labels to the x or y axis. 'Additional' tick positions do not affect the normal tick placement algorithms, they are drawn after the normal ticks. This has two benefits: first, you can add single, important tick positions without disabling the default tick label generation and second, you can draw tick labels 'on top' of others, possibly using different style flags.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xmin=0,xmax=3,ymin=0,ymax=15,
    extra y ticks={2.71828},
    extra y tick labels={$e$},
    extra x ticks={2.2},
    extra x tick style={grid=major,
        tick label style={
            rotate=90,anchor=east}},
    extra x tick labels={Cut},
]
    \addplot {exp(x)};
    \addlegendentry{$e^x$}
\end{axis}
\end{tikzpicture}
```



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={width=5.3cm}}
\begin{tikzpicture}
\begin{loglogaxis}[
    title=Explicitly Provided Limits,
    xtickten={1,2},
    ytickten={-5,-6}]
\addplot coordinates
    {(10,1e-5) (20,5e-6) (40,2.5e-6)};
\end{loglogaxis}
\end{tikzpicture}

\begin{tikzpicture}
\begin{loglogaxis}[
    title=With Extra Ticks,
    xtickten={1,2},
    ytickten={-5,-6},
    extra x ticks={20,40},
    extra y ticks={5e-6,2.5e-6}]
\addplot coordinates
    {(10,1e-5) (20,5e-6) (40,2.5e-6)};
\end{loglogaxis}
\end{tikzpicture}

\begin{tikzpicture}
\begin{loglogaxis}[
    title=With Extra Ticks; $10^e$ format,
    extra tick style={log identify minor tick positions=false},
    xtickten={1,2},
    ytickten={-5,-6},
    extra x ticks={20,40},
    extra y ticks={5e-6,2.5e-6}]
\addplot coordinates
    {(10,1e-5) (20,5e-6) (40,2.5e-6)};
\end{loglogaxis}
\end{tikzpicture}
```

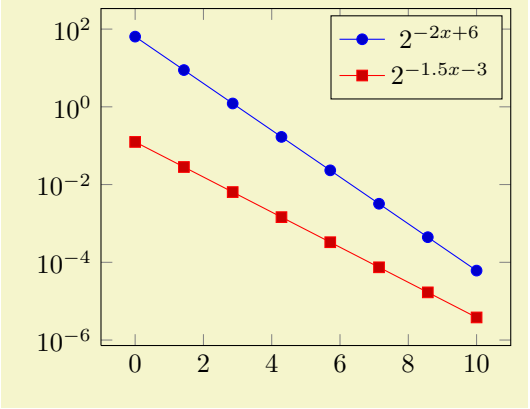
Remarks:

- Use `extra x ticks` to highlight special tick positions. The use of `extra x ticks` does not affect minor tick/grid line generation, so you can place extra ticks at positions $j \cdot 10^i$ in log–plots.
- Extra ticks are always typeset as major ticks.
  They are affected by `major tick length` or options like `grid=major`.
- Use the style `every extra x tick` (`every extra y tick`) to configure the appearance.
- You can also use '`extra x tick style={`⟨*...*⟩`}`' which has the same effect.

/pgfplots/**xtickten**={⟨*exponent base 10 list*⟩}
/pgfplots/**ytickten**={⟨*exponent base 10 list*⟩}
/pgfplots/**ztickten**={⟨*exponent base 10 list*⟩}

These options allow to place ticks at selected positions $10^k, k \in$ {⟨*exponent base 10 list*⟩}. They are only used for logplots. The syntax for {⟨*exponent base 10 list*⟩} is the same as above for `xtick={`⟨*list*⟩`}` or `ytick={`⟨*list*⟩`}`.

Using '`xtickten={1,2,3,4}`' is equivalent to '`xtick={1e1,1e2,1e3,1e4}`', but it requires fewer computational time and it allows to use the short syntax '`xtickten={1,...,4}`'.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}[
    samples=8,
    ytickten={-6,-4,...,4},
    domain=0:10]

\addplot {2^(-2*x + 6)};
\addlegendentry{$2^{-2x + 6}$}

%  or invoke gnuplot to generate coordinates:
\addplot gnuplot[id=pow2]
    {2**(-1.5*x -3)};
\addlegendentry{$2^{-1.5x -3}$}
\end{semilogyaxis}
\end{tikzpicture}
```

In case `log basis x`$\neq 10$, the meaning of `xtickten` changes. In such a case, `xtickten` will still assign the exponent, but for the chosen `log basis x` instead of base 10.

/pgfplots/**xticklabels**={⟨*label list*⟩}
/pgfplots/**yticklabels**={⟨*label list*⟩}
/pgfplots/**zticklabels**={⟨*label list*⟩}

Assigns a *list* of tick *labels* to each tick position. Tick *positions* are assigned using the `xtick` and `ytick`-options.

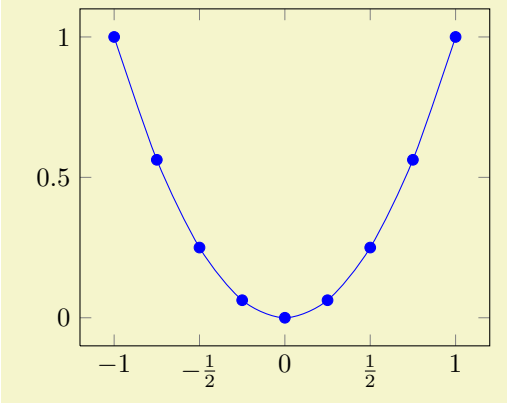
This is one of two options to assign tick labels directly. The other option is `xticklabel`={⟨*command*⟩} (or `yticklabel`={⟨*command*⟩}). The option '`xticklabel`' offers higher flexibility while '`xticklabels`' is easier to use. See also the variant `xticklabels from table`.

The argument ⟨*label list*⟩ has the same format as for ticks, that means

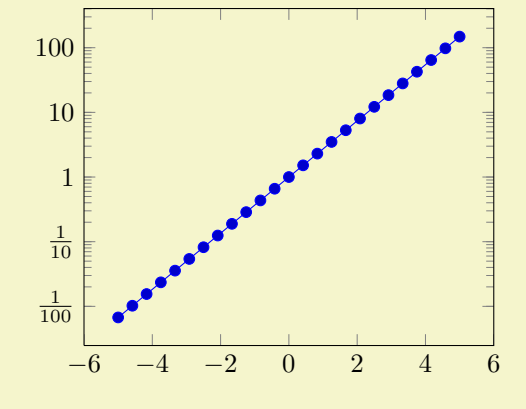
```
xticklabels={$\frac{1}{2}$,$e$}
```

denotes the two–element–list $\{\frac{1}{2}, e\}$. The list indices match the indices of the tick positions. If you need commas inside of list elements, use

```
xticklabels={{0,5}, $e$}.
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xtick={-1.5,-1,...,1.5},
    xticklabels={%
        $-1\frac 12$,
        $-1$,
        $-\frac 12$,
        $0$,
        $\frac 12$,
        $1$},
    %  note: \frac can be done automatically:
    %  xticklabel style={/pgf/number format/frac},
]
\addplot[smooth,blue,mark=*]
coordinates {
    (-1,    1)
    (-0.75, 0.5625)
    (-0.5,  0.25)
    (-0.25, 0.0625)
    (0,     0)
    (0.25,  0.0625)
    (0.5,   0.25)
    (0.75,  0.5625)
    (1,     1)
};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{semilogyaxis}[
  ytickten={-2,-1,0,1,2},
  yticklabels={$\frac{1}{100}$,%
      $\frac{1}{10}$,%
    1,10,100},
]
    \addplot {exp(x)};
\end{semilogyaxis}
\end{tikzpicture}
```

Note that it is also possible to terminate list entries with two backslashes, \\. In that case, the last entry needs to be terminated by \\ as well (it is the same alternative syntax which is also accepted for \legend and cycle list).

Please keep in mind that the arguments *always* refer the a list of tick positions, although it does not alter or define the list of positions. Consequently, you should also provide the list of positions. Note that a list of positions might be longer than what is actually displayed (in case the axis limits clip some of the value away), but the index mapping into ⟨*label list*⟩ still includes the clipped values.

/pgfplots/xticklabel={⟨*command*⟩}
/pgfplots/yticklabel={⟨*command*⟩}
/pgfplots/zticklabel={⟨*command*⟩}

These keys change the TEX-command which creates the tick *labels* assigned to each tick position (see options xtick and ytick).

This is one of the two options to assign tick labels directly. The other option is 'xticklabels={⟨*label list*⟩}' (or yticklabels={⟨*label list*⟩}). The option 'xticklabel' offers higher flexibility while 'xticklabels' is easier to use.

The argument ⟨*command*⟩ can be any TEX-text. The following commands are valid inside of ⟨*command*⟩:

\tick The current element of option xtick (or ytick).

\ticknum The current tick number, starting with 0 (it is a macro containing a number).

\nexttick This command is only valid if the x tick label as interval option is set (or the corresponding variable for *y*). It will contain the position of the next tick position, that means the right boundary of the tick interval.

The default argument is

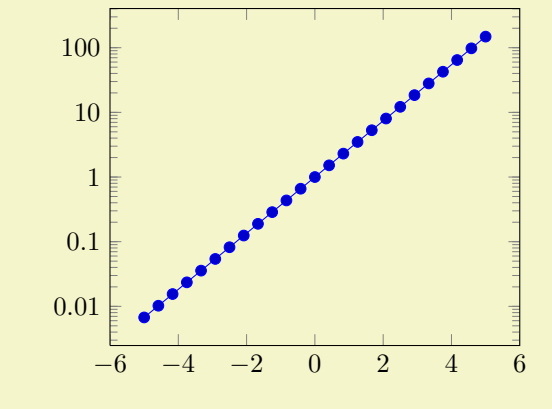
- \axisdefaultticklabel for normal plots:

```
\def\axisdefaultticklabel{$\pgfmathprintnumber{\tick}$}
```

- \axisdefaultticklabellog for logplots:

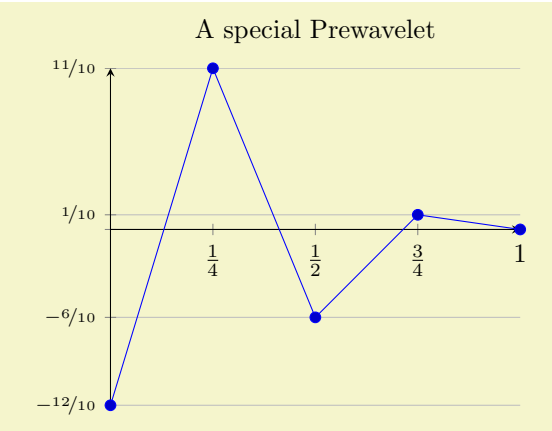
```
\def\axisdefaultticklabellog{%
    \pgfkeysgetvalue{/pgfplots/log number format code/.@cmd}\pgfplots@log@label@style
    \expandafter\pgfplots@log@label@style\tick\pgfeov
}
```

That means you can configure the appearance of linear axis with the number formatting options described in Section 4.12 and logarithmic axis with log number format code, see below.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{semilogyaxis}[
        yticklabel style={/pgf/number format/fixed},
        %  changes tick labels to a number instead
        %  of exponential notation:
        yticklabel={%
            \pgfmathfloatparsenumber{\tick}%
            \pgfmathfloatexp{\pgfmathresult}%
            \pgfmathprintnumber{\pgfmathresult}%
        },
    ]
        \addplot {exp(x)};
    \end{semilogyaxis}
\end{tikzpicture}
```

The following example uses explicitly formatted x tick labels and a small TEX script to format y tick labels as fractions in the form ⟨*sign*⟩⟨*number*⟩`/10` (note that the `/pgf/number format/frac` style can do similar things automatically, see PGFPLOTSTABLE and the documentation therein).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  \usepackage{nicefrace}% required
\begin{tikzpicture}
\begin{axis}[
    %  x ticks explicitly formatted:
    xtick={0,1,0.5,0.25,0.75},
    xticklabels={$0$,$1$,$\frac12$,$\frac14$,$\frac34$},
    %  y ticks automatically by some code fragment:
    ytick=data,
    yticklabel={%
        \scriptsize
        \ifdim\tick pt<0pt %  a TeX \if -- see TeX Book
            \pgfmathparse{-10*\tick}%
            $-\nicefrac{\pgfmathprintnumber{\pgfmathresult}}{10}$%
        \else
            \ifdim\tick pt=0pt
            \else
                \pgfmathparse{10*\tick}%
                $\nicefrac{\pgfmathprintnumber{\pgfmathresult}}{10}$%
            \fi
        \fi
    },
    %  NOTE: this here does the same:
    %  yticklabel style={/pgf/number format/.cd,frac,
    %      frac TeX=\nicefrac,frac whole=false,frac denom=10},
    ymajorgrids,
    title=A special Prewavelet,
    axis x line=center,
    axis y line=left,
    ]
    \addplot coordinates {(0,-1.2) (0.25,1.1)
        (0.5,-0.6) (0.75,0.1) (1,0)};
\end{axis}
\end{tikzpicture}
```

The TEX script takes the \tick macro as input and applies some logic. The \ifdim\tick pt<0pt means "if dimension \tick pt < 0pt". The \ifdim is TEX's only way to compare real fixed point numbers and the author did not want to invoke \pgfmath for this simple task. Since \ifdim expects a dimension, we have to use the pt suffix which is compatible with \pgfmath. The result is that negative numbers, zero and positive numbers are typeset differently.

You can change the appearance of tick labels with

```
\pgfplotsset{tick label style={
    font=\tiny,
    /pgf/number format/sci}}%  this modifies the 'every tick label' style
```

and/or

```
\pgfplotsset{x tick label style={
    above,
    /pgf/number format/fixed zerofill}}%  this modifies the 'every x tick label' style
```

and

```
\pgfplotsset{y tick label style={font=\bfseries}}%  modifies 'every y tick label'
```

/pgfplots/xticklabels from table={⟨\table or filename⟩}{⟨colname⟩}
/pgfplots/yticklabels from table={⟨\table or filename⟩}{⟨colname⟩}
/pgfplots/zticklabels from table={⟨\table or filename⟩}{⟨colname⟩}

A variant of xticklabels={⟨list⟩} which uses each entry in the column named ⟨colname⟩ from a table as tick labels.

The first argument ⟨\table or filename⟩ can be either a loaded table macro (i.e. the result of \pgfplotstableread{⟨file name⟩}{⟨\table⟩}) or just a file name.

The second argument can be a column name, a column alias or a create on use specification (see PgfplotsTable for the latter two). Furthermore, it can be [index]⟨integer⟩ in which case ⟨integer⟩ is a column index.

The behavior of xticklabels from table is the same as if the column ⟨colname⟩ would have been provided as comma separated list to xticklabels. This means the column can contain text, TEX macros or even math mode.

If you have white spaces in your cells, enclose the complete cell in curly braces, {example cell}. The detailed input format for tables is discussed in \addplot table and in the documentation for PgfplotsTable.

/pgfplots/extra x tick label={⟨TEX code⟩}
/pgfplots/extra y tick label={⟨TEX code⟩}
/pgfplots/extra z tick label={⟨TEX code⟩}

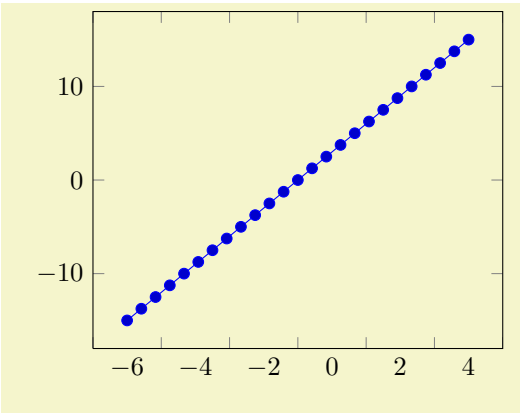
As xticklabel provides code to generate tick labels for each xtick, the key extra x tick label provides code to generate tick labels for every element in extra x ticks.

/pgfplots/extra x tick labels={⟨label list⟩}
/pgfplots/extra y tick labels={⟨label list⟩}
/pgfplots/extra z tick labels={⟨label list⟩}

As xticklabels provides explicit tick labels for each xtick, the key extra x tick labels provides explicit tick labels for every element in extra x ticks.

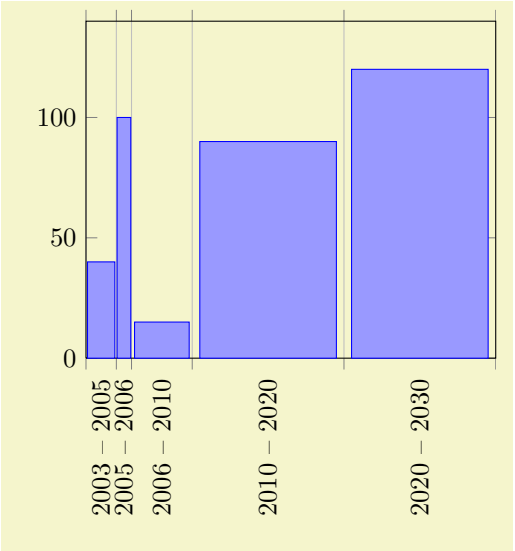
/pgfplots/x tick label as interval=true|false (initially false)
/pgfplots/y tick label as interval=true|false (initially false)
/pgfplots/z tick label as interval=true|false (initially false)

Allows to treat tick labels as intervals; that means the tick positions denote the interval boundaries. If there are n positions, $(n-1)$ tick labels will be generated, one for each interval.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[x tick label as interval]
    \addplot {3*x};
\end{axis}
\end{tikzpicture}
```

This mode enables the use of `\nexttick` inside of `xticklabel` (or `yticklabel`). A common application might be a bar plot.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    ybar interval=0.9,
    x tick label as interval,
    xmin=2003,xmax=2030,
    ymin=0,ymax=140,
    xticklabel={
       $\pgfmathprintnumber{\tick}$
    -- $\pgfmathprintnumber{\nexttick}$},
    xtick=data,
    x tick label style={
        rotate=90,anchor=east,
        /pgf/number format/1000 sep=}
]

    \addplot[draw=blue,fill=blue!40!white]
        coordinates
        {(2003,40) (2005,100) (2006,15)
         (2010,90) (2020,120) (2030,3)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/xminorticks=true|false (initially true)
/pgfplots/yminorticks=true|false (initially true)
/pgfplots/zminorticks=true|false (initially true)
/pgfplots/xmajorticks=true|false (initially true)
/pgfplots/ymajorticks=true|false (initially true)
/pgfplots/zmajorticks=true|false (initially true)
/pgfplots/ticks=minor|major|both|none (initially both)

Enables/disables the small tick lines either for single axis or for all of them. Major ticks are those placed at the tick positions and minor ticks are between tick positions. Please note that minor ticks are automatically disabled if `xtick` is not a uniform range[46].

The key `minor tick length={⟨dimen⟩}` configures the tick length for minor ticks while the `major` variant applies to major ticks. You can configure the appearance using the following styles:

```
\pgfplotsset{every tick/.append style={color=black}} %  applies to major and minor ticks,
\pgfplotsset{every minor tick/.append style={thin}}  %  applies only to minor ticks,
\pgfplotsset{every major tick/.append style={thick}} %  applies only to major ticks.
```

There is also the style "`every tick`" which applies to both, major and minor ticks.

/pgfplots/xtickmin={⟨coord⟩}
/pgfplots/ytickmin={⟨coord⟩}
/pgfplots/ztickmin={⟨coord⟩}

[46] A uniform list means the difference between all elements is the same for linear axis or, for logarithmic axes, log(10).

/pgfplots/xtickmax={⟨*coord*⟩}
/pgfplots/ytickmax={⟨*coord*⟩}
/pgfplots/ztickmax={⟨*coord*⟩}

These keys can be used to modify minimum/maximum values before ticks are drawn. Because this applies to axis discontinuities, it is described on page 174 in Section 4.8.11, "Axis Discontinuities"'.

4.14.2 Tick Alignment: Positions and Shifts

/pgfplots/xtick pos=left|right|both (initially `both`)
/pgfplots/ytick pos=left|right|both (initially `both`)
/pgfplots/ztick pos=left|right|both (initially `both`)
/pgfplots/tick pos=left|right|both

Allows to choose where to place the small tick lines. In the default configuration, this does also affect tick *labels*, see below. The `tick pos` style sets all of them to the same value (aliased by `tickpos`). This option is only useful for boxed axes.

For x, the additional choices `bottom` and `top` can be used which are equivalent to `left` and `right`, respectively. Both are accepted for y.

Changing `tick pos` will also affect the placement of tick labels.

Note that it can also affect the `axis lines` key, although not all combinations make sense. Make sure the settings are consistent.

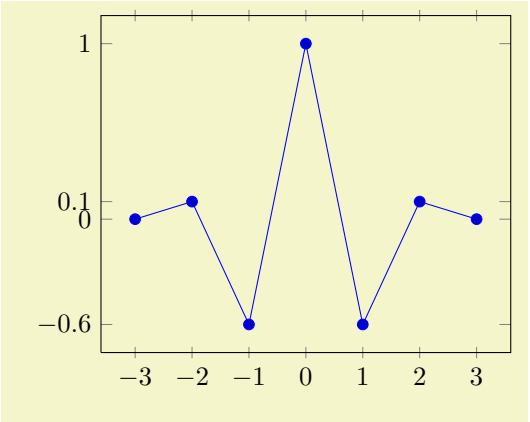
/pgfplots/xticklabel pos=left|right|default (initially `default`)
/pgfplots/yticklabel pos=left|right|default (initially `default`)
/pgfplots/zticklabel pos=left|right|default (initially `default`)
/pgfplots/ticklabel pos=left|right|default (initially `default`)

Allows to choose where to place tick *labels*. The choices `left` and `right` place tick labels either at the left or at the right side of the complete axis. The choice `default` uses the same setting as `xtick pos` (or `ytick pos`). This option is only useful for boxed axes – keep it to `default` for non-boxed figures. The `ticklabel pos` style sets all three of them to the same value.

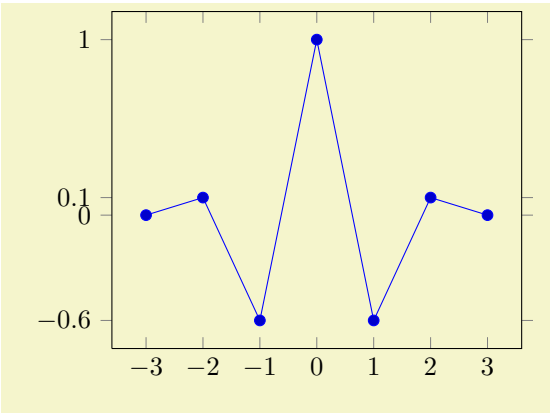
For x, the additional choices `bottom` and `top` can be used which are equivalent to `left` and `right`, respectively. Both are accepted for x.

/pgfplots/xtick align=inside|center|outside (initially `inside`)
/pgfplots/ytick align=inside|center|outside (initially `inside`)
/pgfplots/ztick align=inside|center|outside (initially `inside`)
/pgfplots/tick align=inside|center|outside (initially `inside`)

Allows to change the location of the ticks relative to the axis lines. The `tick align` sets all of them to the same value. Default is "`inside`".

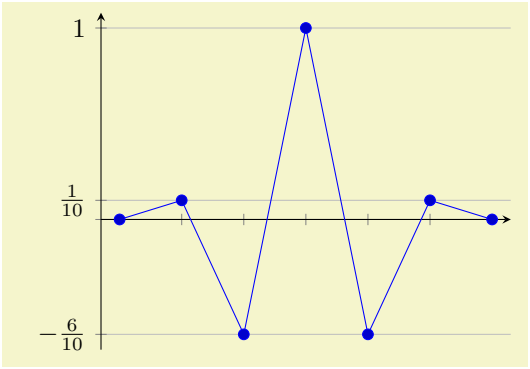



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xtick=data,ytick=data,
    xtick align=center]
\addplot coordinates
    {(-3,0) (-2,0.1) (-1,-0.6)
     (0,1)
     (1,-0.6) (2,0.1) (3,0)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xtick=data,ytick=data,
    ytick align=outside]
\addplot coordinates
    {(-3,0) (-2,0.1) (-1,-0.6)
     (0,1)
     (1,-0.6) (2,0.1) (3,0)};
\end{axis}
\end{tikzpicture}
```

These tick alignment options are set automatically by the `axis x line` and `axis y line` methods (unless one appends an asterisk '`*`'):




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xtick=data,
    axis x line=center,
    xticklabels={,,},
    ytick={-0.6,0,0.1,1},
    yticklabels={
        $-\frac{6}{10}$,,
        $\frac{1}{10}$,$1$},
    ymajorgrids,
    axis y line=left,
    enlargelimits=0.05]
\addplot coordinates
    {(-3,0) (-2,0.1) (-1,-0.6)
     (0,1)
     (1,-0.6) (2,0.1) (3,0)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/xticklabel shift={⟨*dimension*⟩} (initially `empty`)
/pgfplots/yticklabel shift={⟨*dimension*⟩} (initially `empty`)
/pgfplots/zticklabel shift={⟨*dimension*⟩} (initially `empty`)
/pgfplots/ticklabel shift={⟨*dimension*⟩} (initially `empty`)

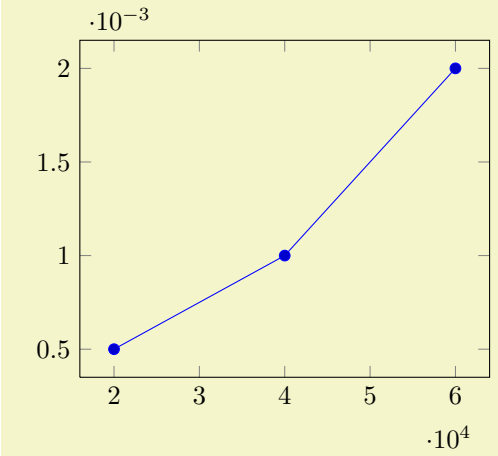
Shifts tick labels in direction of the outer unit normal of the axis by an amount of ⟨*dimension*⟩. The `ticklabel shift` sets the same value for all axes.

This is usually unnecessary as the `anchor` of a tick label already yields enough spacing in most cases.

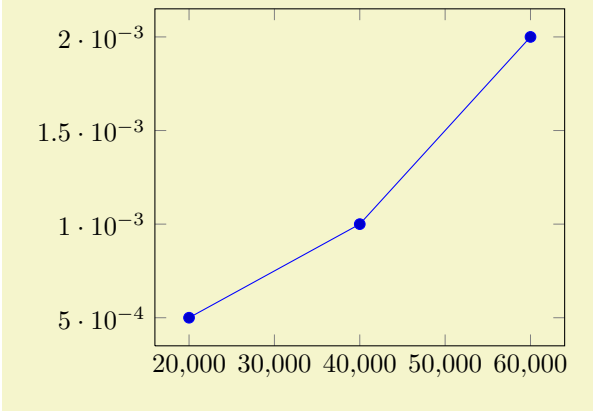
4.14.3 Tick Scaling - Common Factors In Ticks

/pgfplots/scaled ticks=true|false|base 10:⟨*e*⟩|real:⟨*num*⟩|manual:{⟨*label*⟩}{⟨*code*⟩} (initially `true`)
/pgfplots/scaled x ticks=⟨*one of the values*⟩ (initially `true`)
/pgfplots/scaled y ticks=⟨*one of the values*⟩ (initially `true`)
/pgfplots/scaled z ticks=⟨*one of the values*⟩ (initially `true`)

Allows to factor out common exponents in tick labels for *linear axes*. For example, if you have tick labels 20000, 40000 and 60000, you may want to save some space and write $2, 4, 6$ with a separate factor '$\cdot 10^4$'. Use '`scaled ticks=true`' to enable this feature. In case of `true`, tick scaling will be triggered if the data range is either too large or too small (see below).



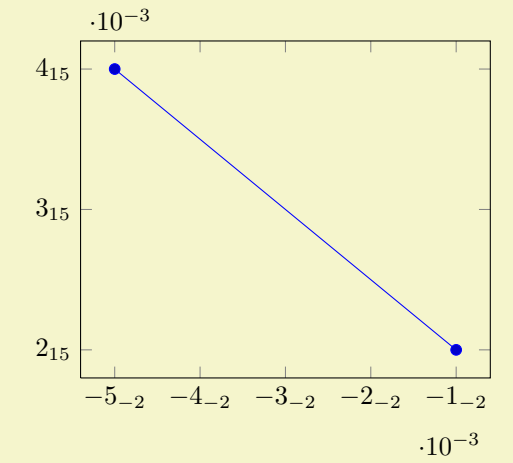

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[scaled ticks=true]
    \addplot coordinates {
        (20000,0.0005)
        (40000,0.0010)
        (60000,0.0020)
    };
\end{axis}
\end{tikzpicture}%
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[scaled ticks=false]
    \addplot coordinates {
        (20000,0.0005)
        (40000,0.0010)
        (60000,0.0020)
    };
\end{axis}
\end{tikzpicture}
```

The `scaled ticks` key is a style which simply sets scaled ticks for both, x and y.

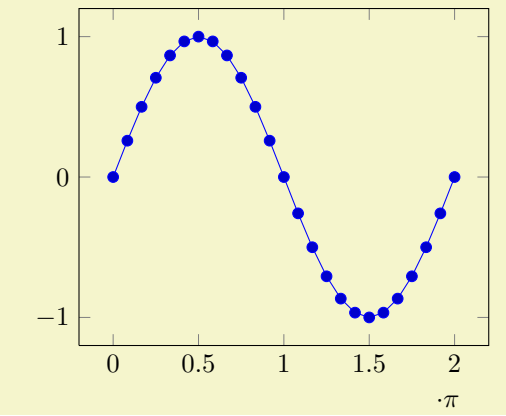
The value `base 10:`$\langle e \rangle$ allows to adjust the algorithm manually. For example, `base 10:3` will divide every tick label by 10^3:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[scaled ticks=base 10:3,
        /pgf/number format/sci subscript]
    \addplot coordinates
        {(-0.00001,2e12) (-0.00005,4e12) };
    \end{axis}
\end{tikzpicture}
```

Here, the `sci subscript` option simply saves space. In general, `base 10:`e will divide every tick by 10^e. The effect is not limited by the "too large or too small" decisions mentioned above.

The value `real:`$\langle num \rangle$ allows to divide every tick by a fixed $\langle num \rangle$. For example, the following plot is physically ranged from 0 to 2π, but the tick scaling algorithm is configured to divide every tick label by π.



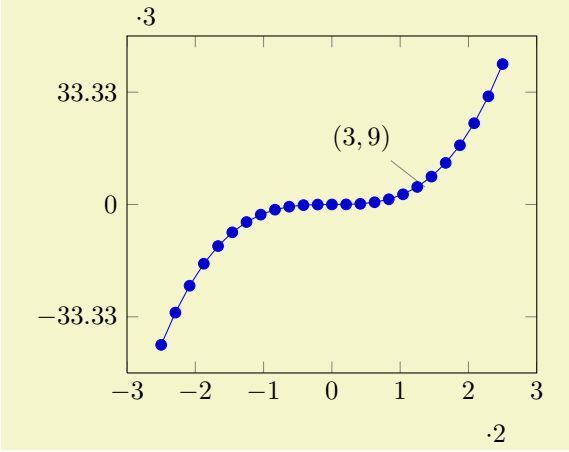

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        xtick={0,1.5708,...,10},
        domain=0:2*pi,
        scaled x ticks={real:3.1415},
        xtick scale label code/.code={$\cdot \pi$}]
    \addplot {sin(deg(x))};
    \end{axis}
\end{tikzpicture}
```

Setting `scaled ticks=real:`⟨*num*⟩ also changes the `tick scale label code` to

```
\pgfkeys{/pgfplots/xtick scale label code/.code=
    {$\pgfkeysvalueof{/pgfplots/tick scale binop} \pgfmathprintnumber{#1}$}}.
```

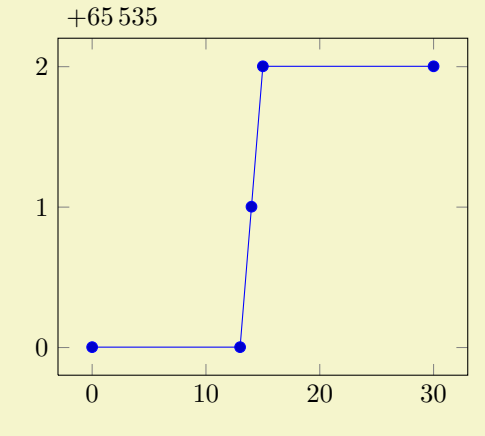
The key `tick scale binop` is described below, it is set initially to `\cdot`.

A further – not very useful – example is shown below. Every x tick label has been divided by 2, every y tick label by 3.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        scaled x ticks=real:2,
        scaled y ticks=real:3]
    \addplot {x^3};
    \node[pin=135:{$(3,9)$}] at (axis cs:3,9) {};
    \end{axis}
\end{tikzpicture}
```

The last option, `scaled ticks=manual:{`⟨*label*⟩`}{`⟨*code*⟩`}` allows even more customization. It allows *full control* over the displayed scaling label *and* the scaling code: ⟨*label*⟩ is used as-is inside of the tick scaling label while ⟨*code*⟩ is supposed to be a one-argument-macro which scales each tick. Example:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    %  warning: the '%' signs are necessary (?)
    scaled y ticks=manual:{$+65\,535$}{%
        \pgfmathparse{#1-65535}%
    },
    yticklabel style={
        /pgf/number format/fixed,
        /pgf/number format/precision=1},
]
\addplot coordinates {
    (0, 65535)
    (13, 65535)
    (14, 65536)
    (15, 65537)
    (30, 65537)
};
\end{axis}
\end{tikzpicture}
```

The example uses `$+65\,535$` as tick scale label content. Furthermore, it defines the customized tick

label formula $y - (+6.5535 \cdot 10^4) = y - 65535$ to generate y tick labels.

The ⟨*label*⟩ can be arbitrary. It is completely in user control. The second argument, ⟨*code*⟩ is supposed to be a one-argument-macro in which `#1` is the current tick position in floating point representation. The macro is expected to assign `\pgfmathresult` (as a number). The PGF manual [5] contains detailed documentation about its math engine.

This feature may also be used do transform coordinates in case they can't be processed with PGFPLOTS: transform them and supply a proper tick scaling method such that tick labels represent the original range.

If ⟨*label*⟩ is empty, the tick scale label won't be drawn (and no space will be occupied).

Tick scaling does *not* work for logarithmic axes.

`/pgfplots/xtick scale label code/.code={⟨...⟩}`
`/pgfplots/ytick scale label code/.code={⟨...⟩}`
`/pgfplots/ztick scale label code/.code={⟨...⟩}`

Allows to change the default code for scaled tick labels. The default is

```
\pgfplotsset{
    xtick scale label code/.code={$\cdot 10^{#1}$}
}
```

More precisely, it is

```
\pgfplotsset{
    xtick scale label code/.code={$\pgfkeysvalueof{/pgfplots/tick scale binop} 10^{#1}$}
}
```

and the initial value of `tick scale binop` is `\cdot`, but it can be changed to `\times` if desired.

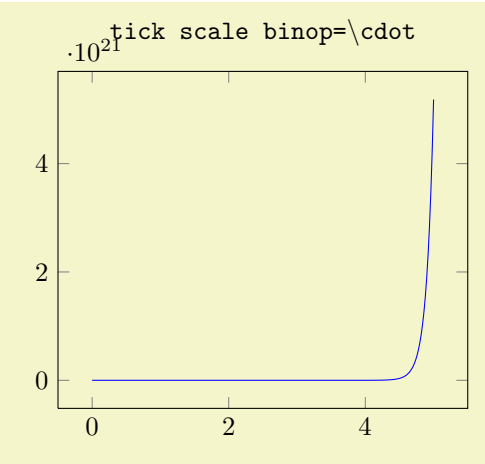
If the code is empty, no tick scale label will be drawn (and no space is consumed).

`/pgfplots/tick scale label code/.code={⟨...⟩}`

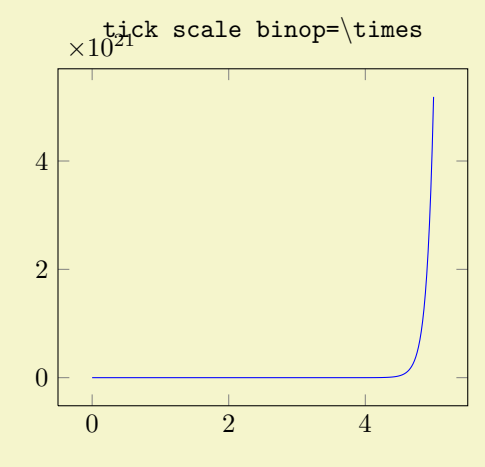
A style which sets `xtick scale label code` and those for y and z.

`/pgfplots/tick scale binop={⟨`*TEX math operator*`⟩}` (initially `\cdot`)

Sets the binary operator used to display tick scale labels.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title=\texttt{tick scale
        binop=\textbackslash cdot}]
\addplot
    [mark=none,blue,samples=250,
     domain=0:5]
    {exp(10*x)};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title=\texttt{tick scale
        binop=\textbackslash times},
    tick scale binop=\times]
\addplot
    [mark=none,blue,samples=250,
     domain=0:5]
    {exp(10*x)};
\end{axis}
\end{tikzpicture}
```

/pgfplots/scale ticks below={⟨*exponent*⟩} (initially -1)

Allows fine tuning of the 'scaled ticks' algorithm: if the axis limits are of magnitude 10^e and $e <$⟨*exponent*⟩, the common prefactor 10^e will be factored out. The default is

/pgfplots/scale ticks above={⟨*exponent*⟩} (initially 3)

Allows fine tuning of the 'scaled ticks' algorithm: if the axis limits are of magnitude 10^e and $e >$⟨*exponent*⟩, the common prefactor 10^e will be factored out.

4.14.4 Tick Fine-Tuning

The tick placement algorithm depends on a number of parameters which can be tuned to get better results.

/pgfplots/max space between ticks={⟨*number*⟩} (initially 35)

Configures the maximum space between adjacent ticks in full points. The suffix "pt" has to be omitted and fractional numbers are not supported.

/pgfplots/try min ticks={⟨*number*⟩} (initially 4)

Configures a loose lower bound on the number of ticks. It should be considered as a suggestion, not a tight limit. This number will increase the number of ticks if 'max space between ticks' produces too few of them.

The total number of ticks may still vary because not all fractional numbers in the axis' range are valid tick positions.

/pgfplots/try min ticks log={⟨*number*⟩} (initially 3)

The same as try min ticks, but for logarithmic axis.

/pgfplots/tickwidth={⟨*dimension*⟩} (initially 0.15cm)
/pgfplots/major tick length={⟨*dimension*⟩} (initially 0.15cm)

Sets the length of major tick lines.

/pgfplots/subtickwidth={⟨*dimension*⟩} (initially 0.1cm)
/pgfplots/minor tick length={⟨*dimension*⟩} (initially 0.1cm)

Sets the length of minor tick lines.

/pgfplots/xtick placement tolerance (initially 0.05pt)
/pgfplots/ytick placement tolerance (initially 0.05pt)
/pgfplots/ztick placement tolerance (initially 0.05pt)

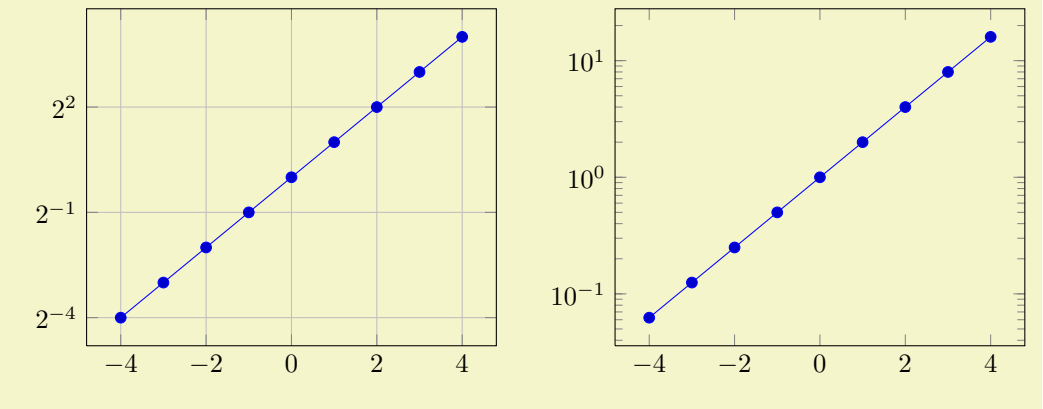
Tick lines and labels will be placed if they are no more than this tolerance beyond the axis limits. This threshold should be chosen such that it does not produce visible differences while still providing fault tolerance.

The threshold is given in paper units of the final figure.

/pgfplots/log basis x={⟨*number*⟩} (initially `empty`)
/pgfplots/log basis y={⟨*number*⟩} (initially `empty`)
/pgfplots/log basis z={⟨*number*⟩} (initially `empty`)

Allows to change the logarithms used for logarithmic axes.

Changing to a different log basis is nothing but a scale. However, it also changes the way tick labels are displayed: they will also be shown in the new basis.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{semilogyaxis}[log basis y=2,grid=major,samples at={-4,...,4}]
        \addplot {2^x};
    \end{semilogyaxis}
\end{tikzpicture}
~
\begin{tikzpicture}
    \begin{semilogyaxis}[log basis y=10,samples at={-4,...,4}]
        \addplot {2^x};
    \end{semilogyaxis}
\end{tikzpicture}
```

The initial setting is '`log basis x=`' which defaults to: the natural logarithm for any coordinates (basis exp(1)), and the logarithm base 10 for the display of tick labels.

If the log basis is changed to something different than the empty string, the chosen logarithm will be applied to any input coordinate (if the axis scale is log as well) and tick labels will be displayed in this basis.

In other words: usually, you see log axes base 10 and that's it. It is only interesting for coordinate filters: the initial setting (with empty ⟨*number*⟩) uses coordinate lists basis e although the display will use basis 10 (i.e. it is rescaled). Any non-empty value ⟨*number*⟩ causes both, coordinate lists *and* display to use ⟨*number*⟩ as basis for the logarithm. The javascript code of the `clickable` library will always use the *display* basis (which is usally 10) when it computes slopes.

Technical remarks. When `log basis x` is used, the style `log basis ticks={`⟨*axis char*⟩`}` will be installed (in this case `log basis ticks=x`). This style in turn will change `log number format code`.

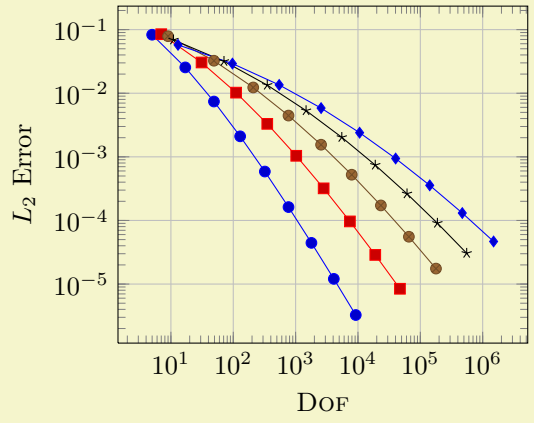
Please note that `xtickten` will be used differently now: it will provide the desired ticks in the new basis! Despite the misleading name "`ten`", `xtickten={1,2,3,4}` will yield ticks at $2^1, 2^2, 2^3, 2^4$ if `log basis x=2` has been set.

4.15 Grid Options

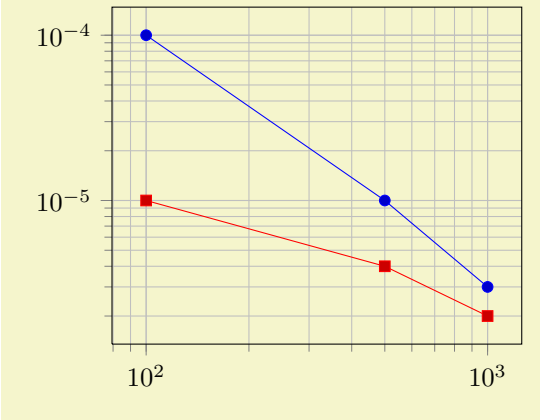
/pgfplots/xminorgrids=true|false (initially `false`)
/pgfplots/yminorgrids=true|false (initially `false`)
/pgfplots/zminorgrids=true|false (initially `false`)
/pgfplots/xmajorgrids=true|false (initially `false`)
/pgfplots/ymajorgrids=true|false (initially `false`)
/pgfplots/zmajorgrids=true|false (initially `false`)
/pgfplots/grid=minor|major|both|none (initially `false`)

Enables/disables different grid lines. Major grid lines are placed at the normal tick positions (see `xmajorticks`) while minor grid lines are placed at minor ticks (see `xminorticks`).

This example employs the coordinates defined on page 17.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel={\textsc{Dof}},
    ylabel={$L_2$ Error},
    grid=major
]
%  see above for this macro:
\plotcoords
\end{loglogaxis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    grid=both,
    tick align=outside,
    tickpos=left]
\addplot coordinates
    {(100,1e-4) (500,1e-5) (1000,3e-6)};
\addplot coordinates
    {(100,1e-5) (500,4e-6) (1000,2e-6)};
\end{loglogaxis}
\end{tikzpicture}
```

Grid lines will be drawn before tick lines are processed, so ticks will be drawn on top of grid lines. You can configure the appearance of grid lines with the styles

```
\pgfplotsset{grid style={help lines}} %  modifies the style 'every axis grid'
\pgfplotsset{minor grid style={color=blue}} %  modifies the style 'every minor grid'
\pgfplotsset{major grid style={thick}} % modifies the style 'every major grid'
```

4.16 Custom Annotations

Often, one may want to add custom drawing elements or descriptive texts to an axis. These graphical elements should be associated to some logical coordinate, grid point, or perhaps they should just be placed somewhere into the axis.

PGFPLOTS assists with the following ways when it comes to annotations:

1. You can explicitly provide any Ti*k*Z instruction like `\draw ... ;` into the axis. Here, the `axis cs` allows to provide coordinates of PGFPLOTS.

   Furthermore, `rel axis cs` allows to position Ti*k*Z elements relatively (like "50% of the axis' width).

2. PGFPLOTS can automatically generate nodes at every coordinate using its `nodes near coords` feature.

3. PGFPLOTS allows you to place nodes on a plot, using the `\addplot ... node[pos=`⟨*fraction*⟩`] {};` feature.

This section explains all of the approaches, except for the `nodes near coords` feature which is documented in its own section.

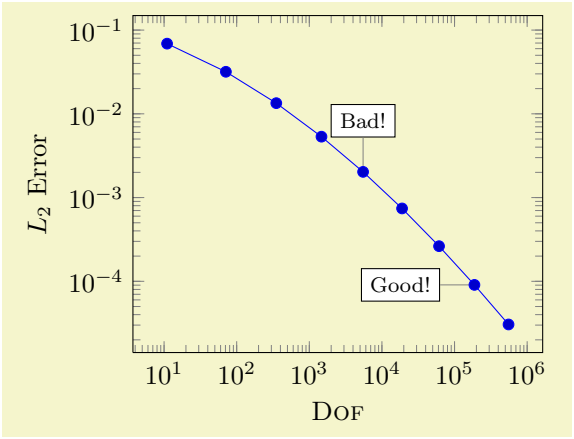
4.16.1 Accessing Axis Coordinates in Graphical Elements

Coordinate system `axis cs`

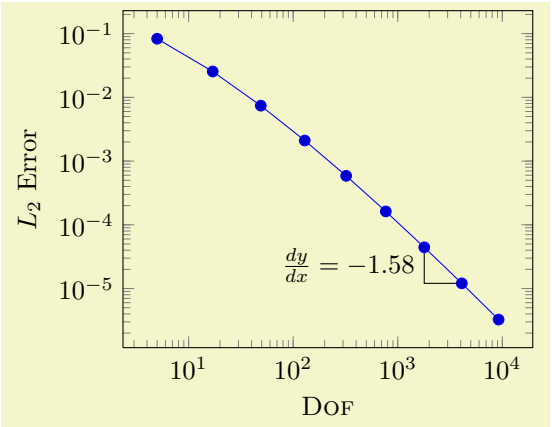
PGFPLOTS provides a new coordinate system for use inside of an axis, the "axis coordinate system", `axis cs`.

It can be used to draw any Ti*k*Z-graphics at axis coordinates. It is used like

```
\draw
   (axis cs:18943,2.873391e-05)
|- (axis cs:47103,8.437499e-06);
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\tikzstyle{every pin}=[fill=white,
    draw=black,
    font=\footnotesize]
\begin{tikzpicture}
    \begin{loglogaxis}[
        xlabel={\textsc{Dof}},
        ylabel={$L_2$ Error}]

    \addplot coordinates {
        (11,     6.887e-02)
        (71,     3.177e-02)
        (351,    1.341e-02)
        (1471,   5.334e-03)
        (5503,   2.027e-03)
        (18943,  7.415e-04)
        (61183,  2.628e-04)
        (187903, 9.063e-05)
        (553983, 3.053e-05)
    };

    \node[coordinate,pin=above:{Bad!}]
        at (axis cs:5503,2.027e-03) {};
    \node[coordinate,pin=left:{Good!}]
        at (axis cs:187903,9.063e-05)    {};
    \end{loglogaxis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[
    xlabel=\textsc{Dof},
    ylabel=$L_2$ Error
]
\draw
        (axis cs:1793,4.442e-05)
    |-  (axis cs:4097,1.207e-05)
    node[near start,left]
    {$\frac{dy}{dx} = -1.58$};

\addplot coordinates {
    (5,    8.312e-02)
    (17,   2.547e-02)
    (49,   7.407e-03)
    (129,  2.102e-03)
    (321,  5.874e-04)
    (769,  1.623e-04)
    (1793, 4.442e-05)
    (4097, 1.207e-05)
    (9217, 3.261e-06)
};
\end{loglogaxis}
\end{tikzpicture}
```

Whenever you draw additional graphics, consider using `axis cs`! It applies any custom transformations (including `symbolic x coords`), logarithms, data scaling transformations or whatever PGFPLOTS usually does and provides a low level PGF coordinate as result.

In case you need only one component (say, the y component) of such a vector, you can use the `\pgfplotstransformcoordinatey` command, see Section 8.4 for details about basic level access.

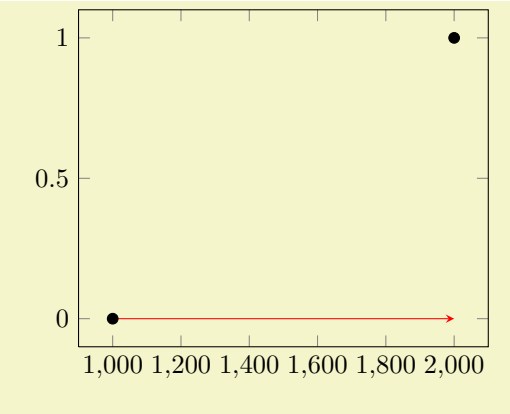
The result of `axis cs` is always an absoute position inside of an axis. This means, in particular, that *adding* two points has unexpected effects: the expression `(axis cs:0,0) ++ (axis cs:1,0)` is not necessarily the same as `(axis cs:1,0)`. The background for such unexpected effects is that PGFPLOTS applies a *shifted* linear transformation which moves the origin in order to support its high accuracy and high data range (compare the documentation of `disabledatascaling`).

In order to express *relative* positions (or lengths), you need to use `axis direction cs`.

Coordinate system `axis direction cs`

While `axis cs` allows to supply *absolute positions*, `axis direction cs` supplies *directions*. It allows to express *relative* positions, includings lengths and dimensions, by means of axis coordinates.

As noted in the documentation for `axis cs`, adding two coordinates by means of the Ti*k*Z `++` operator may have unexpected effects. The correct way for `++` operations is `axis direction cs`:



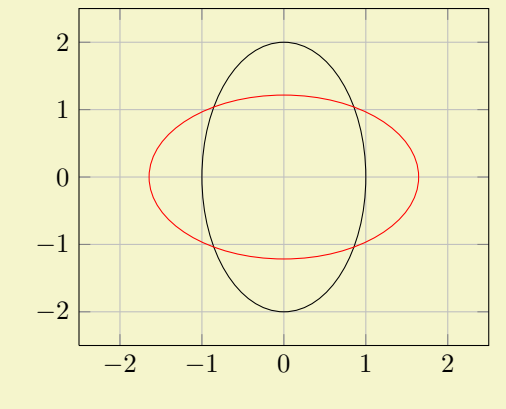

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \draw[red,-stealth]
        (axis cs:1000,0)
        -- %  = line-to
         ++ %  = calculate a vector sum
        (axis direction cs:1000,0);

    \addplot [only marks,mark=*]
        coordinates { (1000,0) (2000,1) };
\end{axis}
\end{tikzpicture}
```

Here, the target of the red arrow is the position `(axis cs:2000,0)` as expected.

Using relative positions is mainly useful for linear axes. Applying this command to log-axes might still work, but it requires more care.

One use-case is to supply lengths – for example in order to support `circle` or `ellipse` paths. The correct way to draw an ellipse in PGFPLOTS would be to specify the two involved radii by means of two `(axis direction cs:`$\langle x,y\rangle$`)` expressions. In general, this is possible if you use the basic level macros `\pgfpathellipse` and `\pgfplotspointaxisdirectionxy`. Please refer to the documentation of `\pgfplotspointaxisdirectionxy` for two examples of drawing arbitrary ellipses by means of this method.

Since drawing circles and ellipses inside of an axis is a common use-case, PGFPLOTS automatically communicates its coordinate system transformations to Ti*k*Z: whenever you write `\draw ellipse[x radius=`$\langle x\rangle$`,y radius=`$\langle y\rangle$`]`, the arguments $\langle x\rangle$ and $\langle y\rangle$ are considered to be PGFPLOTS direction vectors and are handed over to `axis direction cs`. Consequently, ellipses with axis parallel radii are straightforward and use the normal Ti*k*Z syntax:

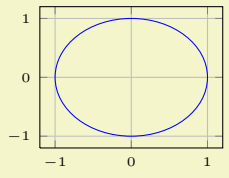



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \pgfplotsset{compat=1.5.1} !
\begin{tikzpicture}
\begin{axis}[
    xmin=-2.5,   xmax=2.5,
    ymin=-2.5,   ymax=2.5,
    xtick={-2,-1,0,1,2},
    ytick={-2,-1,0,1,2},
    grid=major,
]
    %  standard tikz syntax:
    \draw[black] (axis cs:0,0)
        ellipse [
            x radius=1, y radius=2];

    \draw[red]   (axis cs:0,0)
        ellipse [rotate=90,
            x radius=1, y radius=2];
    %  see \pgfplotspointaxisdirectionxy
    %  for arbitrary ellipses
\end{axis}
\end{tikzpicture}
```

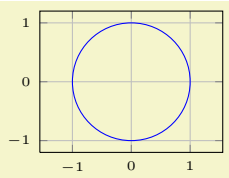
Here, the two ellipses are specified as usual in Ti*k*Z. PGFPLOTS ensures that all necessary transformations are applied to the two radii. Note that PGFPLOTS usually has different axis scales for x and y. As a consequence, the rotated red ellipse does not fit into the axis lines; we would need to use `axis equal` to allow properly rotated ellipses.

Attention: this modification to circles and ellipses requires `\pgfplotsset{compat=1.5.1}`.

The same applies to circles: in the standard view, a circle with `radius`=r will appear as an ellipse due to the different axis scales. Supplying `axis equal` results in true circles:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \pgfplotsset{compat=1.5.1} !
\begin{tikzpicture}
\begin{axis}[tiny,enlargelimits,
    xmin=-1,xmax=1,
    ymin=-1,ymax=1,
    xtick={-1,0,1},
    ytick={-1,0,1},
    grid=major,
]
    \draw[blue] (axis cs:0,0) circle[radius=1];
\end{axis}
\end{tikzpicture}
```

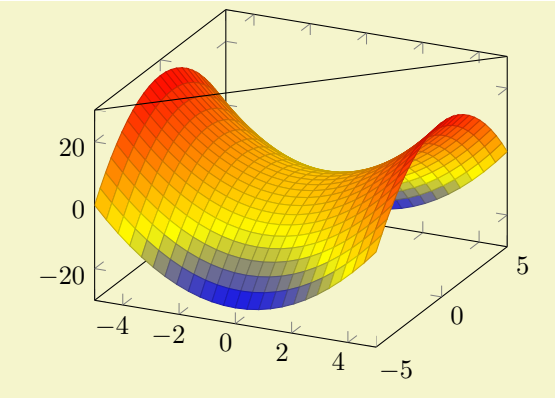



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[tiny,enlargelimits,
    axis equal,
    xmin=-1,xmax=1,
    ymin=-1,ymax=1,
    xtick={-1,0,1},
    ytick={-1,0,1},
    grid=major,
]
    \draw[blue] (axis cs:0,0) circle[radius=1];
\end{axis}
\end{tikzpicture}
```

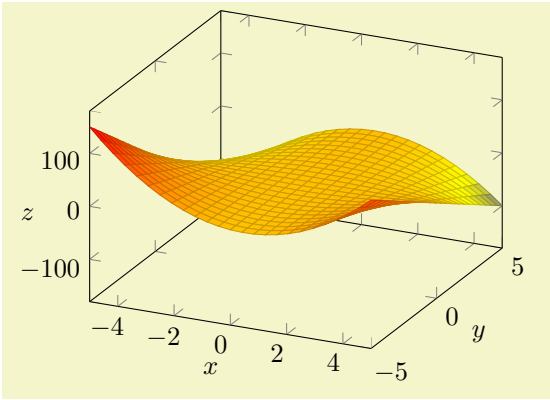
In case you need access to `axis direction cs` inside of math expressions, you can employ the additional math function `transformdirectionx`. It does the same as `axis direction cs`, but only in x direction. The result of `transformdirectionx` is a dimensionless unit which can be interpreted relative to the current PGF x unit vector e_x (see the documentation of `\pgfplotstransformdirectionx` for details). There are the math commands `transformdirectionx`, `transformdirectiony`, and (if the axis is three–dimensional) `transformdirectionz`. Each of them defines `\pgfmathresult` to contain the result of `\pgfplotstransformdirectionx` (or its variants for y and z, respectively).

Coordinate system `rel axis cs`

The "relative axis coordinate system", `rel axis cs`, uses the complete axis vectors as units. That means '$x = 0$' denotes the point on the lower x axis range and '$x = 1$' the point on the upper x axis range (see the remark below for `x dir=reverse`).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}

    \addplot3[surf] {x^2 - y^2};
    \draw  (rel axis cs:0,0,1)
        -- (rel axis cs:1,1,1);
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xlabel=$x$,
    ylabel=$y$,
    zlabel=$z$,
    every axis x label/.style={
        at={(rel axis cs:0.5,-0.15,-0.15)}},
    every axis y label/.style={
        at={(rel axis cs:1.15,0.5,-0.15)}},
    every axis z label/.style={
        at={(rel axis cs:-0.15,-0.15,0.5)}},
]

    \addplot3[surf] {x*(1-x)*y};
\end{axis}
\end{tikzpicture}
```

Points identified by `rel axis cs` use the syntax

`(rel axis cs:`$\langle x \rangle$`,`$\langle y \rangle$`)` or

`(rel axis cs:`$\langle x \rangle$`,`$\langle y \rangle$`,`$\langle z \rangle$`)`

where $\langle x \rangle$, $\langle y \rangle$ and $\langle z \rangle$ are coordinates or constant mathematical expressions. The second syntax is only available in three dimensional axes.

There is one specialty: if you reverse an axis (with `x dir=reverse`), points provided by `rel axis cs` will be *unaffected* by the axis reversal. This is intended to provide consistent placement even for reversed axes. Use `allow reversal of rel axis cs=false` to disable this feature.

There is also a low–level interface to access the transformations and coordinates, see Section 8 on page 359.

Predefined node `current plot begin`

This coordinate will be defined for every plot and can be used is $\langle$*trailing path commands*$\rangle$ or after a plot. It is the first coordinate of the current plot.

Predefined node `current plot end`

This coordinate will be defined for every plot. It is the last coordinate of the current plot.

`/pgfplots/allow reversal of rel axis cs=true|false` (initially `true`)

A fine-tuning key which specifies how to deal with `x dir=reverse` and `rel axis cs` and `ticklabel cs`.

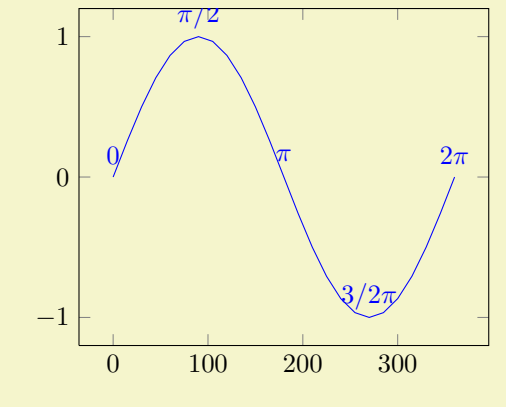
The initial configuration `true` means that points placed with `rel axis cs` and/or `ticklabel cs` will be at the same position inside of the axes even if its ordering has been reversed. The choice `false` will disable the special treatment of `x dir=reverse`.

4.16.2 Placing Nodes on Coordinates of a Plot

The \addplot command is not only used for PGFPLOTS, it can also carry additional drawing instructions which are handed over to Ti*k*Z after the plot's path is complete. Among others, this can be used to add further nodes on the path.

/tikz/pos={⟨*fraction*⟩}

The ⟨*fraction*⟩ identifies a part of the recently completed plot if it is used before the trailing semicolon:



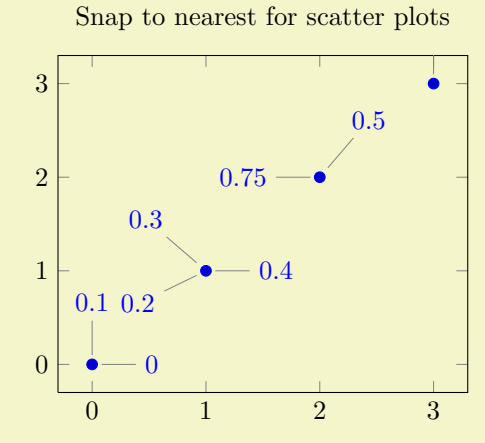

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[blue,domain=0:360] {sin(x)}
    [yshift=8pt]
        node[pos=0] {$0$}
        node[pos=0.25] {$\pi/2$}
        node[pos=0.5] {$\pi$}
        node[pos=0.75] {$3/2\pi$}
        node[pos=1] {$2\pi$}
    ;
    \end{axis}
\end{tikzpicture}
```

Here, the `[yshift=8pt]` tells Ti*k*Z to shift all following nodes upwards. The `node[pos=0] {$0$}` instruction tells Ti*k*Z to add a text node at 0% of the recently completed plot. The relative position 0% (`pos=0`) refers to the first coordinate which has been seen by PGFPLOTS, and 100% (`pos=1`) refers to the last coordinate. Any value between 0 and 1 is interpolated in-between. Note that all these nodes belong to the plot's visualization (which is terminated by the semicolon). Consequently, all these nodes inherit the same graphic settings (like color choices).

The position on the plot is computed by PGFPLOTS using *logical* coordinates. That means: it computes the overall length of the curve before the curve is projected to screen coordinates and identifies the desired position[47]. Afterwards, it projects the final position to screen coordinates. Thus, the position identifies a location on the plot which is always the same, even in case of a rotated three-dimensional axis. PGFPLOTS will linearly interpolate the fraction between successive coordinates.

Valid choices for ⟨*fraction*⟩ are any numbers in the range $[0, 1]$.

Note that the precise meaning of `pos` depends on the current plot handler: for most plot handlers, it defaults to linear interpolation (as in the examples above). For `only marks`, `scatter`, `ybar`, `xbar`, `ybar interval`, and `xbar interval`, it snaps to the nearest encountered coordinate. In this context, "snap to nearest" means that `pos`=p refers to the coordinate with index $i = \text{round}(p \cdot N)$ where N is the total number of points:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[title=Snap to nearest for scatter plots]
\addplot+[only marks]
    coordinates {(0,0) (1,1) (2,2) (3,3)}
    node[pos=0,   pin=0  :0   ] {}
    node[pos=0.1, pin=90 :0.1 ] {}
    node[pos=0.2, pin=200:0.2 ] {}
    node[pos=0.3, pin=135:0.3 ] {}
    node[pos=0.4, pin=0  :0.4 ] {}
    node[pos=0.5, pin=60 :0.5 ] {}
    node[pos=0.75,pin=180:0.75] {}
    node[pos=1,   pin=90 :1   ] {}
;
\end{axis}
\end{tikzpicture}
```

the previous example shows that `pos`=p maps to one of the four available coordinates, namely the one

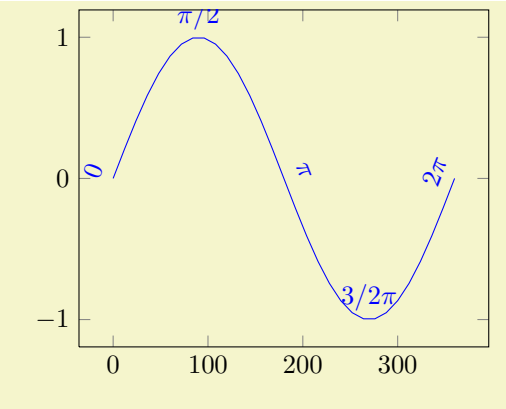
[47] This can be a time-consuming process. Consider using the external library if you have lots of such figures.

whose index is closest to $p \cdot N$. Note that in such a case, the distance between coordinates is irrelevant – only the coordinate index counts.

Note that the fact that PGFPLOTS uses *logical* coordinates to compute the target positions can produce unexpected effects if x and y axis operate on a different scales. Suppose, for example, that x is always of order 10^3 whereas y is of order 10^{-3}. In such a scenario, the y coordinate have no significant contribution to the curve's length – although the rescaled axes clearly show "significant" y dynamics. Consider using `axis equal` together with `pos` to produce comparable effects.

`/tikz/sloped` (initially `false`)

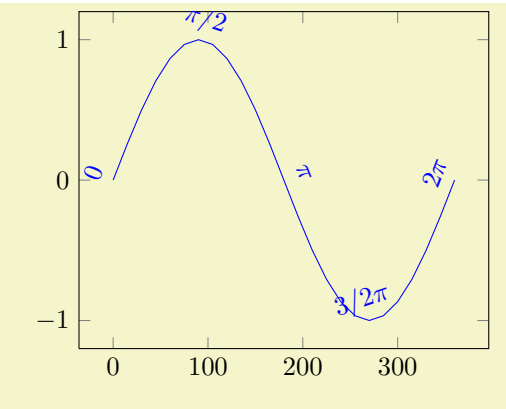
Providing the Ti*k*Z key `sloped` to a node identified by `pos` causes it to be rotated such that it adapts to the plot's gradient.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot[blue,domain=0:360,samples=31] {sin(x)}
    [every node/.style={yshift=8pt},sloped]
        node[pos=0] {$0$}
        node[pos=0.25] {$\pi/2$}
        node[pos=0.5] {$\pi$}
        node[pos=0.75] {$3/2\pi$}
        node[pos=1] {$2\pi$}
    ;
    \end{axis}
\end{tikzpicture}
```

Note that the sequence in which `sloped` and shift transformations are applied is important: if shifts are applied first (as would be the case without the `every node/.style` construction), the shifts do not respect the rotation. If `sloped` is applied first, any subsequent shifts will be applied in the *rotated* coordinates. Thus, the case `every node/.style={yshift=8pt}` shifts every node by `8pt` in direction of its normal vector.

The `sloped` transformation is based on the gradient between two points (the two points adjacent to `pos`). Consequently, it inherits any sampling weaknesses. To see this, consider the example above with a different number of samples:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  same as above with different number of samples
\begin{tikzpicture}
    \begin{axis}
    \addplot[blue,domain=0:360,samples=25] {sin(x)}
    [every node/.style={yshift=8pt},sloped]
        node[pos=0] {$0$}
        node[pos=0.25] {$\pi/2$}
        node[pos=0.5] {$\pi$}
        node[pos=0.75] {$3/2\pi$}
        node[pos=1] {$2\pi$}
    ;
    \end{axis}
\end{tikzpicture}
```

Here, the two extreme points have small slopes due to the sampling. While this does not seriously affect the quality of the plot, it has a huge impact on the transformation matrizes. Keep this in mind when you work with `sloped` (perhaps it even helps to add a further `rotate` argument).

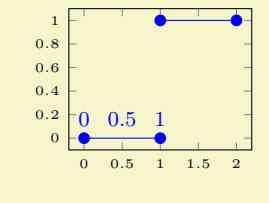
`/tikz/allow upside down=true|false` (initially `false`)

If `/tikz/sloped` is enabled and one has some difficult line plot, the transformation may cause nodes to be drawn upside down. The default configuration `allow upside down=false` will switch the rotation matrix, whereas `allow upside down` allows this case.

`/tikz/pos segment={`⟨*segment index*⟩`}` (initially `empty`)

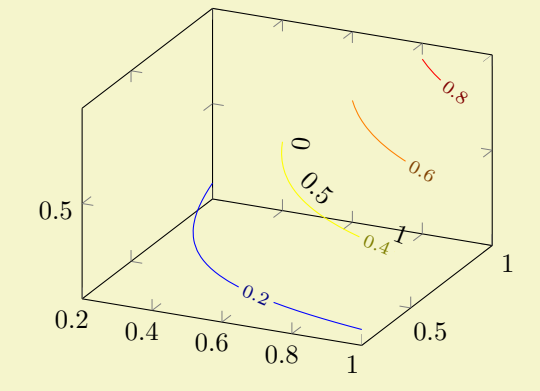
Occasionally, one has a single plot which consists of multiple segments (like those generated by `empty line=jump` or `contour prepared`). The individual segments will typically have different lengths, so it is tedious to identify a position on one of these segments.

If `pos segment=`⟨*segment index*⟩ is non-empty, the key `pos=`⟨*fraction*⟩ is interpreted relatively to the provided segment rather than the whole plot. The argument ⟨*segment index*⟩ is an integer, where 0 denotes the first segment.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[tiny]
\addplot coordinates {
    (0,0) (1,0)

    (1,1) (2,1)}
    [pos segment=0,yshift=7pt,font=\footnotesize]
    node[pos=0] {0}
    node[pos=0.5] {0.5}
    node[pos=1] {1};
\end{axis}
\end{tikzpicture}
```

Here, the plot has two segments. However, all three annotation nodes are placed with `pos segment=0`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
\addplot3[contour gnuplot,domain=0:1] {x*y}
    [sloped,
     allow upside down,
     pos segment=2,
     every node/.style={yshift=7pt}]
    node[pos=0] {0}
    node[pos=0.5] {0.5}
    node[pos=1] {1}
    ;

\end{axis}
\end{tikzpicture}
```

This plot has four segments (which are generated automatically by the plot handler). The annotation nodes are placed on the third segment, where `sloped` causes them to be rotated, `allow upside down` improves the rendering of the '0', and `every node/.style` install a shift in direction of the normal vector (see the documentation of `sloped` for details).

Occasionally, one wants to place a node using `pos` *and* one wants to typeset the coordinates of that point inside of the node. This can be accomplished using `\pgfplotspointplotattime`:

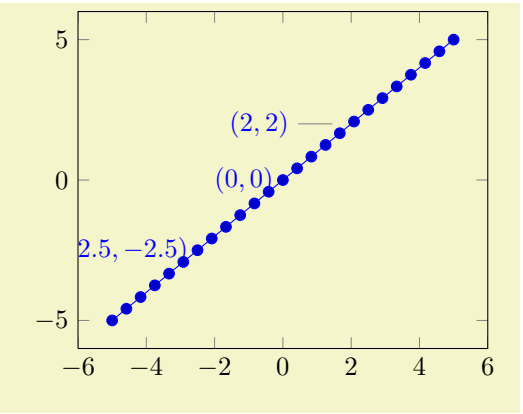
`\pgfplotspointplotattime`
`\pgfplotspointplotattime{`⟨*fraction*⟩`}`

This command is part of the `pos={`⟨*fraction*⟩`}` implementation: it defines the current point of PGF to ⟨*fraction*⟩ of the current plot. Without an argument in curly braces, `\pgfplotspointplotattime` will take the current argument of the `pos` key.

Thus, the command computes the basic PGF coordinates – but it also returns the *logical* coordinates of the resulting point into the following keys:

`/data point/x` (no value)
`/data point/y` (no value)
`/data point/z` (no value)

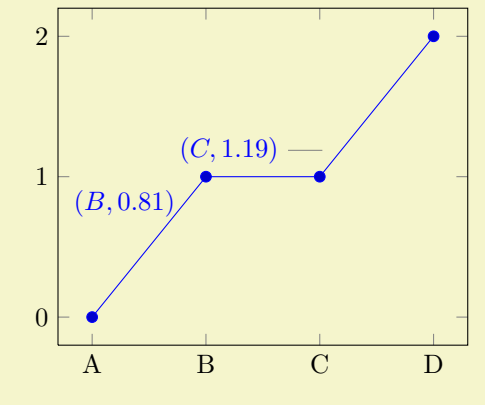
After `\pgfplotspointplotattime` returns, these macros contain the x, y, and z coordinates of the resulting point. They can be used by means of `\pgfkeysvalueof{/data point/x}`, for example.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
\addplot {x}
    [left,/pgf/number format/relative=0]
node[pos=0.5] {%
  \pgfplotspointplotattime
  $(\pgfmathprintnumber
          {\pgfkeysvalueof{/data point/x}},
    \pgfmathprintnumber
        {\pgfkeysvalueof{/data point/y}})$
}
node[pos=0.25] {%
  \pgfplotspointplotattime
  $(\pgfmathprintnumber
          {\pgfkeysvalueof{/data point/x}},
    \pgfmathprintnumber
        {\pgfkeysvalueof{/data point/y}})$
}
node[pos=0.7,pin=180:{%
  \pgfplotspointplotattime{0.7}
  $(\pgfmathprintnumber
          {\pgfkeysvalueof{/data point/x}},
    \pgfmathprintnumber
        {\pgfkeysvalueof{/data point/y}})$
}] {}
    ;
    \end{axis}
\end{tikzpicture}
```

In the example above, three nodes have been placed using different `pos=` arguments. Invoking `\pgfplotspointplotattime` inside of the associated node's body checks if `pos` already has a value and uses that value. The third node displays the coordinates inside of a `pin`. Due to internals of Ti*k*Z, the `pin` knows nothing about the `pos=0.7` argument of its enclosing `node`, so we need to replicate the '`0.7`' argument for `\pgfplotspointplotattime{0.7}`. The `/pgf/number format/relative=0` style causes the number printer to round relative to 10^0 (compare against the same example without this style).

In case you have `symbolic x coords` (or any other `x coord inv tafo` which produces non-numeric results), the output stored in `/data point/x` will be the symbolic expression:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[symbolic x coords={A,B,C,D}]
\addplot coordinates {(A,0) (B,1) (C,1) (D,2)}
    [left]
node[pos=0.3] {%
  \pgfplotspointplotattime
  $(\pgfkeysvalueof{/data point/x},
    \pgfmathprintnumber
        {\pgfkeysvalueof{/data point/y}})$
}
node[pos=0.7,pin=180:{%
  \pgfplotspointplotattime{0.7}
  $(\pgfkeysvalueof{/data point/x},
    \pgfmathprintnumber
        {\pgfkeysvalueof{/data point/y}})$
}] {}
    ;
    \end{axis}
\end{tikzpicture}
```

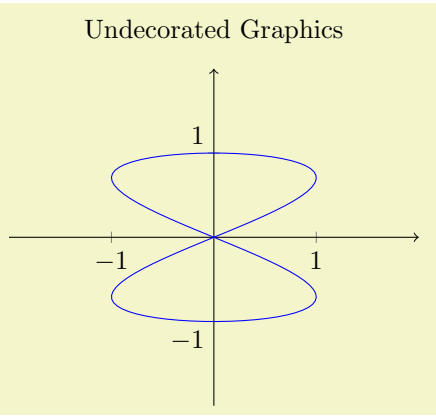
In that specific case, you have to avoid `\pgfmathprintnumber` since the argument is *no* number. Note that `symbolic x coords` cannot return fractions between, say, A and B as you would expect. However, the point will still be placed at the fractional position (unless you have a `scatter` or `bar` plot).

The computation of coordinates for the `pos` feature is computationally expensive for plots with many points. To reduce time, PGFPLOTS will cache computed values: invoking the command `\pgfplotspointplotattime` multiple times with the same argument will reuse the computed value.

4.16.3 Placing Decorations on Top of a Plot

Ti*k*Z comes with the powerful `decorations` library (or better: set of libraries). Decorations allow to replace or extend an existing path by means of fancy additional graphics. An introduction into the decorations functionality of Ti*k*Z is beyond the scope of this manual and the interested reader should read the associated section in [5].

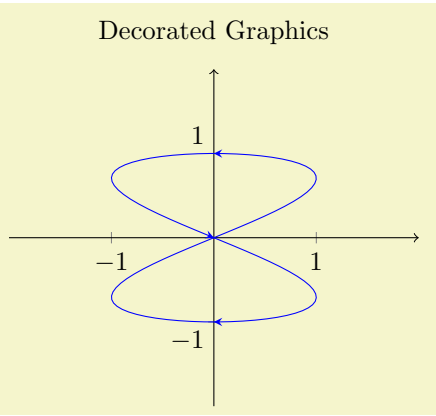
This section shows how to use decorations to enhance plots in PGFPLOTS. Suppose you have some graphics for which you would like to add "direction pointers":




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}[]
%  An undecorated graphics with a lot of
%  pretty-printing styles:
\begin{axis}[
    axis lines=middle,
    title=Undecorated Graphics,
    xmin=-2, xmax=2, ymin=-2, ymax=2,
    xtick={-1,1}, ytick={-1,1},
    %  this disables the standard
    %  tick label *text* (but not the line)
    yticklabel=\ ,
    extra description/.code={
        %  this generates custom y labels to implement
        %  individual styles for every tick:
        \node[below left] at (axis cs:0,-1) {$-1$};
        \node[above left] at (axis cs:0,1) {$1$};
    },
    axis line style={->},
  ]
  \addplot[blue,samples=100,domain=0:2*pi]
    ({sin(deg(2*x))}, {sin(deg(x))});
\end{axis}
\end{tikzpicture}
```

Our aim is to add short pointers indicating the direction of the parameterization.

The solution is to use `\usetikzlibrary{decorations.markings}` and a decoration inside of `\addplot`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usetikzlibrary{decorations.markings}
\begin{tikzpicture}[]
%  Same as in previous example, but with decorations:
\begin{axis}[axis lines=middle,
    title=Decorated Graphics,
    xmin=-2, xmax=2, ymin=-2, ymax=2,
    xtick={-1,1}, ytick={-1,1},
    %  this disables the standard
    %  tick label *text* (but not the line)
    yticklabel=\ ,
    extra description/.code={
        %  this generates custom y labels to implement
        %  individual styles for every tick:
        \node[below left] at (axis cs:0,-1) {$-1$};
        \node[above left] at (axis cs:0,1) {$1$};
    },
    axis line style={->},
  ]
  \addplot[blue,samples=100,domain=0:2*pi,
    postaction={decorate},%  ------
    decoration={markings, %  ------
         mark=at position 0.25 with {\arrow{stealth}},
         mark=at position 0.5  with {\arrow{stealth}},
         mark=at position 0.75 with {\arrow{stealth}}}
    ]
    ({sin(deg(2*x))}, {sin(deg(x))});
\end{axis}
\end{tikzpicture}
```

The only changes are in the option list for `\addplot`: it contains a `postaction={decorate}` which activates the decoration (without replacing the original path) and some specification `decoration` containing details about how to decorate the path.

A discussion of details of the `decorations` libraries is beyond the scope of this manual (see [5] for details), but the main point is to add the required decorations to `\addplot` and its option list.

4.17 Style Options

4.17.1 All Supported Styles

PGFPLOTS provides many styles to customize its appearance and behavior. They can be defined and changed in any place where keys are allowed. Furthermore, own styles are defined easily.

Key handler ⟨*key*⟩`/.style={`⟨*key-value-list*⟩`}`
Defines or redefines a style ⟨*key*⟩. A style is a normal key which will set all options in ⟨*key-value-list*⟩ when it is set.
Use `\pgfplotsset{`⟨*key*⟩`/.style={`⟨*key-value-list*⟩`}}` to (re)define a style ⟨*key*⟩ in the namespace `/pgfplots`.

Key handler ⟨*key*⟩`/.append style={`⟨*key-value-list*⟩`}`
Appends ⟨*key-value-list*⟩ to an already existing style ⟨*key*⟩. This is the preferred method to change the predefined styles: if you only append, you maintain compatibility with future versions.
Use `\pgfplotsset{`⟨*key*⟩`/.append style={`⟨*key-value-list*⟩`}}` to append ⟨*key-value-list*⟩ to the style ⟨*key*⟩. This will assume the prefix `/pgfplots`.

Styles installed for linear/logarithmic axis

`/pgfplots/every axis` (style, initially `empty`)
Installed at the beginning of every axis. Ti*k*Z options inside of it will be used for anything inside of the axis rectangle and any axis descriptions.

`/pgfplots/every semilogx axis` (style, initially `empty`)
Installed at the beginning of every plot with linear x axis and logarithmic y axis, but after '`every axis`'.

`/pgfplots/every semilogy axis` (style, initially `empty`)
Likewise, but with interchanged roles for x and y.

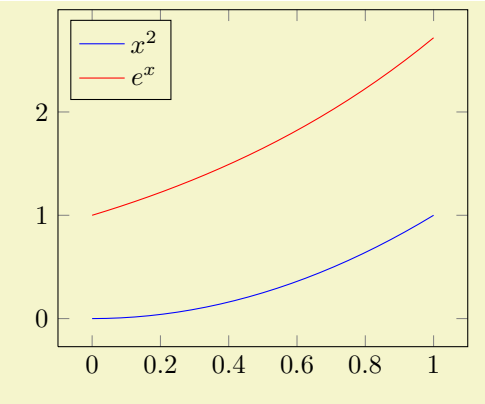
`/pgfplots/every loglog axis` (style, initially `empty`)
Installed at the beginning of every log–log plot.

`/pgfplots/every linear axis` (style, initially `empty`)
Installed at the beginning of every plot with normal axis scaling.

Styles installed for single plots

`/pgfplots/every axis plot` (style, initially `empty`)
Installed for each plot. This style may contain options like samples, gnuplot parameters, error bars and it may contain options which affect the final drawing commands.

`/pgfplots/every axis plot post` (style, initially `empty`)
This style is similar to `every axis plot` in that is applies to any drawing command in `\addplot`. However, it is set *after* any user defined styles or `cycle list` options.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\pgfplotsset{
    every axis plot post/.append style=
        {mark=none}}

\begin{axis}[
    legend style={
        at={(0.03,0.97)},anchor=north west},
    domain=0:1]
    \addplot {x^2};
    \addplot {exp(x)};
    \legend{$x^2$,$e^x$}
\end{axis}
\end{tikzpicture}
```

/pgfplots/every axis plot no # (style, initially empty)

Used for every #th plot where $\# = 0, 1, 2, 3, 4, \ldots$.

/pgfplots/every forget plot (style, initially empty)

Used for every plot which has forget plot activated.

/pgfplots/forget plot style={⟨*key-value-list*⟩}

An abbreviation for every forget plot/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every forget plot.

Styles for axis descriptions

/pgfplots/every axis label (style, initially empty)

Used for all axis label (like xlabel and ylabel).

/pgfplots/label style={⟨*key-value-list*⟩}

An abbreviation for every axis label/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every axis label.

/pgfplots/every axis x label (style, no value)
/pgfplots/every axis y label (style, no value)
/pgfplots/every axis z label (style, no value)

Used only for x, y, or z labels, respectively and installed after 'every axis label'.

The initial settings are set by xlabel absolute and its variants (if the initial configuration compat=pre 1.3 is active) or xlabel near ticks which provides the better spacing as it incorporates the tick label sizes to compute the position.

Attention: These styles will be overwritten by axis x line and/or axis y line. Please remember to place your modifications after the axis line variations.

/pgfplots/x label style={⟨*key-value-list*⟩}
/pgfplots/y label style={⟨*key-value-list*⟩}
/pgfplots/z label style={⟨*key-value-list*⟩}
/pgfplots/xlabel style={⟨*key-value-list*⟩}
/pgfplots/ylabel style={⟨*key-value-list*⟩}
/pgfplots/zlabel style={⟨*key-value-list*⟩}

Different abbreviations for every axis x label/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every axis y label/.append style={⟨*key-value-list*⟩}, and z, every axis z label/.append style={⟨*key-value-list*⟩}).

/pgfplots/every axis title (style, no value)

Used for any axis title. The at=(⟨*x*,*y*⟩) syntax will place the title using axis description cs.

The initial setting is

```
\pgfplotsset{every axis title/.style={at={(0.5,1)},above,yshift=6pt}}
```

To be more precise, the yshift doesn't use the hardcoded 6pt: it uses the value of

/pgfplots/every axis title shift={⟨*default shift*⟩} (initially 6pt)

which can be reset if needed.

/pgfplots/title style={⟨*key-value-list*⟩}

An abbreviation for every axis title/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every axis title.

/pgfplots/every axis legend (style, no value)

Installed for each legend. As described for axis description cs, the legend's position can be placed using coordinates between 0 and 1 (it employs axis description cs automatically).

The initial setting is

```
\pgfplotsset{every axis legend/.style={
        cells={anchor=center},
        inner xsep=3pt,inner ysep=2pt,nodes={inner sep=2pt,text depth=0.15em},
        anchor=north east,
        shape=rectangle,
        fill=white,draw=black,
        at={(0.98,0.98)}}}
```

/pgfplots/legend style={⟨*key-value-list*⟩}

An abbreviation for every axis legend/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every axis legend.

/pgfplots/every legend image post (style, no value)

Allows to change the appearance of the small legend images *after* the options of the plot style have been applied. Thus, legend formatting can be changed independently of the plot style using every legend image post.

This key is also documented on page 159.

/pgfplots/legend image post style={⟨*key-value-list*⟩}

An abbreviation for every legend image post/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every legend image post.

/pgfplots/every legend to name picture (style, no value)

A style for use with `legend to image`, see the documentation therein.

/pgfplots/every colorbar (style, no value)

A style to change the colorbar. See page 180 for the reference documentation of every colorbar.

/pgfplots/colorbar style={⟨*key-value-list*⟩}

An abbreviation for every colorbar/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every colorbar.

Styles for axis lines

/pgfplots/every outer x axis line (style, initially `empty`)
/pgfplots/every outer y axis line (style, initially `empty`)
/pgfplots/every outer z axis line (style, initially `empty`)

Installed for every axis line which lies on the outer box.

If you want arrow heads, you may also need to check the separate axis lines boolean key.

/pgfplots/every inner x axis line (style, initially `empty`)
/pgfplots/every inner y axis line (style, initially `empty`)
/pgfplots/every inner z axis line (style, initially `empty`)

Installed for every axis line which is drawn using the `center` or `middle` options.

/pgfplots/axis line style={⟨*key-value-list*⟩}
/pgfplots/inner axis line style={⟨*key-value-list*⟩}
/pgfplots/outer axis line style={⟨*key-value-list*⟩}
/pgfplots/x axis line style={⟨*key-value-list*⟩}
/pgfplots/y axis line style={⟨*key-value-list*⟩}
/pgfplots/z axis line style={⟨*key-value-list*⟩}

These options modify parts of the axis line styles. They append options to every inner x axis line and every outer x axis line and the respective y/z variants.

Please refer to Section 4.8.9 on page 171 for details about styles for axis lines.

/pgfplots/every 3d box foreground (style, no value)

Installed for the parts drawn by 3d box=complete. This affects axis lines, tick lines and grid lines drawn in the *foreground*. The background drawing operations have already been done when this style is evaluated.

/pgfplots/3d box foreground style={⟨*key-value-list*⟩}

An abbreviation for every 3d box foreground/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every 3d box foreground.

/pgfplots/every colorbar sampled line (style, no value)

To be used in conjunction with colorbar sampled line, see the documentation therein.

/pgfplots/colorbar sampled line style={⟨*key-value-list*⟩}

An abbreviation for every colorbar sampled line/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every colorbar sampled line.

Styles for ticks

/pgfplots/every tick (style, initially very thin,gray)

Installed for each of the small tick *lines*.

/pgfplots/tick style={⟨*key-value-list*⟩}

An abbreviation for every tick/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every tick.

/pgfplots/every minor tick (style, initially empty)

Used for each minor tick line, installed after 'every tick'.

/pgfplots/minor tick style={⟨*key-value-list*⟩}

An abbreviation for every minor tick/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every minor tick.

/pgfplots/every major tick (style, initially empty)

Used for each major tick line, installed after 'every tick'.

/pgfplots/major tick style={⟨*key-value-list*⟩}

An abbreviation for every major tick/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every major tick.

/pgfplots/every tick label (style, initially empty)

Used for each x and y tick labels.

/pgfplots/tick label style={⟨*key-value-list*⟩}
/pgfplots/ticklabel style={⟨*key-value-list*⟩}

Different abbreviations for every tick label/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every tick label/.append style={⟨*key-value-list*⟩}, and z, every tick label/.append style={⟨*key-value-list*⟩}).

/pgfplots/every x tick label (style, initially empty)
/pgfplots/every y tick label (style, initially empty)
/pgfplots/every z tick label (style, initially empty)

Used for each x (or y or z, respectively) tick label, installed after 'every tick label'.

/pgfplots/x tick label style={⟨*key-value-list*⟩}
/pgfplots/y tick label style={⟨*key-value-list*⟩}

/pgfplots/z tick label style={⟨*key-value-list*⟩}
/pgfplots/xticklabel style={⟨*key-value-list*⟩}
/pgfplots/yticklabel style={⟨*key-value-list*⟩}
/pgfplots/zticklabel style={⟨*key-value-list*⟩}

Different abbreviations for `every x tick label/.append style={`⟨*key-value-list*⟩`}` (or the respective styles for y, `every y tick label/.append style={`⟨*key-value-list*⟩`}`, and z, `every z tick label/.append style={`⟨*key-value-list*⟩`}`).

/pgfplots/every x tick scale label (style, no value)
/pgfplots/every y tick scale label (style, no value)
/pgfplots/every z tick scale label (style, no value)

Configures placement and display of the nodes containing the order of magnitude of tick labels, see Section 4.14.3 for more information about `scaled ticks`.

The initial settings are

```
\pgfplotsset{
    every x tick scale label/.style={at={(1,0)},yshift=-2em,left,inner sep=0pt},
    every y tick scale label/.style={at={(0,1)},above right,inner sep=0pt,yshift=0.3em},
    every z tick scale label/.style={
      at={(zticklabel cs:1.2,-\pgfplotsvalueoflargesttickdimen z -0.3em)},
      anchor=near zticklabel,inner sep=0pt},
    }
```

/pgfplots/x tick scale label style={⟨*key-value-list*⟩}
/pgfplots/y tick scale label style={⟨*key-value-list*⟩}
/pgfplots/z tick scale label style={⟨*key-value-list*⟩}

An abbreviation for `every x tick scale label/.append style={`⟨*key-value-list*⟩`}` (or the respective styles for y, `every y tick scale label/.append style={`⟨*key-value-list*⟩`}`, and the z–axis, `every z tick scale label/.append style={`⟨*key-value-list*⟩`}`).

It appends options to the already existing style `every x tick scale label`.

/pgfplots/every x tick (style, initially `empty`)
/pgfplots/every y tick (style, initially `empty`)
/pgfplots/every z tick (style, initially `empty`)

Installed for tick *lines* on either x or y axis.

/pgfplots/xtick style={⟨*key-value-list*⟩}
/pgfplots/ytick style={⟨*key-value-list*⟩}
/pgfplots/ztick style={⟨*key-value-list*⟩}

An abbreviation for `every x tick/.append style={`⟨*key-value-list*⟩`}` (or the respective styles for y, `every y tick/.append style={`⟨*key-value-list*⟩`}`, and the z–axis, `every z tick/.append style={`⟨*key-value-list*⟩`}`).

It appends options to the already existing style `every x tick`.

/pgfplots/every minor x tick (style, initially `empty`)
/pgfplots/every minor y tick (style, initially `empty`)
/pgfplots/every minor z tick (style, initially `empty`)

Installed for minor tick lines on either x or y axis.

/pgfplots/minor x tick style={⟨*key-value-list*⟩}
/pgfplots/minor y tick style={⟨*key-value-list*⟩}
/pgfplots/minor z tick style={⟨*key-value-list*⟩}

An abbreviation for `every minor x tick/.append style={`⟨*key-value-list*⟩`}` (or the respective styles for y, `every minor y tick/.append style={`⟨*key-value-list*⟩`}`, and the z–axis, `every minor z tick/.append style={`⟨*key-value-list*⟩`}`).

It appends options to the already existing style `every minor x tick`.

/pgfplots/every major x tick (style, initially `empty`)

/pgfplots/every major y tick (style, initially empty)
/pgfplots/every major z tick (style, initially empty)

Installed for major tick lines on either x or y axis.

/pgfplots/major x tick style={⟨*key-value-list*⟩}
/pgfplots/major y tick style={⟨*key-value-list*⟩}
/pgfplots/major z tick style={⟨*key-value-list*⟩}

An abbreviation for every major x tick/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every major y tick/.append style={⟨*key-value-list*⟩}, and the z–axis, every major z tick/.append style={⟨*key-value-list*⟩}).

It appends options to the already existing style every major x tick.

/pgfplots/every extra x tick (style, no value)
/pgfplots/every extra y tick (style, no value)
/pgfplots/every extra z tick (style, no value)

Allows to configure the appearance of 'extra x ticks'. This style is installed before touching the first extra x tick. It is possible to set any option which affects tick or grid line generation.

The initial setting is

```
\pgfplotsset{
    every extra x tick/.style={/pgfplots/log identify minor tick positions=true},
    every extra y tick/.style={/pgfplots/log identify minor tick positions=true}}
```

Useful examples are shown below.

```
\pgfplotsset{every extra x tick/.append style={grid=major}}
\pgfplotsset{every extra x tick/.append style={major tick length=0pt}}
\pgfplotsset{every extra x tick/.append style={/pgf/number format=sci subscript}}
\pgfplotsset{extra x tick style={
        color=red,
        tickwidth=3mm,
        %  the initial 'every tick style' defines a 'line width'.
        %  this here redefines it:
        tick style={
            line width=2mm,
        },
    }
}
```

/pgfplots/extra x tick style={⟨*key-value-list*⟩}
/pgfplots/extra y tick style={⟨*key-value-list*⟩}
/pgfplots/extra z tick style={⟨*key-value-list*⟩}

An abbreviation for every extra x tick/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every extra y tick/.append style={⟨*key-value-list*⟩}, and the z–axis, every extra z tick/.append style={⟨*key-value-list*⟩}).

It appends options to the already existing style every extra x tick.

/pgfplots/extra tick style={⟨*key-value-list*⟩}

An abbreviation which appends ⟨*key-value-list*⟩ to every extra x tick, every extra y tick and every extra z tick.

Styles for grid lines

/pgfplots/every axis grid (style, initially thin,black!25)

Used for each grid line.

/pgfplots/grid style={⟨*key-value-list*⟩}

An abbreviation for every axis grid/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every axis grid.

/pgfplots/every minor grid (style, initially empty)

Used for each minor grid line, installed after 'every axis grid'.

/pgfplots/minor grid style={⟨*key-value-list*⟩}

An abbreviation for every minor grid/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every minor grid.

/pgfplots/every major grid (style, initially empty)

Likewise, for major grid lines.

/pgfplots/major grid style={⟨*key-value-list*⟩}

An abbreviation for every major grid/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every major grid.

/pgfplots/every axis x grid (style, initially empty)
/pgfplots/every axis y grid (style, initially empty)
/pgfplots/every axis z grid (style, initially empty)

Used for each grid line in either x or y direction.

/pgfplots/x grid style={⟨*key-value-list*⟩}
/pgfplots/y grid style={⟨*key-value-list*⟩}
/pgfplots/z grid style={⟨*key-value-list*⟩}

An abbreviation for every axis x grid/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every axis y grid/.append style={⟨*key-value-list*⟩}, and the z–axis, every axis z grid/.append style={⟨*key-value-list*⟩}).

It appends options to the already existing style every axis x grid.

/pgfplots/every minor x grid (style, initially empty)
/pgfplots/every minor y grid (style, initially empty)
/pgfplots/every minor z grid (style, initially empty)

Used for each minor grid line in either x or y direction.

/pgfplots/minor x grid style={⟨*key-value-list*⟩}
/pgfplots/minor y grid style={⟨*key-value-list*⟩}
/pgfplots/minor z grid style={⟨*key-value-list*⟩}

An abbreviation for every minor x grid/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every minor y grid/.append style={⟨*key-value-list*⟩}, and the z–axis, every minor z grid/.append style={⟨*key-value-list*⟩}).

It appends options to the already existing style every minor x grid.

/pgfplots/every major x grid (style, initially empty)
/pgfplots/every major y grid (style, initially empty)
/pgfplots/every major z grid (style, initially empty)

Used for each major grid line in either x or y direction.

/pgfplots/major x grid style={⟨*key-value-list*⟩}
/pgfplots/major y grid style={⟨*key-value-list*⟩}
/pgfplots/major z grid style={⟨*key-value-list*⟩}

An abbreviation for every major x grid/.append style={⟨*key-value-list*⟩} (or the respective styles for y, every major y grid/.append style={⟨*key-value-list*⟩}, and the z–axis, every major z grid/.append style={⟨*key-value-list*⟩}).

It appends options to the already existing style every major x grid.

Styles for error bars

/pgfplots/every error bar (style, initially thin)

Installed for every error bar.

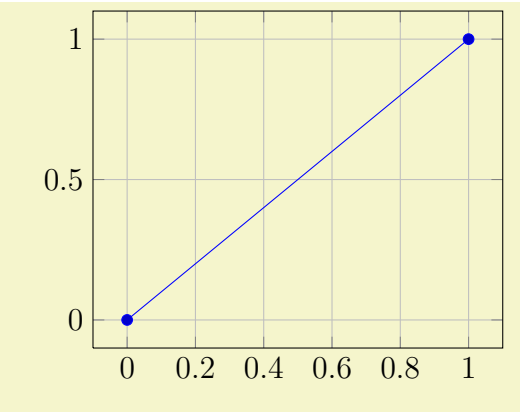
/pgfplots/error bars/error bar style={⟨*key-value-list*⟩}

An abbreviation for every error bar/.append style={⟨*key-value-list*⟩}.

It appends options to the already existing style every error bar.

4.17.2 (Re)Defining Own Styles

Use \pgfplotsset{⟨*style name*⟩/.style={⟨*key-value-list*⟩}} to create own styles. If ⟨*style name*⟩ exists already, it will be replaced. Please note that it is *not* possible to use the Ti*k*Z-command \tikzstyle{⟨*style name*⟩}=[] in this context[48].




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{my personal style/.style=
    {grid=major,font=\large}}

\begin{tikzpicture}
\begin{axis}[my personal style]
    \addplot coordinates {(0,0) (1,1)};
\end{axis}
\end{tikzpicture}
```

4.18 Alignment Options and Bounding Box Control

4.18.1 Basic Alignment

Alignment works with two main methods: a coordinate where the axis shall be drawn and an "anchor" inside of the axis which shall be drawn at this particular coordinate. This methodology is common for each Ti*k*Z node – and an axis is nothing but a (special) Ti*k*Z node. The coordinate can be specified using the at key, while the anchor can be specified with the anchor key. In most cases, it is sufficient to provide only an anchor – unless one needs more than one axis in the same picture environment.

/pgfplots/at={⟨*coordinate expression*⟩}

Assigns a position for the complete axis image. This option works similarly to the at-option of \node[at={⟨*coordinate expression*⟩}], see [5]. The common syntax is at={(⟨*x*,*y*⟩)}.

The idea is to provide an ⟨*coordinate expression*⟩ where the axis will be placed. The axis' anchor will be placed at ⟨*coordinate expression*⟩.

/pgfplots/anchor={⟨*name*⟩} (initially south west)

Chooses one of the different possible positions inside of an axis which is placed with at. The at key defines the position where to place the axis inside of the embedding picture, the anchor key defines which point of the axis shall be positioned by 'at'. The initial configuration assumes at={(0,0)}. Thus, anchor=center will place the axis' center at the logical picture position $(0, 0)$. Similarly, anchor=south west will position the lower left corner of the axis at $(0, 0)$.

For users who are familiar with Ti*k*Z: an axis is actually a very special node, so anchors work as in [5].

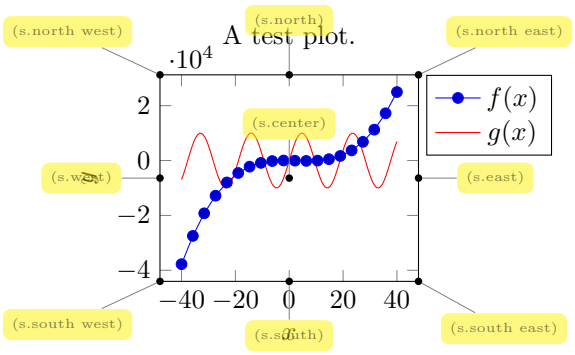
Anchors are useful in conjunction with horizontal or vertical alignment of plots, see the examples below.

There are four sets of anchors available: anchors positioned on the axis bounding box, anchors on the outer bounding box and anchors which have one coordinate on the outer bounding box and the other one at a position of the axis rectangle. Finally, one can place anchors near the origin.

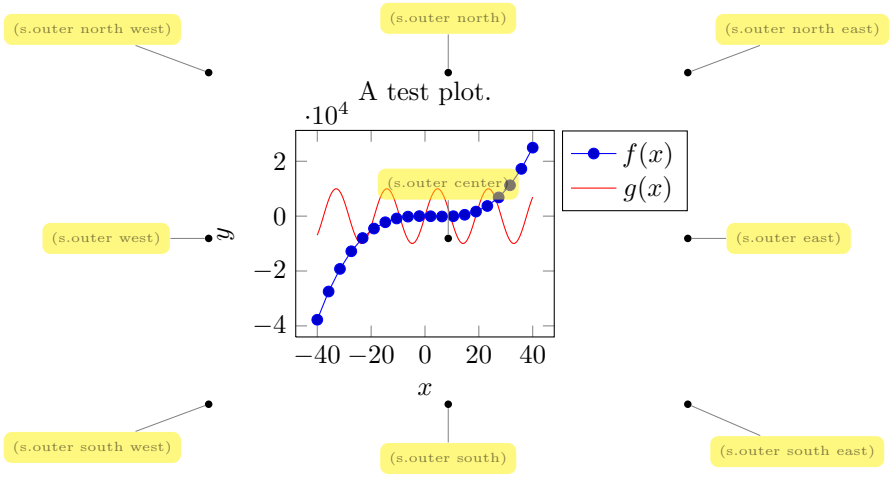
In more detail, we have anchors on the axis rectangle (the bounding box around the axis)[49],

[48] This was possible in a previous version and is still supported for backwards compatibility. But in some cases, it may not work as expected.

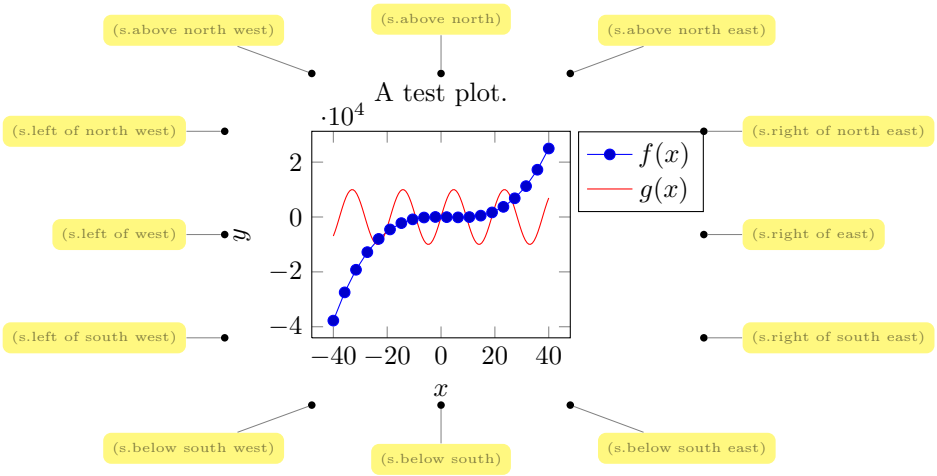
[49] Versions prior to PGFPLOTS v.1.3 did *not* use the bounding box of the axis, they used axis coordinates to orient these anchors. This has been fixed. If you *really* want to undo the bugfix, see compat/anchors.



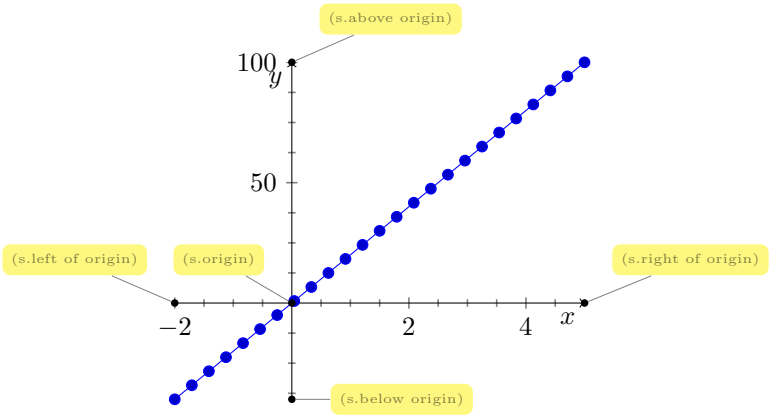
Anchors on the outer bounding box,



There are anchors which have one coordinate on the outer bounding box, and one on the axis rectangle,

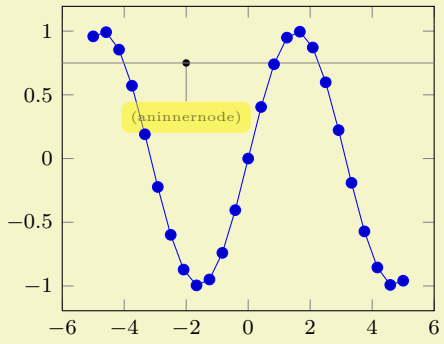


And finally, we have origin anchors which are especially useful when axis lines pass through the origin,



There is a fifth anchor which is not directly related to the axis: you can provide the anchor of *a named inner node*. Thus, you can define your own anchor, by writing `\node (⟨name⟩) at (⟨point coordinate⟩) {};` as follows (using the `baseline` option described below):

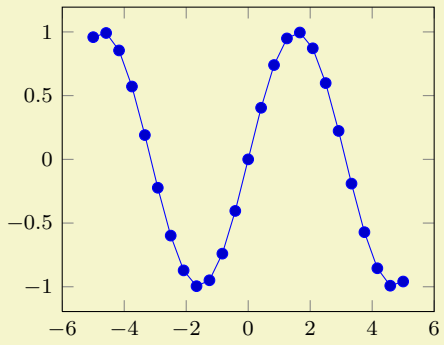
Aligning at .....




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
Aligning at .....
\begin{tikzpicture}[baseline]
\begin{axis}[small,anchor=aninnernode.center]
    \addplot {sin(deg(x))};
    \node
        [pin=-90:(aninnernode),fill=black,circle,scale=0.3]
        (aninnernode) at (axis cs:-2,0.75) {};
    \draw[help lines] (axis cs:-6,0.75) -- (axis cs:6,0.75);
\end{axis}
\end{tikzpicture}
```

What happens is that a node is placed at `(axis cs:-2,0.75)`. Note that the options `[pin=...]` are merely to show the `\node` (the pin style has been defined by the PGFPLOTS manual). Since a name can also be assigned using `name=`⟨*node's name*⟩ and since any PGFPLOTS description is also a `\node`, you can align your plot at selected axis descriptions:

Aligning at .....




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
Aligning at .....
\begin{tikzpicture}[baseline]
\begin{axis}[
    small,
    title={The function $\sin x$ is very pretty.},
    title style={name=MyTitleNode},
    anchor=MyTitleNode.base,
]
    \addplot {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```

The default value is `anchor=south west`. You can use anchors in conjunction with the Ti*k*Z `baseline` option and/or `\begin{pgfinterruptboundingbox}` to perform alignment.

Remarks: Each of the anchors on the axis rectangle has an equivalent to a coordinate in the `axis description cs` described in Section 4.8.1. That means the first set of anchors actually lives on the

tight bounding box around the axis (without any ticks or descriptions). The `south west` anchor will always be the lower left corner of this bounding box, even in case of a rotated or skewed coordinate system[50]. Similar statements hold for the other anchors.

4.18.2 Vertical Alignment with `baseline`

/tikz/baseline (no value)

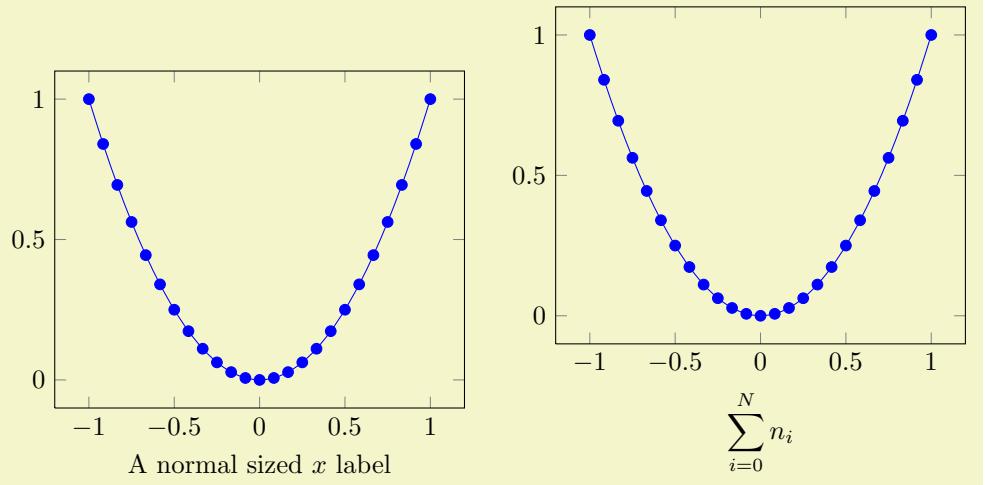
The `baseline` option should be provided as argument to a `tikzpicture`. It configures Ti*k*Z to shift the picture position $y = 0$ to the embedding text's baseline:

This is a picture, here another one.

```
This is \tikz[baseline]\fill[red] (0,0) circle(3pt); a picture,
here \tikz[baseline]\fill[red] (0,10pt) circle(3pt); another one.
```

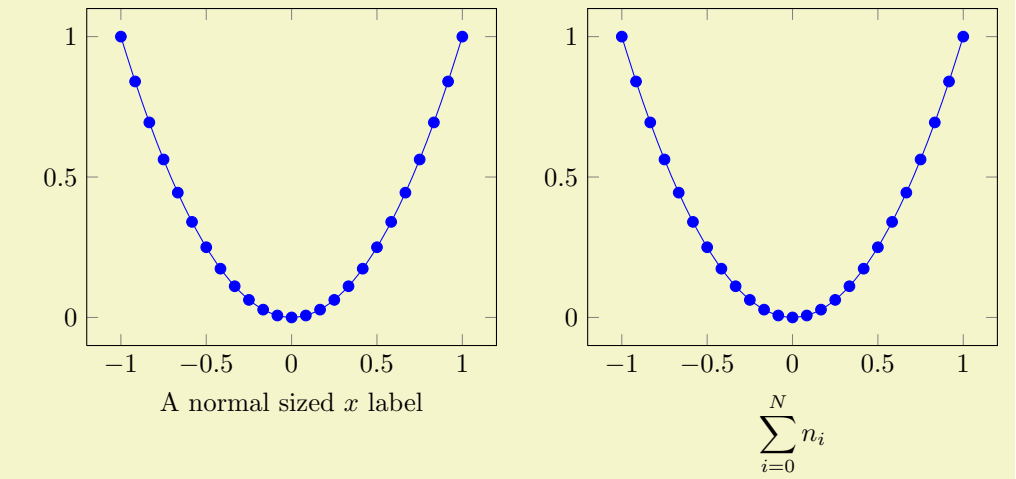
Consequently, the `baseline` option allows to align different `tikzpicture`s. An axis is, by default, placed with `at={(0,0)}`, and the `anchor` key specifies which part of the axis is placed at `(0,0)`. Consequently, the `baseline` option, together with `anchor`, allows to align different axes with the embedding text.

The default axis anchor is `south west`, which means that the picture coordinate $(0, 0)$ is the lower left corner of the axis. As a consequence, the Ti*k*Z option "`baseline`" allows vertical alignment of adjacent plots:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  1. Unaligned:
\pgfplotsset{domain=-1:1}
\begin{tikzpicture}
    \begin{axis}[xlabel=A normal sized $x$ label]
    \addplot[smooth,blue,mark=*] {x^2};
    \end{axis}
\end{tikzpicture}%
\hspace{0.15cm}
\begin{tikzpicture}
    \begin{axis}[xlabel={$\displaystyle \sum_{i=0}^N n_i $ }]
    \addplot[smooth,blue,mark=*] {x^2};
    \end{axis}
\end{tikzpicture}
```

[50] Note that this is only true for versions since 1.3.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  2. Aligned:
\pgfplotsset{domain=-1:1}
\begin{tikzpicture}[baseline]
    \begin{axis}[xlabel=A normal sized $x$ label]
    \addplot[smooth,blue,mark=*] {x^2};
    \end{axis}
\end{tikzpicture}%
\hspace{0.15cm}
\begin{tikzpicture}[baseline]
    \begin{axis}[xlabel={$\displaystyle \sum_{i=0}^N n_i $ }]
    \addplot[smooth,blue,mark=*] {x^2};
    \end{axis}
\end{tikzpicture}
```

Note that it is also possible to write `baseline=5cm` in which case the image offset at $y =$`5cm` will be used as baseline.

The `baseline` key is related to `\begin{minipage}[`⟨*alignment*⟩`]` or `\begin{tabular}[`⟨*alignment*⟩`]`: the ⟨*alignment*⟩ tells LaTeX which part of the `minipage` or `tabular` shall be positioned on the baseline. Thus, `baseline` does the same for pictures (with more freedom for ⟨*alignment*⟩).

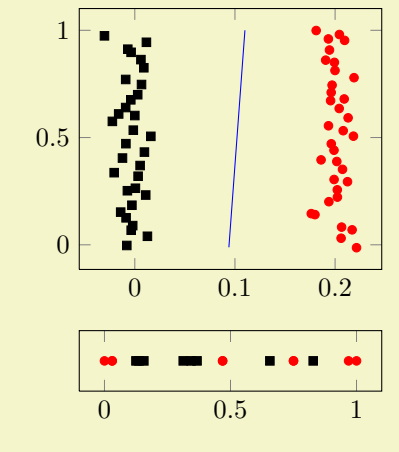
4.18.3 Horizontal Alignment

Horizontal alignment can be done in two ways:

1. Using separate `tikzpicture` environments which have reduced bounding boxes or
2. A single `tikzpicture` environment in which the complete alignment is done.

The first approach requires the use of reduced bounding boxes and is discussed in Section 4.18.6.

The second approach, a single `tikzpicture` environment, employs the `at` and `anchor` keys to align parts of the images. For example, if you place multiple `axes` into a single `tikzpicture` and use the '`anchor`'-option, you can control horizontal alignment:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\pgfplotsset{every axis/.append style={
cycle list={
    {red,only marks,mark options={
        fill=red,scale=0.8},mark=*},
    {black,only marks,mark options={
        fill=black,scale=0.8},mark=square*}}}}

\begin{axis}[width=4cm,scale only axis,
    name=main plot]
\addplot file
    {plotdata/pgfplots_scatterdata1.dat};
\addplot file
    {plotdata/pgfplots_scatterdata2.dat};
\addplot[blue] coordinates {
    (0.093947,    -0.011481)
    (0.101957,    0.494273)
    (0.109967,    1.000027)};
\end{axis}

\begin{axis}[
    at={(main plot.below south west)},yshift=-0.1cm,
    anchor=north west,
    width=4cm,scale only axis,height=0.8cm,
    ytick=\empty]

\addplot file
  {plotdata/pgfplots_scatterdata1_latent.dat};
\addplot file
  {plotdata/pgfplots_scatterdata2_latent.dat};
\end{axis}
\end{tikzpicture}
```

Here, the second axis uses `at={(main plot.below south west)}` to be placed below the first one. Furthermore, it has `yshift=-0.1cm` in order to leave additional space, and it uses `anchor=north west` to place the upper left corner at the specified position. Instead of the `at={}` construction, we could also have used `yshift` with larger negative shift.

4.18.4 Alignment In Array Form (Subplots)

Sometimes multiple alignment axes in array form are desired. PGFPLOTS supports this task in several ways which are described in the following. There are basically three related, yet different, approaches:

1. Simply place `\begin{tikzpicture}...\end{tikzpicture}` into a LaTeX table. This is straight–forward; you would do the very same thing with `\includegraphics`.
   In addition to `\includegraphics`, the `baseline` feature allows simple yet effective vertical alignment. In addition, the `trim left` and `trim right` features allow simple yet effective horizontal alignment (see below).
2. Use a *single* picture which contains an array of axes, i.e. a pattern like
   `\begin{tikzpicture} \matrix{` ⟨*multiple axes*⟩`}; \end{tikzpicture}`.
   This allows considerably simpler alignment! Alas, it needs special handling for `legend entries` due to a weakness of `\matrix`. If you use the `external` library (which is recommended), it takes more time since the picture gets larger.
3. Use the `groupplots` library shipped with PGFPLOTS. It is specialized on axes in array form with particular strength if the axes are closely related (for example if they share axis descriptions like `xlabel` or even tick labels). Note, however, that the other approaches are better when it comes to automatic handling of bounding boxes.

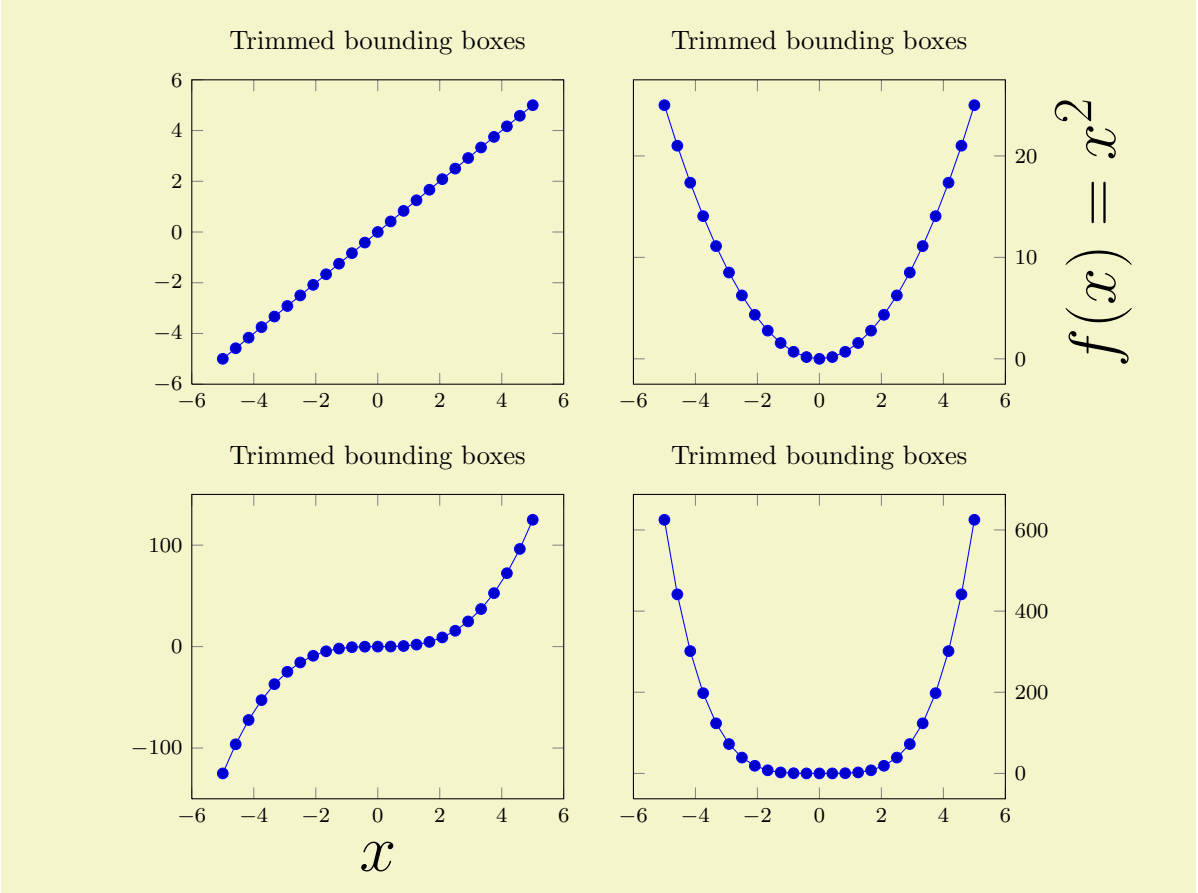
The `groupplots` library is discussed in all detail in Section 5.5. This section discusses the other two approaches.

Array Alignment using LaTeX Tables The idea is simple: use a LaTeX table and provide one `tikzpicture` for every cell. You are probably familiar with this sort of alignment, perhaps together with `\includegraphics`. It works in the very same way for PGFPLOTS. The approach is the simplest

one since it doesn't need special knowledge. Its disadvantage, however, is more difficulty to control positions *inside* of the image (like differently sized axis descriptions).

Is is strongly recommended to employ the `baseline` option for each cell picture, which simplifies vertical alignment considerably. If you want a simple solution to place separate axes in array form, and you prefer to use one `tikzpicture` for every axis, the probably most simple and most effective way to get horizontal alignment are the `trim left` and `trim right` features – or styles based on them:

The `trim axis left` feature can be used to exclude axis descriptions on the left from the bounding box, and the `trim axis right` can exclude axis descriptions on the right from the bounding box. Thus, alignment is done using the vertical axis lines. Since both keys effectively modify the bounding box, they are documented in Section 4.18.6 "Bounding Box Restrictions". Here is just a small example for array alignment by means of `tabular`, `baseline` and the `trim left`/`trim right` features:



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{
    small,
    title=Trimmed bounding boxes
}
\begin{center}
\begin{tabular}{rl}
    \begin{tikzpicture}[baseline,trim axis left]
        \begin{axis}
            \addplot {x};
        \end{axis}
    \end{tikzpicture}
    &
    \begin{tikzpicture}[baseline,trim axis right]
    \begin{axis}[
        ylabel={$f(x)=x^2$},
        yticklabel pos=right,
        ylabel style={font=\Huge}]
        \addplot {x^2};
    \end{axis}
    \end{tikzpicture}
    \\
    %
    \begin{tikzpicture}[baseline,trim axis left]
    \begin{axis}[xlabel=$x$,xlabel style={font=\Huge}]
        \addplot {x^3};
    \end{axis}
    \end{tikzpicture}%
    &
    \begin{tikzpicture}[baseline,trim axis right]
    \begin{axis}[yticklabel pos=right]
        \addplot {x^4};
    \end{axis}
    \end{tikzpicture}%
    \\
\end{tabular}%
\end{center}
```

The example has 2×2 axes. The `baseline` feature controls the vertical alignment: the lower axis lines are always on the same height. The `trim axis left` key is a style which tells Ti*k*Z to trim everything which is left of the left axis line. Similarly, the `trim axis right` key does not include picture parts right of the right axis line. Together with `\begin{center}` and the `yticklabel pos=right` key, we get correct horizontal and vertical alignment together with centering at the left- and right axis lines (without descriptions).

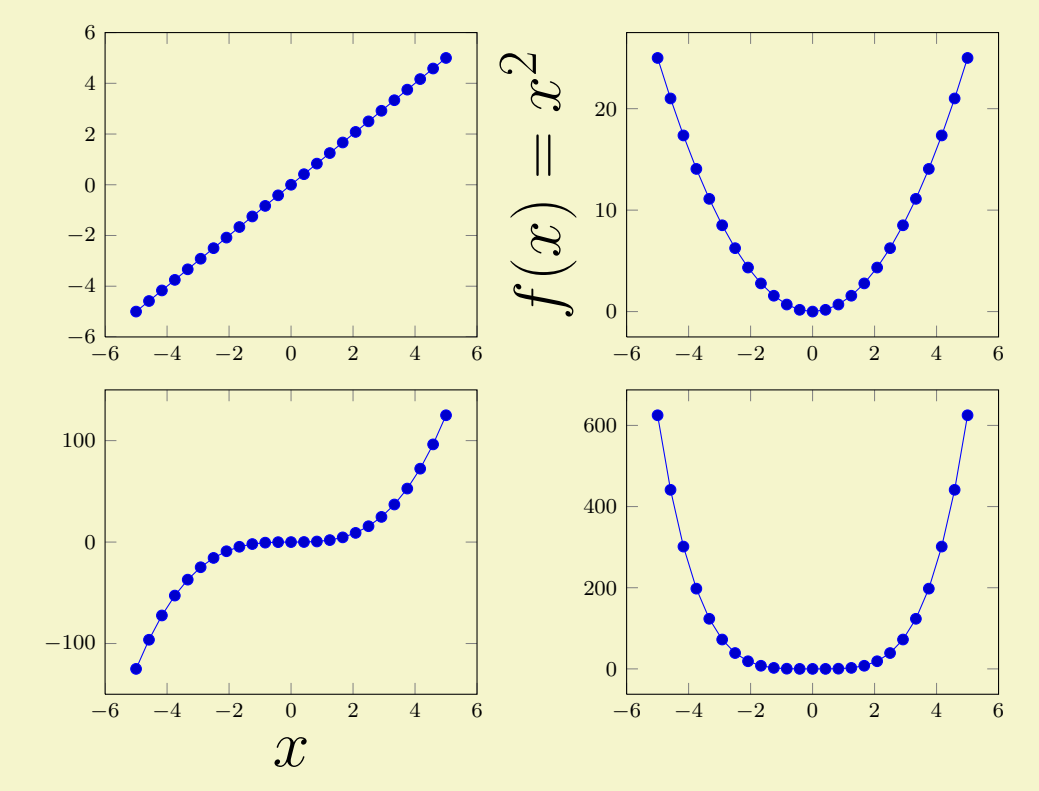
A strong advantage is that this type of alignment requires almost no changes to your pictures. Thus, you can copy–paste existing images (TEX code) relatively simple.

Note that the approach is fully compatible with the image `external`ization library: each picture is exported separately, and the bounding box restrictions (and the `baseline` offset) are stored in separate `.dpth` files. The `trim left`/`trim right` approach for horizontal alignment is the *only* supported way for reduced bounding boxes and image externalization.

Array Alignment using Ti*k*Z Matrices While it is possible to use (for example) `tabular` combined with the vertical and horizontal alignment methods discussed above, it might be better to use a Ti*k*Z `matrix` since it automatically handles the size of axis descriptions.

A Ti*k*Z matrix is some sort of "graphical" table. It knows everything about picture alignment and it has more flexibility than `tabular` when it comes to graphics. The idea is to pack the complete array into a *single* picture.

The complete documentation of a Ti*k*Z matrix is beyond the scope of this manual, please refer to [5] for details. But we provide an example here:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \pgfplotsset{small}
    \matrix {
        \begin{axis}
            \addplot {x};
        \end{axis}
        &
        %  differently large labels are aligned automatically:
        \begin{axis}[ylabel={$f(x)=x^2$},ylabel style={font=\Huge}]
            \addplot {x^2};
        \end{axis}
        \\
        %
        \begin{axis}[xlabel=$x$,xlabel style={font=\Huge}]
            \addplot {x^3};
        \end{axis}
        &
        \begin{axis}
            \addplot {x^4};
        \end{axis}
        \\
    };
\end{tikzpicture}
```

So, a matrix is a picture element inside of `tikzpicture`. Its cells are separated by '`&`' as in tabular (or, if '`&`' causes problems, with `\pgfmatrixnextcell`). Its rows are separated by '`\\`'. Each cell is aligned using the cells' anchor. Since, by default, the anchor of an axis is placed at the lower left corner, the example above is completely aligned, without the need for any bounding box modifications – even the labels are aligned correctly. If another anchor shall be used, simply place

```
\pgfplotsset{anchor=....}
\matrix {
  ...
};
```

in front of the matrix. This will use the same configuration for every sub-plot.

Attention: Unfortunately, the array alignment with `\matrix` needs special *attention with legends*. A legend is also a `\matrix` and Ti*k*Z matrices can't be nested. You will need to use the `legend to name` feature (or to assemble a legend by means of `\label` and `\ref`) to overcome this weakness (see Section 4.8.6 for details).

4.18.5 Miscellaneous for Alignment

Predefined node `current axis`

A node which refers to the current axis or the last typeset axis.

You can use this node in axis descriptions, for example to place axis labels or titles.

Remark: If you use `current axis` inside of axis descriptions, the "current axis" is not yet finished. That means you *can't use any outer anchor* inside of axis descriptions.

It is also possible to use `current axis` in any drawing or plotting commands inside of an axis (but no outer anchor as these are not defined when drawing commands are processed). This usage is similar to the `axis description cs`.

4.18.6 Bounding box restrictions

Bounding box restrictions can be archieved with several methods of PGF:

1. The `overlay` option,
2. The `pgfinterruptboundingbox` environment,
3. The `\pgfresetboundingbox` command,
4. The `\useasboundingbox` path,
5. The `trim left` and `trim right` feature (which is the *only* supported way of restricted bounding boxes and image externalization; at least for PDF output).

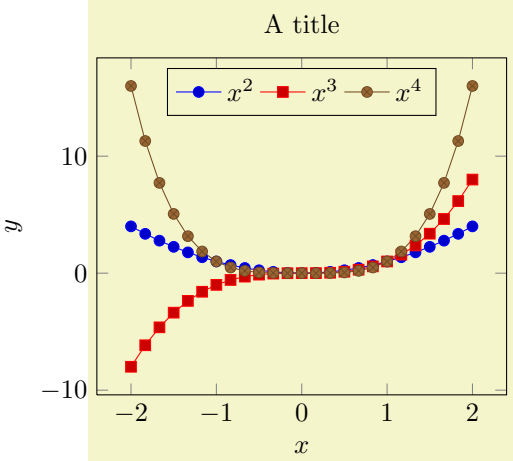
Note that image externalization (the `external` library) is more or less incompatible with methods (1.)–(4.). The problem is that `pdflatex` crops everything outside of the bounding box away. There are only two safe ways to "restrict" bounding boxes of external `.pdf` images: the first is the mentioned `trim left`/`trim right` feature and the second is to use negative `\hspace` or `\vspace` commands (or options to `\includegraphics`).

`/tikz/overlay` (no value)

A special key of PGF which disables bounding box updates for (parts of) the image. The effect is that those parts are an "overlay" over the document.

For PGFPLOTS, `overlay` can be useful to position legends or other axis descriptions outside of the axis – without affecting its size (and without affecting alignment).

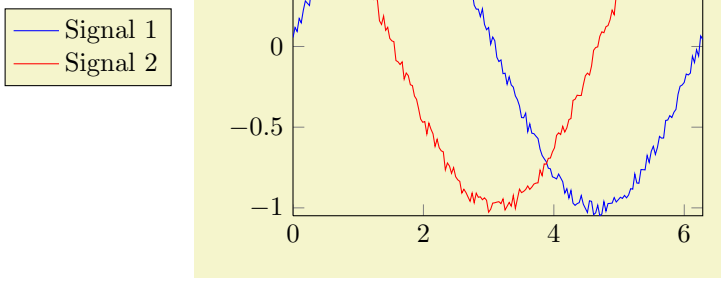
For example, one may want to include only certain parts of the axis into the final bounding box. This would allow horizontal alignment (centering):




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}%
   \begin{axis}[
      title=A title,
      ylabel style={overlay},
      yticklabel style={overlay},
      xlabel={$x$},
      ylabel={$y$},
      legend style={at={(0.5,0.97)},
         anchor=north,legend columns=-1},
      domain=-2:2
   ]
   \addplot {x^2};
   \addplot {x^3};
   \addplot {x^4};
   \legend{$x^2$,$x^3$,$x^4$}
   \end{axis}
\end{tikzpicture}%
```

Now, the left axis descriptions (y label and y ticks) stick out of the bounding box.

The following example places a legend somewhere without affecting the bounding box.

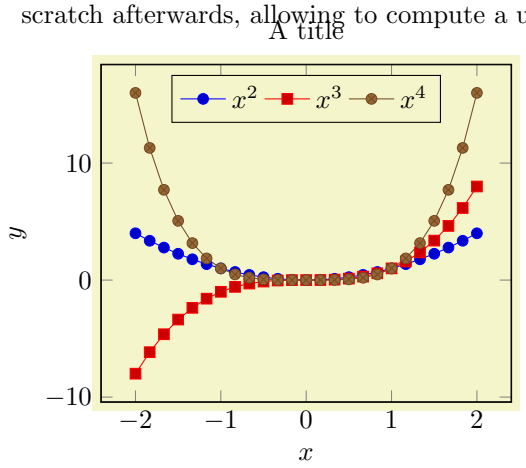



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
   \begin{axis}[
      domain=0:6.2832,samples=200,
      legend style={
         overlay,
         at={(-0.5,0.5)},
         anchor=center},
      every axis plot post/.append style={mark=none},
      enlargelimits=false]

   \addplot {sin(deg(x)+3)+rand*0.05};
   \addplot {cos(deg(x)+2)+rand*0.05};
   \legend{Signal 1,Signal 2}
   \end{axis}
\end{tikzpicture}
```

More information about the `overlay` option can be found in the PGF manual [5].

\pgfresetboundingbox

This command of PGF resets the bounding box of the current picture. The computation starts from scratch afterwards, allowing to compute a user–defined bounding box.

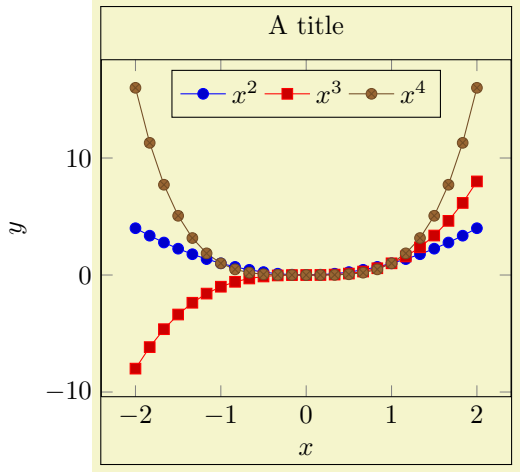



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\setlength{\fboxsep}{0pt}%
\fbox{%
\begin{tikzpicture}%
    \begin{axis}[
        title=A title,
        xlabel={$x$},
        ylabel={$y$},
        legend style={at={(0.5,0.97)},
            anchor=north,legend columns=-1},
        domain=-2:2
    ]
    \addplot {x^2};
    \addplot {x^3};
    \addplot {x^4};
    \legend{$x^2$,$x^3$,$x^4$}
    \end{axis}

    \pgfresetboundingbox
    \path
                (current axis.south west)
    rectangle (current axis.north east);
\end{tikzpicture}%
}%
```

The example draws a normal picture, containing an axis. Afterwards, it throws the bounding box away and creates a new one based on the `current axis` node and its anchors.

```
\begin{pgfinterruptboundingbox}
  ⟨environment contents⟩
\end{pgfinterruptboundingbox}
```

Yet another approach with the same effect is shown below: the bounding box is interrupted manually, and resumed afterwards.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\setlength{\fboxsep}{0pt}%
\fbox{%
\begin{tikzpicture}%
    \begin{pgfinterruptboundingbox}
    \begin{axis}[
        title=A title,
        xlabel={$x$},
        ylabel={$y$},
        legend style={at={(0.5,0.97)},
            anchor=north,legend columns=-1},
        domain=-2:2
    ]
    \addplot {x^2};
    \addplot {x^3};
    \addplot {x^4};
    \legend{$x^2$,$x^3$,$x^4$}
    \end{axis}
    \end{pgfinterruptboundingbox}

    \useasboundingbox
              (current axis.below south west)
    rectangle (current axis.above north east);
\end{tikzpicture}%
}%
```

The pgfinterruptboundingbox environment does not include its content into the image's bounding box, and \useasboundingbox sets the pictures bounding box to the following argument (see [5]).

/tikz/trim left={⟨*x coordinate or point*⟩} (default 0pt)
/tikz/trim right={⟨*x coordinate or point*⟩}

These two keys allow to reduce the size of the bounding box.

The trim left key expects either a single x coordinate like 1cm or a point like (current axis.west). If a point is provided, is uses only the x coordinate of that point. Then, the left end of the bounding box is set to the resulting x coordinate and everything left of it is outside of the bounding box.

The trim right key has the same effect, only for the right end of the bounding box.

More detailed documentation can be found in the Ti*k*Z manual.

/tikz/trim axis left (style, no value)

A style with value trim left=(current axis.south west).

The style needs to be provided as argument to \begin{tikzpicture}[trim axis left]. It expects (at least) one PGFPLOTS environment in the picture. The effect is to trim everything which is left of the last axis' anchor south west (i.e. everything left of the left axis boundary).

/tikz/trim axis right (style, no value)

A style with value trim right=(current axis.south east).

It works similarly to trim axis left: the effect is that everything right of the right axis line of the last axis environment is truncated from the bounding box.

/tikz/trim axis group left (style, no value)

A style which has the same effect as trim axis left, but is tailored for the groupplots library.

It has the value trim left=(group c1r1.south west).

The style needs to be provided as argument to \begin{tikzpicture}[trim axis group left]. It expects (at least) one groupplot environment in the picture. The effect is to trim everything which is left of the first group axis' anchor south west (i.e. everything left of the left axis boundary).

/tikz/trim axis group right (style, no value)

A style which has the same effect as trim axis right, but is tailored for the groupplots library.

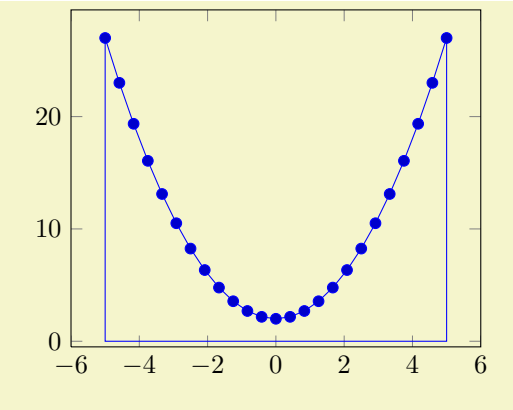
It works similarly to trim axis group left: the effect is that everything right of the rightmost axis in a group plot (the last element of the groupplot environment) is truncated from the bounding box.

4.19 Closing Plots (Filling the Area Under Plots)

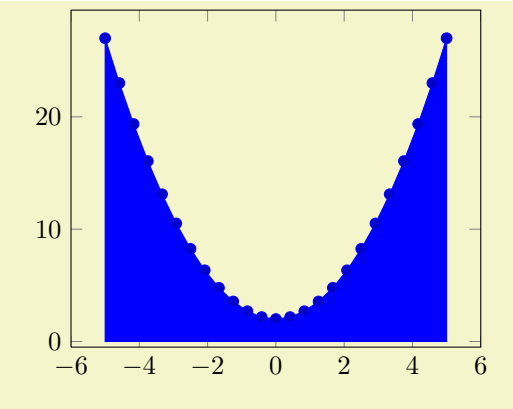
\closedcycle

Provide \closedcycle as ⟨*trailing path commands*⟩ after \addplot to draw a closed line from the last plot coordinate to the first one.

Use \closedcycle whenever you intend to fill the area under a plot.

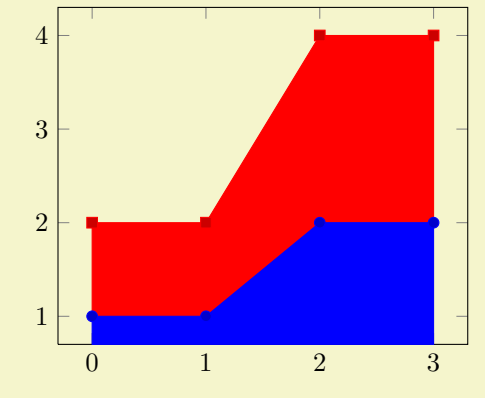



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot {x^2+2} \closedcycle;
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot+[fill] {x^2+2} \closedcycle;
    \end{axis}
\end{tikzpicture}
```

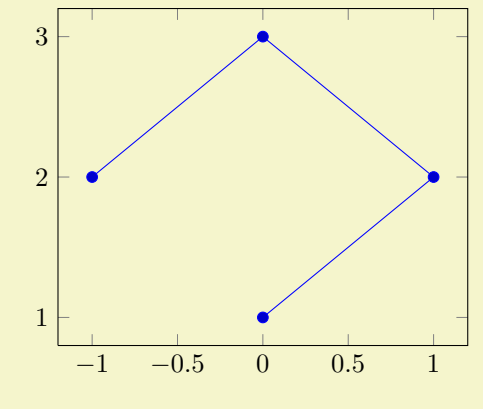
In case of stacked plots, \closedcycle connects the current plot with the previous plot instead of connecting with the x axis[51].



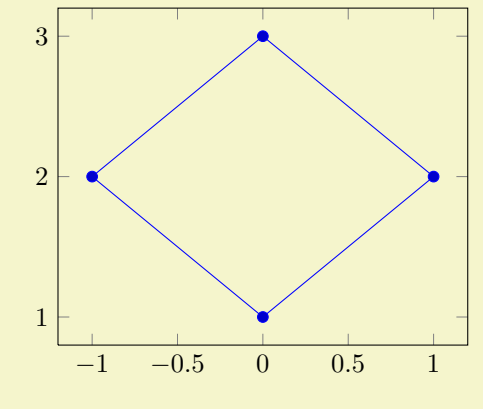

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[stack plots=y]
    \addplot+[fill] coordinates
        {(0,1) (1,1) (2,2) (3,2)} \closedcycle;
    \addplot+[fill] coordinates
        {(0,1) (1,1) (2,2) (3,2)} \closedcycle;
    \end{axis}
\end{tikzpicture}
```

Note that \closedcycle has been designed for functions (i.e. for a plot where every x has at most one y value). For arbitrary curves, you can safely use the TikZ path --cycle instead which simply connects the last and the first path element:

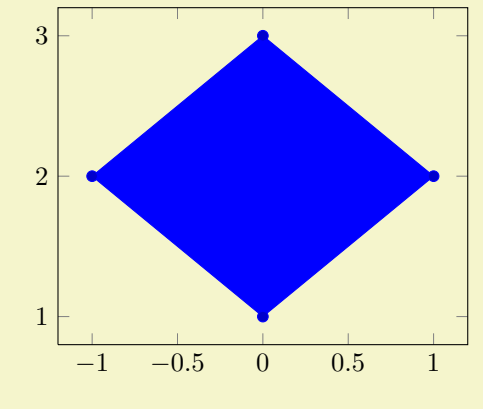
[51] The implementation for stacked plots requires some additional logic to determine the filled area: `\closedcycle` will produce a `plot coordinates` command with *reversed* coordinates of the previous plot. This is usually irrelevant for end users, but it assumes that the plot's type is symmetric. Since constant plots are inherently asymmetric, `\closedcycle` will use `const plot mark right` as reversed sequence for `const plot mark left`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot coordinates
        {(0,1) (1,2) (0,3) (-1,2)};
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot coordinates
        {(0,1) (1,2) (0,3) (-1,2)} --cycle;
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot+[fill] coordinates
        {(0,1) (1,2) (0,3) (-1,2)} --cycle;
    \end{axis}
\end{tikzpicture}
```

The `--cycle` is actually a path instruction of [5]; it connects the first and the last coordinate of one path. Note that this is automatically done for `fill`ed paths.

4.20 Symbolic Coordinates and User Transformations

PGFPLOTS supports user transformations which can be applied to input and output coordinates. Suppose the plot shall display days versus account statements over time. Then, one wants to visualize date versus credit balance. But: dates need to be transformed to numbers before doing so! Furthermore, tick labels shall be displayed as dates as well. This, and more general transformations, can be realized using the `x coord trafo` and `y coord trafo` keys.

Remark: This section applies to users who want to have non-standard input *coordinates*. If you have normal numbers which don't need to be transformed and you like to have special symbols as tick labels, you should consider using the `xticklabels` (`yticklabels`) key described on page 224.

`/pgfplots/x coord trafo/.code={⟨...⟩}`
`/pgfplots/y coord trafo/.code={⟨...⟩}`

/pgfplots/z coord trafo/.code={⟨...⟩}
/pgfplots/x coord inv trafo/.code={⟨...⟩}
/pgfplots/y coord inv trafo/.code={⟨...⟩}
/pgfplots/z coord inv trafo/.code={⟨...⟩}

These code keys allow arbitrary coordinate transformations which are applied to input coordinates and output tick labels.

The `x coord trafo` and `y coord trafo` command keys take one argument which is the input coordinate. They are expected to set `\pgfmathresult` to the final value.

At this level, the input coordinate is provided as it is found in the `\addplot` statement. For example, if x coordinates are actually of the form ⟨*year*⟩-⟨*month*⟩-⟨*day*⟩, for example `2008-01-05`, then a useful coordinate transformation would transform this string into a number (see below for a predefined realization).

In short, *no* numerics has been applied to input coordinates when this transformation is applied[52].

The input coordinate transformation is applied to

- any input coordinates (specified with `\addplot` or `axis cs`),
- any user-specified `xtick` or `ytick` options,
- any user-specified `extra x ticks` and `extra y ticks` options,
- any user-specified axis limits like `xmin` and `xmax`.

The output coordinate transformation `x coord inv trafo` is applied to tick positions just before evaluating the `xticklabel` and `yticklabel` keys. The argument to `x coord inv trafo` is a fixed point number (which may have trailing zeros after the period). The tick label code may use additional macros defined by the inverse transformation.

Remark: PGFPLOTS will continue to produce tick positions as usual, no extra magic is applied. It may be necessary to provide tick positions explicitly if the default doesn't respect the coordinate space properly.

The initial value of these keys is

```
\pgfplotsset{
    x coord trafo/.code={},
    x coord inv trafo/.code={}}
```

which simply disables the transformation (the same for y, of course).

Remark: It might be necessary to set

```
\pgfplotsset{
    xticklabel={\tick},
    scaled x ticks=false,
    plot coordinates/math parser=false,
}
```

in order to avoid number formatting routines on `\tick` or numerics for tick scale methods. This is done automatically by the predefined symbolic coordinate styles (see below).

4.20.1 String Symbols as Input Coordinates

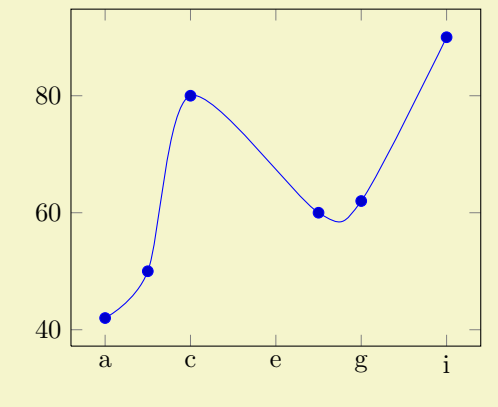
It is possible to provide a string dictionary to PGFPLOTS. An input coordinate can then use any symbol provided in that dictionary.

/pgfplots/symbolic x coords={⟨*dictionary*⟩}
/pgfplots/symbolic y coords={⟨*dictionary*⟩}
/pgfplots/symbolic z coords={⟨*dictionary*⟩}

A style which sets `x coord trafo` and `x coord inv trafo` (or the respective `y` or `z` variants) such that any element in ⟨*dictionary*⟩ is a valid input coordinate. The ⟨*dictionary*⟩ can be a comma separated list or a list terminated with '\\'. In both cases, white space is considered to be part of the names (use '%' at end of lines).

[52] Of course, if coordinates have been generated by gnuplot or PGF, this does no longer hold.

The dictionary will assign integer numbers to every element. These integers are used internally for arithmetics. Finally, the inverse transformation takes a fixed point number and maps it to the nearest integer, and that integer is mapped into the dictionary.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[symbolic x coords={a,b,c,d,e,f,g,h,i}]
    \addplot+[smooth] coordinates {
        (a,42)
        (b,50)
        (c,80)
        (f,60)
        (g,62)
        (i,90)};
\end{axis}
\end{tikzpicture}
```

The effect of the transformation is simply that input coordinates can be elements of the dictionary and tick labels will be chosen out of this dictionary as well.

See also the option to add tick and/or grid lines at every encountered coordinate using `xtick=data` (or `minor xtick=data`).

4.20.2 Dates as Input Coordinates

The already mentioned application of using dates as input coordinates has been predefined, together with support for hours and minutes. It relies on the PGF calendar library which converts dates to numbers in the Julian calendar. Then, one coordinate unit is one day.

```
\usepgfplotslibrary{dateplot} % LaTeX and plain TeX
\usepgfplotslibrary[dateplot] % ConTeXt
\usetikzlibrary{pgfplots.dateplot} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.dateplot] % ConTeXt
```

Loads the coordinate transformation code.

/pgfplots/date coordinates in=⟨*coordinate*⟩

Installs `x coord trafo` and `x coord inv trafo` (or the respective variant for ⟨*coordinate*⟩) such that ISO dates of the form ⟨*year*⟩-⟨*month*⟩-⟨*day*⟩ are accepted. Here, ⟨*coordinate*⟩ is usually one of `x`, `y`, or `z`, but it can also contain stuff like `hist/data`.

After installing this style, input values like `2006-02-28` will be converted to an "appropriate" integer using the Julian calender. Input coordinates may be of the form

⟨*year*⟩-⟨*month*⟩-⟨*day*⟩

or they may contain times as

⟨*year*⟩-⟨*month*⟩-⟨*day*⟩ ⟨*hour*⟩:⟨*minute*⟩.

The result of the transformation are numbers where one unit is one day and times are fractional numbers.

The transformation is realized using the PGF-calendar module, see [5, Calendar Library]. This reference also contains more information about extended syntax options for dates.

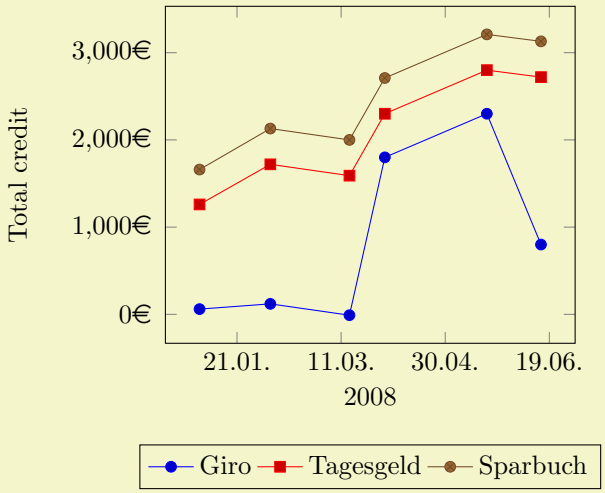
The inverse transformation provides the following macros which are available during tick label evaluation (i.e. when used inside of `xticklabel` or `yticklabel`):

- `\year` expands to the year component,
- `\month` expands to the month component,
- `\day` expands to the day component,
- `\hour` expands to the hour component (using two digits),
- `\Hour` expands to the hour component (but omits leading zeros),

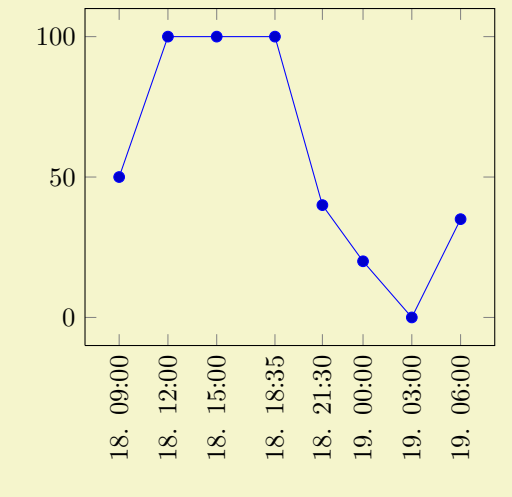
- `\minute` expands to the minute component (two digits),
- `\Minute` expands to the minute component (omits leadings zeros),
- `\lowlevel` expands to the low level number representing the tick,
- `\second` will always be `00`.

This allows to use `\day.\month.\year` or `\day. \hour:\minute` inside of `xticklabel`, for example. A complete example (with fictional data) is shown below.

date	account1	account2	account3
2008-01-03	60	1200	400
2008-02-06	120	1600	410
2008-03-15	-10	1600	410
2008-04-01	1800	500	410
2008-05-20	2300	500	410
2008-06-15	800	1920	410




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usepgfplotslibrary{dateplot} !

\pgfplotstabletypeset[string type]{plotdata/accounts.dat}

\begin{tikzpicture}
    \begin{axis}[
        date coordinates in=x,
        xticklabel={\day.\month.},
        xlabel={2008},
        stack plots=y,
        yticklabel={\pgfmathprintnumber{\tick}\EUR{}}, %  <- requires \usepackage{eurosym}
        ylabel=Total credit,
        ylabel style={yshift=10pt},
        legend style={
            at={(0.5,-0.3)},anchor=north,legend columns=-1}]

    \addplot table[x=date,y=account1] {plotdata/accounts.dat};
    \addplot table[x=date,y=account2] {plotdata/accounts.dat};
    \addplot table[x=date,y=account3] {plotdata/accounts.dat};
    \legend{Giro,Tagesgeld,Sparbuch}
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usepgfplotslibrary{dateplot} !
\begin{tikzpicture}
  \begin{axis}[
    date coordinates in=x,
    xtick=data,
    xticklabel style=
        {rotate=90,anchor=near xticklabel},
    xticklabel=\day. \hour:\minute,
    date ZERO=2009-08-18,%  <- improves precision!
  ]
  \addplot coordinates {
    (2009-08-18 09:00,  050)
    (2009-08-18 12:00,  100)
    (2009-08-18 15:00,  100)
    (2009-08-18 18:35,  100)
    (2009-08-18 21:30,  040)
    (2009-08-19,        020)
    (2009-08-19 3:00,   000)
    (2009-08-19 6:0,    035)
  };
  \end{axis}
\end{tikzpicture}
```

Attention: If you intend to use hours and minutes, you should *always* provide the `date ZERO` to maintain adequate precision!

`/pgfplots/date ZERO=`⟨*year*⟩`-`⟨*month*⟩`-`⟨*day*⟩ (initially `2006-01-01`)

A technical key which defines the 0 coordinate of `date coordinates in`. Users will never see the resulting numbers, so one probably never needs to change it. However, the resulting numbers may become very large and a mantisse of 6 significant digits may not be enough to get accurate results. In this case, `date ZERO` should be set to a number which falls into the input date range.

4.21 Skipping Or Changing Coordinates – Filters

`/pgfplots/x filter/.code=`{⟨...⟩}
`/pgfplots/y filter/.code=`{⟨...⟩}
`/pgfplots/z filter/.code=`{⟨...⟩}
`/pgfplots/filter point/.code=`{⟨...⟩}

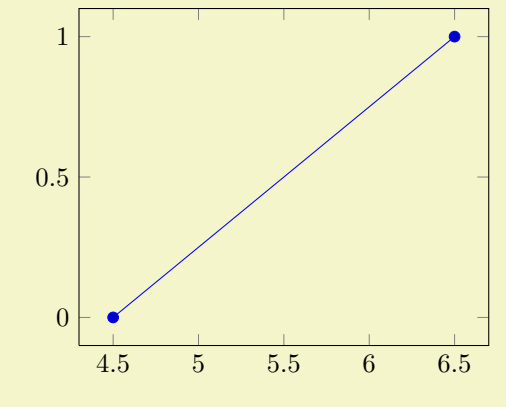
The code keys `x filter` and `y filter` allow coordinate filtering which are based on a *single* coordinate. A coordinate filter gets an input coordinate as `#1` (on input, the same value is stored in `\pgfmathresult`), applies some operation and writes the result into the macro `\pgfmathresult`. If `\pgfmathresult` is empty afterwards, the coordinate is discarded. You can also set `\pgfmathresult` to `nan` or `inf` in which case the coordinate can be either discarded (if `unbounded coords=discard` is set) or the plot can be interrupted (the case `unbounded coords=jump`).

The `filter point/.code` filter allows filtering depending on all components forming a complete point (x, y and z); it is described below.

It is allowed that filters do not change `\pgfmathresult`. In this case, the unfiltered coordinate will be used.

Coordinate filters are useful in automatic processing system, where PGFPLOTS is used to display automatically generated plots. You may not want to filter your coordinates by hand, so these options provide a tool to do this automatically.

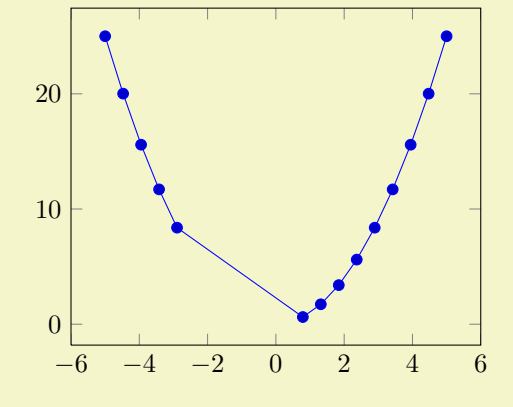
The following filter adds 0.5 to every x coordinate.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[x filter/.code=
    {\pgfmathadd{#1}{0.5}}]
\addplot coordinates {
    (4,0)
    (6,1)
};
\end{axis}
\end{tikzpicture}
```

Please refer to [5, pgfmath manual] for details about the math engine of PGF. Please keep in mind that the math engine works with limited TEX precision.

During evaluation of the filter, the macro `\coordindex` contains the number of the current coordinate (starting with 0). Thus, the following filter discards all coordinates after the 5th and before the 10th.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    samples=20,
    x filter/.code={
        \ifnum\coordindex>4
            \ifnum\coordindex<11
                \def\pgfmathresult{}
            \fi
        \fi
    }]
\addplot {x^2};
\end{axis}
\end{tikzpicture}
```

There is also a style key which simplifies selection by index, see below.

PGFPLOTS invokes the filter with argument `#1` set to the input coordinate. For x-filters, this is the x-coordinate as it is specified to `\addplot`, for y-filters it is the y-coordinate.

If the corresponding axis is logarithmic, `#1` is the *logarithm* (see `log basis x` and its variants) of the coordinate as a real number, for example `#1=4.2341`. In case the logarithm was undefined, the argument will be empty.

The arguments to coordinate filters are minimally preprocessed: first, for logarithmic axes, the *log* of the argument is supplied. Second, any high level coordinate maps like `x coord trafo` (which may be used to map dates to numbers or string to numbers or so) are applied. In consequence, the `#1` argument is supposed to be a number. No further transformation has been applied.

Occasionally, it might be handy to get the "raw", completely unprocessed input coordinate as it has been reported by the coordinate input routine. This unprocessed data is available in the three math parser constants `rawx`, `rawy` and `rawz` (use `\pgfmathrawx`, `\pgfmathrawy` and `\pgfmathrawz` as a way to assign the value of interest to `\pgfmathresult`). All these values are ready for use in filters (and some other methods influence plots as well).

If key filters are invoked for `plot table`, access to the current row's data can be achieved using `\thisrow{`⟨*column name*⟩`}` (and its variants). This includes all columns of the table.

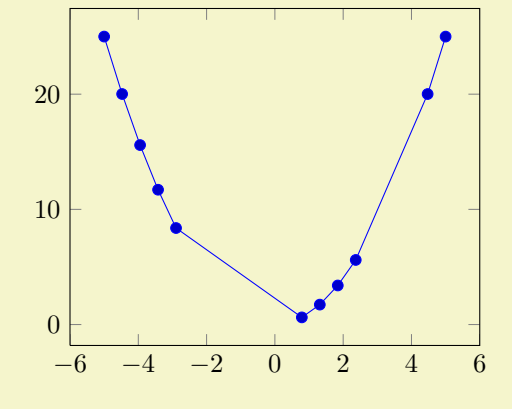
The `filter point` key is more technical. It doesn't take an argument: its arguments are given in terms of the `pgfkeys` variables `/data point x`, `/data point y` and `/data point z`. It may change its coordinates using `\pgfkeyssetvalue{/data point x}{`⟨*new value*⟩`}`; access to variables can be accessed with `\pgfkeysvalueof{/data point/x}` or, if the argument shall be written into a macro, with `\pgfkeysgetvalue`. This filter is evaluated after the other ones.

Note that you can provide different `x filter`/`y filter` arguments to each `\addplot` command. It seems there are only problems with the '`#1`' argument, and I haven't yet found out why. Please use

\pgfmathresult in place of #1 if you provide \addplot[x filter/.code={...}].

/pgfplots/skip coords between index={⟨*begin*⟩}{⟨*end*⟩}

A style which appends an x filter which discards selected coordinates. The selection is done by index where indexing starts with 0, see \coordindex. Every coordinate with index $\langle begin\rangle \le i < \langle end\rangle$ will be skipped.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    samples=20,
    skip coords between index={5}{11},
    skip coords between index={15}{18}]

\addplot {x^2};
\end{axis}
\end{tikzpicture}
```

Technical note : this style usually applies to x coordinates (i.e. it counts x coordinates). In case you want to apply it to something like hist/data or quiver/u, you can

1. append an asterisk '*' to the style's name and
2. provide the target coordinate's name as first argument.

For example, skip coords between index*={hist/data}{2} applies to hist/data.

/pgfplots/each nth point={⟨*integer*⟩}

A style which appends an x filter which discards all but each nth input coordinate.

Note that there is also a mark repeat style which applies the same operation to plot marks only.

Technical note : this style usually applies to x coordinates (i.e. it counts x coordinates). In case you want to apply it to something like hist/data or quiver/u, you can

1. append an asterisk '*' to the style's name and
2. provide the target coordinate's name as first argument.

For example, each nth point*={hist/data}{2} applies to hist/data.

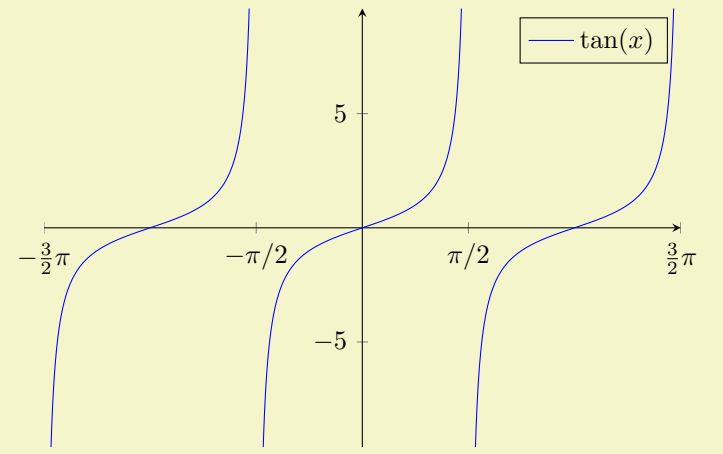
/pgfplots/restrict x to domain=⟨*min*⟩:⟨*max*⟩
/pgfplots/restrict y to domain=⟨*min*⟩:⟨*max*⟩
/pgfplots/restrict z to domain=⟨*min*⟩:⟨*max*⟩
/pgfplots/restrict x to domain*=⟨*min*⟩:⟨*max*⟩
/pgfplots/restrict y to domain*=⟨*min*⟩:⟨*max*⟩
/pgfplots/restrict z to domain*=⟨*min*⟩:⟨*max*⟩

These keys append x (or y or z) coordinate filters to restrict the respective coordinate to a domain.

The versions without star (like restrict x to domain) will assign the value -inf if the coordinate is below ⟨*min*⟩ and +inf if the coordinate is above ⟨*max*⟩. The starred versions (like restrict x to domain*) will truncate coordinates to [⟨*min*⟩, ⟨*max*⟩], i.e. they assign the value ⟨*min*⟩ if the coordinate falls outside of the lower limit and ⟨*max*⟩ if the value falls outside of the upper limit.

For logarithmic axes, ⟨*min*⟩ and ⟨*max*⟩ are *logs* of the respective values. A variant which uses the non-logarithmic number might be to use restrict expr to domain={\pgfmathrawx}{⟨*min*⟩}{⟨*max*⟩}.

The non-starred versions also set unbounded coords=jump which leads to interrupted plots.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    restrict y to domain=-10:10,
    samples=1000,
    %  some fine-tuning for the display:
    width=10cm, height=210pt,
    xmin=-4.7124, xmax=4.7124,
    xtick={-4.7124,-1.5708,...,10},
    xticklabels={$-\frac32 \pi$,$-\pi/2$,$\pi/2$,$\frac32 \pi$},
    axis x line=center,
    axis y line=center]

\addplot[blue] gnuplot[id=tangens,domain=-1.5*pi:1.5*pi] {tan(x)};
\legend{$\tan(x)$}
\end{axis}
\end{tikzpicture}
```

/pgfplots/restrict expr to domain={⟨*expression*⟩}{⟨⟨*min*⟩:⟨*max*⟩⟩}
/pgfplots/restrict expr to domain*={⟨*expression*⟩}{⟨⟨*min*⟩:⟨*max*⟩⟩}

Appends an x coordinate filter which sets the x coordinate to `-inf` if the ⟨*expression*⟩ evaluates to something less than ⟨*min*⟩ and to `inf` if ⟨*expression*⟩ evaluates to something larger than ⟨*max*⟩.

The starred variant, `restrict to domain*` assigns ⟨*min*⟩ if ⟨*expression*⟩ is less then the lower limit and ⟨*max*⟩ if it is larger than the upper limit.

The non-starred version also sets `unbounded coords=jump` which leads to interrupted plots.

In contrast to `restrict x to domain`, ⟨*expression*⟩ can depend on anything which is valid during `\addplot`, in particular `\coordindex` or table columns (`\thisrow{`⟨*column name*⟩`}` and friends). The expression doesn't need to depend on x at all.

/pgfplots/@restrict to domain={⟨*filter name*⟩}{⟨*expression*⟩}{⟨⟨*min*⟩:⟨*max*⟩⟩}{0|1}

A low–level (technical) key which allows to apply the `restrict * to ...` features also to something like `hist/data`.

For example, `@restrict to domain={hist/data}{}{0:1}{0}` applies the domain-restriction to the histogram-input `hist/data`. The final '0' means that it works as the key `restrict x to domain=0:1`, i.e. it skips everything which is outside of $[0,1]$. In a similar way, `@restrict to domain={hist/data}{}{0:1}{1}` applies the functionality of `restrict x to domain*=0:1` to `hist/data`: it truncates values outside of $[0,1]$ to the domain's end-points.

The ⟨*filter name*⟩ is expected to be a coordinate name like `x`, `y`, `z` (or `hist/data`).

The ⟨*expression*⟩ configures an expression which will be used rather than the value of ⟨*filter name*⟩. It can be empty.

The ⟨*min*⟩:⟨*max*⟩ are as described above.

If the last argument is `1`, any coordinate outside of the allowed domain will take the domain boundary as value. If it is `0`, such a coordinate will get either `inf` or `-inf`.

/pgfplots/filter discard warning=true|false (initially `true`)

Issues a notification in your logfile whenever coordinate filters discard coordinates.

You can find somewhat more on coordinate filtering in Section 4.4.12: “Interrupted Plots”.

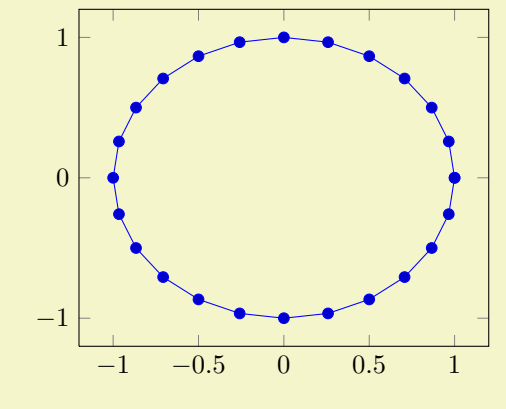
4.22 Transforming Coordinate Systems

Usually, PGFPLOTS works with cartesian coordinates. However, one may want to provide coordinates in a different coordinate system.

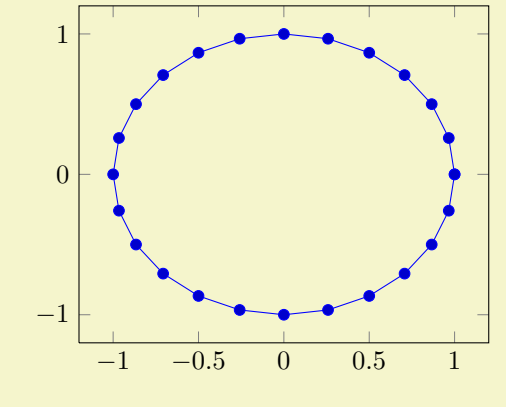
In this case, the `data cs` key can be used to identify the input coordinate system:

`/pgfplots/data cs=cart|polar|polarrad` (initially `cart`)

Defines the coordinate system (‘cs’) of the input coordinates. PGFPLOTS will apply transformations if ⟨*name*⟩ does not match the expected coordinate system.

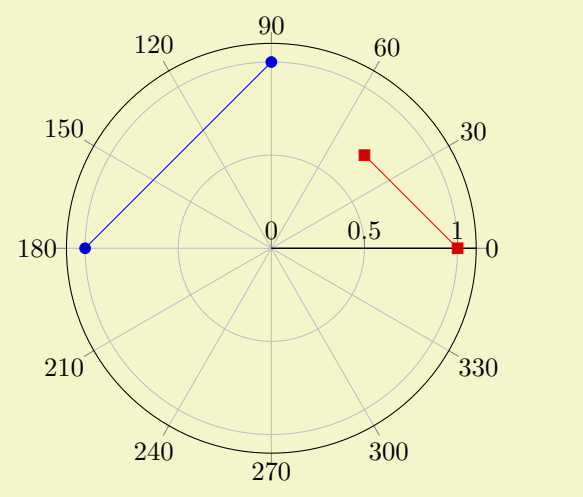



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot+[data cs=polar,domain=0:360] (\x,1);
    \end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}
    \addplot+[data cs=polarrad,domain=0:2*pi] (\x,1);
    \end{axis}
\end{tikzpicture}
```

Every axis type has its own coordinate system. For example, a normal `axis` expects the `cart` coordinate system, whereas a `polaraxis` expects a `polar` coordinate system. If the argument to `data cs` does not match the expected coordinate system, PGFPLOTS will transform it:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usepgfplotslibrary{polar}
\begin{tikzpicture}
    \begin{polaraxis}
    \addplot coordinates {(90,1) (180,1)};
    \addplot+[data cs=cart]
        coordinates {(1,0) (0.5,0.5)};
    \end{polaraxis}
\end{tikzpicture}
```

At the time of this writing, PGFPLOTS supports the following values for ⟨*name*⟩:

The `data cs=cart` denotes the cartesian coordinate system. It is expected by the usual `axis` (or its logarithmic variants). It can have three components, x, y, and z.

The `data cs=polar` is the (two–dimensional) coordinate system with (angle, radius). The angle is a periodic number in the range $[0, 360)$; the radius is any number. If a `polar` coordinate has a z component, it is taken as-is (the transformations ignore it).

The `data cs=polarrad` is similar to `polar`, but it expects the angle in radians, i.e. in the periodic range $[0, 2\pi)$.

At the point of this writing, the `data cs` method will work for most plot handlers. But for complicated plot handlers, further logic may be needed which is not yet available (for example, the `quiver` plot handler might not be able to convert its direction vectors correctly)[53].

`\pgfplotsaxistransformcs{`⟨*fromname*⟩`}{`⟨*toname*⟩`}`

Expects the current point in a set of keys, provided in the coordinate system ⟨*fromname*⟩ and replaces them by the same coordinates represented in ⟨*toname*⟩.

On input, the coordinates are stored in `/data point/x`, `/data point/y`, and `/data point/z` (the latter may be empty). The macro will test if there is a declared coordinate transformation from ⟨*fromname*⟩ to ⟨*toname*⟩ and invoke it. If there is none, it will attempt to convert to `cart` first and then from `cart` to ⟨*toname*⟩. If that does not exist either, the operation fails.

`\pgfplotsdefinecstransform{`⟨*fromname*⟩`}{`⟨*toname*⟩`}{`⟨*code*⟩`}`

Defines a new coordinate system transformation. The ⟨*code*⟩ is expected to get input and write output as described for `\pgfplotsaxistransformcs`.

Implementing a new coordinate system immediately raises the question in which math mode the operations shall be applied. PGFPLOTS supports different so–called "coordinate math systems" for generic operations, and for each individual coordinate as well. These coordinate math systems can either use basic PGF math arithmetics, the `fpu`, or perhaps there will come a LuaTEX library.

The documentation of this system is beyond the scope of this manual[54]. Please consider reading the source-code comments and the source of existing transformations if you intend to write own transformations.

4.23 Fitting Lines – Regression

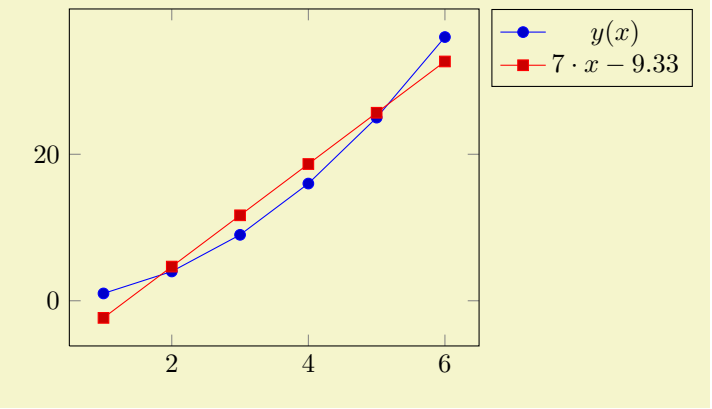
This section documents the attempts of PGFPLOTS to fit lines to input coordinates. PGFPLOTS currently supports `create col/linear regression` applied to columns of input tables. The feature relies on PGFPLOTSTABLE, it is actually implemented as a table postprocessing method.

`/pgfplots/table/create col/linear regression={`⟨*key-value-config*⟩`}`

[53] In case you run into problems, consider writing a bug report or ask others in TEX online discussion forums.
[54] Which is quite comprehensive even without API documentation, as you will certainly agree...

A style for use in `\addplot table` which computes a linear (least squares) regression $y(x) = a \cdot x + b$ using the sample data (x_i, y_i) which has to be specified inside of ⟨*key-value-config*⟩ (see below).

It creates a new column on-the-fly which contains the values $y(x_i) = a \cdot x_i + b$. The values a and b will be stored (globally) into `\pgfplotstableregressiona` and `\pgfplotstableregressionb`.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[legend pos=outer north east]
    \addplot table {%  plot X versus Y. This is original data.
        X Y
        1 1
        2 4
        3 9
        4 16
        5 25
        6 36
    };
    \addplot table[
        y={create col/linear regression={y=Y}}] %  compute a linear regression from the input table
    {
        X Y
        1 1
        2 4
        3 9
        4 16
        5 25
        6 36
    };
    % \xdef\slope{\pgfplotstableregressiona} %<-- might be handy occasionally
    \addlegendentry{$y(x)$}
    \addlegendentry{%
        $\pgfmathprintnumber{\pgfplotstableregressiona} \cdot x
        \pgfmathprintnumber[print sign]{\pgfplotstableregressionb}$}
    \end{axis}
\end{tikzpicture}
```

The example above has two plots: one showing the data and one containing the `linear regression` line. We use `y={create col/linear regression={}}` here, which means to create a new column[55] containing the regression values automatically. As arguments, we need to provide the y column name explicitly[56]. The x value is determined from context: `linear regression` is evaluated inside of `\addplot table`, so it uses the same x as `\addplot table` (i.e. if you write `\addplot table[x={`⟨*col name*⟩`}]`, the regression will also use ⟨*col name*⟩ as its `x` input). Furthermore, it shows the line parameters a and b in the legend.

The following ⟨*key-value-config*⟩ keys are accepted as comma–separated list:

`/pgfplots/table/create col/linear regression/table={`⟨*macro* or *file name*⟩`}` (initially `empty`)

Provides the table from where to load the `x` and `y` columns. It defaults to the currently processed one, i.e. to the value of `\pgfplotstablename`.

[55] The `y={create col/` feature is available for any other PgfplotsTable postprocessing style, see the `create on use` documentation in the PgfplotsTable manual.

[56] In fact, pgfplots sees that there are only two columns and uses the second by default. But you need to provide it if there are at least 3 columns.

/pgfplots/table/create col/linear regression/x={⟨*column*⟩} (initially `empty`)
/pgfplots/table/create col/linear regression/y={⟨*column*⟩} (initially `empty`)

Provides the source of x_i and y_i data, respectively. The argument ⟨*column*⟩ is usually a column name of the input table, yet it can also contain `[index]`⟨*integer*⟩ to designate column indices (starting with 0), `create on use` specifications or `aliases` (see the PgfplotsTable manual for details on `create on use` and `alias`).

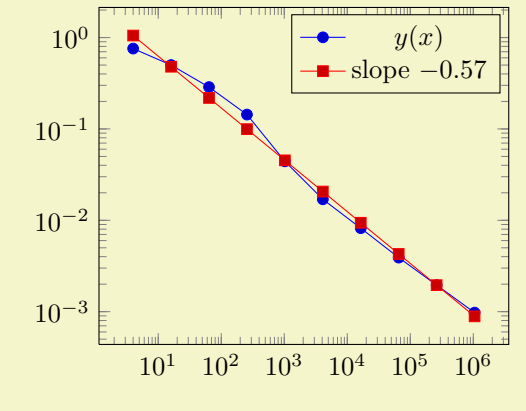
The initial configuration (an empty value) checks the context where the `linear regression` is evaluated. If it is evaluated inside of `\pgfplotstabletypeset`, it uses the first and second table columns. If it is evaluated inside of `\addplot table`, it uses the same x input as the `\addplot table` statement. The `y` key needs to be provided explicitly (unless the table has only two columns).

/pgfplots/table/create col/linear regression/xmode=auto|linear|log (initially `auto`)
/pgfplots/table/create col/linear regression/ymode=auto|linear|log (initially `auto`)

Enables or disables processing of logarithmic coordinates. Logarithmic processing means to apply ln before computing the regression line and exp afterwards.

The choice `auto` checks if the column is evaluated inside of a PGFPLOTS axis. If so, it uses the axis scaling of the embedding axis. Otherwise, it uses `linear`.

In case of logarithmic coordinates, the `log basis x` and `log basis y` keys determine the basis.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}
 \addplot table[x=dof,y=error2]
    {pgfplotstable.example1.dat};
  \addlegendentry{$y(x)$}

 \addplot table[
      x=dof,
      y={create col/linear regression={y=error2}}]
    {pgfplotstable.example1.dat};

  %  might be handy occasionally:
  % \xdef\slope{\pgfplotstableregressiona}
 \addlegendentry{slope
   $\pgfmathprintnumber{\pgfplotstableregressiona}$}
\end{loglogaxis}
\end{tikzpicture}
```

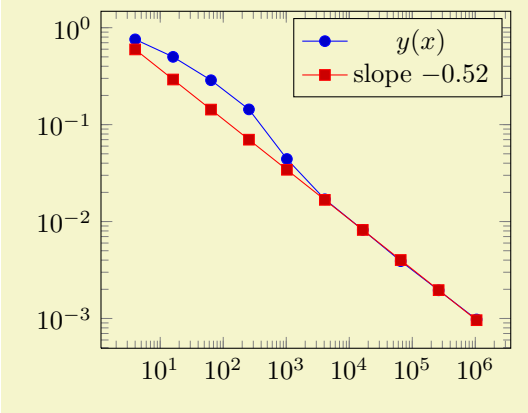
The (commented) line containing `\slope` is explained above; it allows to remember different regression slopes in our example.

/pgfplots/table/create col/linear regression/variance list={⟨*list*⟩} (initially `empty`)
/pgfplots/table/create col/linear regression/variance={⟨*column name*⟩} (initially `empty`)

Both keys allow to provide uncertainties (variances) to single data points. A high (relative) variance indicates an unreliable data point, a value of 1 is standard.

The `variance list` key allows to provide variances directly as comma–separated list, for example `variance list={1000,1000,500,200,1,1}`.

The `variance` key allows to load values from a table ⟨*column name*⟩. Such a column name is (initially, see below) loaded from the same table where data points have been found. The ⟨*column name*⟩ may also be a `create on use` name.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}
 \addplot table[x=dof,y=error2]
    {pgfplotstable.example1.dat};
  \addlegendentry{$y(x)$}

 \addplot table[
      x=dof,
        y={create col/linear regression={
        y=error2,
        variance list={1000,800,600,500,400}}
      }
]
   {pgfplotstable.example1.dat};

 \addlegendentry{slope
  $\pgfmathprintnumber{\pgfplotstableregressiona}$}
\end{loglogaxis}
\end{tikzpicture}
```

If both, `variance list` and `variance` are given, the first one will be preferred. Note that it is not necessary to provide variances for every data point.

`/pgfplots/table/create col/linear regression/variance src={`⟨*\table or file name*⟩`}` (initially `empty`)

Allows to load the `variance` from another table. The initial setting is empty. It is acceptable if the `variance` column in the external table has fewer entries than expected, in this case, only the first ones will be used.

Limitations: Currently, PGFPLOTS supports only linear regression, and it only supports regression together with `\addplot table`. Furthermore, long input tables might need quite some time.

4.24 Miscellaneous Options

`/pgfplots/disablelogfilter=true|false` (initially `false, default true`)

Disables numerical evaluation of $\log(x)$ in TEX. If you specify this option, any plot coordinates and tick positions must be provided as $\log(x)$ instead of x. This may be faster and – possibly – more accurate than the numerical log. The current implementation of $\log(x)$ normalizes x to $m \cdot 10^e$ and computes

$$\log(x) = \log(m) + e\log(10)$$

where $y = \log(m)$ is computed with a Newton method applied to $\exp(y) - m$. The normalization involves string parsing without TEX-registers. You can safely evaluate $\log(1 \cdot 10^{-7})$ although TEX-registers would produce an underflow for such small numbers.

`/pgfplots/disabledatascaling=true|false` (initially `false, default true`)

Disables internal re-scaling of input data. Normally, every input data like plot coordinates, tick positions or whatever, are parsed without using TEX's limited number precision. Then, a transformation like

$$T(x) = 10^{q-m} \cdot x - a$$

is applied to every input coordinate/position where m is "the order of x" base 10. Example: $x = 1234 = 1.234 \cdot 10^3$ has order $m = 4$ while $x = 0.001234 = 1.234 \cdot 10^{-3}$ has order $m = -2$. The parameter q is the order of the axis' width/height.

The **effect** of the transformation is that your plot coordinates can be of *arbitrary magnitude* like 0.0000001 and 0.0000004. For these two coordinates, PGFPLOTS will use 100pt and 400pt internally. The transformation is quite fast since it relies only on period shifts. This scaling allows precision beyond TEX's capabilities.

The option "`disabledatascaling`" disables this data transformation. This has two consequences: first, coordinate expressions like `(`⟨`axis cs:`*x,y*⟩`)` have the same effect as `(`⟨*x,y*⟩`)`, no re-scaling is applied. Second, coordinates are restricted to what TEX can handle[57].

[57] Please note that the axis' scaling requires to compute $1/(x_{\max} - x_{\min})$. The option `disabledatascaling` may lead to overflow or underflow in this context, so use it with care! Normally, the data scale transformation avoids this problem.

So far, the data scale transformation applies only to normal axes (logarithmic scales do not need it).

/pgfplots/execute at begin plot={⟨*commands*⟩}

This axis option allows to invoke ⟨*commands*⟩ at the beginning of each `\addplot` command. The argument ⟨*commands*⟩ can be any TEX content.

You may use this in conjunction with `x filter=...` to reset any counters or whatever. An example would be to change every 4th coordinate.

/pgfplots/execute at end plot={⟨*commands*⟩}

This axis option allows to invoke ⟨*commands*⟩ after each `\addplot` command. The argument ⟨*commands*⟩ can be any TEX content.

/pgfplots/execute at begin axis={⟨*commands*⟩}

Allows to invoke ⟨*commands*⟩ at the end of `\begin{axis}` (or the other "begin axis" statements).

The statement is execute as (almost) last statement before the preparation has been completed.

/pgfplots/execute at end axis={⟨*commands*⟩}

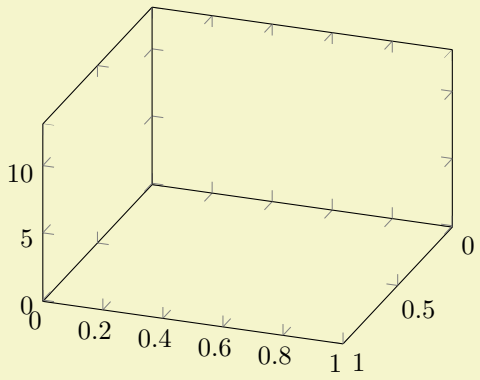
The counterpart for `execute at begin axis`. The hook is actually superfluos, it is executed immediately after `before end axis`. It is executed in the same TEX group as `execute at begin axis`.

/pgfplots/execute at begin plot visualization={⟨*commands*⟩}

Allows to add customized code which is executed at the beginning of each plot visualization. In contrast to `execute at begin plot`, this happens not immediately during `\addplot`, but late during the postprocessing of `\end{axis}` when actual drawing commands are generated.

One possible application is shown below: suppose you want to use `\usepackage{ocg}` in order to switch layers dynamically, for example in a beamer package. This can be implemented as follows:

Dynamic PDF Layer Support (see Acrobat Layers)



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usepackage[pdftex]{ocg}
\begin{tikzpicture}
\begin{axis}[
    title=Dynamic PDF Layer Support (see Acrobat Layers),
    view={110}{35}]
\addplot3+[
    execute at begin plot visualization=\begin{ocg}{First Layer}{FirstLayer}{0},
    execute at end plot visualization=\end{ocg},
]
    coordinates {(0,0,12) (0,1,2) (1,0,6) (0,0,12)};

\addplot3+[
    execute at begin plot visualization=\begin{ocg}{Second Layer}{SecondLayer}{0},
    execute at end plot visualization=\end{ocg},
]
    coordinates {(0,0,9) (0,1,8) (1,0,4) (0,0,9)};

\addplot3+[
    execute at begin plot visualization=\begin{ocg}{Third Layer}{ThirdLayer}{0},
    execute at end plot visualization=\end{ocg},
]
    coordinates {(0,0,1) (0,1,7) (1,0,3) (0,0,1)};
\end{axis}
\end{tikzpicture}
```

The `execute *` hooks insert the OCG-statements at the correct positions, and the single plot commands are added to different dynamic layers. Use the Acrobat Reader and its "Layers" Tab to switch each of them on or off. Note that it would not be enough to add the `\begin{ocg}...` statements right into the text since PGFPLOTS postpones drawing commands until `\end{axis}` (splitting of survey and visualization phase).

See http://www.texample.net/weblog/2008/nov/02/creating-pdf-layers for more details on OCG and how to obtain it.

Technical note: these hooks are also inserted for `\pgfplotsextra` commands.

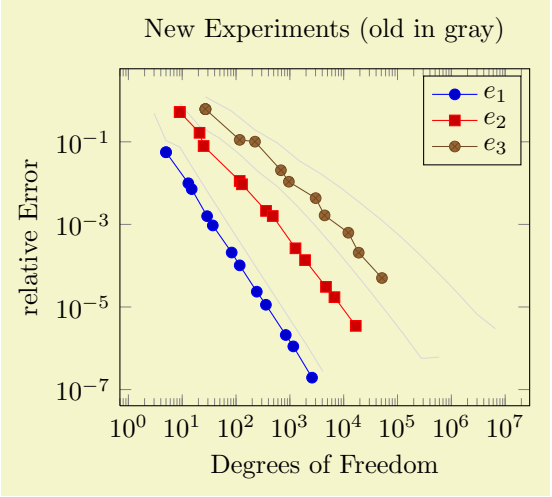
/pgfplots/execute at end plot visualization={⟨*commands*⟩}

This is the counter–part of `execute at begin plot visualization`.

/pgfplots/forget plot={⟨*true,false*⟩} (initially `false`)

Allows to include plots which are not remembered for legend entries, which do not increase the number of plots and which are not considered for cycle lists.

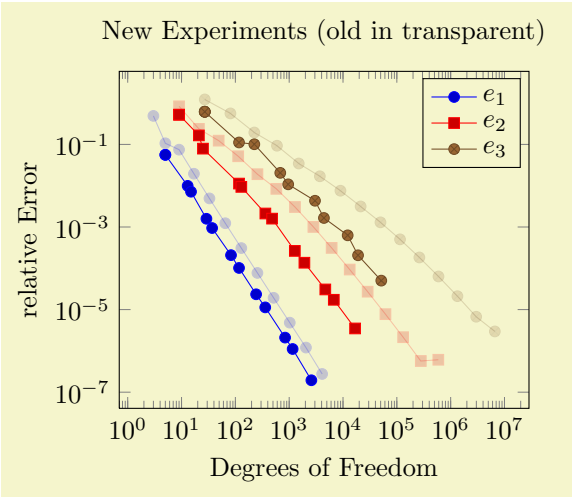
A forgotten plot can be some sort of decoration which has a separate style and does not influence the axis state, although it is processed as any other plot. Provide this option to `\addplot` as in the following example.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[
        %  some descriptions:
        table/x=Basis,
        table/y={L2/r},
        xlabel=Degrees of Freedom,
        ylabel=relative Error,
        title=New Experiments (old in gray),
        legend entries={$e_1$,$e_2$,$e_3$}
    ]
    \addplot[black!15,forget plot]
        table {plotdata/oldexperiment1.dat};
    \addplot[black!15,forget plot]
        table {plotdata/oldexperiment2.dat};
    \addplot[black!15,forget plot]
        table {plotdata/oldexperiment3.dat};
    \addplot table {plotdata/newexperiment1.dat};
    \addplot table {plotdata/newexperiment2.dat};
    \addplot table {plotdata/newexperiment3.dat};
    \end{loglogaxis}
\end{tikzpicture}
```

Since forgotten plots won't increase the plot index, they will use the same `cycle list` entry as following plots.

The style `every forget plot` can be used to configure styles for each such plot:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[
        forget plot style={opacity=0.2},
        %  same as above:
        table/x=Basis,
        table/y={L2/r},
        xlabel=Degrees of Freedom,
        ylabel=relative Error,
        title=New Experiments (old in transparent),
        legend entries={$e_1$,$e_2$,$e_3$},
    ]
    \foreach \exp in {1,2,3} {
      \addplot+[forget plot]
          table {plotdata/oldexperiment\exp.dat};
      \addplot table {plotdata/newexperiment\exp.dat};
    }
    \end{loglogaxis}
\end{tikzpicture}
```

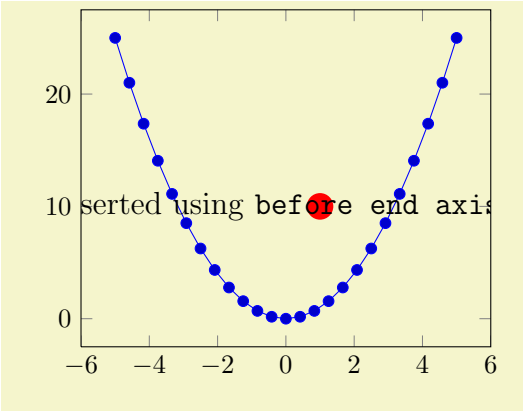
Here, the `\addplot+` command means we are using the same `cycle list` as the following plot and `forget plot style` modifies `every forget style` and yields transparency of the "old experiments".

Please note that `every plot no` ⟨*index*⟩ styles are not applicable here.

A forgotten plot will be stacked normally if `stack plots` is enabled!

`/pgfplots/before end axis/.code={`⟨`...`⟩`}`

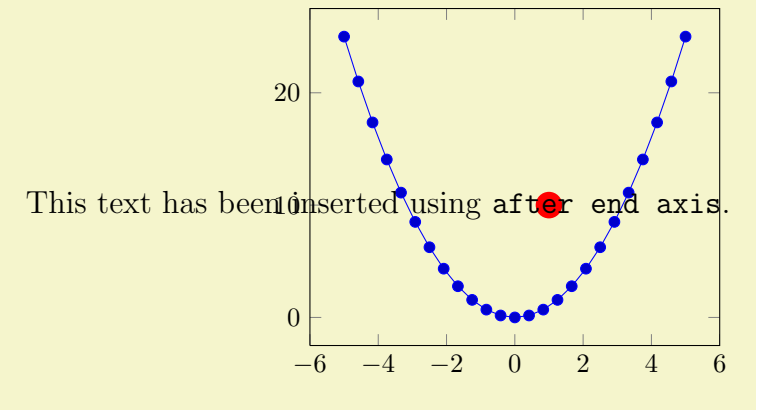
Allows to insert ⟨*commands*⟩ just before the axis is ended (see also `execute at end axis`). This option takes effect inside of the clipped area.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={
    before end axis/.code={
        \fill[red] (axis cs:1,10) circle(5pt);
        \node at (axis cs:-4,10)
            {\large This text has been inserted
             using \texttt{before end axis}.};
    }}}
\begin{tikzpicture}
    \begin{axis}
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

`/pgfplots/after end axis/.code={`⟨`...`⟩`}`

Allows to insert ⟨*commands*⟩ right after the end of the clipped drawing commands. While `before end axis` has the same effect as if ⟨*commands*⟩ had been placed inside of your axis, `after end axis` allows to access axis coordinates without being clipped.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\pgfplotsset{every axis/.append style={
    after end axis/.code={
        \fill[red] (axis cs:1,10) circle(5pt);
        \node at (axis cs:-4,10)
            {\large This text has been inserted using \texttt{after end axis}.};
    }}}
\begin{tikzpicture}
    \begin{axis}
    \addplot {x^2};
    \end{axis}
\end{tikzpicture}
```

/pgfplots/clip marker paths=true|false (initially `false`)

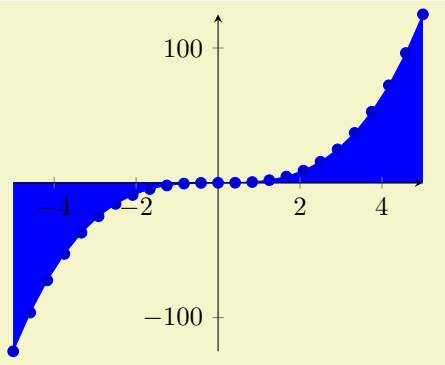
The initial choice `clip marker paths=false` causes markers to be drawn *after* the clipped region. Only their positions will be clipped. As a consequence, markers will be drawn completely, or not at all. The value `clip marker paths=true` is here for backwards compatibility: it does not introduce special marker treatment, so markers may be drawn partially if they are close to the clipping boundary[58].

/pgfplots/clip=true|false (initially `true`)

Whether any paths inside an axis shall be clipped.

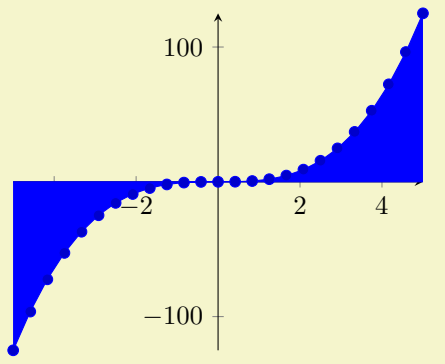
/pgfplots/axis on top=true|false (initially `false`)

If set to `true`, axis lines, ticks, tick labels and grid lines will be drawn on top of plot graphics.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        axis on top=true,
        axis x line=middle,
        axis y line=middle]
    \addplot+[fill] {x^3} \closedcycle;
    \end{axis}
\end{tikzpicture}
```

[58] Please note that clipped marker paths may be slightly faster during TEX compilation.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[
        axis on top=false,
        axis x line=middle,
        axis y line=middle]
    \addplot+[fill] {x^3} \closedcycle;
    \end{axis}
\end{tikzpicture}
```

Please note that this feature does not affect plot marks. I think it looks unfamiliar if plot marks are crossed by axis descriptions.

/pgfplots/visualization depends on=⟨*macro*⟩ (initially `empty`)
/pgfplots/visualization depends on=⟨*expression*⟩\as⟨*macro*⟩ (initially `empty`)
/pgfplots/visualization depends on=value ⟨*content*⟩\as⟨*macro*⟩ (initially `empty`)

Allows to communicate data to PGFPLOTS which is essential to perform the visualization although PGFPLOTS isn't aware of it.

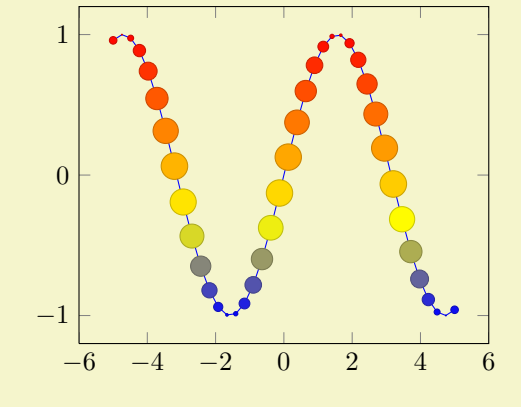
Suppose you want a scatter plot, which depends on the (x, y) coordinates, the `point meta` data to draw individual colors and furthermore data which influences the `mark size`. Thus, you need a total of 4 coordinates for every data point, although PGFPLOTS supports only 3 in its initial configuration.

Before we actually come to the main point of the problem, we'll talk about how to get a scatter plot which has individual colors *and* individual sizes. It is not sufficient to set `mark size` alone, since `mark size` is evaluated only once, before markers are processed (the same holds for `every mark`). Thus, we can use `scatter` combined with

`scatter/@pre marker code/.append style={/tikz/mark size=\perpointmarksize}`.

The `@pre marker code` is installed for every marker of a scatter plot individually. Now, we come to the problem as such: where can we get the value for `mark size`, in our case called `\perpointmarksize`?

A solution is `visualization depends on` (using the second input syntax at this point):




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}
    \addplot+[
        scatter,
        scatter src=y,
        samples=40,
        visualization depends on=
            {5*cos(deg(x)) \as \perpointmarksize},
        scatter/@pre marker code/.append style=
            {/tikz/mark size=\perpointmarksize}
    ]
        {sin(deg(x))};
\end{axis}
\end{tikzpicture}
```

Here, we define `\perpointmarksize` as `5*cos(deg(x))`. The expression will be evaluated together with all other coordinates. Thus, everything which is available during the survey phase can be used here. This includes the final coordinates `x`, `y`, `z`; the constant `meta` expands to the current per point meta data. Furthermore, `\thisrow{`⟨*colname*⟩`}` expands to the value of a table column.

The command `visualization depends on` evaluates and remembers every value in internal data structures. The remembered value is then available as ⟨*macro*⟩ during the visualization phase. In our example, the `@pre marker code` is evaluated during the visualization phase and applies `mark size=5*cos(deg(x))`.

The first syntax, `visualization depends on=`⟨*macro*⟩, tells PGFPLOTS to use an already defined ⟨*macro*⟩. The second syntax with ⟨*content*⟩`\as`⟨*macro*⟩ provides also the value.

There can be more than one `visualization depends on` phrase.

In case the stored value is not of numerical type[59], you can use the prefix '`value`' before the argument, i.e.

`visualization depends on=value` ⟨*macro*⟩ or

`visualization depends on=value` ⟨*content*⟩`\as` ⟨*macro*⟩.

Such a value will be expanded and stored, but not parsed as number (at least not by PGFPLOTS).

`/pgf/fpu={`⟨*true,false*⟩`}` (initially `true`)

This key activates or deactivates the floating point unit. If it is disabled (`false`), the core PGF math engine written by Mark Wibrow and Till Tantau will be used for `plot expression`. However, this engine has been written to produce graphics and is not suitable for scientific computing. It is limited to fixed point numbers in the range ±16384.00000.

If the `fpu` is enabled (`true`, the initial configuration) the high-precision floating point library of PGF written by Christian Feuersänger will be used. It offers the full range of IEEE double precision computing in TEX. This FPU is also part of PGFPLOTSTABLE, and it is activated by default for `create col/expr` and all other predefined mathematical methods.

Use

```
\pgfkeys{/pgf/fpu=false}
```

in order to de-activate the extended precision. If you prefer using the `fp` (fixed point) package, possibly combined with Mark Wibrows corresponding PGF library, the `fpu` will be deactivated automatically. Please note, however, that `fp` has a smaller data range (about $\pm 10^{17}$) and may be slower.

4.25 TikZ Interoperability

PGFPLOTS is built on top of TikZ/PGF, and it inherits most of power to visualize plots. However, their coordinate systems do not match up – for good reason: PGFPLOTS operates on logical (data) coordinates whereas TikZ operates on image coordinates.

Occasionally, one may want to synchronize both in order to generate a graphic – and the question arises how to match the coordinates from TikZ to PGFPLOTS and vice–versa. This section explains how to match coordinates and it discusses the necessary restrictions.

There are a couple of keys in PGFPLOTS which control the mapping of coordinates. The purpose of these keys is to implement visualization techniques, but they do things different than TikZ (and they should). To match coordinates with TikZ, one needs the following aspects:

1. Restrict your visualization type: a logarithmic axis simply may not fit into TikZ (to be more precise: it may fit, but a TikZ unit will correspond to a log–unit in PGFPLOTS).

2. Configure matching unit vectors by means of the `x` and `y` keys. The default configuration of TikZ is to use `x=1cm,y=1cm,z={(0,0)}`. Note that these settings are usually overridden by PGFPLOTS in order to respect `width` and `height` (and `view` for three–dimensional axes).

3. Disable the data scaling by means of `disabledatascaling`: PGFPLOTS will internally apply linear coordinate transformations in order to provide the data range required for floating point arithmetics (using approximately floating point precision). Disabling the data scaling means to restrict yourself to the (small) data range supported by TikZ—but that's probably what you want in that case.

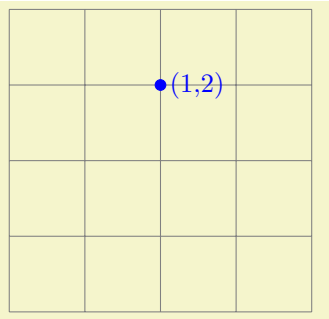
4. Define `anchor` and position of the `axis`, probably using `anchor=origin,at={(0,0)}`. The `at={(0,0)}` configures PGFPLOTS to place the axis at the TikZ position `(0,0)` whereas `anchor=origin` means that PGFPLOTS will place its data origin $(0,0,0)$ at the place designated by `at` (see Section 4.18 for details).

5. Make sure that the PGFPLOTS axis contains the data origin $(0,0,0)$ in the displayed data range (i.e. configure `xmin`, `xmax`, `ymin`, and `ymax` appropriately).

   Without this, the `anchor=origin` key required in the previous item will be truncated to the next coordinate which is part of the displayed range.

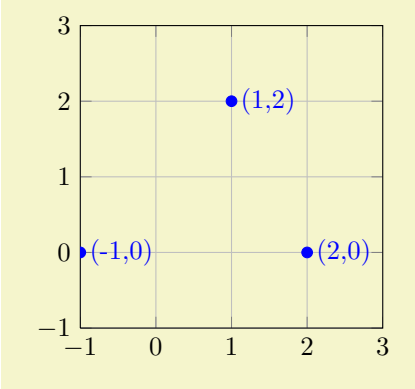
[59] Or if it is just a constant and you'd like to improve speed.

Here is a simple example, first with Ti*k*Z:




```
\begin{tikzpicture}
\coordinate (Point) at (1,2);

\draw [gray] (-1,-1) grid (3,3);
\draw [blue,fill] (Point) circle (2pt)
    node [right] {(1,2)};
\end{tikzpicture}
```

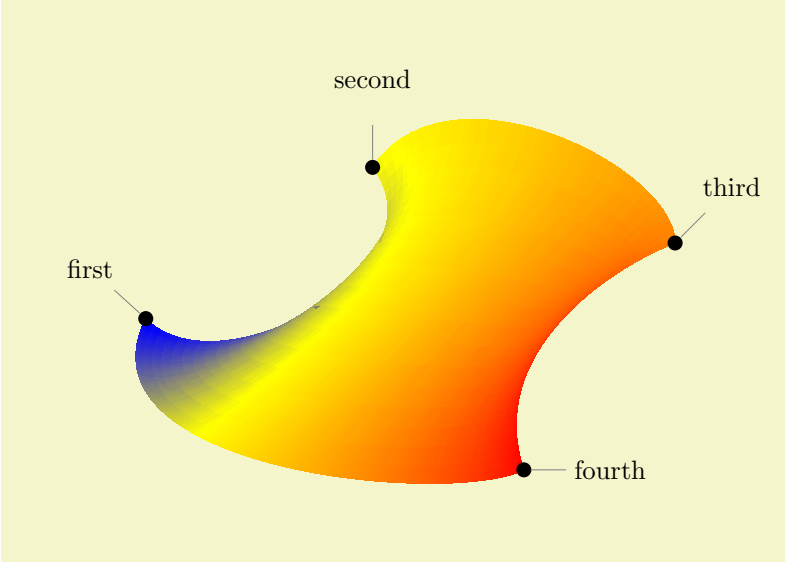
it displays a grid with $x, y \in [-1, 3]$ and shows a node inside of it. Now, we apply the keys discussed above to match this setting in PGFPLOTS:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \coordinate (Point) at (1,2);

\begin{axis}[
    %  tell pgfplots to "grab" the axis at its
    %  internal (0,0) coord:
    anchor=origin,
    %  tell pgfplots to place its anchor at (0,0):
    %  (This is actually the default and can
    %  be omitted)
    at={(0pt,0pt)},
    %  tell pgfplots to use the "natural" dimensions:
    disabledatascaling,
    %  tell pgfplots to use the same unit vectors
    %  as tikz:
    x=1cm,y=1cm,
    %
    %  this is just as usual in pgfplots. I guess
    %  it is only useful if (0,0) is part of the
    %  range... try it out.
    xmin=-1,xmax=3, ymin=-1,ymax=3,grid=both]
%  this uses the point defined OUTSIDE of the axis
\draw [blue,fill] (Point) circle (2pt)
    node [right] {(1,2)};

%  this uses a TIKZ coordinate (2,0) in the axis:
\draw [blue,fill] (2,0) circle (2pt)
    node [right] {(2,0)};

%  this here will always work inside of an axis:
\draw [blue,fill] (axis cs:-1,0) circle (2pt)
    node [right] {(-1,0)};
\end{axis}
\end{tikzpicture}
```

The example demonstrates several things: first, it defines a coordinate in the enclosing `tikzpicture` and uses it inside of the `axis` (at the correct position). Second, it uses the standard Ti*k*Z coordinate `(2,0)` inside of the `axis`, and it is placed at the expected position. Third, it uses the approach provided by PGFPLOTS by using the `axis cs` to designate a coordinate (this last approach does also work without the coordinate matching).

Here is an example which inserts a PGFPLOTS graphics correctly into a `tikzpicture`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  requires \usepgfplotslibrary{patchplots}
\begin{tikzpicture}
\begin{axis}[
    %  tell pgfplots to "grab" the axis at its internal (0,0) coord:
    anchor=origin,
    %  tell pgfplots to place its anchor at (0,0):
    %  (This is actually the default and can be omitted)
    at={(0pt,0pt)},
    %  tell pgfplots to use the "natural" dimensions:
    disabledatascaling,
    %  tell pgfplots to use the same unit vectors as tikz:
    x=1cm,y=1cm,
    %
    hide axis,
]
\addplot[patch,patch type=coons,
    shader=interp,point meta=explicit]
coordinates {
    (0,0)   [0] %  first corner
    (1,-1)  [0] %  bezier control point between (0) and (3)
    (4,0.7) [0] %  bezier control point between (0) and (3)
    %
    (3,2)   [1] %  second corner
    (4,3.5) [1] %  bezier control point between (3) and (6)
    (7,2)   [1] %  bezier control point between (3) and (6)
    %
    (7,1)      [2] %  third corner
    (6,0.6)    [2] %  bezier control point between (6) and (9)
    (4.5,-0.5) [2] %  bezier control point between (6) and (9)
    %
    (5,-2)   [3] %  fourth corner
    (4,-2.5) [3] %  bezier control point between (9) and (0)
    (-1,-2)  [3] %  bezier control point between (9) and (0)
};
\end{axis}

%  this requires pgf 2.10
\begin{scope}[every node/.style={circle,inner sep=2pt,fill=black}]
\node[pin=140:first] at (0,0) {};
\node[pin=second]    at (3,2) {};
\node[pin=45:third]  at (7,1) {};
\node[pin=0:fourth]  at (5,-2) {};
\end{scope}
\end{tikzpicture}
```

The example employs one of the `patch` plots of the `patchplots` library. Since these graphical elements typically require depth information (`z buffer`ing) and color data (`point meta`), they are only available inside of PGFPLOTS. However, the configuration above ensures that coordinates match one-to-one between PGFPLOTS and Ti*k*Z. The `hide axis` flag disables anything of PGFPLOTS, so only the visualized `patch` plot

remains[60].

[60]Note that the $(0, 0, 0)$ coordinate of PGFPLOTS is part of the data range here.

5 Related Libraries

This section describes some libraries which come with PGFPLOTS, but they are more or less special and need to be activated separately.

5.1 Clickable Plots

```
\usepgfplotslibrary{clickable} % LaTeX and plain TeX
\usepgfplotslibrary[clickable] % ConTeXt
\usetikzlibrary{pgfplots.clickable} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.clickable] % ConTeXt
```

A library which generates small popups whenever one clicks into a plot. The popup displays the coordinate under the mouse pointer, supporting the optional snap–to–nearest `clickable coords` feature with customizable displayed information. Furthermore, the library allows to display slopes if one holds the mouse pressed and drags it to another point in the plot.

The library has two purposes: to compute slopes in a simple way[61] and to provide related, optional information to single data points which are not important enough to be listed in the main text (like prototype parameters or other technical things).

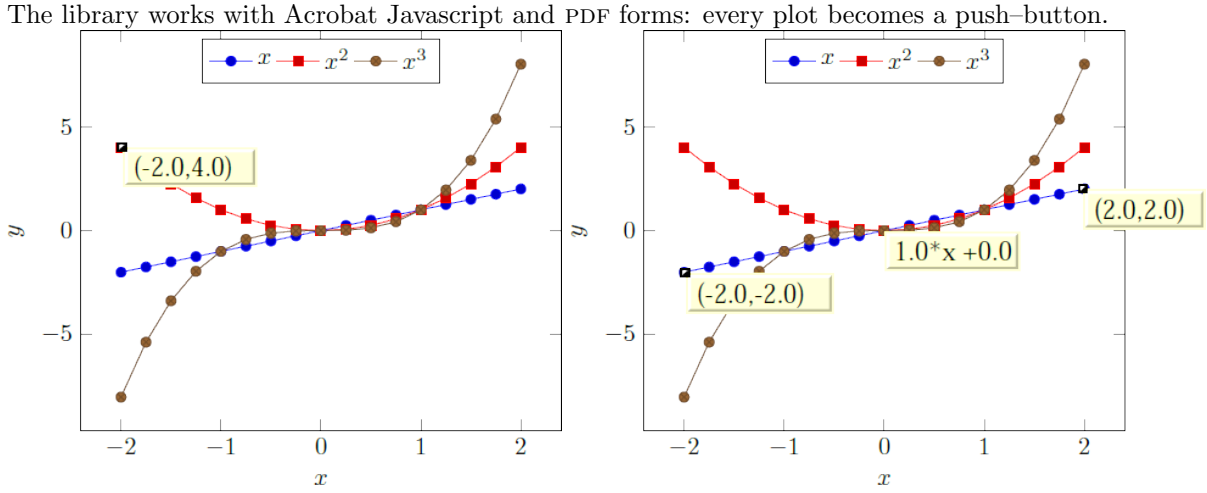
5.1.1 Overview

It is completely sufficient to write

```
\usepgfplotslibrary{clickable}
```

in the document preamble. This will automatically prepare every plot.

The library works with Acrobat Javascript and PDF forms: every plot becomes a push–button.

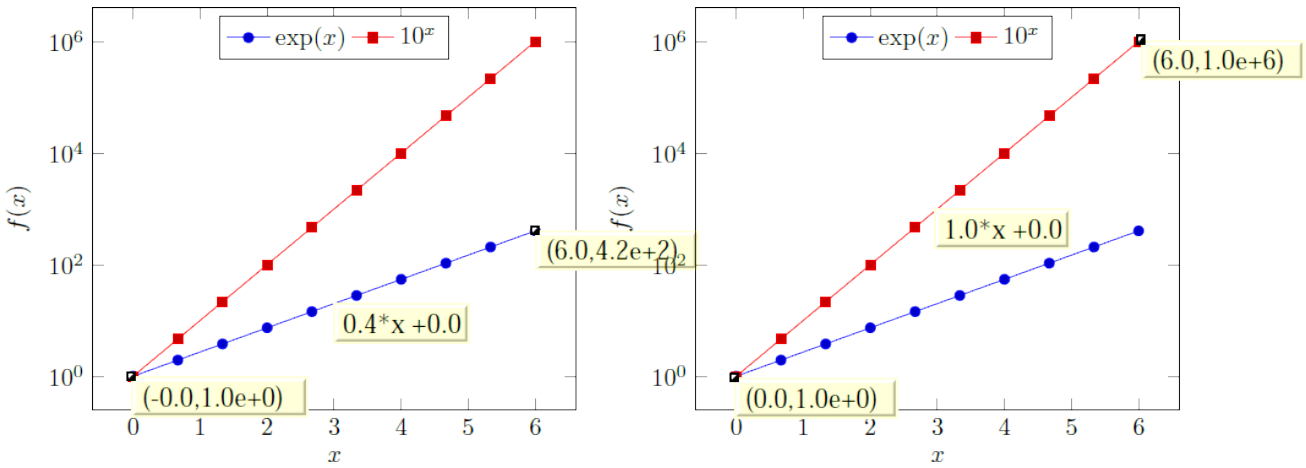


These screenshots show the result of clicking into the axis range (left column) and of dragging from one point to another (right column). The second case shows the result of Drag-and-Drop: it displays start- and end points and the equation for the line segment between between the first point of the drag- and drop and the second point where the mouse has been released. The line segment is

$$l(x; x_0, y_0, x_1, y_1) = m \cdot x + n$$

where $m = (y_1 - y_0)/(x_1 - x_0)$ is the slope and n the offset chosen such that $l(x_0; \dots) = y_0$. For logarithmic plots, logarithms will be applied before computing slopes.

[61] The author is applied mathematician...



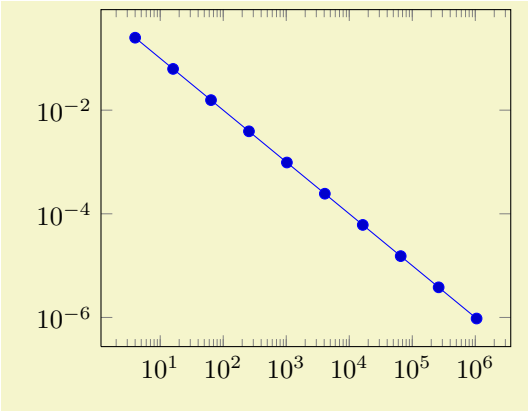
These screen shots show the result of drag- and drop for *logarithmic* axes: the end points show, again, the coordinates (without logs) and the form field in the middle shows the slope and offset of the linear equation in log coordinates.

The log basis for any logarithmic axes is usually 10, but it respects the current setting of `log basis x` and `log basis y`. The applied log will always use the same logarithm which is also used for the axis descriptions (this is not necessarily the same as used by PgfplotsTable!).

This document has been produced with the `clickable` library, so it is possible to load it into Acrobat Reader and simply click into a plot.

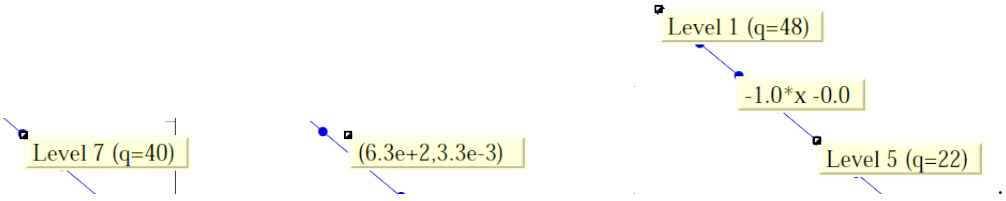
`/pgfplots/clickable coords={`⟨*displayed text*⟩`}`

Activates a snap–to–nearest feature when clicking onto plot coordinates. The ⟨*displayed text*⟩ is the coordinate's x and y value by default (i.e. you write just `clickable coords` without an equal sign).



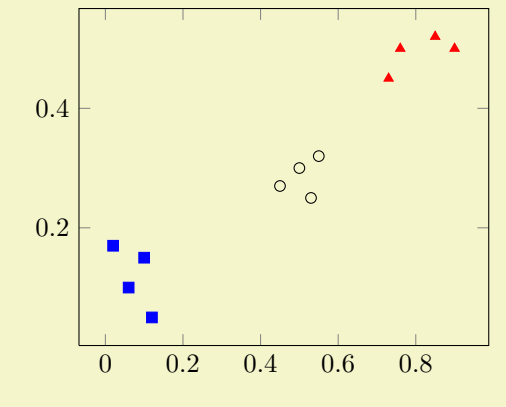
```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{loglogaxis}[clickable coords=
        {Level \thisrow{level} (q=\thisrow{q})}]
    \addplot table[x=dof,y=error] {
level  dof     error           q
1      4       2.50000000e-01  48
2      16      6.25000000e-02  25
3      64      1.56250000e-02  41
4      256     3.90625000e-03  8
5      1024    9.76562500e-04  22
6      4096    2.44140625e-04  46
7      16384   6.10351562e-05  40
8      65536   1.52587891e-05  3
9      262144  3.81469727e-06  1
10     1048576 9.53674316e-07  9
    };

    \end{loglogaxis}
\end{tikzpicture}
```

Now, clicking onto a data point yields 'Level 7 (q=40)' whereas clicking besides a data point results in the click coordinates as before,



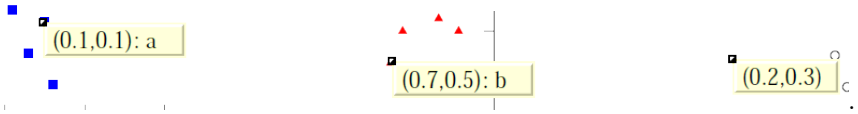
Note that logarithmic slopes work as before.

If you want the (x, y) values to be displayed, use the special placeholder string '`(xy)`' inside of ⟨*displayed text*⟩. As an example, we consider again the `scatter/classes` example of page 77:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[%
    clickable coords={(xy): \thisrow{label}},%
    scatter/classes={%
        a={mark=square*,blue},%
        b={mark=triangle*,red},%
        c={mark=o,draw=black}}]
    \addplot[scatter,only marks,%
        scatter src=explicit symbolic]%
    table[meta=label] {
x      y      label
0.1    0.15   a
0.45   0.27   c
0.02   0.17   a
0.06   0.1    a
0.9    0.5    b
0.5    0.3    c
0.85   0.52   b
0.12   0.05   a
0.73   0.45   b
0.53   0.25   c
0.76   0.5    b
0.55   0.32   c
    };
    \end{axis}
\end{tikzpicture}
```

Here, we find popups like



The ⟨*displayed text*⟩ is a richtext string displayed with *Javascript*. For most purposes, it is used like an unformatted C string: it contains characters, perhaps line breaks with '`\n`' or tabulators with '`\t`', but it should not contain TEX formatting instructions, especially no math mode (the '`(xy)`' replacement text is formatted with `sprintf`, see below). Consider `clickable coords code` in case you'd like to preprocess data before displaying it. If you experience problems with special characters, try prepending a backslash to them. If that doesn't work either, try to prefix the word with '`\\`' and/or with '`\string`'. Consider using `clickable coords size` if you intend to work with multiline fields and the size allocation needs improvements.

In fact, ⟨*displayed text*⟩ can even contain richtext (=XHTML) formatting instructions like '`<br/>`' (note the final slash) or '`<span style="color:\#7E0000;">text</span>`' (note the backslash before '`#`') which changes the color for `text`. The `<span style="">` arguments are CSS fields, consider an HTML reference for a list of CSS attributes.

It is possible to use `clickable coords` together with three dimensional axes. Note that dynamic (clickable) features of a three dimensional axis without `clickable coords` will be disabled (they appear to be useless). Furthermore, three dimensional axes do not support slope calculations; only the snap–to–nearest feature is available.

Consider using `annot/snap dist=6` to increase the snap–to–nearest distance.

The `clickable coords` can be specified for all plots in an axis (as in the examples above), but also once for every single `\addplot` commands for which the snap–to–nearest feature is desired (with different ⟨*displayed text*⟩).

If multiple `clickable coords` are on the same position, each click chooses the next one (in the order of appearance).

`/pgfplots/clickable coords code={`⟨*TEX code which defines* `\pgfplotsretval`⟩`}`

A variant of `clickable coords` which allows to prepare the displayed information before it is handed over to Javascript.

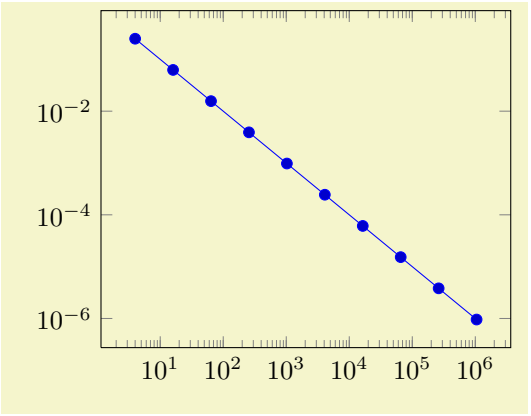
The value should be TEX code which defines `\pgfplotsretval` somehow. The result is used as simple, unformatted string which is associated to coordinates.

Consider using

`\pgfmathprintnumberto[verbatim]{`⟨*number*⟩`}\macroname`

`\edef\pgfplotsretval{Number=\macroname}`

to provide number printing. The `\pgfmathprintnumberto[verbatim]` doesn't use math mode to format a number[62], and it writes its result into `\macroname`. The name '`\macroname`' is arbitrary, use anything like '`\eps`' or '`\info`'. The `\edef` means "expanded definition" and has the effect of expanding all macros to determine the value, in our case "Number= ⟨*the value*⟩". The following example uses it twice to pretty–print the data:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{loglogaxis}[clickable coords code={%
    \pgfmathprintnumberto[verbatim,precision=1]%
        {\thisrow{error}}%
        \error%
    \pgfmathprintnumberto[verbatim,frac]%
        {\thisrow{frac}}%
        \fraccomp%
    \edef\pgfplotsretval{error \error, R=\fraccomp}%
}]%
\addplot table[x=dof,y=error] {
level  dof     error           frac
1      4       2.50000000e-01  0.5
2      16      6.25000000e-02  0.75
3      64      1.56250000e-02  0.1
4      256     3.90625000e-03  0.2
5      1024    9.76562500e-04  0.5
6      4096    2.44140625e-04  0.8
7      16384   6.10351562e-05  0.125
8      65536   1.52587891e-05  0.725
9      262144  3.81469727e-06  0.625
10     1048576 9.53674316e-07  1
};

\end{loglogaxis}
\end{tikzpicture}
```

resulting in

The ⟨*TEX code*⟩ is evaluated inside of a local scope, all locally declared variables are freed afterwards (that's why you can use any names you want).

`/pgfplots/clickable coords size=auto` or `{`⟨*max chars*⟩`}` or `{`⟨*max chars x,max chars y*⟩`}` (initially `auto`)

This is actually just another name for `annot/popup size snap`, see its documentation below.

5.1.2 Requirements for the Library

- The library relies on the LaTeX packages `insdljs` ("Insert document level Javascript") and `eforms` which are both part of the freely available `AcroTeX` education bundle [4][63]. The `insdljs` package creates a temporary file with extension `.djs`.
- At the time of this writing, only Adobe Acrobat Reader interpretes Javascript and Forms properly. The library doesn't have any effect if the resulting document is used in other viewers (as far as I know).

Note that although this library has been written for PGFPLOTS, it can be used independently of a PGFPLOTS environment.

Compatibility issues: There a several restrictions when using this library. Most of them will vanish in future versions – but up to now, I can't do magic.

- The library does not yet support rotated axes. Use `clickable=false` for those axes.

[62] See the PgfplotsTable manual for details about number printing.
[63] These packages rely on LaTeX, so the library is only available for LaTeX, not for plain TeX or ConTeXt.

- The library works only with `pdflatex`; `dvips` or `dvipdfm` are not supported[64].
- Up to now, it is *not* possible to use this library together with the `external` library and other image externalization methods of Section 7.

  To be more precise, you can (with two extra preamble lines, see below) get correctly annotated, exported PDF documents, but the `\includegraphics` command does not import the dynamic features.

  In case you decide to use this work–around, you need to insert

```
%  \maxdeadcycles=10000 % in case you get the error 'Output loop---<N> consecutive dead cycles.'
\usepackage[pdftex]{eforms}
```

  before loading PGF, Ti*k*Z or PGFPLOTS. The `\maxdeadcycles` appears to be necessary for large documents, try it out.

  As long as you are working on a draft version of your document, you might want to use

```
\pgfkeys{/pgf/images/include external/.code={\href{file:#1}{\pgfimage{#1}}}}
```

  in your preamble. This will generate hyperlinks around the graphics files which link to the exported figures. Clicking on the hyperlinks opens the exported figure which, in turn, has been generated with the `clickable` library and allows dynamic features[65].
- The library automatically calls `\begin{Form}` at `\begin{document}` and `\end{Form}` at the end of the document. This environment of `hyperref` is necessary for dynamic user interaction and should be kept in mind if the document contains other form elements.

Acknowledgements:

- I have used a Javascript `sprintf` implementation of Kevin van Zonneveld [6] (the Javascript API has only a limited set of conversions).

5.1.3 Customization

It is possible to customize the library with several options.

`/pgfplots/clickable=true|false` (initially `true`)

Allows to disable the library for single plots.

`/pgfplots/annot/js fillColor={`⟨*Javascript color*⟩`}` (initially `["RGB",1,1,.855]`)

Sets the background (fill) color of the short popup annotations.

Possible choices are `transparent`, gray, RGB or CMYK color specified as four–element–arrays of the form `["RGB",` ⟨*red*⟩`,`⟨*green*⟩`,`⟨*blue*⟩`]`. Each color component is between 0 and 1.

Again: this option is for Javascript. It is *not* possible to use colors as in PGF.

`/pgfplots/annot/point format={`⟨*sprintf-format*⟩`}` (initially `(%.1f,%.1f)`)
`/pgfplots/annot/point format 3d={`⟨*sprintf-format*⟩`}` (initially `(%.1f,%.1f,%.1f)`)

Allows to provide an `sprintf` format string which is used to fill the annotations with text. The first argument to `sprintf` is the x-coordinate and the second argument is the y-coordinate.

The `point format 3d` variant is used for any three–dimensional axis whereas the `point format` is used (only) for two–dimensional ones.

The `every semilogx axis`, `every semilogy axis` and `every loglog axis` styles have been updated to

```
\pgfplotsset{
    every semilogy axis/.append style={/pgfplots/annot/point format={(\% .1f,\%.1e)}},
    every semilogx axis/.append style={/pgfplots/annot/point format={(\% .1e,\%.1f)}},
    every loglog axis/.append style={/pgfplots/annot/point format={(\% .1e,\%.1e)}}
}
```

such that every logarithmic coordinate is displayed in scientific format.

[64] In fact, they should be. I don't really know why they don't ... any hint is welcome.

[65] This special treatment needs the external files in the same base directory as the main document, so this approach is most certainly *not* suitable for a final document.

/pgfplots/annot/slope format={⟨*sprintf-format*⟩} (initially `%.1f*x %+.1f`)

Allows to provide an `sprintf` format string which is used to fill the slope–annotation with text. The first argument is the slope and the second the line offset.

/pgfplots/annot/printable=true|false (initially `false`)

Allows to configure whether the small annotations will be printed. Otherwise, they are only available on screen.

/pgfplots/annot/font={⟨*Javascript font name*⟩} (initially `font.Times`)

Allows to choose a Javascript font for the annotations. Possible choices are limited to what Javascript accepts (which is *not* the same as LaTeX). The default fonts and its names are shown below.

Font Name	Name in Javascript
Times-Roman	font.Times
Times-Bold	font.TimesB
Times-Italic	font.TimesI
Times-BoldItalic	font.TimesBI
Helvetica	font.Helv
Helvetica-Bold	font.HelvB
Helvetica-Oblique	font.HelvI
Helvetica-BoldOblique	font.HelvBI
Courier	font.Cour
Courier-Bold	font.CourB
Courier-Oblique	font.CourI
Courier-BoldOblique	font.CourBI
Symbol	font.Symbol
ZapfDingbats	font.ZapfD

/pgfplots/annot/textSize={⟨*Size in Point*⟩} (initially `11`)

Sets the text size of annotations in points.

/pgfplots/annot/popup size generic=auto or {⟨x⟩} or {⟨x,y⟩} (initially `auto`)
/pgfplots/annot/popup size snap=auto or {⟨x⟩} or {⟨x,y⟩} (initially `auto`)
/pgfplots/annot/popup size={⟨*value*⟩}

The first key defines the size of popups if you just click into an axis. The second key defines the size of popups for the snap–to–nearest feature (i.e. those prepared by `clickable coords`). The third key sets both to the same ⟨*value*⟩.

The argument can be `auto` in which case PGFPLOTS tries to be smart and counts characters. This may fail for multiline texts. The choice ⟨x⟩ provides the *horizontal* size only, in units of `annot/textSize`. Thus, `annot/popup size generic=6` makes the popup $6 \cdot 11$ points wide. In this case, only one line will be allocated. Finally, ⟨x,y⟩ allows to provide horizontal and vertical size, both in units of `annot/textSize`.

See also `clickable coords size` which is an alias for `annot/popup size snap`.

/pgfplots/annot/snap dist={⟨*Size in Point*⟩} (initially `4`)

Defines the size within two mouse clicks are considered to be equivalent, meased in points (Euclidean distance).

/pgfplots/annot/richtext=true|false (initially `true`)

Enables or disables richtext formatting in `clickable coords` arguments. Richtext is kind of XHTML and allows CSS styles like colors, font changes and other CSS attributes, see the documentation for `clickable coords` for details.

The case `annot/richtext=false` is probably more robust.

5.1.4 Using the Clickable Library in Other Contexts

This library provides essentially one command, `\pgfplotsclickablecreate` which creates a clickable area of predefined size, combined with Javascript interaction code. It can be used independently of PGFPLOTS.

`\pgfplotsclickablecreate[`⟨*required key-value-options*⟩`]`

Creates an area which is clickable. A click produces a popup which contains information about the point under the cursor.

The complete (!) context needs to be provided using key-value-pairs, either set before calling this method of inside of `[`⟨*required key-value-options*⟩`]`.

This command actually creates an AcroForm which invokes Javascript whenever it is clicked. A Javascript Object is created which represents the context (axis limits and options). This Javascript object is available at runtime.

This method is public and it is *not* restricted to PGFPLOTS. The PGFPLOTS hook simply initializes the required key-value-pairs.

This method does not draw anything. It initializes only a clickable area and Javascript code.

The required key-value-pairs are documented below.

Attention: Complete key-value validation is *not* performed here. It can happen that invalid options will produce Javascript bugs when opened with Acrobat Reader. Use the Javascript console to find them.

All options described in the following are only interesting for users who intend to use this library without PGFPLOTS.

`/pgfplots/annot/width=`{⟨*dimension*⟩} (initially -)

This required key communicates the area's width to `\pgfplotsclickablecreate`. It must be a TEX dimension like `5cm`.

`/pgfplots/annot/height=`{⟨*dimension*⟩} (initially -)

This required key communicates the area's height to `\pgfplotsclickablecreate`. It must be a TEX dimension like `5cm`.

`/pgfplots/annot/jsname=`{⟨*string*⟩} (initially -)

This required key communicates a unique identifier to `\pgfplotsclickablecreate`. This identifier is used to identify the object in Javascript, so there can't be more than one of them. If it is empty, a default identifier will be generated.

`/pgfplots/annot/xmin=`{⟨*number*⟩}
`/pgfplots/annot/xmax=`{⟨*number*⟩}
`/pgfplots/annot/ymin=`{⟨*number*⟩}
`/pgfplots/annot/ymax=`{⟨*number*⟩} (initially `empty`)

These required keys communicate the axis limits to `\pgfplotsclickablecreate`. They should be set to numbers which can be assigned to a Javascript floating point number (standard IEEE double precision).

`/pgfplots/annot/collected plots=`{⟨*nested arrays*⟩} (initially `empty`)

The low level interface to implement a snap–to–nearest feature. The value is an array of plots, where each plot is again an array of coordinates and each coordinate is an array of three elements, x, y and text. Please consult the code comments for details and examples.

5.2 Colormaps

An extension by Patrick Häcker

```
\usepgfplotslibrary{colormaps} % LaTeX and plain TeX
\usepgfplotslibrary[colormaps] % ConTeXt
\usetikzlibrary{pgfplots.colormaps} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.colormaps] % ConTeXt
```

A small library providing a number of additional colormaps. Many of these colormaps originate from the free Matlab package "SC — powerful image rendering" of Oliver Woodford.

The purpose of this library is to provide further colormaps to all users and to provide some of them which are similar to those used by Matlab (®).

/pgfplots/colormap/autumn (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={autumn}{rgb255=(255,0,0) rgb255=(255,255,0)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/bled (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={bled}{rgb255=(0,0,0) rgb255=(43,43,0) rgb255=(0,85,0)
        rgb255=(0,128,128) rgb255=(0,0,170) rgb255=(213,0,213) rgb255=(255,0,0)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/bright (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={bright}{rgb255=(0,0,0) rgb255=(78,3,100) rgb255=(2,74,255)
        rgb255=(255,21,181) rgb255=(255,113,26) rgb255=(147,213,114) rgb255=(230,255,0)
        rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/bone (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={bone}{[1cm]rgb255(0cm)=(0,0,0) rgb255(3cm)=(84,84,116)
        rgb255(6cm)=(167,199,199) rgb255(8cm)=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/cold (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={cold}{rgb255=(0,0,0) rgb255=(0,0,255) rgb255=(0,255,255)
        rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/copper (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={copper}{[1cm]rgb255(0cm)=(0,0,0) rgb255(4cm)=(255,159,101)
        rgb255(5cm)=(255,199,127)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/copper2 (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={copper2}{rgb255=(0,0,0) rgb255=(68,62,63) rgb255=(170,112,95)
        rgb255=(207,194,138) rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/earth (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={earth}{rgb255=(0,0,0) rgb255=(0,28,15) rgb255=(42,39,6)
        rgb255=(28,73,33) rgb255=(67,85,24) rgb255=(68,112,46) rgb255=(81,129,83)
        rgb255=(124,137,87) rgb255=(153,147,122) rgb255=(145,173,164) rgb255=(144,202,180)
        rgb255=(171,220,177) rgb255=(218,229,168) rgb255=(255,235,199) rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/gray (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={gray}{rgb255=(0,0,0) rgb255=(255,255,255)}
}
```

This colormap is an alias for the standard colormap/blackwhite.

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/hot2 (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={hot2}{[1cm]rgb255(0cm)=(0,0,0) rgb255(3cm)=(255,0,0)
        rgb255(6cm)=(255,255,0) rgb255(8cm)=(255,255,255)}
}
```

Note that this particular choice ships directly with PGFPLOTS, you do not need to load the colormaps library for this value.

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/hsv (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={hsv}{rgb255=(255,0,0) rgb255=(255,255,0) rgb255=(0,255,0)
        rgb255=(0,255,255) rgb255=(0,0,255) rgb255=(255,0,255) rgb255=(255,0,0)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/hsv2 (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={hsv2}{rgb255=(0,0,0) rgb255=(128,0,128) rgb255=(0,0,230)
        rgb255=(0,255,255) rgb255=(0,255,0) rgb255=(255,255,0) rgb255=(255,0,0)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/jet (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={jet}{rgb255(0cm)=(0,0,128) rgb255(1cm)=(0,0,255)
        rgb255(3cm)=(0,255,255) rgb255(5cm)=(255,255,0) rgb255(7cm)=(255,0,0)
        rgb255(8cm)=(128,0,0)}
}
```

Note that this particular choice ships directly with PGFPLOTS, you do not need to load the colormaps library for this value.

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/pastel (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={pastel}{rgb255=(0,0,0) rgb255=(120,0,5) rgb255=(0,91,172)
        rgb255=(215,35,217) rgb255=(120,172,78) rgb255=(255,176,24) rgb255=(230,255,0)
        rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/pink (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={pink}{rgb255=(0,0,0) rgb255=(12,16,46) rgb255=(62,22,43)
        rgb255=(53,53,65) rgb255=(79,72,58) rgb255=(122,80,67) rgb255=(147,91,102)
        rgb255=(147,115,140) rgb255=(144,145,154) rgb255=(173,163,146) rgb255=(216,171,149)
        rgb255=(250,179,179) rgb255=(255,198,227) rgb255=(246,229,255) rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/sepia (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={sepia}{rgb255(0cm)=(0,0,0) rgb255(1cm)=(26,13,0)
        rgb255(18cm)=(255,230,204) rgb255(20cm)=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/spring (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={spring}{rgb255=(255,0,255) rgb255=(255,255,0)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/summer (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={summer}{rgb255=(0,128,102) rgb255=(255,255,102)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/temp (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={temp}{rgb255=(36,0,217) rgb255=(25,29,247) rgb255=(41,87,255)
        rgb255=(61,135,255) rgb255=(87,176,255) rgb255=(117,211,255) rgb255=(153,235,255)
        rgb255=(189,249,255) rgb255=(235,255,255) rgb255=(255,255,235) rgb255=(255,242,189)
        rgb255=(255,214,153) rgb255=(255,172,117) rgb255=(255,120,87) rgb255=(255,61,61)
        rgb255=(247,40,54) rgb255=(217,22,48) rgb255=(166,0,33)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/thermal (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={thermal}{rgb255=(0,0,0) rgb255=(77,0,179) rgb255=(255,51,0)
        rgb255=(255,255,0) rgb255=(255,255,255)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

/pgfplots/colormap/winter (style, no value)

A style which is equivalent to

```
\pgfplotsset{
    /pgfplots/colormap={winter}{rgb255=(0,0,255) rgb255=(0,255,128)}
}
```

This colormap is similar to one shipped with Matlab (®) under a similar name.

5.3 Dates as Input Coordinates

```
\usepgfplotslibrary{dateplot} % LaTeX and plain TeX
\usepgfplotslibrary[dateplot] % ConTeXt
\usetikzlibrary{pgfplots.dateplot} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.dateplot] % ConTeXt
```

A library which allows to use dates like `2008-01-01` or dates with time like `2008-01-01 11:35` as input coordinates in plots. The library converts dates to numbers and tick labels will be pretty-printed dates (or times).

This library is documented in Section 4.20 on page 268.

5.4 Image Externalization

```
\usepgfplotslibrary{external} % LaTeX and plain TeX
\usepgfplotslibrary[external] % ConTeXt
\usetikzlibrary{pgfplots.external} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.external] % ConTeXt
```

The `external` library offers a convenient method to export every single `tikzpicture` into a separate `.pdf` (or `.eps`). Later runs of LaTeX will simply include these graphics, thereby reducing typesetting time considerably.

This library is documented in more detail in Section 7.1 “Export to PDF/EPS”.

The `external` library has been written by Christian Feuersänger (author of PGFPLOTS). It has been contributed to Ti*k*Z as general purpose library, so the reference documentation along with all tweaks can be found in [5, Section “Externalization Library”]. The command `\usepgfplotslibrary{external}` is actually just a wrapper which loads `\usetikzlibrary{external}` or, if this library does not yet exist because the installed PGF has at most version 2.00, it will load a copy which is shipped with PGFPLOTS.

5.5 Grouping plots

by Nick Papior Andersen

```
\usepgfplotslibrary{groupplots} % LaTeX and plain TeX
\usepgfplotslibrary[groupplots] % ConTeXt
\usetikzlibrary{pgfplots.groupplots} % LaTeX and plain TeX
```

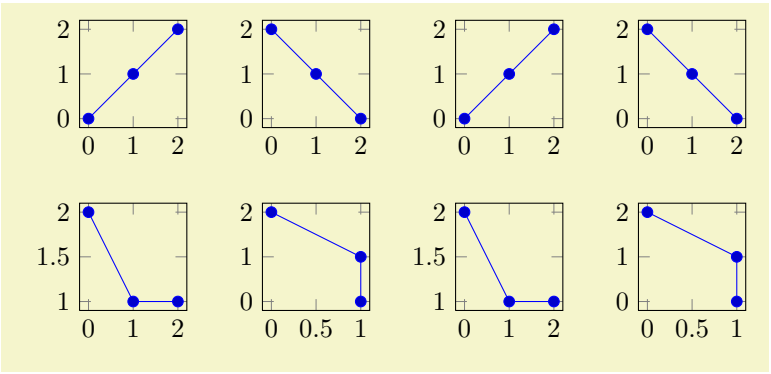
`\usetikzlibrary[pgfplots.groupplots] % ConTeXt`

A library which allows the user to typeset several plots in a matrix like structure. Often one has to compare two plots to one another, or you simply need to display two plots in conjunction with each other. Either way the following section describes this library which makes matrix structure easier than alternative methods discussed in Section 4.18.4.

`\begin{groupplot}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{groupplot}`

Once you have loaded the `groupplots` library you will gain access to this environment. This environment is limited to the same restrictions as the `axis` environment. It actually utilizes this environment so consider it as an extension of this. What is important to note is that [⟨*options*⟩] are applied to all plots in the entire environment. This can be really handy when you need the same `xmin`, `xmax`, `ymin` and `ymax`.

With such an environment one can typeset plots in matrix like styles



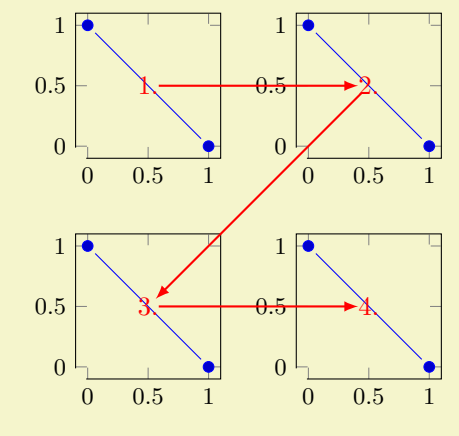

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
%  Example using groupplots library
\begin{tikzpicture}
  \begin{groupplot}[group style={group size=2 by 2},height=3cm,width=3cm]
    \nextgroupplot
    \addplot coordinates {(0,0) (1,1) (2,2)};
    \nextgroupplot
    \addplot coordinates {(0,2) (1,1) (2,0)};
    \nextgroupplot
    \addplot coordinates {(0,2) (1,1) (2,1)};
    \nextgroupplot
    \addplot coordinates {(0,2) (1,1) (1,0)};
  \end{groupplot}
\end{tikzpicture}
%  Same example created as done without the library
\begin{tikzpicture}
  \begin{axis}[name=plot1,height=3cm,width=3cm]
    \addplot coordinates {(0,0) (1,1) (2,2)};
  \end{axis}
  \begin{axis}[name=plot2,at={($(plot1.east)+(1cm,0)$)},anchor=west,height=3cm,width=3cm]
    \addplot coordinates {(0,2) (1,1) (2,0)};
  \end{axis}
  \begin{axis}[name=plot3,at={($(plot1.south)-(0,1cm)$)},anchor=north,height=3cm,width=3cm]
    \addplot coordinates {(0,2) (1,1) (2,1)};
  \end{axis}
  \begin{axis}[name=plot4,at={($(plot2.south)-(0,1cm)$)},anchor=north,height=3cm,width=3cm]
    \addplot coordinates {(0,2) (1,1) (1,0)};
  \end{axis}
\end{tikzpicture}
```

The equivalent code is seen as the second example and it is clear that you have to type a lot less. So how do you use it? First of all you need to utilize the new environment `groupplot`. Within this environment the following command works.

`\nextgroupplot[`⟨*axis options*⟩`]` ⟨*normal plot commands*⟩

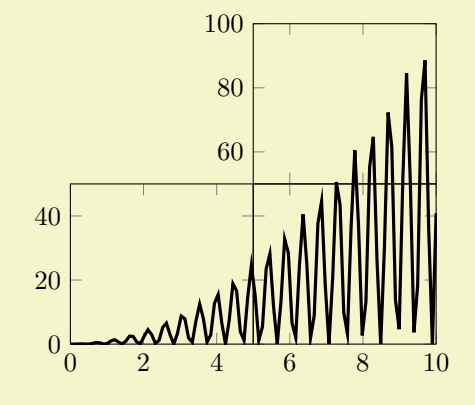
This command shifts the placement of the plot. Therefore one should always start the environment `groupplot` with the command `\nextgroupplot` in order to create the first plot. The [⟨*axis options*⟩]

are the options that are supplied to the following axes until the next `\nextgroupplot` command is seen by TEX. The order in which figures are typeset are as seen in the next example.



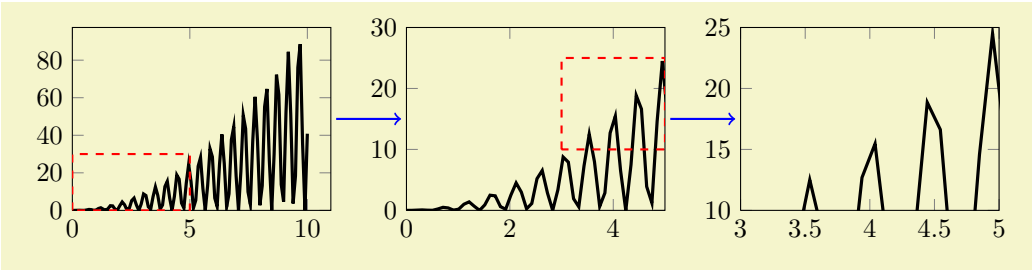

```
\begin{tikzpicture}[shorten >=4pt,shorten <=4pt]
  \begin{groupplot}[group style={group size=2 by 2},
    height=3.5cm,width=3.5cm,/tikz/font=\small]
    \nextgroupplot% 1
    \addplot coordinates {(0,1) (1,0)};
    \nextgroupplot% 2
    \addplot coordinates {(0,1) (1,0)};
    \nextgroupplot% 3
    \addplot coordinates {(0,1) (1,0)};
    \nextgroupplot% 4
    \addplot coordinates {(0,1) (1,0)};
  \end{groupplot}
  \draw[thick,>=latex,->,red]
    (group c1r1.center) node {1.}  --
    (group c2r1.center) node {2.};
  \draw[thick,>=latex,->,red]
    (group c2r1.center)  --
    (group c1r2.center) node {3.};
  \draw[thick,>=latex,->,red]
    (group c1r2.center)  --
    (group c2r2.center) node {4.};
\end{tikzpicture}
```

The plot first fills the first row, then the next row and so on. Just like a table, thus the names `group c⟨column⟩r⟨row⟩`. The power of the `groupplot` is to quickly create an aligned structure of plots. But you can also utilize it to structure data more creatively. Consider the next example.



```
\begin{tikzpicture}
 \begin{groupplot}[group style={group size=2 by 2,
      horizontal sep=0pt,vertical sep=0pt,
      xticklabels at=edge bottom},
      xmin=0,ymin=0,
      height=3.7cm,width=4cm,no markers]
  \nextgroupplot[group/empty plot]
  \nextgroupplot[xmin=5,xmax=10,ymin=50,ymax=100]
  \addplot[very thick] file {plotdata/group-1.dat};
  \nextgroupplot[xmax=5,ymax=50]
  \addplot[very thick] file {plotdata/group-1.dat};
  \nextgroupplot[xmin=5,xmax=10,ymax=50,
     yticklabels={}]
  \addplot[very thick] file {plotdata/group-1.dat};
 \end{groupplot}
\end{tikzpicture}
```

Or for instance zooming in on data as in the next example.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{groupplot}[group style={group size=3 by 1},xmin=0,ymin=0,height=4cm,width=5cm,no markers]
    \nextgroupplot
    \addplot[very thick] file {plotdata/group-1.dat};
    \draw[red,dashed,thick] (axis cs:0,0) rectangle (axis cs:5,30);
    \nextgroupplot[xmax=5,ymax=30]
    \addplot[very thick] file {plotdata/group-1.dat};
    \draw[red,dashed,thick] (axis cs:3,10) rectangle (axis cs:5,25);
    \nextgroupplot[xmin=3,xmax=5,ymin=10,ymax=25]
    \addplot[very thick] file {plotdata/group-1.dat};
  \end{groupplot}
  \draw[thick,blue,->,shorten >=2pt,shorten <=2pt]
        (group c1r1.east) -- (group c2r1.west);
  \draw[thick,blue,->,shorten >=2pt,shorten <=2pt]
        (group c2r1.east) -- (group c3r1.west);
\end{tikzpicture}
```

5.5.1 Grouping options

/pgfplots/group style={⟨*options with* group/ *prefix*⟩}

This key sets all ⟨*options*⟩ using the /pgfplots/group/ prefix.

Note that the distinction between group/ and normal options is important as some of them are quite similar.

For example, the following statements are all equivalent:

```
\pgfplotsset{group style={a=2,b=3}}
\pgfplotsset{group/a=2,group/b=3}
\pgfplotsset{group/.cd,a=2,b=3}
```

All the following keys are in the subdirectory group.

/pgfplots/group/group size=⟨*columns*⟩ by ⟨*rows*⟩ (initially 1 by 1)
/pgfplots/group/columns=⟨*columns*⟩ (initially 1)
/pgfplots/group/rows=⟨*rows*⟩ (initially 1)

These keys determine the total number of plots that can be in one environment groupplot. It is thus important not to add more \nextgroupplot in the environment than ⟨*columns*⟩×⟨*rows*⟩. This is critical to set if one uses more than 1 more plot. As the key group size uses columns and rows you should stick to either group size or both columns and rows.

/pgfplots/group/horizontal sep=⟨*dimension*⟩ (initially 1cm)
/pgfplots/group/vertical sep=⟨*dimension*⟩ (initially 1cm)

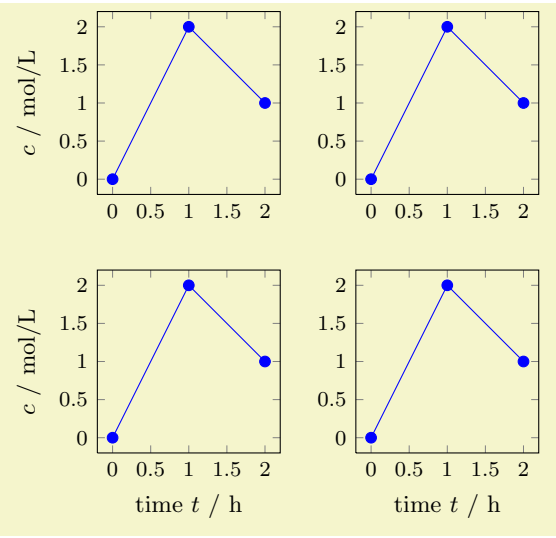
The spacing between the plots in the horizontal and vertical direction, respectively. If you thus want them to be *glued* together you should set them both to a length of 0pt.

/pgfplots/group/every plot/.style={⟨*style*⟩} (initially empty)

This style is used on every plot as the first style. It is thus equivalent as ⟨*options*⟩ in the groupplot environment.

/pgfplots/group/xlabels at=all|edge bottom|edge top (initially all)
/pgfplots/group/ylabels at=all|edge left|edge right (initially all)

In order to determine which plots get labels typeset one can use these keys. By default all axes get typeset normally and thus have both x and y axis labels.




```
\begin{tikzpicture}
    \begin{groupplot}[
        group style={
            group name=my plots,
            group size=2 by 2,
            xlabels at=edge bottom,
            ylabels at=edge left,
        },
        footnotesize,
        width=4cm,
        height=4cm,
        %
        xlabel=time $t$ / h,
        ylabel=$c$ / mol/L,
    ]
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \end{groupplot}
\end{tikzpicture}
```

In the example above, only the bottom row gets the label defined in the beginning `groupplot`-environment on the x axis and only the first column of plots gets labels on the y axis on their left side. These keys are especially handy when using *glued* plots.

`/pgfplots/group/xticklabels at=all|edge top|edge bottom` (initially `all`)
`/pgfplots/group/yticklabels at=all|edge left|edge right` (initially `all`)

In order to determine which plots get tick labels typeset one can use these keys. By default all axes gets typeset normally and thus have both x and y axis tick labels. If one sets

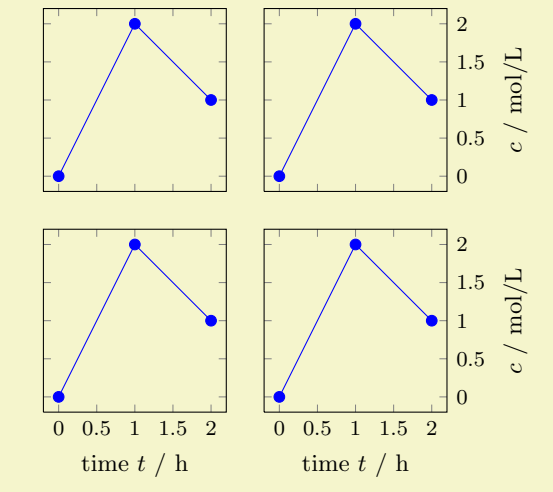
```
\pgfplotsset{group/xticklabels at=edge bottom,group/yticklabels at=edge right}
```

only the bottom row gets tick labels on the x axis and only the last column gets tick labels on the y axis on their right side. These keys are specially handy when using *glued* plots.

Keep in mind that this is implies the same ticks for all plots.

`/pgfplots/group/x descriptions at=all|edge top|edge bottom` (initially `all`)
`/pgfplots/group/y descriptions at=all|edge left|edge right` (initially `all`)

These are simply a short hand for using both `xticklabels at` and `xlabels at` simultaneously:




```
\begin{tikzpicture}
    \begin{groupplot}[
        group style={
            group name=my plots,
            group size=2 by 2,
            %
            x descriptions at=edge bottom,
            y descriptions at=edge right,
            horizontal sep=0.5cm,
            vertical sep=0.5cm,
        },
        footnotesize,
        width=4cm,
        height=4cm,
        %
        xlabel=time $t$ / h,
        ylabel=$c$ / mol/L,
    ]
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
        \addplot coordinates{(0,0) (1,2) (2,1)};
    \end{groupplot}
\end{tikzpicture}
```

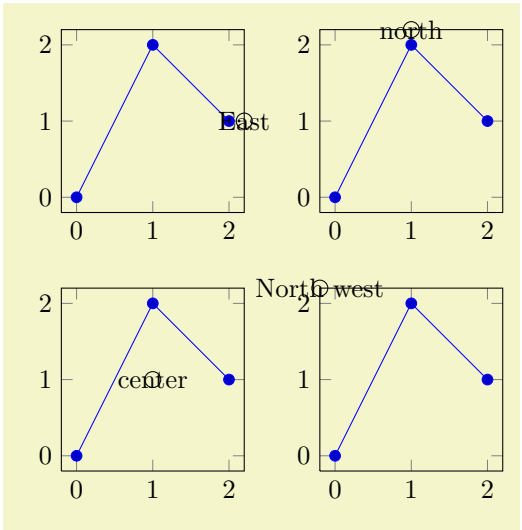
Here, `x descriptions at=edge bottom` yields that x descriptions (`xlabel` and `xticklabel`) are only used for the lowest row. Furthermore, `y descriptions at=edge right` places y descriptions only for the rightmost column. Consider modifying the `horizontal sep` and `vertical sep` for your needs.

As for `xticklabels at`, usage of this key implies the same ticks for all plots.

This might *require* `compat=1.3` (or newer).

`/pgfplots/group/group name={`⟨*name*⟩`}` (initially `group`)

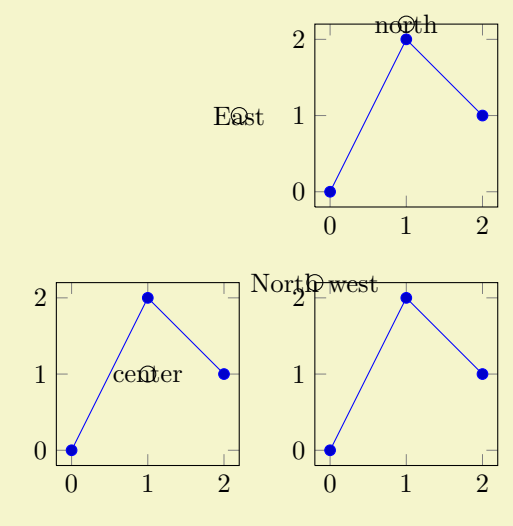
This sets what you can refer the plots to after typesetting. Thus you can use their anchors later. See the following example




```
\begin{tikzpicture}
  \begin{groupplot}[group style={
        group name=my plots,group size=2 by 2},
    width=4cm,height=4cm]
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
  \end{groupplot}
  \draw (my plots c1r1.east)
      circle (3pt) node {East};
  \draw (my plots c2r1.north)
      circle (3pt) node {north};
  \draw (my plots c1r2.center)
      circle (3pt) node {center};
  \draw (my plots c2r2.north west)
      circle (3pt) node {North west};
\end{tikzpicture}
```

`/pgfplots/group/empty plot/.style={`⟨*style*⟩`}` (initially `/pgfplots/hide axis`)

This key can be used as an option to the command `\nextgroupplot`. This makes the next plot invisible (only the axes) but maintains it anchors and name. If you want it to behave in another style then you can redefine it. Consider the same example as before.




```
\begin{tikzpicture}
  \begin{groupplot}[group style={
        group name=my plots,group size=2 by 2},
    width=4cm,height=4cm]
    \nextgroupplot[group/empty plot]
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
    \nextgroupplot
    \addplot coordinates{(0,0) (1,2) (2,1)};
  \end{groupplot}
  \draw (my plots c1r1.east)
      circle (3pt) node {East};
  \draw (my plots c2r1.north)
      circle (3pt) node {north};
  \draw (my plots c1r2.center)
      circle (3pt) node {center};
  \draw (my plots c2r2.north west)
      circle (3pt) node {North west};
\end{tikzpicture}
```

Notice that you need to call a `\nextgroupplot` againwards to jump to the next plot.

5.6 Patchplots Library

```
\usepgfplotslibrary{patchplots} % LaTeX and plain TeX
\usepgfplotslibrary[patchplots] % ConTeXt
\usetikzlibrary{pgfplots.patchplots} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.patchplots] % ConTeXt
```

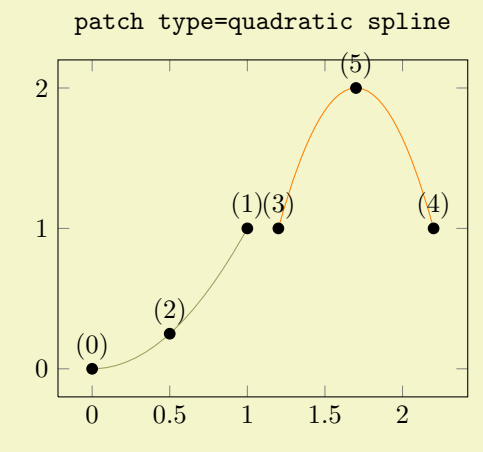
A library for advanced `patch` plots. It has been designed to visualize shaded isoparametric finite element meshes of higher order. Its core is an interface to the generation of smoothly shaped elements with interpolated color values (based on `.pdf` Shading Type 6 and `.pdf` Shading Type 7), with additional (limited) support for constant color filling without shadings.

5.6.1 Additional Patch Types

`/pgfplots/patch type=default|rectangle|triangle|line|quadratic spline|cubic spline|bilinear|triangle quadr|biquadratic|coons|polygon|tensor bezier` (initially `default`)

The `patchplots` library supports several new `patch type`s in addition to the initially available choices (which are `rectangle`,`triangle` and `line`). The documentation of the two–dimensional choices from page 114 is repeated here.

There are two new one–dimensional patch types, namely `quadratic spline` and `cubic spline`. Here, `patch type=quadratic spline` consists of quadratic patches of $n = 3$ vertices each. The vertices are interpolated exactly:

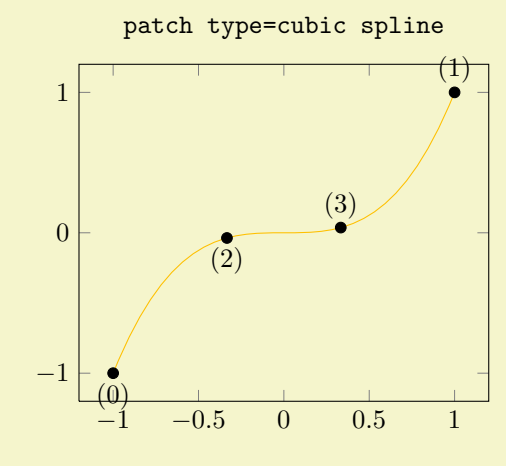



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title={\texttt{patch type=quadratic spline}}]
\addplot[
    mark=*,
    patch,mesh,%  without mesh, pgfplots tries to fill
    patch type=quadratic spline]
coordinates {
    %  left, right, middle-> first segment
    (0,0) (1,1) (0.5,0.5^2)
    %  left, right, middle-> second segment
    (1.2,1) (2.2,1) (1.7,2)
};
\end{axis}
\end{tikzpicture}
```

In our example, the first segment interpolates $f(x) = x^2$ at the points $\{0, 1/2, 1\}$. The `quadratic`

`spline` is actually nothing but piecewise Lagrangian interpolation with quadratic polynomials: it expects three points in the sequence '(left end), (right end), (middle)' and interpolates these three points with a quadratic polynomial. Unlike the default 1d `mesh` visualization (which uses `patch type=line` implicitly), you have to use the special syntax above (or the equivalent approach by means of `patch table`). Note that `patch type=quadratic spline` results in correct shapes, but uses *just constant color* for each segment; high–order color shading is only supported approximately using `patch refines`.

The `patch type=cubic spline` is very similar: it expects patches of $n = 4$ vertices and interpolates them with a cubic polynomial:



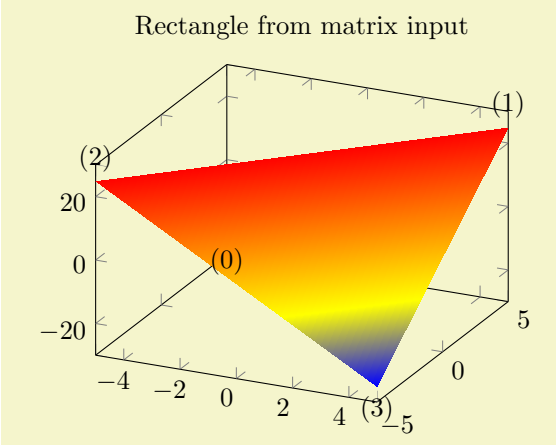

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title={\texttt{patch type=cubic spline}}]
\addplot[
    mark=*,
    patch,mesh,
    patch type=cubic spline]
coordinates {
    %  left, right, left middle, right middle
    (-1,-1)
    (1,1)
    (-1/3,{(-1/3)^3})
    (1/3,{(1/3)^3})
};
\end{axis}
\end{tikzpicture}
```

Here, we interpolated $f(x) = x^3$ at the four equidistant points $\{-1, -1/3, 1/3, 1\}$ with a cubic polynomial (which is x^3). The `cubic spline` expects a sequence of patches, each with four coordinates, given in the sequence '(left end), (right end), (interpolation point at $1/3$), (interpolation point at $2/3$)'. It has limitations and features like `quadratic spline`, see above.

The `patchplots` library is especially strong for `shader=interp`, so this is our main focus in the remaining documentation here.

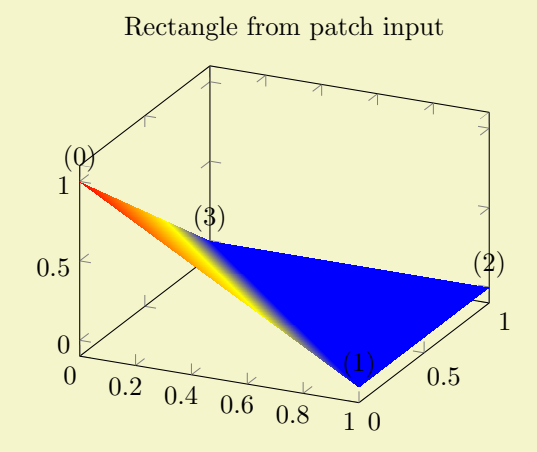
Attention: At the time of this writing, many free pdf viewers do not fully support the following shadings[66]. The preferred viewer is Adobe Acrobat Reader.

The choice `rectangle` expects one or more rectangular patches with $n = 4$ vertices each. These vertices are either encoded as a matrix or as individual patches (using `mesh input=patches`), in the sequence in which you would connect the vertices:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Rectangle from matrix input]
%  note that surf implies 'patch type=rectangle'
\addplot3[surf,shader=interp,samples=2,
    patch type=rectangle]
    {x*y};
\end{axis}
\end{tikzpicture}
```

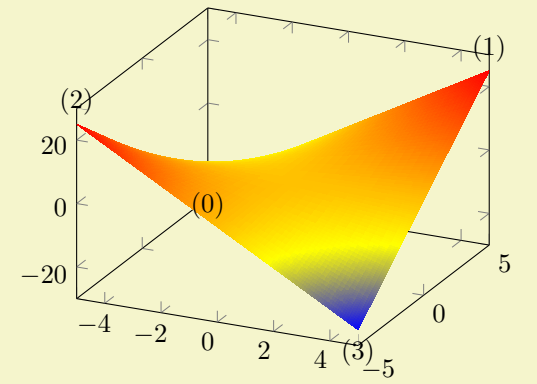
[66] The author of this package has submitted bugfixes to Linux viewers based on xpdf/libpoppler, so the problem will (hopefully) vanish in future versions.



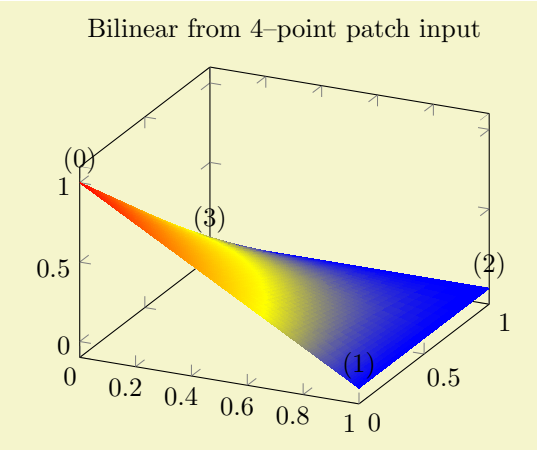

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Rectangle from patch input]
\addplot3[patch,shader=interp,patch
type=rectangle] coordinates {
    (0,0,1) (1,0,0) (1,1,0) (0,1,0)
};
\end{axis}
\end{tikzpicture}
```

As already documented on page 114, the `shader=interp` implementation for `rectangle` uses two triangles and interpolates them linearly. The differences between the two examples above arise due to z buffering approaches: the matrix input reorders the matrix in linear time, whereas the second example would sort complete rectangles. In our case, this yields to the different corner sequence.

The choice `bilinear` is essentially the same as `rectangular` with respect to its input formats and stroke paths, but it uses correct bilinear shading for `shader=interp`. The two examples from above now become

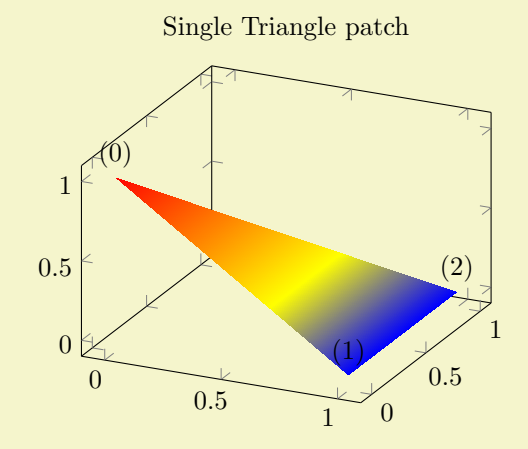



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Bilinear from $2\times 2$ matrix input]
%  note that surf implies 'patch type=rectangle'
\addplot3[surf,shader=interp,samples=2,
    patch type=bilinear]
    {x*y};
\end{axis}
\end{tikzpicture}
```



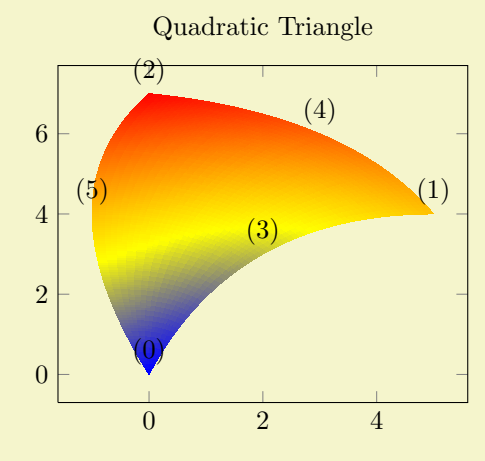

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Bilinear from $4$--point patch input]
\addplot3[patch,shader=interp,patch type=bilinear]
coordinates {
    (0,0,1) (1,0,0) (1,1,0) (0,1,0)
};
\end{axis}
\end{tikzpicture}
```

The choice `triangle` expects a sequence of linear triangles, each encoded using $n = 3$ vertices:

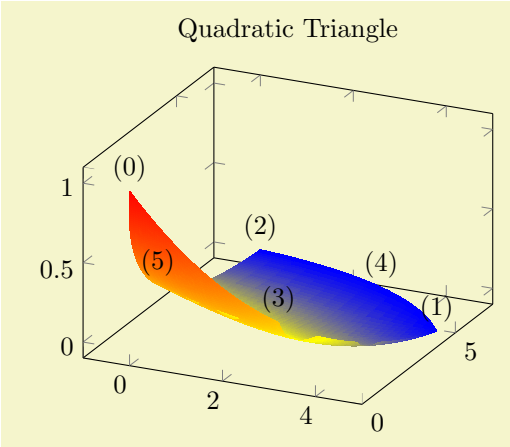



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[enlargelimits,
    nodes near coords={(\coordindex)},
    title=Single Triangle patch]
\addplot3[patch,shader=interp] coordinates {
    (0,0,1)
    (1,0,0)
    (1,1,0)
};
\end{axis}
\end{tikzpicture}
```

The choice `triangle quadr` expects a sequence of isoparametric quadratic triangles, each defined by $n = 6$ vertices:



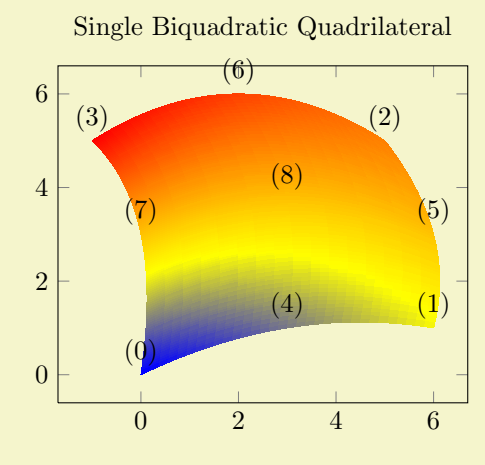

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Quadratic Triangle]
\addplot[patch,patch type=triangle quadr,
    shader=interp,point meta=explicit]
coordinates {
    (0,0) [1] (5,4) [2] (0,7) [3]
    (2,3) [1] (3,6) [2] (-1,4)  [3]
};
\end{axis}
\end{tikzpicture}
```



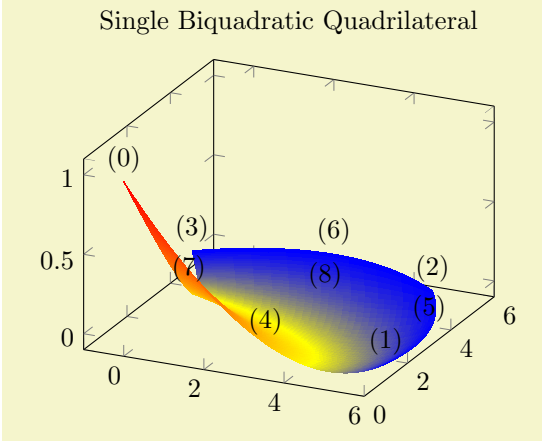

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Quadratic Triangle]
\addplot3[patch,patch type=triangle quadr,
    shader=interp]
coordinates {
    (0,0,1) (5,4,0) (0,7,0)
    (2,3,0) (3,6,0) (-1,4,0)
};
\end{axis}
\end{tikzpicture}
```

Here, the edges have the correct quadratic shape. However, the color interpolation is just *bilinear*; using the color values of the corners and ignoring the rest (consider using `patch refines` to improve the color interpolation). For three dimensions, PGFPLOTS checks the depth of corners to determine foreground/background. For two dimensions, strongly distorted elements may fold over each other in unexpected ways.

The choice `biquadratic` expects a sequence of isoparametric biquadratic quadrilaterals (rectangles), each defined by $n = 9$ vertices:



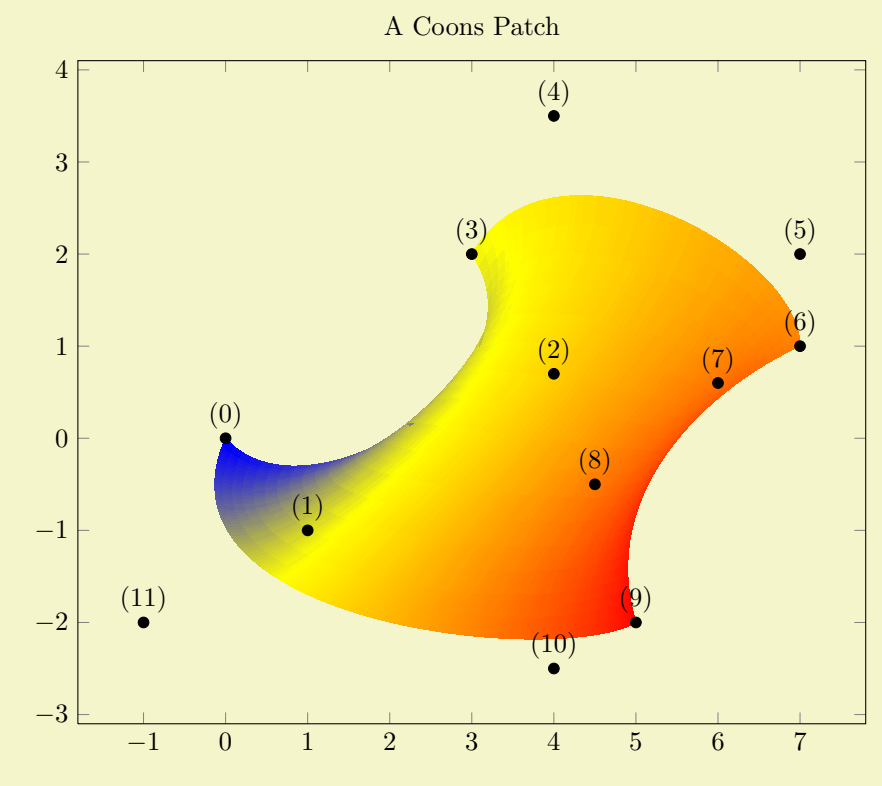

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Single Biquadratic Quadrilateral]
\addplot[patch,patch type=biquadratic,
    shader=interp,point meta=explicit]
coordinates {
    (0,0) [1] (6,1) [2] (5,5) [3] (-1,5) [4]
    (3,1) [1] (6,3) [2] (2,6) [3] (0,3) [4]
    (3,3.75) [4]
};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={(\coordindex)},
    title=Single Biquadratic Quadrilateral]
\addplot3[patch,patch type=biquadratic,shader=interp]
coordinates {
    (0,0,1) (6,1,0) (5,5,0) (-1,5,0)
    (3,1,0) (6,3,0) (2,6,0) (0,3,0)
    (3,3.75,0)
};
\end{axis}
\end{tikzpicture}
```

Similar to `triangle quadr`, the edges have the correct quadratic shape – but the color interpolation is just *bilinear*; using the color values of the corners and ignoring the rest. Again, consider using `patch refines` to improve the color interpolation.

The choice `coons` expects a sequence of one or more Coons patches, made up of $n = 12$ points each. A Coons patch is delimited by four cubic Bézier curves, with the end points attached to each other – and the n points provide the required control points for these curves in a specific ordering which is illustrated in the following example:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[nodes near coords={{\coordindex}},
    width=12cm,
    title=A Coons Patch]
\addplot[mark=*,patch,patch type=coons,
    shader=interp,point meta=explicit]
coordinates {
    (0,0)   [0] %  first corner
    (1,-1)  [0] %  Bezier control point between (0) and (3)
    (4,0.7) [0] %  Bezier control point between (0) and (3)
    %
    (3,2)   [1] %  second corner
    (4,3.5) [1] %  Bezier control point between (3) and (6)
    (7,2)   [1] %  Bezier control point between (3) and (6)
    %
    (7,1)      [2] %  third corner
    (6,0.6)    [2] %  Bezier control point between (6) and (9)
    (4.5,-0.5) [2] %  Bezier control point between (6) and (9)
    %
    (5,-2)   [3] %  fourth corner
    (4,-2.5) [3] %  Bezier control point between (9) and (0)
    (-1,-2)  [3] %  Bezier control point between (9) and (0)
};
\end{axis}
\end{tikzpicture}
```

The four cubic Bézier curves are *equivalent* to `curveto` paths of PGF, i.e. (⟨*corner 1*⟩)`..controls`(⟨*control point A*⟩)` and `(⟨*control point B*⟩)` .. `(⟨*corner 2*⟩) paths. The interpolated shading is bilinear. More precisely, a bilinear shading in the unit cube $[0,1]^2$ is initialised which is then mapped into the Coons patch such that the corners match. The color interpolation uses only the color data of the four corners, color values of intermediate control points are ignored for the shading (although their value will be respected for the upper and lower limit of color data). In contrast to the finite element patches, a Coons patch is inherently two–dimensional. While you can still use three–dimensional coordinates, PGFPLOTS will draw the shading as you provide it, without checking for the depth information (as it does for the other `patch types`). In other words: depending on the current `view` angle, the shading might fold over itself in unexpected ways.

Even for two dimensions, Coons patches may fold over themselves. To determine which part is

foreground and which part is background, the following rule applies: the four corner points (0), (3), (6), (9) are associated to the unit cube points $(u, v) = (0, 0)$, $(0, 1)$, $(1, 1)$ and $(1, 0)$, respectively. The edge between corner (3) and (6) (i.e. the one with $v = 1$) is foreground, the edge between (1) and (9) is background. Thus, large values of v are drawn on top of small values of v. If v is constant, large values of u are drawn on top of small values of u. Thus, reordering the patch vertices (choosing a different first vertex and/or reversing the sequence) allows to get different foreground/background configurations[67].

The choice `tensor bezier` is similar to `patch type=coons`: it allows to define a bezier patch. However, it allows more freedom: it has 16 control points instead of the 12 of a `coons` patch. The four additional control points are situated in the center of each patch. This `patch type` generates `.pdf` shadings of type 7 (whereas `coons` patches are shadings of type 6). It has been added for reasons of completeness, although it has not been tested properly. Please refer to the specification of the `.pdf` format for details[68].

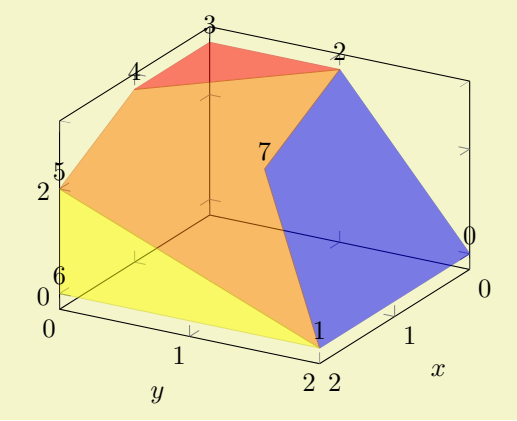
The choice `polygon` expects polygons with a fixed number of vertices. This `patch type` requires the number of vertices as argument:

`/pgfplots/vertex count=`⟨*count*⟩

The number of vertices to be used for `patch type=polygon`. The number can be arbitrary. All input patches are expected to have this many vertices – but it is acceptable if a patch uses the same vertex multiple times. This means that `patch type=polygon` accepts polygons with different numbers of vertices, but you need to apply some sort of "manual padding".

This parameter is (currently) mandatory.

A `patch` plot with `patch type=polygon` simply connects the n=`vertex count` vertices in their order of appearance and closes the resulting path:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[view/h=120,xlabel=$x$,ylabel=$y$]
    \addplot3[
        opacity=0.5,
        table/row sep=\\,
        patch,
        patch type=polygon,
        vertex count=5,
        patch table with point meta={%
            %  pt1 pt2 pt3 pt4 pt5 cdata
            0 1 7 2 2 0\\
            1 6 5 5 5 1\\
            1 5 4 2 7 2\\
            2 4 3 3 3 3\\
    }]
    table {
        x y z\\
        0 2 0\\% 0
        2 2 0\\% 1
        0 1 3\\% 2
        0 0 3\\% 3
        1 0 3\\% 4
        2 0 2\\% 5
        2 0 0\\% 6
        1 1 2\\% 7
    };
%  replicate the vertex list to show \coordindex:
\addplot3[only marks,nodes near coords=\coordindex]
table[row sep=\\] {
0 2 0\\ 2 2 0\\ 0 1 3\\ 0 0 3\\
1 0 3\\ 2 0 2\\ 2 0 0\\ 1 1 2\\
};
    \end{axis}
\end{tikzpicture}
```

The example above defines the `patch` by means of a connectivity table (`patch table with point`

[67] Internally, PGFPLOTS employs such mechanisms to map the higher order isoparametric patch types to Coons patches, sorting according their corner's depth information.

[68] If someone is willing to test it and document it, feel free to email me!

meta) and a vertex list (the normal input coordinates of the plot): there are 8 vertices and 4 polygons. Note that 2 of these polygons are triangles, one has 4 corners and only of them actually has all 5 allocated corners. This effect can be achieved by replicating one of the corners. The connectivity table in our example defines a unique color for each polygon: 0 for the first patch, 1 for the second, 2 for the third, and 3 for the last. These numbers map into the current colormap.

The patch type=polygon supports *neither* triangulation *nor* shading *nor* refinement. The order of appearance of the input points is supposed to be the order in which the line–to operations of the resulting path are generated.

5.6.2 Automatic Patch Refinement and Triangulation

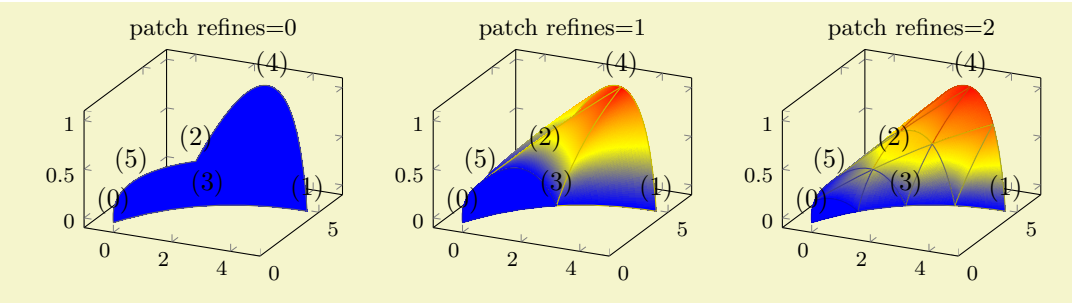
PGFPLOTS supports automatic patch refinement for most of its patch types. There are mainly two purposes for patch refinement: to increase the quality of z buffer=sort and/or to improve color interpolation for high–order patches.

/pgfplots/patch refines={⟨*levels*⟩} (initially 0)

This key controls patch refinement. The initial choice patch refines=0 disables refinement and visualizes elements as they have been found in input files.

A positive ⟨*levels*⟩ enables (recursive) patch refinement: each patch is refined individually.

The following example illustrates the patch refines feature for a triangle quadr shape function on an edge. Note that since PGFPLOTS uses only first order shading which is based on the corner points (0), (1) and (2), the specified shape function of patch refines=0 has constant color. Higher ⟨*levels*⟩ approximate the patch with increasing quality:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\foreach \level in {0,1,2} {%
    \begin{tikzpicture}
    \begin{axis}[
        nodes near coords={(\coordindex)},
        footnotesize,
        title={patch refines=\level}]

    \addplot3[patch,patch type=triangle quadr,
        shader=faceted interp,patch refines=\level]
    coordinates {
        (0,0,0) (5,4,0) (0,7,0)
        (2,3,0) (3,6,1) (-1,4,0)
    };
    \end{axis}
    \end{tikzpicture}
}
```

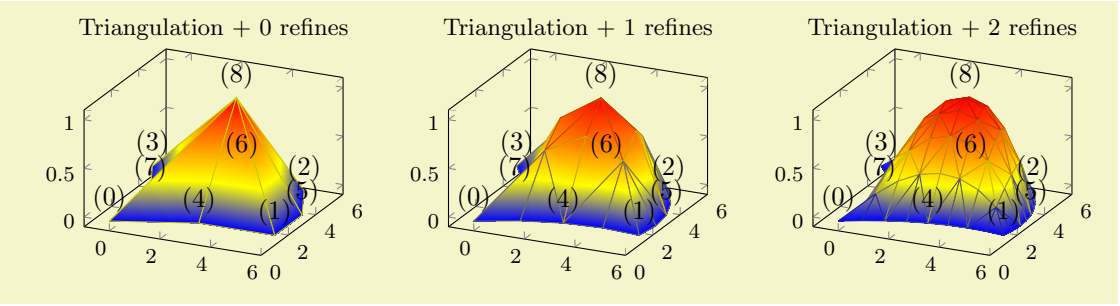
In this example, patch refinement makes a huge difference since it is just one element with huge displacements. For practical examples, you probably won't need many refinement levels.

The refined patches reproduce the geometry's shape exactly. In addition, they improve color interpolation. Note that its purpose is just visualization, therefor hanging nodes are allowed (and will be generated by patch refine for most patch types).

Patch refinement is implemented for all supported patches except for patch type=coons, tensor bezier, and polygon.

/pgfplots/patch to triangles=true|false (initially false)

Occasionally, one has a complicated `patch type` on input and would like to visualize it as a `triangle` mesh. PGFPLOTS supports automatic triangulation of patches. Triangulation means to replace each individual input patch by one or more triangles. Triangulation can be combined with `patch refines` in which case `patch refines` is applied first and the resulting refined patches are then triangulated.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\foreach \level in {0,1,2} {%
    \begin{tikzpicture}
    \begin{axis}[
        nodes near coords={{\coordindex}},
        footnotesize,
        title={Triangulation + \level\ refines}]

    \addplot3[patch,patch type=biquadratic,shader=faceted interp,
        patch to triangles,patch refines=\level]
    coordinates {
        (0,0,0) (6,1,0) (5,5,0) (-1,5,0)
        (3,1,0) (6,3,0) (2,6,0) (0,3,0)
        (3,3.75,1)
    };
    \end{axis}
    \end{tikzpicture}%
}
```

For one–dimensional `patch type`s like `quadratic spline`, `patch to triangles` results in approximation by means of `patch type=line` instead of `triangle`.

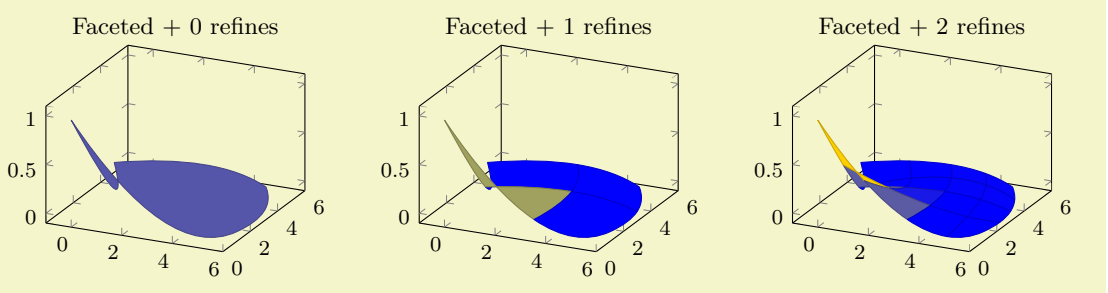
The `patch to triangles` feature is implemented for all supported patches except for `patch type=coons`, `tensor bezier`, and `polygon`.

5.6.3 Peculiarities of Flat Shading and High Order Patches

The `patchplots` library has been optimized for use with interpolated shadings, i.e. for `shader=interp`: it allows the filled area to fold over itself or to be outside of the patch boundaries.

PGFPLOTS also supports `shader=flat` and `shader=faceted` by simply stroking and/or filling the patch boundaries. Naturally, such an approach works only if the enclosed patch boundary and the filled area are essentially the same! Consider using `shader=flat` or `shader=faceted` only if the *mesh width is small enough* such that patches do not fold over themselves.

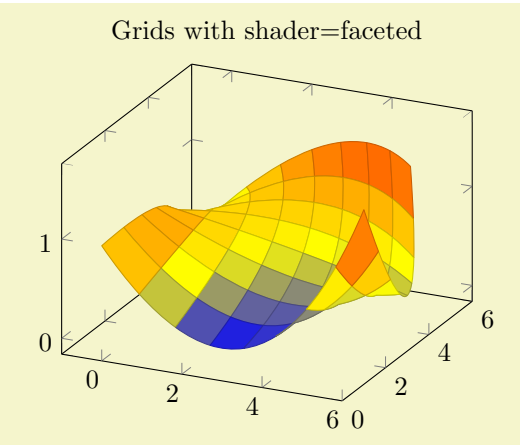
The following example illustrates the effect: the coarse single element on the left folds over itsself, resulting in strange fill patterns. Refining the mesh reduces the effect.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\foreach \level in {0,1,2} {%
    \begin{tikzpicture}
    \begin{axis}[
        footnotesize,
        title={Faceted + \level\ refines}]

    \addplot3[patch,patch type=biquadratic,shader=faceted,
        patch refines=\level]
    coordinates {
        (0,0,1) (6,1,0) (5,5,0) (-1,5,0)
        (3,1,0) (6,3,0) (2,6,0) (0,3,0)
        (3,3.75,0)
    };
    \end{axis}
    \end{tikzpicture}
}
```

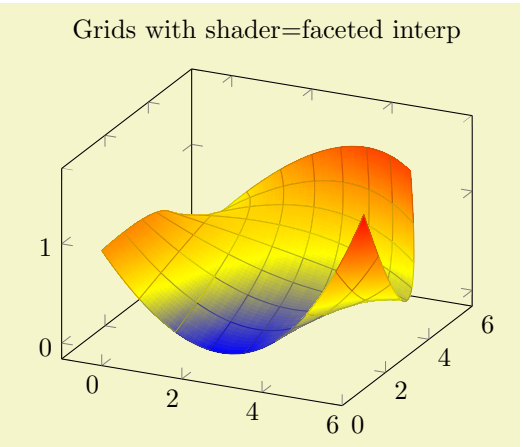
5.6.4 Drawing Grids

The `patchplots` library supports grid (`mesh`) visualization in the same way as for two/three–dimensional `mesh`- and `surf` plots. This includes four different approaches: the first is `shader=faceted`, which uses constant fill color and `faceted color` for stroke paths (as we already saw in Section 5.6.3). The second approach is to use `shader=faceted interp` which uses interpolated shadings for filling and issues stroke paths on top of each interpolated element. The third approach is to issue two `\addplot` commands, one with the filled `patch` plot, and one with a `patch,mesh` style which only draws (colored) grid lines on top of the previous plot. The three approaches are shown below.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Grids with shader=faceted}]

\addplot3[patch,patch type=biquadratic,
    shader=faceted,patch refines=3]
coordinates {
    (0,0,1) (6,1,1.6) (5,5,1.3) (-1,5,0)
    (3,1,0) (6,3,0.4) (2,6,1.1) (0,3,0.9)
    (3,3.75,0.5)
};
\end{axis}
\end{tikzpicture}
```

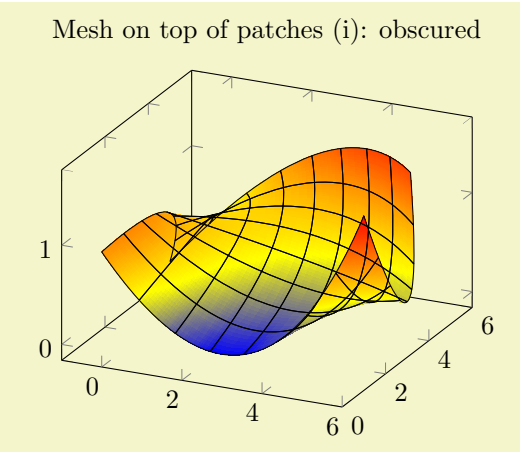
As already discussed in Section 5.6.3, the approach with `shader=faceted` works well if the mesh width is small enough (such that single patches do not overlap and their fill area is within the patch boundaries).



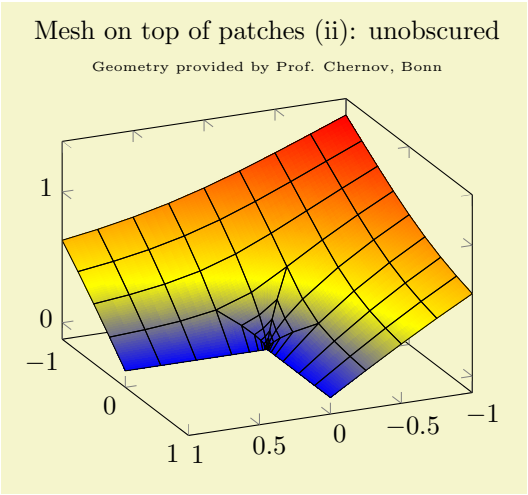

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Grids with shader=faceted interp}]

\addplot3[patch,patch type=biquadratic,
    shader=faceted interp,patch refines=3]
coordinates {
    (0,0,1) (6,1,1.6) (5,5,1.3) (-1,5,0)
    (3,1,0) (6,3,0.4) (2,6,1.1) (0,3,0.9)
    (3,3.75,0.5)
};
\end{axis}
\end{tikzpicture}
```

Here, grid lines are defined to be the patch boundary, so it may occasionally happen for coarse patches that grid lines cross the filled area. If you experience problems, consider using the `patch refines` key.

The `shader=faceted interp` supports `z buffer` – at the cost of generating one shading for *each* patch element (the stroke path is drawn immediately after the patch element is shaded). This can become quite expensive[69] at display time and may lead to huge pdf files. However, `shader=faceted interp` provides smooth shadings and, at the same time, good grid lines which are drawn in the correct order.



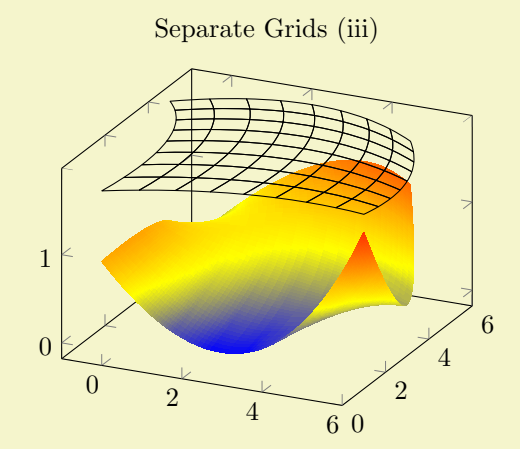

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Mesh on top of patches (i): obscured}]

\addplot3[patch,patch type=biquadratic,shader=interp,
    patch refines=3]
coordinates {
    (0,0,1) (6,1,1.6) (5,5,1.3) (-1,5,0)
    (3,1,0) (6,3,0.4) (2,6,1.1) (0,3,0.9)
    (3,3.75,0.5)
};
\addplot3[patch,patch type=biquadratic,mesh,black,
    patch refines=3]
coordinates {
    (0,0,1) (6,1,1.6) (5,5,1.3) (-1,5,0)
    (3,1,0) (6,3,0.4) (2,6,1.1) (0,3,0.9)
    (3,3.75,0.5)
};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Mesh on top of patches (ii): unobscured\\
      \tiny Geometry provided by Prof. Chernov, Bonn},
    title style={align=center},
    view={156}{28}]
\addplot3[patch,patch type=bilinear,
    shader=interp,
    patch table=plotdata/patchexample_conn.dat]
    file {plotdata/patchexample_verts.dat};

\addplot3[patch,patch type=bilinear,
    mesh,black,
    patch table=plotdata/patchexample_conn.dat]
    file {plotdata/patchexample_verts.dat};
\end{axis}
\end{tikzpicture}
```

The approach to draw grids separately is realized by means of two `\addplot` statements; the first using `patch` as before, the second using `patch,mesh`. This configures PGFPLOTS to visualize just the mesh. Make sure you provide '`mesh`' after '`patch`' since the latter activates filled `surf` visualization. The approach of meshes on top of patches implies to draw grid lines simply over any previous drawing operations. Thus, depth information is lost (as displayed in the first example above). Overlaying grid lines on top of the surface works in special cases (see bottom picture). An approach which always works is to provide the mesh at a fixed z position as displayed in the following example:

[69] I would really like to hear any well–founded ideas how to improve this issue. In case you have an idea– let me know!




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title={Separate Grids (iii)}]

\addplot3[patch,patch type=biquadratic,shader=interp,
    patch refines=3]
coordinates {
    (0,0,1) (6,1,1.6) (5,5,1.3) (-1,5,0)
    (3,1,0) (6,3,0.4) (2,6,1.1) (0,3,0.9)
    (3,3.75,0.5)
};
\addplot3[patch,patch type=biquadratic,
    mesh,black,
    z filter/.code={\def\pgfmathresult{1.8}},
    patch refines=3]
coordinates {
    (0,0,1) (6,1,1.6) (5,5,1.3) (-1,5,0)
    (3,1,0) (6,3,0.4) (2,6,1.1) (0,3,0.9)
    (3,3.75,0.5)
};
\end{axis}
\end{tikzpicture}
```

Here, the first `\addplot3` command is the same as above, just with `shader=interp`. The second reproduces the same geometry, but uses a `z filter` to fix the z coordinate (in this case to $z = 1.8$). This effectively overrules all z coordinates.

Thus, grid lines can be realized either by means of flat fill color with `shader=faceted` (efficient), by means of interpolated fill colors with `shader=faceted interp` (inefficient, see above) or, for special applications, using a separate `patch,mesh` plot which is drawn on top of the patches (efficient). In any case, the mesh visualization considers the `faceted color` which can depend on `mapped color`.

5.7 Polar Axes

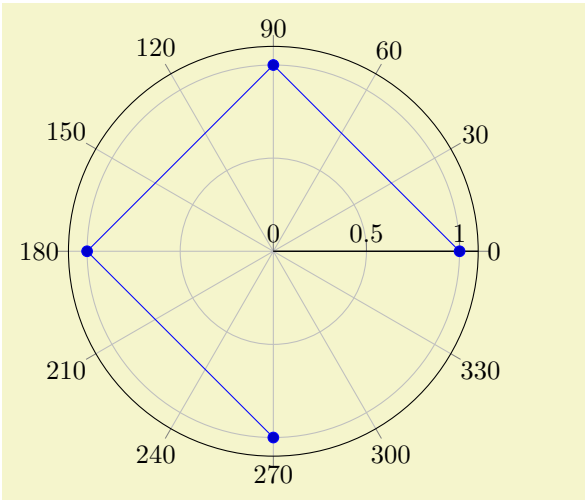
```
\usepgfplotslibrary{polar} % LaTeX and plain TeX
\usepgfplotslibrary[polar] % ConTeXt
\usetikzlibrary{pgfplots.polar} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.polar] % ConTeXt
```

A library to draw polar axes and plot types relying on polar coordinates, represented by angle (in degrees or, optionally, in radians) and radius.

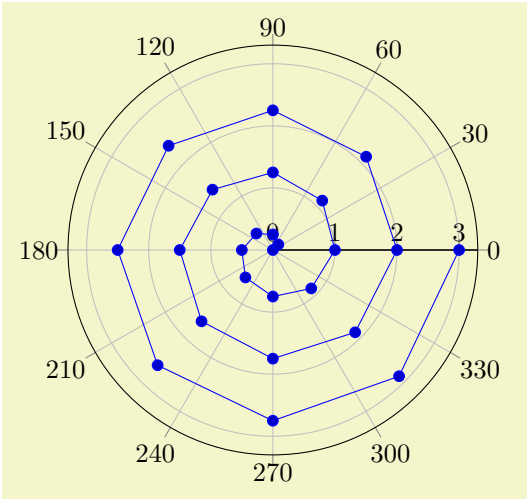
5.7.1 Polar Axes

```
\begin{polaraxis}
  ⟨environment contents⟩
\end{polaraxis}
```

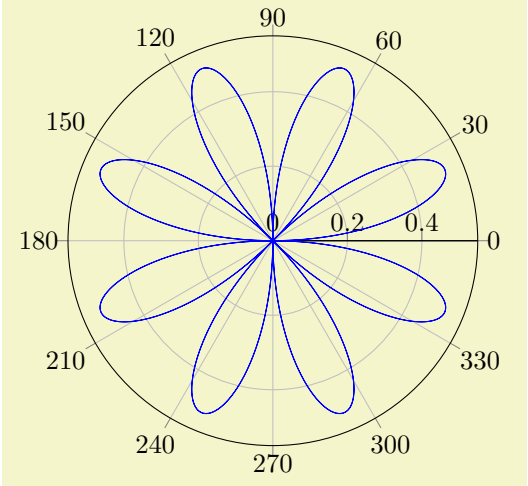
The `polar` library provides the `polaraxis` environment. Inside of such an environment, all coordinates are expected to be given in polar representation of the form $(\langle angle \rangle, \langle radius \rangle)$, i.e. the x coordinate is always the angle and the y coordinate the radius:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}
    \addplot coordinates {(0,1) (90,1)
        (180,1) (270,1)};
    \end{polaraxis}
\end{tikzpicture}
```

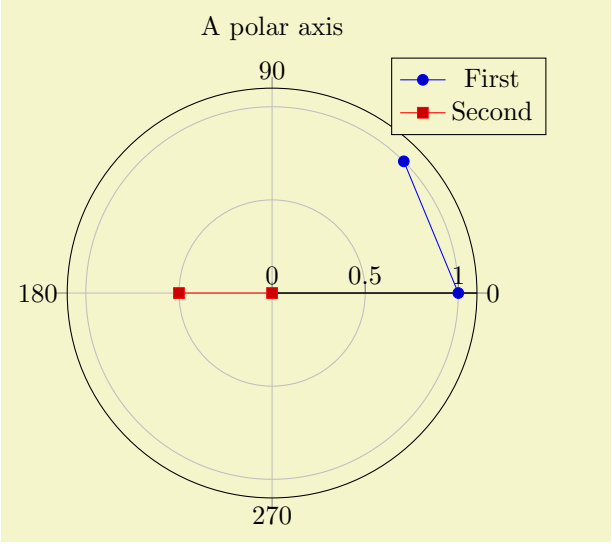



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}
    \addplot+[domain=0:3] (360*x,x); %  (angle,radius)
    \end{polaraxis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}
    \addplot+[mark=none,domain=0:720,samples=600]
        {sin(2*x)*cos(2*x)};
    %  equivalent to (x,{sin(..)cos(..)}), i.e.
    %  the expression is the RADIUS
    \end{polaraxis}
\end{tikzpicture}
```

Polar axes support most of the PGFPLOTS user interface, i.e. `legend entries`, any axis descriptions, `xtick`/`ytick` and so on:



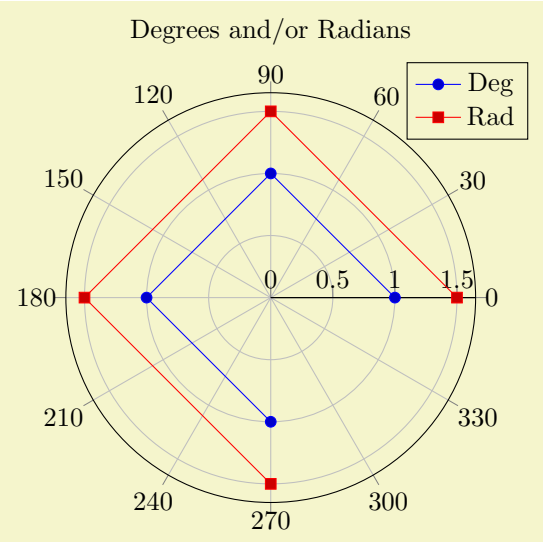

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[
        xtick={0,90,180,270},
        title=A polar axis]

    \addplot coordinates {(0,1) (45,1)};
    \addlegendentry{First}

    \addplot coordinates {(180,0.5) (0,0)};
    \addlegendentry{Second}
    \end{polaraxis}
\end{tikzpicture}
```

Furthermore, you can use all of the supported input coordinate methods (like `\addplot coordinates`, `\addplot table`, `\addplot expression`). The only difference is that polar axes interpret the (first two) input coordinates as polar coordinates of the form $(\langle \textit{angle in degrees} \rangle, \langle \textit{radius} \rangle)$.

It is also possible to provide `\addplot3`; in this case, the third coordinate will be ignored (although it can be used as color data using `point meta=z`). An example can be found below in Section 5.7.3.

5.7.2 Using Radians instead of Degrees

The initial configuration uses degrees for the angle (x component of every input coordinate). PGFPLOTS also supports to provide the angle in radians using the `data cs=polarrad` switch:



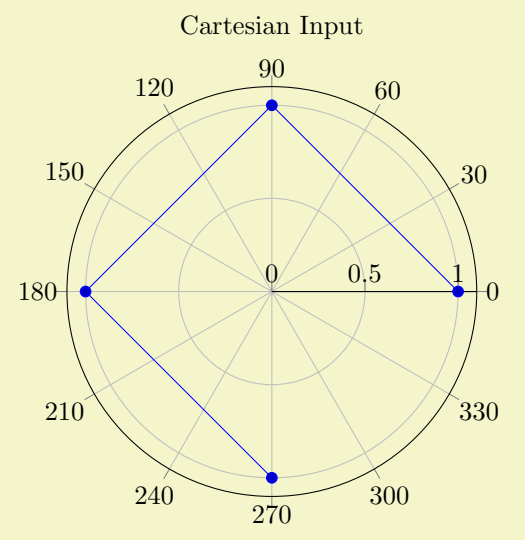

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[title={Degrees and/or Radians}]
    \addplot
        coordinates {(0,1) (90,1) (180,1) (270,1)};
    \addlegendentry{Deg}

    \addplot+[data cs=polarrad]
        coordinates {(0,1.5) (pi/2,1.5)
          (pi,1.5) (pi*3/2,1.5)};
    \addlegendentry{Rad}
    \end{polaraxis}
\end{tikzpicture}
```

The `data cs` key is described in all detail on page 274; it tells PGFPLOTS the coordinate system of input data. PGFPLOTS will then take steps to automatically transform each coordinate into the required coordinate

system (in our case, this is `data cs=polar`).

5.7.3 Mixing With Cartesian Coordinates

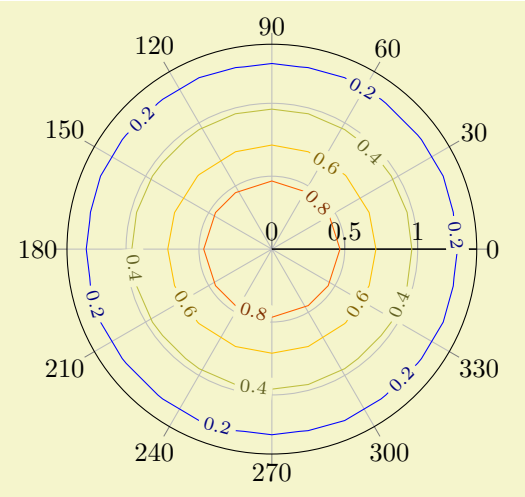
Similarly to the procedure described above, you can also provide Cartesian coordinates inside of a polar axis: simply tell PGFPLOTS that it should automatically transform them to polar representation by means of `data cs=cart`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[title=Cartesian Input]
    \addplot+[data cs=cart]
        coordinates {(1,0) (0,1) (-1,0) (0,-1)};
    \end{polaraxis}
\end{tikzpicture}
```

More details about the `data cs` key can be found on page 274.

This does also allow more involved visualization techniques which may operate on Cartesian coordinates. The following example uses `\addplot3` to sample a function $f\colon \mathbb{R}^2 \to \mathbb{R}$, computes `contour` lines (with the help of `gnuplot`) and displays the result in a `polaraxis`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}
    \addplot3[contour gnuplot,domain=-3:3,
      data cs=cart]
        {exp(-x^2-y^2)};
    \end{polaraxis}
\end{tikzpicture}
```

What happens is that $z = \exp(-x^2 - y^2)$ is sampled for $x, y \in [-3, 3]$, then contour lines are computed on (x, y, z), then the resulting triples (x, y, z) are transformed to polar coordinates (α, r, z) (leaving z intact). Finally, the z coordinate is used as `point meta` to determine the color.

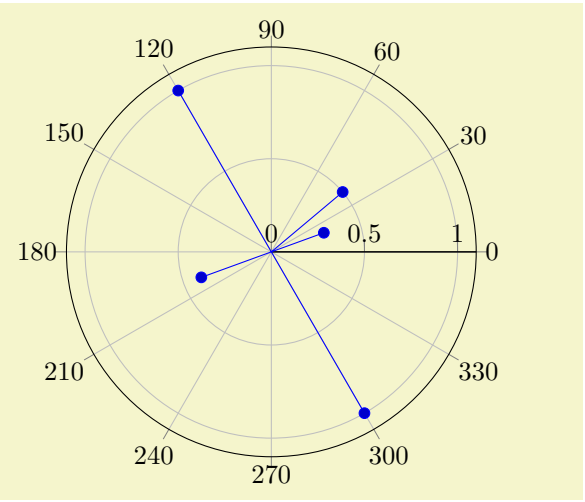
Note that `\addplot3` allows to process three–dimensional input types, but the result will always be two–dimensional (the z coordinate is ignored for point placement in `polaraxis`). However, the z coordinate can be used to determine point colors (using `point meta=z`).

5.7.4 Special Polar Plot Types

`/tikz/polar comb` (no value)

`\addplot+[polar comb]`

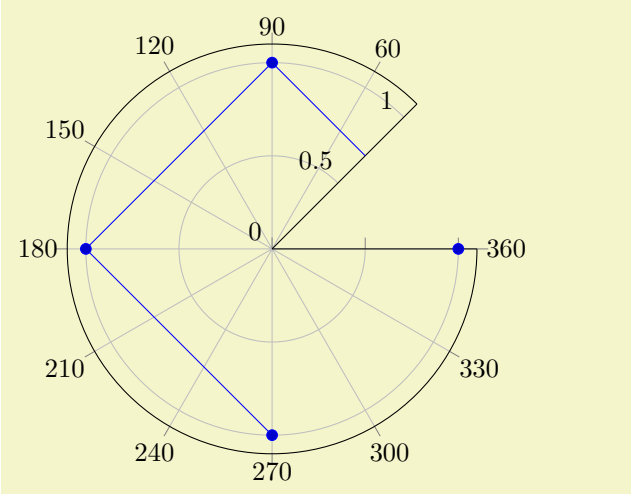
The polar comb plot handler is provided by Ti*k*Z; it draws paths from the origin to the designated position and places marks at the positions (similar to the comb plot handler). Since the paths always start at the origin, it is particularly suited for polaraxis:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}
    \addplot+[polar comb]
        coordinates {(300,1) (20,0.3) (40,0.5)
            (120,1) (200,0.4)};
    \end{polaraxis}
\end{tikzpicture}
```

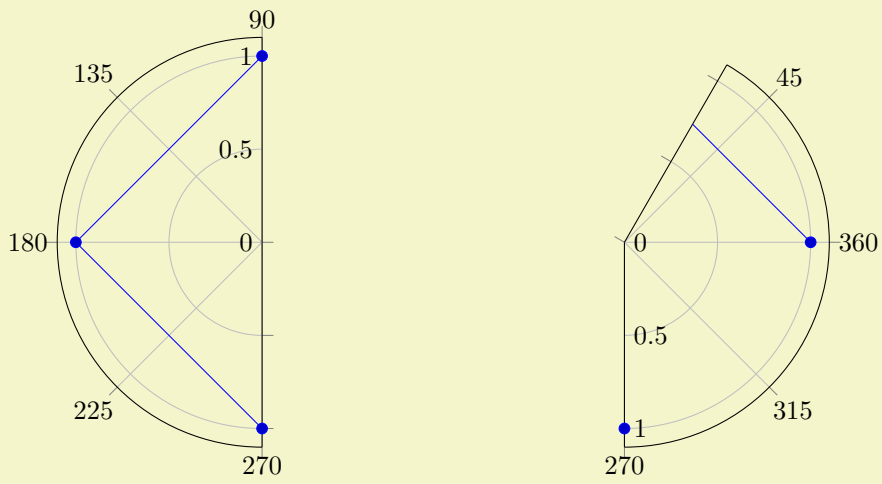
5.7.5 Partial Polar Axes

The polar library also supports partial axes. If you provide xmin/xmax, you can restrict the angles used for the axis:



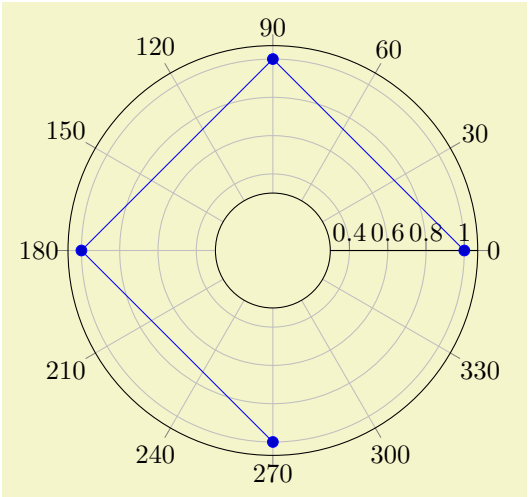

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[xmin=45,xmax=360]
    \addplot coordinates {(0,1) (90,1) (180,1) (270,1)};
    \end{polaraxis}
\end{tikzpicture}
```

Currently, the first angle must be lower than the second one. But you can employ the periodicity to get pies as follows:



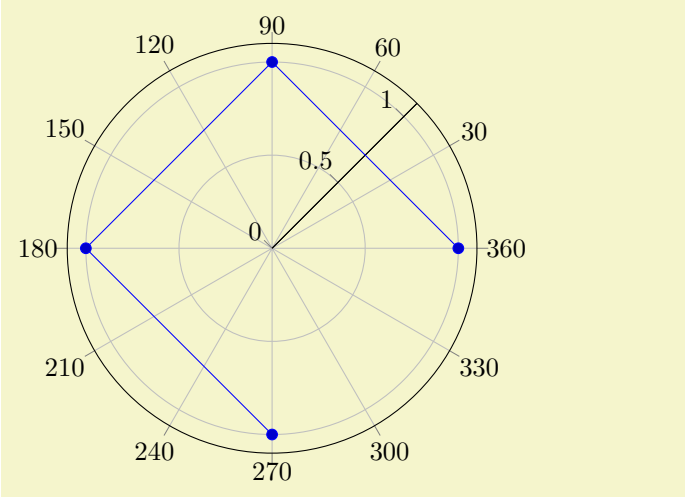

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[xmin=90,xmax=270]
    \addplot coordinates {(0,1) (90,1) (180,1) (270,1)};
    \end{polaraxis}
\end{tikzpicture}~%
\begin{tikzpicture}
    \begin{polaraxis}[xmin=270,xmax=420]
    \addplot coordinates {(0,1) (90,1) (180,1) (270,1)};
    \end{polaraxis}
\end{tikzpicture}
```

Similarly, an explicitly provided value for `ymin` allows to reduce the displayed range away from 0:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[ymin=0.3]
    \addplot coordinates {(0,1) (90,1)
      (180,1) (270,1)};
    \end{polaraxis}
\end{tikzpicture}
```

Modifying `xmin` and `xmax` manually can also be used to move the y axis line (the line with `ytick` and `yticklabels`):




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{polaraxis}[xmin=45,xmax=405]
    \addplot coordinates {(0,1) (90,1) (180,1) (270,1)};
    \end{polaraxis}
\end{tikzpicture}
```

5.8 Smith Charts

```
\usepgfplotslibrary{smithchart} % LaTeX and plain TeX
\usepgfplotslibrary[smithchart] % ConTeXt
\usetikzlibrary{pgfplots.smithchart} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.smithchart] % ConTeXt
```

A library to draw Smith Charts.

A Smith Chart maps the complex half plane with positive real parts to the unit circle. The `smithchart` library allows PGFPLOTS to visualize Smith Charts: it visualizes two–dimensional input coordinates $z \in \mathbb{C}$ of the form $z = x + jy \in \mathbb{C}$ (j being the imaginary unit, $j^2 = -1$) with $x \geq 0$ using the map

$$r\colon [0,\infty] \times [-\infty,\infty] \to \{a + jb \mid a^2 + b^2 = 1\}, \quad r(z) = \frac{z-1}{z+1}$$

using complex number division. The result is always in the unit circle.

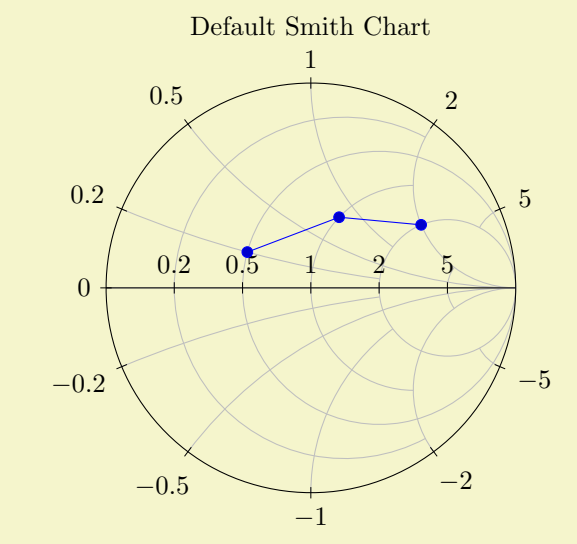
The main application for Smith Charts is in the area of electrical and electronics engineers specializing in radio frequency: to show the reflection coefficient $r(z)$ for normalised impedance z. It is beyond the scope of this manual to delve into the radio frequency techniques; for us, it is important to note that the `smithchart` library supports

- the data map $r(z)$ shown above,
- an axis class which interprets x as the real components and y as the imaginary components,
- a visualization of grid lines as arcs,
- the possibility to stop grid lines to allow uniform spacing in Smith Charts,
- a large set of the PGFPLOTS axis fine tuning parameters,
- input of already mapped coordinates $r(z)$ (i.e. Cartesian coordinates in the unit circle),
- many of the PGFPLOTS plot handlers.

5.8.1 Smith Chart Axes

`\begin{smithchart}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{smithchart}`

The `\begin{smithchart}` environment draws Smith Charts. It accepts the same ⟨*options*⟩ as `\begin{axis}`. In fact, it is equivalent to `\begin{axis}[`⟨*options*⟩`,axis type=smithchart]`.




```
\begin{tikzpicture}
    \begin{smithchart}[title=Default Smith Chart]
    \addplot coordinates {(0.5,0.2) (1,0.8) (2,2)};
    \end{smithchart}
\end{tikzpicture}
```

The example above visualizes three data points using the initial configuration of Smith Charts; the data points are interpreted as complex numbers $z = x + jy$ and are mapped using $r(z)$.

5.8.2 Size Control

A Smith Chart can be resized by providing either `width` or `height` as argument to the axis. If you provide both, the Chart is drawn as an ellipsis.

The tick and grid positions for `smithchart` axes are realized by means of three manually tuned sets of grid lines: one for small-sized plots, one for medium-sized plots and one for huge plots. The actual parameters for `width` or `height` are considered to select one of the following sets:

`/pgfplots/few smithchart ticks` (style, no value)

This produces the output of the example above – it constitutes the initial configuration for Smith Chart which has a width of less than `14cm`.

The `few smithchart ticks` style is defined by:

```
\pgfplotsset{
    few smithchart ticks/.style={
        default smithchart xtick/.style={
            xtick={0.2,0.5,1,2,5},
        },
        default smithchart ytick/.style={
            ytick={%
                0,%
                 0.2, 0.5, 1, 2, 5,%
                -0.2,-0.5,-1,-2,-5},
        },
        default smithchart xytick/.style={
            xgrid each nth passes y={2},
            ygrid each nth passes x={2},
        },
    },
}
```

Note that `few smithchart ticks` contains syntactical overhead to distinguish between "default ticks" and final tick positions: it does not assign `xtick` and `ytick` directly. Instead, it provides them as separate `default xtick style` arguments. The purpose of this distinction is to mark them as "default" arguments – the underlying styles `smithchart/every default xtick` is used if and only if there is no `xtick` value given.

In case you want to override this default, you can either

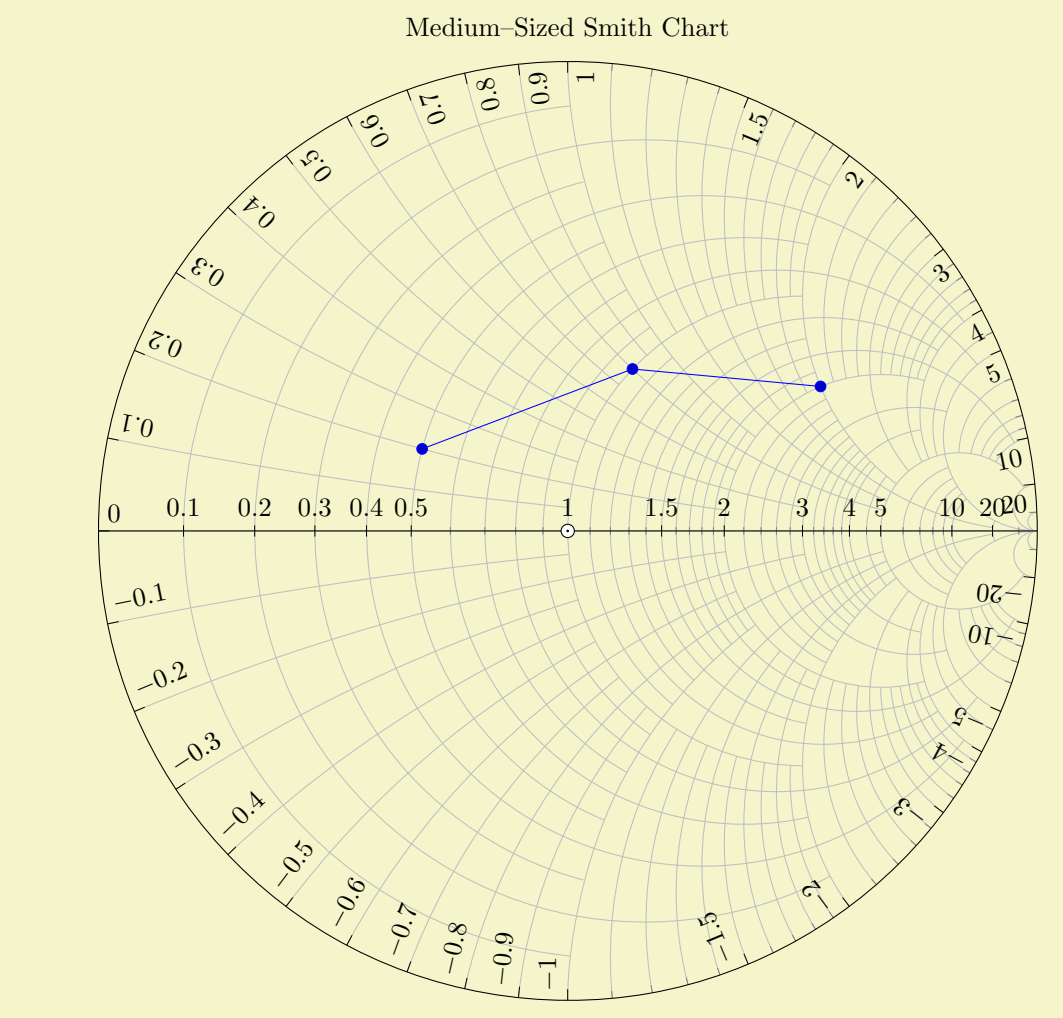
- copy–paste the definition above and adjust it or
- omit all the `default smithchart xtick/.style` stuff and write `xtick={`⟨*your list*⟩`}` directly.

As mentioned, the only purpose of the default smithchart xtick/.style overhead is to distinguish between \begin{smithchart}[xtick={⟨*user defined*⟩}] and default arguments (see the documentation of default smithchart xtick/.style for more about this technical detail).

For fine tuning of the scaling decisions, see the smith chart ticks by size key.

/pgfplots/many smithchart ticks (style, no value)

The many smithchart ticks style is used for every Smith Chart whose width exceeds 14cm although it is less than 20cm:




```
\begin{tikzpicture}
    \begin{smithchart}[
        title=Medium--Sized Smith Chart,
        width=14cm]
    \addplot coordinates {(0.5,0.2) (1,0.8) (2,2)};
    \end{smithchart}
\end{tikzpicture}
```

We see that many smithchart ticks has different placement and alignment options than few smithchart ticks: it uses sloped tick labels inside of the unit circle for the y descriptions (imaginary axis).

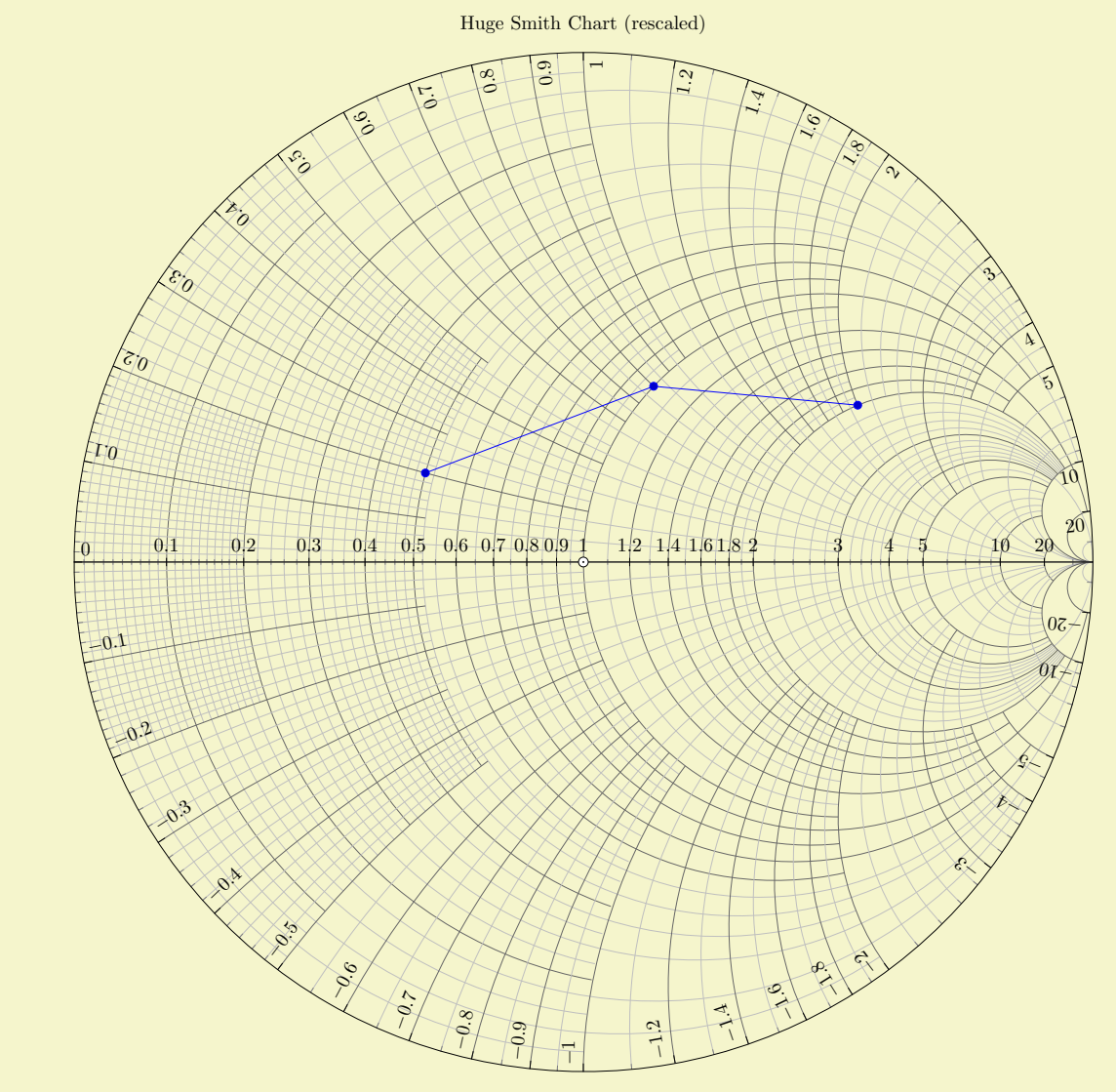
The initial configuration is realized by means of *two* separate styles: one which defines only the tick positions (the many smithchart ticks* style) and one which also changes placement and alignment options. The initial configuration can be changed individually (see the end of this section for examples). The initial configuration is:

```
\pgfplotsset{
    many smithchart ticks*/.style={
        default smithchart xtick/.style={
            xtick={
                0.1,0.2,0.3,0.4,0.5,1,1.5,2,3,4,5,10,20%
            },
            minor xtick={0.6,0.7,0.8,0.9,1.1,1.2,1.3,1.4,1.6,1.7,1.8,1.9,
              2.2,2.4,2.6,2.8,3.2,3.4,3.6,3.8,4.5,6,7,8,9,50},
        },
        default smithchart ytick/.style={
            ytick={%
                0,%
                0.1,0.2,...,1,1.5,2,3,4,5,10,20,%
                -0.1,-0.2,...,-1,-1.5,-2,-3,-4,-5,-10,-20%
            },
            minor ytick={%
                1.1,1.2,1.3,1.4,1.6,1.7,1.8,1.9,2.2,2.4,2.6,2.8,3.2,3.4,3.6,3.8,
                  4.5,6,7,8,9,50,%
                -1.1,-1.2,-1.3,-1.4,-1.6,-1.7,-1.8,-1.9,-2.2,-2.4,-2.6,-2.8,
                  -3.2,-3.4,-3.6,-3.8,-4.5,-6,-7,-8,-9,-50%
            },
        },
        default smithchart xytick/.style={
            xgrid each nth passes y={1,2,4,5,10,20},
            ygrid each nth passes x={1,2,3,5,10:3,20:3},
        },
    },
    /pgfplots/many smithchart ticks/.style={
        many smithchart ticks*,
        yticklabel in circle,
        show origin=true,
    },
}
```

See the documentation for `few smithchart ticks` for an explanation of the `default smithchart xtick/.style` overhead.

`/pgfplots/dense smithchart ticks` (style, no value)

The `dense smithchart ticks` style assigns the set of tick positions for every Smith Chart whose width is at least `20cm`:




```
\begin{tikzpicture}[scale=0.75]
    \begin{smithchart}[
        title=Huge Smith Chart (rescaled),
        width=20cm]
    \addplot coordinates {(0.5,0.2) (1,0.8) (2,2)};
    \end{smithchart}
\end{tikzpicture}
```

Attention: This style might change in future versions!

Similarly to `many smithchart ticks` (see above), the initial configuration is realized by means of *two* separate styles: one which defines only the tick positions (the `many smithchart ticks*` style) and one which also changes placement- and alignment options:

```
\pgfplotsset{
    dense smithchart ticks*/.style={
        default smithchart xtick/.style={
            xtick={
                0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,1.2,1.4,1.6,1.8,2,3,4,5,10,20%
            },
            minor xtick={%
                0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09,0.11,0.12,0.13,0.14,0.15,0.16,0.17,
                0.18,0.19,0.22,0.24,0.26,0.28,0.32,0.34,0.36,0.38,0.42,0.44,0.46,0.48,%
                0.52,%  This is sub-optimal and will (hopefully) be improved in the future.
                0.55,0.65,0.75,0.85,0.95,%
                % 0.6,0.7,0.8,0.9,%
                1.1,1.3,1.5,1.7,1.9,%
                2.2,2.4,2.6,2.8,3.2,3.4,3.6,3.8,4.5,6,7,8,9,50},
        },
        default smithchart ytick/.style={
            ytick={%
                0,%
                0.1,0.2,...,1,1.2,1.4,1.6,1.8,2,3,4,5,10,20,%
                -0.1,-0.2,...,-1,-1.2,-1.4,-1.6,-1.8,-2,-3,-4,-5,-10,-20%
            },
            minor ytick={%
                0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09,0.11,0.12,0.13,0.14,0.15,0.16,0.17,
                0.18,0.19,0.22,0.24,0.26,0.28,0.32,0.34,0.36,0.38,0.42,0.44,0.46,0.48,%
                0.55,0.65,0.75,0.85,0.95,%
                1.1,1.3,1.5,1.7,1.9,2.2,2.4,2.6,2.8,3.2,3.4,3.6,3.8,4.5,6,7,8,9,50,%
                -0.01,-0.02,-0.03,-0.04,-0.05,-0.06,-0.07,-0.08,-0.09,-0.11,-0.12,-0.13,-0.14,
                -0.15,-0.16,-0.17,-0.18,-0.19,-0.22,-0.24,-0.26,-0.28,-0.32,-0.34,-0.36,-0.38,
                -0.42,-0.44,-0.46,-0.48,-0.55,-0.65,-0.75,-0.85,-0.95,%
                -1.1,-1.3,-1.5,-1.7,-1.9,-2.2,-2.4,-2.6,-2.8,-3.2,-3.4,-3.6,-3.8,-4.5,-6,-7,-8,
                -9,-50%
            },
        },
        default smithchart xytick/.style={
            xgrid each nth passes y={0.2 if < 0.2001,0.5 if < 0.50001,1 if < 1.001,2,4,5,10,20},
            ygrid each nth passes x={0.2 if < 0.2001,0.52 if < 0.52001,1 if < 1.001,2,3,5,10:3,20:3},
        },
    },
    dense smithchart ticks/.style={
        yticklabel in circle,
        dense smithchart ticks*,
        show origin=true,
        every major grid/.style={black!60},
    },
}
```

See the documentation for `few smithchart ticks` for an explanation of the `default smithchart xtick/.style` overhead.

`/pgfplots/default smithchart xtick` (no value)
`/pgfplots/default smithchart ytick` (no value)
`/pgfplots/default smithchart xytick` (no value)

The `default smithchart xtick` style is installed if and only if you do not provide `xtick` manually.

Similarly, the `default smithchart ytick` style is installed if and only if you do not provide `ytick` manually.

Finally, the `default smithchart xytick` style is installed if and only if you provide neither `xtick` nor `ytick`.

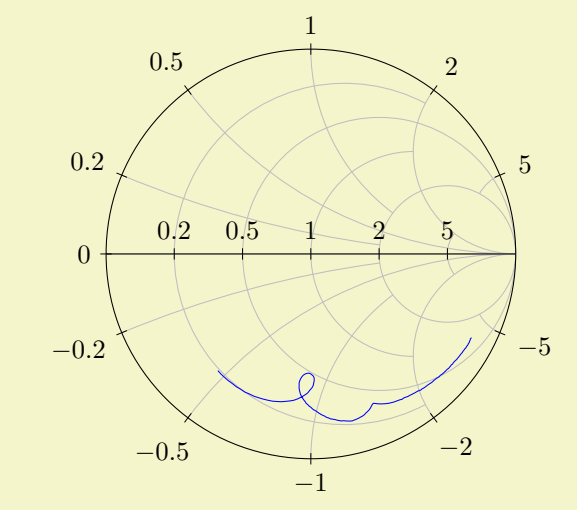
These styles are usually defined in `few smithchart ticks` and its variants, see above.

5.8.3 Working with Prepared Data

`/pgfplots/is smithchart cs=true|false` (initially `false`)

Occasionally, you may already have input data transformed into unit–circle Cartesian coordinate $r(z) = (x, y)$.

You can provide them to PGFPLOTS with the `is smithchart cs` key:



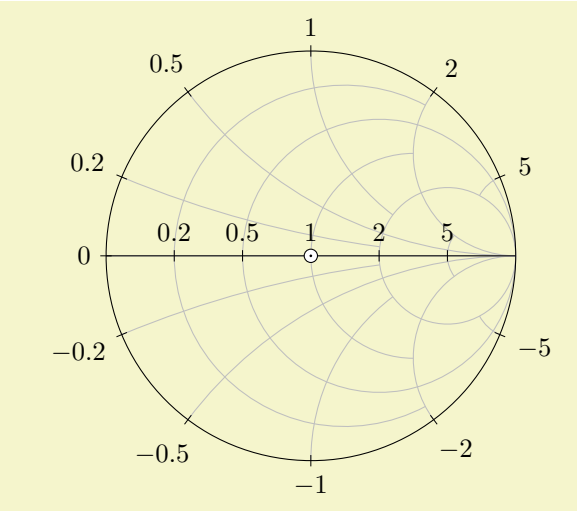

```
\begin{tikzpicture}
    \begin{smithchart}
    %  smithchart_data.dat contains
    %  0.78395 -0.40845
    %  0.78165 -0.41147
    %  0.77934 -0.41466
    %  0.77774 -0.41869
    %  ...
    \addplot[blue,is smithchart cs]
      file {plotdata/smithchart_data.dat};
    \end{smithchart}
\end{tikzpicture}
```

Using `is smithchart cs` tells PGFPLOTS to skip the transformation $r(z)$.

5.8.4 Appearance Control and Styles

`/pgfplots/show origin=true|false` (initially `false`)

Allows to place an extra description at the point $(0, 0)$ to mark the origin.




```
\begin{tikzpicture}
    \begin{smithchart}[show origin]
    \end{smithchart}
\end{tikzpicture}
```

`/pgfplots/show origin code/.code={⟨...⟩}`

Allows to redefine the code to draw the origin marker. The initial configuration is

```
\pgfplotsset{
    show origin code/.code={%
        \path[draw=black,fill=white] (0pt,0pt) circle (2.5pt);
        \path[fill=black] (0pt,0pt) circle (0.5pt);
    }
}
```

`/pgfplots/yticklabel in circle` (style, no value)

This style draws Smith Chart tick labels for imaginary components (the `ytick` arguments) inside of the circle.

It installs transformations to rotate and shift tick labels. See the `many smithchart ticks` style for an example.

The initial configuration for this style is

```
\pgfplotsset{
    yticklabel in circle/.style={
        ytick align=inside,
        yticklabel style={
            rotate=90,
            sloped like y axis={%
                execute for upside down={\tikzset{anchor=north east}},
                % allow upside down,
                reset nontranslations=false},
            anchor=south west,
            % font=\tiny,
        }
    }
}
```

/pgfplots/every smithchart axis (style, no value)

This style is installed for every Smith Chart. It is defined as

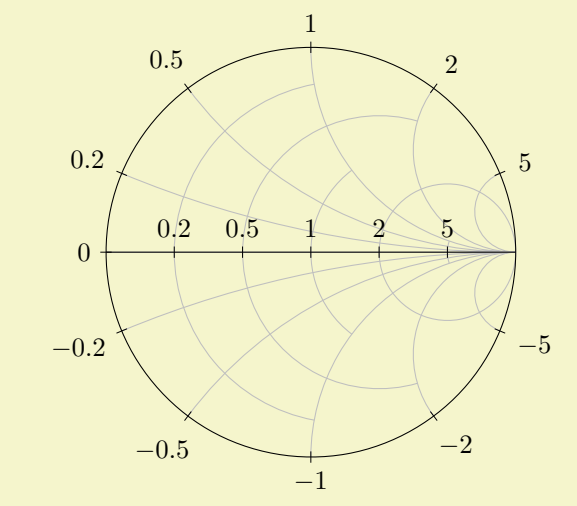
```
\pgfplotsset{
    every smithchart axis/.style={
        grid=both,
        xmin=0,
        scaled ticks=false, %  never draw the \cdot 10^4 labels
        major tick style={draw=black},
        xtick align=center,
        ytick align=center,
    },
}
```

5.8.5 Controlling Arcs and Their Stop Points

This section allows advanced control over Smith Chart arcs (grid lines). The two features `xgrid each nth passes y` and `xgrid stop at y` (and their counterparts for y) allow to draw only partial arcs in order to get a more uniform appearance.

/pgfplots/xgrid each nth passes y={⟨*list of stop entries*⟩} (initially `empty`)

This key constitutes the main idea to draw only partial arcs: you provide a couple of y tick coordinates which constitute "boundaries". Then, only each (say) second x grid line is allowed to pass these boundaries:




```
\begin{tikzpicture}
    \begin{smithchart}[
        xtick={0.2,0.5,1,2,5},
        ytick={
            0,
             0.2, 0.5, 1, 2, 5,
            -0.2,-0.5,-1,-2,-5},
        xgrid each nth passes y={1,2},
    ]
    \end{smithchart}
\end{tikzpicture}
```

The example overwrites the `default smithchart ticks` to define a new layout: now, every `ytick` uses the complete arc, but some of the grid lines for `xtick` stop at $y = 1$ and, if they pass, they may stop at $y = 2$.

The argument ⟨*list of stop entries*⟩ is a comma–separated list of entries. Each entry is, in the simplest case, a y coordinate (it should be a coordinate which appears in the `ytick` list). This simplest case means "only each second x grid line may pass the grid line for this y". The second syntax allows to

provide a natural number, using ⟨*y coord*⟩:⟨*number*⟩. This means to let only each ⟨*number*⟩'s x grid line pass the designated y grid line. The third syntax also allows to write `if <` ⟨*x value*⟩. It means the entry is considered only for x grid lines which are less than ⟨*x value*⟩. To summarize: there are the three possible forms of entries

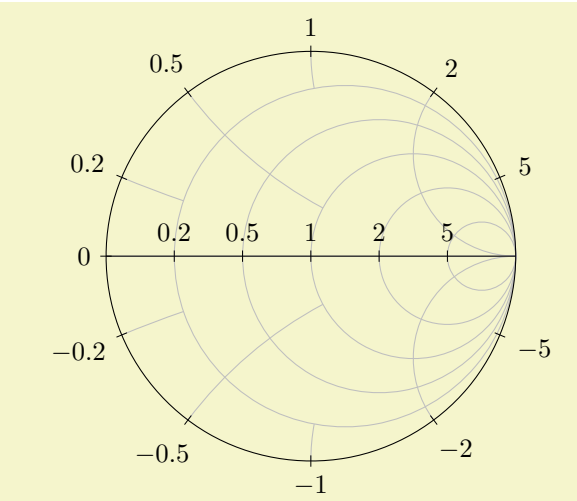
1. single y coordinates, for example `xgrid each nth passes y={1,2}` or
2. the same as above, followed by an integer, for example `xgrid each nth passes y={1:3,2:2}` or
3. an additional restriction clause like `xgrid each nth passes y={0.2 if <0.3}`.
   In this case, the all x grid lines which fulfill $x \leq 0.3$ will be checked if they are allowed to pass $y = 0.2$. All x grid lines with $x > 0.3$ are not affected by the constraint. See the `dense smithchart ticks` style for an application example.

Note that `xgrid each nth passes y` always employs symmetry; you do not need to provide y and $-y$ (if you want to, you may use the `xgrid stop at y` key to overrule the "each nth"-strategy).

In order to check if a given `xtick` argument is the "nth" grid line, PGFPLOTS collects all `xtick` and `minor xtick` arguments into *one* large array and sorts it. Then, it uses the resulting sequence to assign the indices. Consequently, you can freely intermix minor and major ticks; it will still work. The only way to affect the counting is the `xgrid each nth passes y start` key, see below.

`/pgfplots/ygrid each nth passes x={`⟨*list of stop entries*⟩`}` (initially `empty`)

As you may already have guessed, this is the y counterpart of `xgrid each nth passes y`. It restricts the arcs for y grid lines by provided x ticks:




```
\begin{tikzpicture}
    \begin{smithchart}[
        xtick={0.2,0.5,1,2,5},
        ytick={
            0,
             0.2, 0.5, 1, 2, 5,
            -0.2,-0.5,-1,-2,-5},
        ygrid each nth passes x={0.2,1:2},
    ]
    \end{smithchart}
\end{tikzpicture}
```

The syntax is exactly the same as explained for `xgrid each nth passes y`. The only difference is that the `if <` syntax uses absolute values `y` (to maintain symmetry).

Now, we know how to use `xgrid each nth passes y` and the corresponding `ygrid each nth passes x` *separately*. Can we use both keys at the same time? Yes – but it may happen that lines end in white space! PGFPLOTS applies some logic to avoid arcs ending in white space by extending them to the next feasible stopping point. The result of mixing both of these keys is thus corrected automatically.

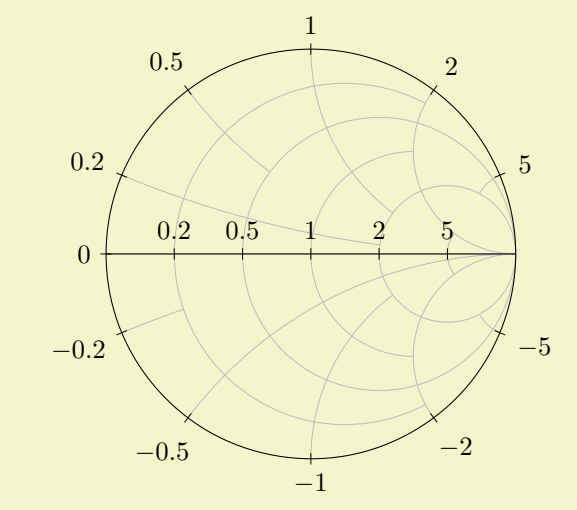
`/pgfplots/xgrid each nth passes y start={`⟨*integer*⟩`}` (initially `0`)
`/pgfplots/ygrid each nth passes x start={`⟨*integer*⟩`}` (initially `0`)

Allows to modify where the "each nth" counting starts. The argument can be considered as a shift. I consider this key to be more or less experimental – in the hope it may be useful. Try it out.

`/pgfplots/xgrid stop at y={`⟨*list*⟩`}` (initially `empty`)
`/pgfplots/ygrid stop at x={`⟨*list*⟩`}` (initially `empty`)

These keys allow to provide *individual* stop points for explicitly chosen tick positions. These explicit stop points have higher precedence over the each nth features described above.

The `ygrid stop at x` key accepts a comma–separated list of entries ⟨*y coord*⟩:⟨*x stop point*⟩:




```
\begin{tikzpicture}
    \begin{smithchart}[
        ygrid stop at x={0.5:0.5,-0.2:0.2}
    ]
    \end{smithchart}
\end{tikzpicture}
```

In this example, the $y = 0.5$ arc stops at the $x = 0.5$ arc whereas the $y = -0.2$ arc stops at $x = 0.2$. The `ygrid stop at x` key allows unsymmetric layouts (different stop points for y and $-y$).

5.9 Ternary Diagrams

```
\usepgfplotslibrary{ternary} % LaTeX and plain TeX
\usepgfplotslibrary[ternary] % ConTeXt
\usetikzlibrary{pgfplots.ternary} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.ternary] % ConTeXt
```

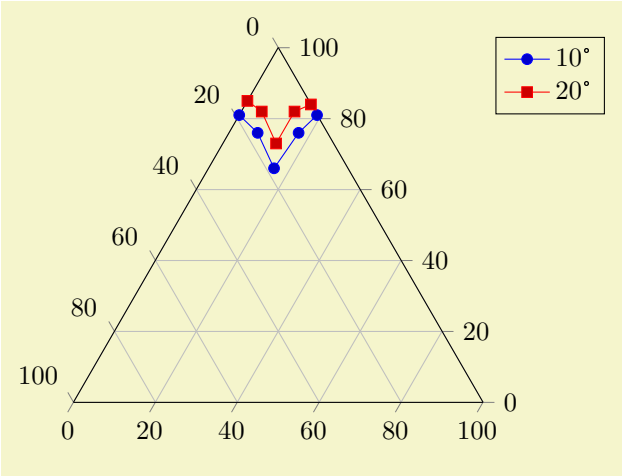
A library to draw ternary diagrams.

A ternary diagram visualizes three–component systems such that the sum of them yields 100%. Ternary diagrams are triangular axes.

5.9.1 Ternary Axis

```
\begin{ternaryaxis}[⟨options⟩]
  ⟨environment contents⟩
\end{ternaryaxis}
```

The axis environment for ternary axes.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}
    \addplot3 coordinates {
        (0.81,    0.19,    0.00)
        (0.76,    0.17,    0.07)
        (0.66,    0.16,    0.16)
        (0.76,    0.07,    0.17)
        (0.81,    0.00,    0.19)
    };

    \addplot3 coordinates {
        (0.85,    0.15,    0.00)
        (0.82,    0.13,    0.05)
        (0.73,    0.14,    0.13)
        (0.82,    0.06,    0.13)
        (0.84,    0.00,    0.16)
    };
    \legend{$10$\textdegree, $20$\textdegree}
\end{ternaryaxis}
\end{tikzpicture}
```

A `ternaryaxis` works with *relative coordinates*: each data point consists of three components x, y, z. Their sum forms a compound entity which has 100% (of whatever). In the standard configuration, we have $x, y, z \in [0, 1]$. The unit interval is not necessary: you can as well choose *absolute data ranges* $x \in [x_{\min}, x_{\max}]$, $y \in [y_{\min}, y_{\max}]$ and $z \in [z_{\min}, z_{\max}]$. The important thing is that the relative values

$$\tilde{x} := \frac{x - x_{\min}}{x_{\max} - x_{\min}},\ \tilde{y} := \frac{y - y_{\min}}{y_{\max} - y_{\min}},\ \tilde{z} := \frac{z - z_{\min}}{z_{\max} - z_{\min}}$$

sum up to 100%, i.e. $\tilde{x} + \tilde{y} + \tilde{z} = 1$. Thus, PGFPLOTS computes $\tilde{x}$, $\tilde{y}$ and $\tilde{z}$ and interpretes them as barycentric (triangular) coordinates.

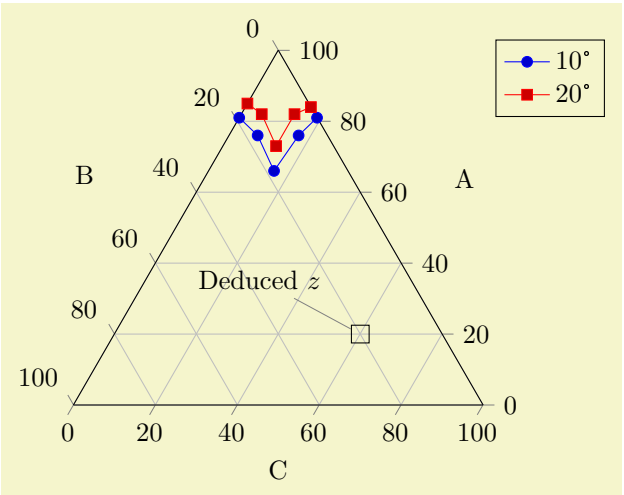
For this to work, it is **crucial to provide** `xmin`, `xmax`, `ymin`, `ymax` and `zmin`, `zmax` precisely! The initial configuration fixes them to the unit interval.

What happens behind the scenes is that a data point (x, y, z) is placed at X, Y determined by

$$\begin{bmatrix} X(x,y,z) \\ Y(x,y,z) \end{bmatrix} = \tilde{x}A + \tilde{y}B + \tilde{z}C = \begin{bmatrix} \frac{1}{2}\tilde{x} + 2\tilde{z} \\ \frac{\sqrt{3}}{2}\tilde{x} \end{bmatrix}$$

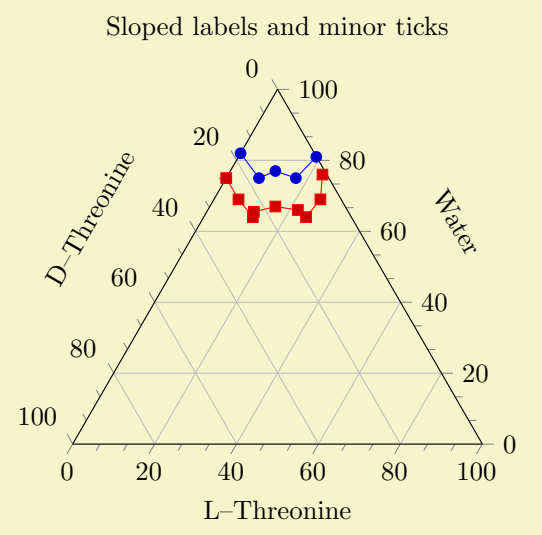
where $A = (1/2, \sqrt{3}/2)$ is top corner of the triangle, $B = (0, 0)$ the lower left and $C = (1, 0)$ the lower right one. The $\tilde{y}$ component is not really necessary due to the linear dependency $\tilde{x} + \tilde{y} + \tilde{z} = 1$.

The input coordinate $(100\%, 0\%, 0\%)$ is mapped to A, the input coordinate $(0\%, 100\%, 0\%)$ to B and $(0\%, 0\%, 100\%)$ to C (Acrobat Reader: click into the axis to verify it).

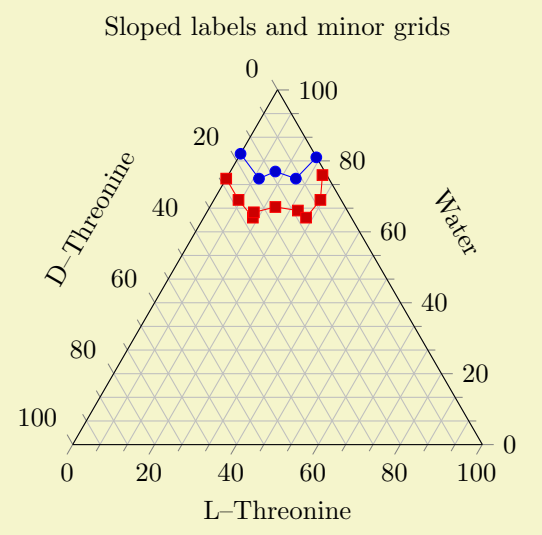


```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[xlabel=A,ylabel=B,zlabel=C]
    \addplot3 coordinates {
        (0.81,  0.19,  0.00)
        (0.76,  0.17,  0.07)
        (0.66,  0.16,  0.16)
        (0.76,  0.07,  0.17)
        (0.81,  0.00,  0.19)
    };

    \addplot3 coordinates {
        (0.85,  0.15,  0.00)
        (0.82,  0.13,  0.05)
        (0.73,  0.14,  0.13)
        (0.82,  0.06,  0.13)
        (0.84,  0.00,  0.16)
    };

    \node[pin=130:Deduced $z$,draw=black] at (axis cs:0.2,0.2) {};

    \legend{$10$\textdegree, $20$\textdegree}
\end{ternaryaxis}
\end{tikzpicture}
```

A `ternaryaxis` can contain zero, one or more `\addplot3` commands, just as a usual `axis`. In case you provide only two–dimensional coordinates (for example using `\addplot` or `axis cs`), the third component is deduced automatically such that components sum to 100%. The `\addplot3` command can use any of the accepted input formats, for example using `coordinates`, `table`, `expression` or whatever – but the input is always interpreted as barycentric coordinates (three components summing up to 100%).

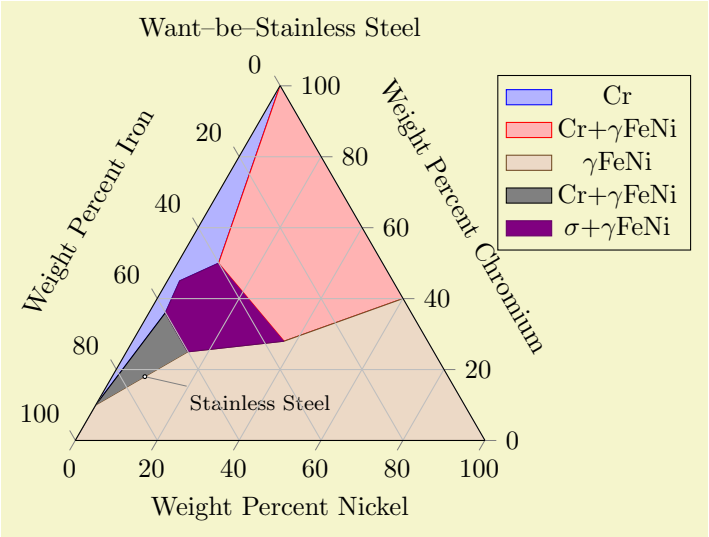



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    title=Sloped labels and minor ticks,
    xlabel=Water,
    ylabel=D--Threonine,
    zlabel=L--Threonine,
    label style={sloped},
    minor tick num=2,
]
    \addplot3 coordinates {
        (0.82,  0.18,  0.00)
        (0.75,  0.17,  0.08)
        (0.77,  0.12,  0.11)
        (0.75,  0.08,  0.17)
        (0.81,  0.00,  0.19)
    };
    \addplot3 coordinates {
        (0.75,  0.25,  0.00)
        (0.69,  0.25,  0.06)
        (0.64,  0.24,  0.12)
        (0.655, 0.23,  0.115)
        (0.67,  0.17,  0.16)
        (0.66,  0.12,  0.22)
        (0.64,  0.11,  0.25)
        (0.69,  0.05,  0.26)
        (0.76,  0.01,  0.23)
    };
\end{ternaryaxis}
\end{tikzpicture}
```



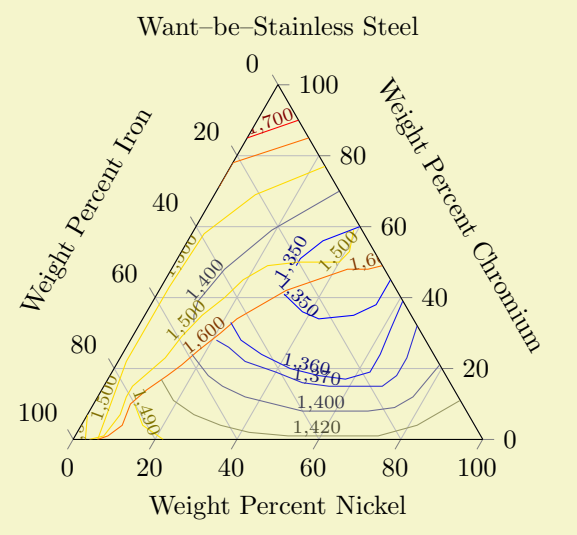

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    title=Sloped labels and minor grids,
    xlabel=Water,
    ylabel=D--Threonine,
    zlabel=L--Threonine,
    label style={sloped},
    minor tick num=2,
    grid=both,
]
    \addplot3 coordinates {
        (0.82,  0.18,  0.00)
        (0.75,  0.17,  0.08)
        (0.77,  0.12,  0.11)
        (0.75,  0.08,  0.17)
        (0.81,  0.00,  0.19)
    };
    \addplot3 coordinates {
        (0.75,  0.25,  0.00)
        (0.69,  0.25,  0.06)
        (0.64,  0.24,  0.12)
        (0.655, 0.23,  0.115)
        (0.67,  0.17,  0.16)
        (0.66,  0.12,  0.22)
        (0.64,  0.11,  0.25)
        (0.69,  0.05,  0.26)
        (0.76,  0.01,  0.23)
    };
\end{ternaryaxis}
\end{tikzpicture}
```

A `ternaryaxis` supports (most of) the PGFPLOTS axis interface, among them the `grid` option, the `xtick={⟨positions⟩}` way to provide ticks, including `extra x ticks` and its variants. Of course, it can also contain any of the `mark`, `color` and `cycle list` options of a normal axis.
The following example is a (crude) copy of an example of
`http://www.sv.vt.edu/classes/MSE2094_NoteBook/96ClassProj/experimental/ternary2.html`
and uses `area style` to change `cycle list` and the legend appearance.



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    title=Want--be--Stainless Steel,
    xlabel=Weight Percent Chromium,
    ylabel=Weight Percent Iron,
    zlabel=Weight Percent Nickel,
    label style=sloped,
    area style,
]
    \addplot3 table {
    A B C
    1 0 0
    0.5 0.4 0.1
    0.45 0.52 0.03
    0.36 0.6 0.04
    0.1 0.9 0
    };
    \addlegendentry{Cr}

    \addplot3 table {
    A B C
    1 0 0
    0.5 0.4 0.1
    0.28 0.35 0.37
    0.4 0 0.6
    };
    \addlegendentry{Cr+$\gamma$FeNi}

    \addplot3 table {
    0.4 0 0.6
    0.28 0.35 0.37
    0.25 0.6 0.15
    0.1 0.9 0
    0 1 0
    0 0 1
    };
    \addlegendentry{$\gamma$FeNi}

    \addplot3 table {
    0.1 0.9 0
    0.36 0.6 0.04
    0.25 0.6 0.15
    };
    \addlegendentry{Cr+$\gamma$FeNi}

    \addplot3 table {
    0.5 0.4 0.1
    0.45 0.52 0.03
    0.36 0.6 0.04
    0.25 0.6 0.15
    0.28 0.35 0.37
    };
    \addlegendentry{$\sigma$+$\gamma$FeNi}

    \node[inner sep=0.5pt,circle,draw,fill=white,pin=-15:\footnotesize Stainless Steel]
      at (axis cs:0.18,0.74,0.08) {};

\end{ternaryaxis}
\end{tikzpicture}
```

Ternary plots can also use `contour prepared` to plot contour lines. The following example is a (crude) copy of an example of
http://www.sv.vt.edu/classes/MSE2094_NoteBook/96ClassProj/experimental/ternary2.html:



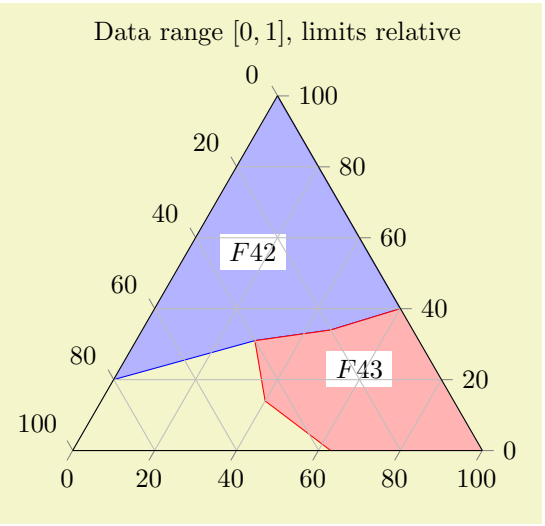

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    title=Want--be--Stainless Steel,
    xlabel=Weight Percent Chromium,
    ylabel=Weight Percent Iron,
    zlabel=Weight Percent Nickel,
    label style=sloped,
]

%  plotdata/pgfplotsternary.example1.dat:
%
%  Chromium Iron Nickel Temperature
%  0.90     0.0     0.10    1700
%  0.85     0.14    0.00    1700
%
%  0.85     0.00    0.15    1600
%  0.78     0.22    0.00    1600
%  0.71     0.29    0.00    1600
%  ....
\addplot3[contour prepared={labels over line},
        point meta=\thisrow{Temperature}]
    table[x=Chromium,y=Iron,z=Nickel]
    {plotdata/pgfplotsternary.example1.dat};
\end{ternaryaxis}
\end{tikzpicture}
```

The `contour prepared={labels over line}` installs the display style `contour/labels over line` and expects precomputed contour lines from the input stream. Here, the input stream is a table, consisting of the three relative components for Chromium, Iron and Nickel – and the `point meta` is set to be the Temperature column. The `contour prepared` style uses the (x, y, z) coordinate to plot the data point and the `point meta` to determine contour labels (the initial configuration of `contour prepared` is to use `point meta=z`). The output thus allows to use both barycentric coordinates (ternary components) *and* contour labels.

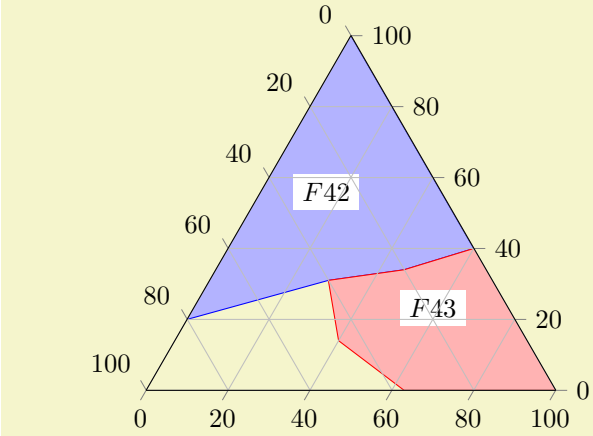
`/pgfplots/ternary limits relative=true|false` (initially `true`)
`/pgfplots/ternary relative limits=true|false` (initially `true`)

Allows to switch tick labels between relative numbers in the range $[0, 100]$ or absolute numbers.

The choice `ternary limits relative=true` accepts data in any input number range, for example $(x, y, z) \in [0, 1]^3$, or $(x, y, z) \in [0, 100]^3$ or in any absolute scala of the form $x_i \in [\underline{x}_i, \overline{x}_i]$ for $x_i \in \{x, y, z\}$ (remember that it is crucial to communicate these limits to PGFPLOTS explicitly using `xmin`, `xmax`, `ymin`, `ymax` and `zmin`, `zmax` such that relative coordinates can be computed, see the description above for details). In every case, relative tick labels are drawn, i.e. tick labels in the range $[0, 100]$.



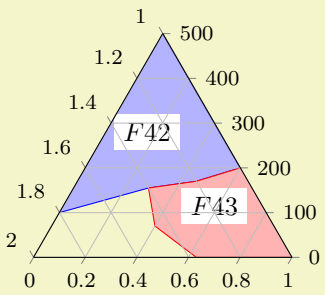

```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    ternary limits relative,
    title={Data range $[0,1]$, limits relative},
    area style]
\addplot3 coordinates {
    (0.2,0.8,0)
    (0.31,0.4,0.29)
    (0.34,0.2,0.46)
    (0.4,0,0.6)
    (1,0,0)
};
\addplot3 coordinates {
    (0.4,0,0.6)
    (0.34,0.2,0.46)
    (0.31,0.4,0.29)
    (0.14,0.46,0.4)
    (0,0.37,0.63)
    (0,0,1)
};
\node[fill=white]
    at (axis cs:0.56,0.28,0.16) {$F 42$};
\node[fill=white]
    at (0.7,0.2) {$F 43$};
\end{ternaryaxis}
\end{tikzpicture}
```



```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    xmax=500,ymin=1,ymax=2,
    ternary limits relative,
    title={Data range $x\in[0,500]$,
        $y\in[1,2]$, $z\in[0,1]$ limits relative},
    area style]
\addplot3 coordinates {
    (100,1.8,0)
    (155,1.4,0.29)
    (170,1.2,0.46)
    (200,1,0.6)
    (500,1,0)
};
\addplot3 coordinates {
    (200,1,0.6)
    (170,1.2,0.46)
    (155,1.4,0.29)
    (70,1.46,0.4)
    (0,1.37,0.63)
    (0,1,1)
};
\node[fill=white]
    at (axis cs:280,1.28,0.16) {$F 42$};
\node[fill=white]
    at (0.7,0.2) {$F 43$};
\end{ternaryaxis}
\end{tikzpicture}
```

The choice `ternary limits relative=false` accepts the same data ranges, but it draws tick labels in the very same data ranges.

Data range $x \in [0, 500]$, $y \in [1, 2]$, $z \in [0, 1]$ limits absolute

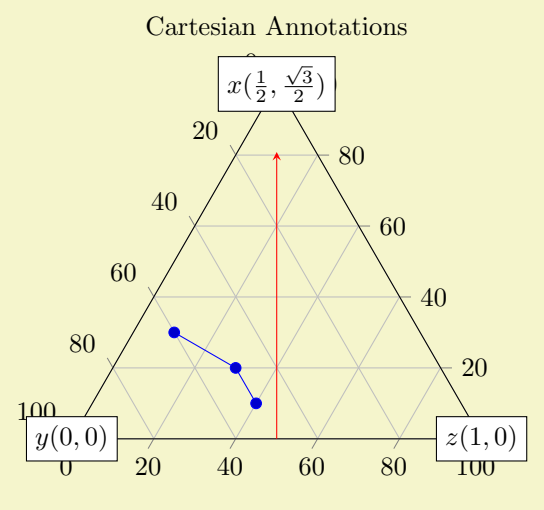


```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    ternary limits relative=false,
    xmax=500,ymin=1,ymax=2,
    title={Data range $x\in[0,500]$,
        $y\in[1,2]$, $z\in[0,1]$ limits absolute},
    footnotesize, %  just for the sake of demonstration...
    area style]
\addplot3 coordinates {
    (100,1.8,0)
    (155,1.4,0.29)
    (170,1.2,0.46)
    (200,1,0.6)
    (500,1,0)
};
\addplot3 coordinates {
    (200,1,0.6)
    (170,1.2,0.46)
    (155,1.4,0.29)
    (70,1.46,0.4)
    (0,1.37,0.63)
    (0,1,1)
};
\node[fill=white]
    at (axis cs:280,1.28,0.16) {$F 42$};
\node[fill=white]
    at (0.7,0.2) {$F 43$};
\end{ternaryaxis}
\end{tikzpicture}
```

Coordinate system `cartesian cs`

A coordinate system which allows Cartesian coordinates. The lower left point has coordinate $(0, 0)$, the lower right point has $(1, 0)$ and the upper point of the triangle is at $(1/2, \sqrt{3}/2)$.

If you use the standard point syntax (x, y) in path commands inside of the axis, you'll get Cartesian coordinates. If you want to use it for axis descriptions (like `xlabel`), you'll have to write `cartesian cs:0,0` explicitly (axis labels have the default coordinate system `axis description cs`).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{ternaryaxis}[
        title=Cartesian Annotations,
        clip=false]

    \addplot3 coordinates {
        (0.1,0.5,0.4)
        (0.2,0.5,0.3)
        (0.3,0.6,0.1)
    };

    \node[fill=white,draw] at (0,0) {$y (0,0)$};
    \node[fill=white,draw] at (1,0) {$z (1,0)$};
    \node[fill=white,draw] at (0.5,{sqrt(3)/2})
        {$x (\frac12,\frac{\sqrt3}{2})$};

    \draw[red,-stealth] (0.5,0) -- (0.5,0.7);
    \end{ternaryaxis}
\end{tikzpicture}
```

`/pgfplots/every ternary axis` (style, no value)

A style which is installed at the beginning of every ternary axis. It is used to adjust some of the PGFPLOTS keys to fit the triangular shape.

The initial configuration is

```
\pgfplotsset{
    every ternary axis/.style={
        tick align=outside,
        grid=major,
        xticklabel style={anchor=west},
        every 3d description/.style={},
        every axis x label/.style={at={(ticklabel cs:0.5)},anchor=near ticklabel},
        every axis y label/.style={at={(ticklabel cs:0.5)},anchor=near ticklabel},
        every axis z label/.style={at={(ticklabel cs:0.5)},anchor=near ticklabel},
        every x tick scale label/.style=
          {at={(xticklabel cs:0.95,5pt)},anchor=near xticklabel,inner sep=0pt},
        every y tick scale label/.style=
          {at={(yticklabel cs:0.95,5pt)},anchor=near yticklabel,inner sep=0pt},
        every z tick scale label/.style=
          {at={(yticklabel cs:0.95,5pt)},anchor=near yticklabel,inner sep=0pt},
        every axis title shift=15pt,
        every axis legend/.style={
            cells={anchor=center},
            inner xsep=3pt,inner ysep=2pt,nodes={inner sep=2pt,text depth=0.15em},
            shape=rectangle,
            fill=white,
            draw=black,
            at={(1.03,1.03)},
            anchor=north west,
        },
        annot/point format 3d/.initial={(\% .2f, \%.2f, \%.2f)},
    },
}
```

5.9.2 Tieline Plots

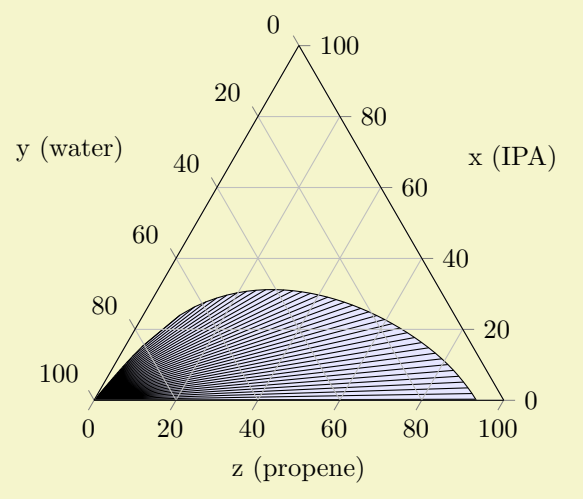
/tikz/tieline={⟨*options with* tieline/ *prefix*⟩}

\addplot+[tieline={⟨*options with* tieline/ *prefix*⟩}]

A plot handler for use in ternary diagrams which plots tie lines and binodal curves.

On input, it accepts *pairs* of coordinates, $A^{(i)} = (A_x^{(i)}, A_y^{(i)}, A_z^{(i)})$ and $B^{(i)} = (B_x^{(1)}, B_y^{(2)}, B_z^{(3)})$, for $i = 1, \dots, N$ (i.e. it requires a total of six coordinates, perhaps plus additional color data).

On output, it connects the pairs, i.e. for every fixed $i = 1, \dots, N$, it connects $A^{(i)}$ — $B^{(i)}$ (the so-called "tie lines"). In addition, it also draws the binodal curve, which is made up by connecting all $A^{(i)}$ and then, in reverse ordering, all $B^{(i)}$: $A^{(1)}$ — $A^{(2)}$ — ⋯ — $A^{(N)}$ — $B^{(N)}$ — $B^{(N-1)}$ — ⋯ — $B^{(1)}$:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    xlabel=x (IPA),
    ylabel=y (water),
    zlabel=z (propene),
    axis on top,
]
%  plotdata/ternary_data.txt is a table of the form
% A_propene A_water A_IPA  B_propene B_water B_IPA
%  0.0009   0.9990  0      0.9333    0.0667  0
%  0.0009   0.9988  0.0002 0.9303    0.0665  0.0032
%  0.0011   0.9975  0.0013 0.9135    0.0673  0.0191
%  0.0013   0.9962  0.0024 0.8956    0.0693  0.0351
% ...
    \addplot3[tieline,fill=blue!10]
    table [x=A_IPA,y=A_water,z=A_propene]
        {plotdata/ternary_data.txt};
\end{ternaryaxis}
\end{tikzpicture}
```

We see that each input line has six columns, and each six columns are taken into account (this is different from other plot handlers!). The six columns make up the three components of the A and B points, respectively. In the example above, we used explicit column names and provided A_x using `x=A_IPA`, A_y using `y=A_water` and A_z using `z=A_propene`. Note that these keys are the common input method for `\addplot table`; they are nothing special (that means we could also use `x index` instead). The three columns for B can be provided manually (see below), or deduced automatically: in our case, the value

for B_x has been found in the third column after `x=A_IPA` (which is `B_IPA`); the value for B_y has been found in the third column after `y=A_water` and B_z is made up from the third column after `z=A_propene`. In other words, the B value is searched (by default) by adding 3 to the column index of the respective A coordinate.

You do not need to provide *any* column names; in this case, the first three columns make up A (in the order of appearance) and the following three make up B.

The only supported input type for `tieline` plots is table input. It is optimized to use `\addplot3 table` (as described above). To use the two–dimensional variant `\addplot table`, you need to tell PGFPLOTS explicitly which columns make up A_x, A_y, B_x, B_y; the z coordinates are deduced automatically such that the result sums to 100%.

`/pgfplots/table/tie end x={`⟨*colname*⟩`}` (initially `empty`)
`/pgfplots/table/tie end y={`⟨*colname*⟩`}` (initially `empty`)
`/pgfplots/table/tie end z={`⟨*colname*⟩`}` (initially `empty`)
`/pgfplots/table/tie end x index={`⟨*col index*⟩`}` (initially `empty`)
`/pgfplots/table/tie end y index={`⟨*col index*⟩`}` (initially `empty`)
`/pgfplots/table/tie end z index={`⟨*col index*⟩`}` (initially `empty`)

These keys can be used to provide column names or column indices for B_x, B_y and B_z, respectively. They can be provided like

```
\addplot3[tieline] table[tie end y=B_water] ....
```

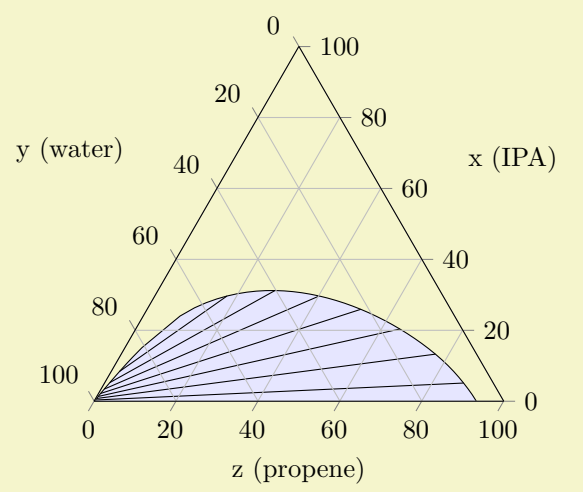
Note that the `tie end x` keys are *only* available if the `tieline` option has been used before.

The values for A are provided with `table/x`, `table/x index` and its variants as for any other plot type.

The `tieline` plot handler accepts several options to customize the appearance. You can provide them as argument after `tieline`, using `tieline={`⟨*options*⟩`}`. In this case, the `tieline/` prefix can be omitted. The keys are described in the following:

`/pgfplots/tieline/each nth tie={`⟨*number*⟩`}` (initially `empty`)

Allows to draw only each nth tie line, even though the binodal curve uses all provided coordinates:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    xlabel=x (IPA),
    ylabel=y (water),
    zlabel=z (propene),
    axis on top,
]
%  plotdata/ternary_data.txt is a table of the form
% A_propene A_water A_IPA  B_propene B_water B_IPA
%  0.0009   0.9990  0      0.9333    0.0667  0
%  0.0009   0.9988  0.0002 0.9303    0.0665  0.0032
%  0.0011   0.9975  0.0013 0.9135    0.0673  0.0191
%  0.0013   0.9962  0.0024 0.8956    0.0693  0.0351
% ...
    \addplot3[
        tieline={each nth tie=5},
        fill=blue!10,
    ]
    table [x=A_IPA,y=A_water,z=A_propene]
        {plotdata/ternary_data.txt};
\end{ternaryaxis}
\end{tikzpicture}
```

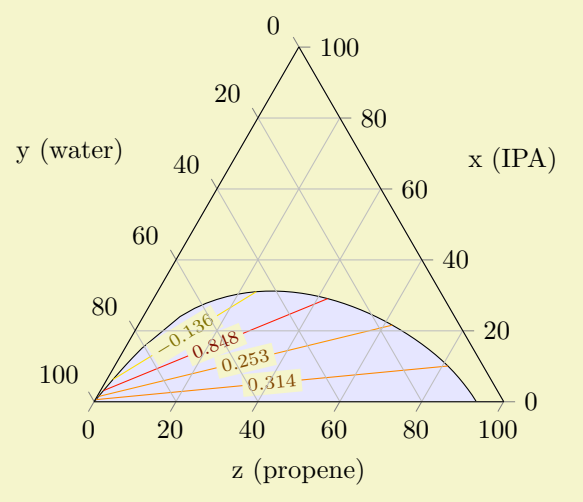
Note that plot `mark`s (if any) are drawn on every input position, use the `mark repeat` option to change that.

`/pgfplots/tieline/tieline style={`⟨*options*⟩`}`

Appends ⟨*options*⟩ to the style `tieline/every tieline`.

Useful ⟨*options*⟩ are, for example, other plot handlers to adjust the appearance of tie lines. Suppose that you have additional color data for every tie line (which might have been provided as further

input column). In our case, we provide random color data using `point meta=rand`, and visualize the single tielines as with `contour prepared`:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{ternaryaxis}[
    xlabel=x (IPA),
    ylabel=y (water),
    zlabel=z (propene),
    axis on top,
]
%  plotdata/ternary_data.txt is a table of the form
% A_propene A_water A_IPA  B_propene B_water B_IPA
%  0.0009   0.9990  0      0.9333    0.0667  0
%  0.0009   0.9988  0.0002 0.9303    0.0665  0.0032
%  0.0011   0.9975  0.0013 0.9135    0.0673  0.0191
%  0.0013   0.9962  0.0024 0.8956    0.0693  0.0351
% ...
    \addplot3[
        point meta=rand,
        tieline={
            each nth tie=8,
            tieline style={contour prepared}
        },
        fill=blue!10,
    ]
    table [x=A_IPA,y=A_water,z=A_propene]
        {plotdata/ternary_data.txt};
\end{ternaryaxis}
\end{tikzpicture}
```

The effect here is that contour labels and line colors are chosen for every tie line, where the actual color is determined using `point meta` and `colormap`. Other choices for plot handlers in `tieline style` might be the `mesh`.

`/pgfplots/tieline/curve style={`⟨*options*⟩`}`

Appends ⟨*options*⟩ to the style `tieline/every curve`.

The `curve style` allows to customize the plot handler for the curve. A possible choice might be `curve style={smooth}` or a separate fill/draw color.

5.10 Units in Labels

by Nick Papior Andersen

```
\usepgfplotslibrary{units} % LaTeX and plain TeX
\usepgfplotslibrary[units] % ConTeXt
\usetikzlibrary{pgfplots.units} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.units] % ConTeXt
```

A library which allows to use automatic typesetting of units in labels. The library utilizes different keys to typeset the final output in a consistent way. Calling one of the commands automatically sets the key '`use units=true`' so one does not have to worry about this.

PGFPLOTS has the capability of supporting units. This provides quick customization of the plot as well as the addition of units in labels.

Loading the library automatically enables the typesetting of units in labels. Currently it only supports predefined SI units but a per-user customization is also implemented such that it can be used in any way you like.

First the key which enables you to switch on/off the unit system.

`/pgfplots/use units={`⟨*boolean*⟩`}` (initially `true`)

This key simply enables PGFPLOTS to use what is described next. This key will be set to true if you load the library. You can use this to temporarily determine whether the unit library should be used in plots.

`/pgfplots/x unit={`⟨*unit*⟩`}` (initially `empty`)

/pgfplots/y unit={⟨*unit*⟩} (initially empty)
/pgfplots/z unit={⟨*unit*⟩} (initially empty)

These keys set the unit in their respective axis. In SI units you could for instance set the x unit in Newton as x unit=N.

/pgfplots/x unit prefix={⟨*prefix*⟩} (initially empty)
/pgfplots/y unit prefix={⟨*prefix*⟩} (initially empty)
/pgfplots/z unit prefix={⟨*prefix*⟩} (initially empty)

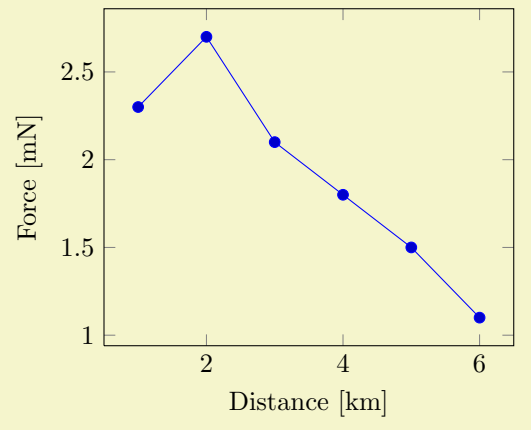
These keys set the prefix of the unit. If a value on the y axis is in kilo you would set the y unit prefix=k. Prefix will be typeset in front of the unit.

This command will not intervene with the basis of the axis system. I.e. a prefix as just mentioned will not divide every y axis number by 1000. In order to do this, see key ⟨*axis*⟩ SI prefix, see Section 5.10.1.

Notice that if the ⟨*axis*⟩ unit isn't set the entire unit will not be typeset.

Remarks: Remember that all typesetting of labels occur within math mode (i.e. within $$ delimiters). Therefore one can use \frac and other mathematics commands.

Often one just has to utilize the above mentioned keys. It is the basis of the unit typesetting system provided by PGFPLOTS.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{axis}[use units,
    x unit=m,x unit prefix=k,
    y unit=N,y unit prefix=m,
    xlabel=Distance,ylabel=Force]
    \addplot coordinates {
        (1,2.3)
        (2,2.7)
        (3,2.1)
        (4,1.8)
        (5,1.5)
        (6,1.1)
    };
  \end{axis}
\end{tikzpicture}
```

Below is an example of what would be obtained according to the styles

```
%  x label becomes ``Temperature [T]'', y label becomes ``Nothing''
\pgfplotsset{use units,x unit=T,xlabel=Temperature,ylabel=Nothing}
%  x label becomes ``Temperature'', y label becomes ``Nothing''
\pgfplotsset{use units,x unit prefix=m,xlabel=Temperature,ylabel=Nothing}
```

Notice the second example. Only setting the prefix will not activate the unit typesetting. Therefore one should ensure to use the x unit key if the typesetting of the labels should be done.

For typesetting the units one can also change the appearance. For instance one might not like the square brackets which surround the unit. These can luckily be changed using the below keys.

/pgfplots/unit marking pre={⟨*pre*⟩} (initially \left[)
/pgfplots/unit marking post={⟨*post*⟩} (initially \right])
/pgfplots/unit markings=parenthesis|square brackets|slash space (initially square brackets)

These keys set the surroundings of the unit. The initial yields $\left[\frac{1}{2}\right]$ such that you can typeset fractions in units. Be aware that you can only obtain large fractions if you use \dfrac. These can easily be set using the option key unit markings where the options typesets as the following

```
\pgfplotsset{x unit=T,unit markings=parenthesis} %  x unit becomes `` \left(T\right)''
\pgfplotsset{x unit=T,unit markings=square brackets} %  x unit becomes `` \left[T\right]''
\pgfplotsset{x unit=T,unit markings=slash space} %  x unit becomes `` / T''
```

Notice that all typesetting of units first inserts a space and then the unit marking pre code.

Of course you can just manually set each of them with the unit marking pre and unit marking post keys. Just remember that they are typeset within a $$.

One will typically typeset the unit with a specific font. To do so an option of changing the typesetting command is supplied.

/pgfplots/unit code/.code 2 args={⟨...⟩}

This can be utilized to great extent. By default, units are typeset as `\mathrm{`⟨*unit prefix*⟩⟨*unit*⟩`}`. But if one for instance wishes to utilize the package `siunitx`, which has great capabilities in typesetting both units, numbers and angles, one can just set the key as

```
\pgfplotsset{unit code/.code 2 args={\si{#1#2}}}
```

which would yield the unit as `\si{`⟨*unit prefix*⟩⟨*unit*⟩`}`.

The first argument is typeset as ⟨*unit prefix*⟩ and the second argument is ⟨*unit*⟩.

The most important thing is that the command needs exactly two arguments. So if you would like a command that typesets the prefix in bold face and the unit in normal roman font you should call

```
\pgfplotsset{unit code/.code 2 args={\mathbf{#1}\mathrm{#2}}
```

5.10.1 Preset SI prefixes

To support the SI system a number of preset keys are defined. This should yield a more intuitive way of supplying the prefix as well as add some more functionality. For instance it provides an easy scaling mechanism.

/pgfplots/x SI prefix=yocto|...|milli|centi|deci|deca|hecto|kilo|...|yotta (initially `none`)
/pgfplots/y SI prefix=yocto|...|milli|centi|deci|deca|hecto|kilo|...|yotta (initially `none`)
/pgfplots/z SI prefix=yocto|...|milli|centi|deci|deca|hecto|kilo|...|yotta (initially `none`)
/pgfplots/change x base=true|false (initially `false`)
/pgfplots/change y base=true|false (initially `false`)
/pgfplots/change z base=true|false (initially `false`)

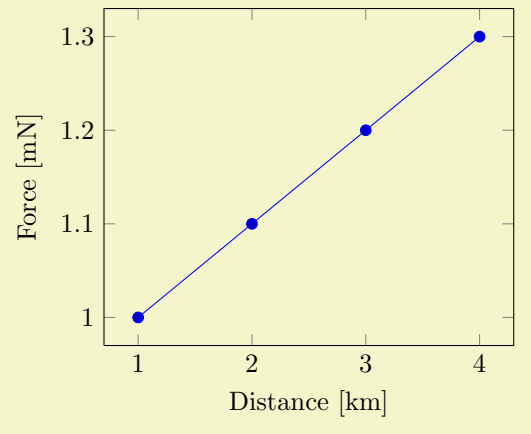
These keys sets the prefix of the unit. The allowed prefixes are:

Prefix	Power
yocto	−24
zepto	−21
atto	−18
femto	−15
pico	−12
nano	−9
micro	−6
milli	−3
centi	−2
deci	−1

Prefix	Power
deca	1
hecto	2
kilo	3
mega	6
giga	9
tera	12
peta	15
exa	18
zetta	21
yotta	24

As well as resetting the base of the axis if the key `change` ⟨*axis*⟩ `base=true`. Just **remember** to set the `change` ⟨*axis*⟩ `base` before using the ⟨*axis*⟩ `SI prefix` key.

See the utilization as in the example below.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
  \begin{axis}[change x base,
    x SI prefix=kilo,x unit=m,
    y SI prefix=milli,y unit=N,
    xlabel=Distance,ylabel=Force]
    \addplot coordinates {
        (1000,1)
        (2000,1.1)
        (3000,1.2)
        (4000,1.3)
    };
  \end{axis}
\end{tikzpicture}
```

Notice that the `x axis` has changed base without displaying the $\cdot 10^3$. This is done by using the key `change x base`. Even though you have used the key `y SI prefix=milli` the base isn't changed on the `y axis`. Try adding `change y base` just after `change x base` and see the result!

The above keys are the easy implementation of the base change. Below is a further customization of the base change. It makes it easy to implement a prefix with a custom base change.

`/pgfplots/axis base prefix`=axis {⟨*axis*⟩} base {⟨*base*⟩} prefix {⟨*prefix*⟩} (initially `empty`)

One can utilize this key to customize further of the base and setting the prefix.

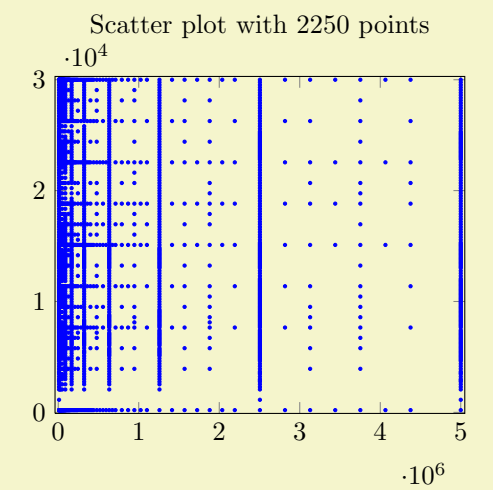
```
\pgfplotsset{change x base,axis base prefix={axis x base -3 prefix k}}
\pgfplotsset{change x base,x SI prefix=kilo}
```

The above two commands are thus equivalent. Remember that the base should operate in opposite of prefix!

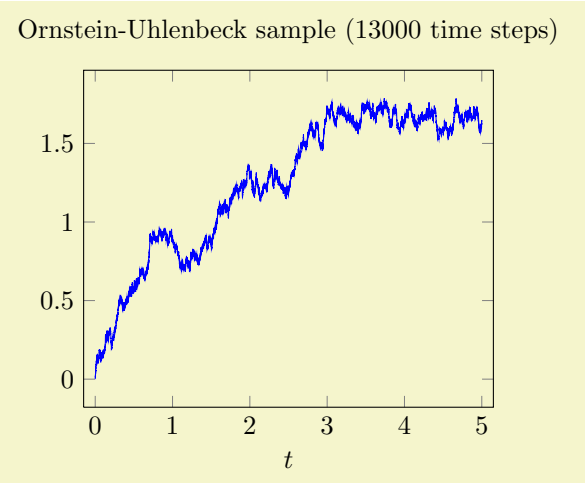
6 Memory and Speed considerations

6.1 Memory Limits of TEX

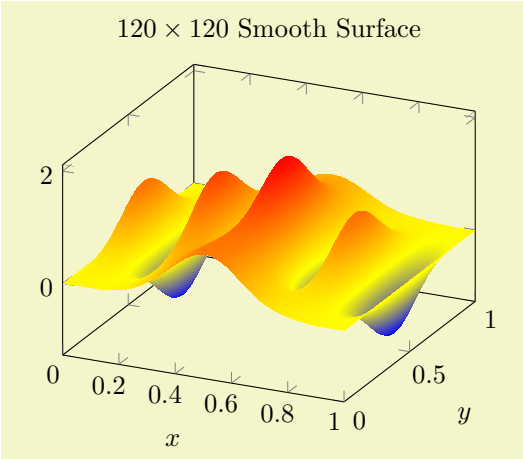
PGFPLOTS can typeset plots with several thousand points if memory limits of TEX are configured properly. Its runtime is roughly proportional to the number of input points[70].




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    enlargelimits=0.01,
    title style={yshift=5pt},
    title=Scatter plot with $2250$ points]

\addplot[blue,
    mark=*,only marks,mark options={scale=0.3}]
    file[skip first]
    {plotdata/pgfplots_scatterdata3.dat};

\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    enlarge x limits=0.03,
    title=Ornstein-Uhlenbeck sample
        ($13000$ time steps),
    xlabel=$t$]

\addplot[blue] file {plotdata/ou.dat};
\end{axis}
\end{tikzpicture}
```




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    title=$120 \times 120$ Smooth Surface,
    xlabel=$x$,
    ylabel=$y$]
\addplot3[surf,samples=120,shader=interp,domain=0:1]
    {sin(deg(8*pi*x))* exp(-20*(y-0.5)^2)
    + exp(-(x-0.5)^2*30
        - (y-0.25)^2 - (x-0.5)*(y-0.25))};
\end{axis}
\end{tikzpicture}
```

PGFPLOTS relies completely on TEX to do all typesetting. It uses the front-end-layer and basic layer of PGF to perform all drawing operations. For complicated plots, this may take some time, and you may want

[70] In fact, the runtime is pseudo–linear: starting with about 100,000 points, it will become quadratic. This limitation applies to the path length of PGF paths as well. Furthermore, the linear runtime is not possible yet for stacked plots.

to read Section 7 for how to write single figures to external graphics files. Externalization is the best way to reduce typesetting time.

However, for large scale plots with a lot of points, limitations of TEX's capacities are reached easily.

6.2 Memory Limitations

The default settings of most TEX-distributions are quite restrictive, so it may be necessary to adjust them.

Usually, the log–file or the final error message contains a summary about the used resources, giving a hint which parameter needs to be increased.

6.2.1 MikTEX

For MikTEX, memory limits can be increased in two ways. The first is to use command line switches:

```
pdflatex
    --stack-size=n --save-size=n
    --main-memory=n --extra-mem-top=n --extra-mem-bot=n
    --pool-size=n --max-strings=n
```

Experiment with these settings if MikTEX runs out of memory. Usually, one doesn't invoke `pdflatex` manually: there is a development aid which does all the invocations, so this one needs to be adjusted.

Sometimes it might be better to adjust the MikTEX configuration file permanently, for example to avoid reconfiguring the TEX development program. This can be realized using the command

```
initexmf --edit-config-file=pdflatex
```

which can be typed either on a command prompt in Windows or using Start ≫ Execute. As a result, an editor will be opened with the correct config file. A sample config file could be

```
main_memory=90000000
save_size=80000
```

or any of the config file entries which are listed below can be entered. Thanks to "LeSpocky" for his documentation in

http://blog.antiblau.de/2009/04/21/speicherlimits-von-miktex-erhoehen.

6.2.2 TEXLive or similar installations

For Unix installations, one needs to adjust config files. This can be done as follows:

1. Locate `texmf.cnf` on your system. On my Ubuntu installation, it is in

   `/usr/share/texmf/web2c/texmf.cnf`.

2. Either change `texmf.cnf` directly, or copy it to some convenient place. If you copy it, here is how to proceed:

   - keep only the changed entries in your local copy to reduce conflicts. TEX will always read *all* config files found in its search path.
   - Adjust the search path to find your local copy. This can be done using the environment variable `TEXMFCNF`. Assuming your local copy is in `~/texmf/mytexcnf/texmf.cnf`, you can write

     ```
     export TEXMFCNF=~/texmf/mytexcnf:
     ```

     to search first in your directory, then in all other system directories.

3. You should change the entries

   ```
   main_memory = n
   extra_mem_top = n
   extra_mem_bot = n
   max_strings = n
   param_size = n
   save_size = n
   stack_size = n
   ```

   The log–file usually contains information about the parameter which needs to be enlarged.

An example of this config file thing is shown below. It changes memory limits.

1. Create the file `~/texmf/mytexcnf/texmf.cnf` (and possibly the paths as well).

```
%  newly created file ~/texmf/mytexcnf/texmf.cnf:
%  If you want to change some of these sizes only for a certain TeX
%  variant, the usual dot notation works, e.g.,
%  main_memory.hugetex = 20000000
main_memory = 230000000 %  words of inimemory available; also applies to inimf&mp
extra_mem_top = 10000000     %  extra high memory for chars, tokens, etc.
extra_mem_bot = 10000000     %  extra low memory for boxes, glue, breakpoints, etc.
save_size = 150000    %  for saving values outside current group
stack_size = 150000    %  simultaneous input sources

%  Max number of characters in all strings, including all error messages,
%  help texts, font names, control sequences.  These values apply to TeX and MP.
% pool_size = 1250000
%  Minimum pool space after TeX/MP's own strings; must be at least
%  25000 less than pool_size, but doesn't need to be nearly that large.
% string_vacancies = 90000
%  Maximum number of strings.
% max_strings = 100000
%  min pool space left after loading .fmt
% pool_free = 47500
```

2. Run `texhash` such that TEX updates its `~/texmf/ls-R` database.

3. Create the environment variable `TEXMFCNF` and assign the value '`~/texmf/mytexcnf:`' (including the trailing '`:`'!). For my linux system, this can be done using by adding

```
export TEXMFCNF=~/texmf/mytexcnf:
```

   to `~/.bashrc`.

Unfortunately, TEX does not allow arbitrary memory limits, there is an upper bound hard coded in the executables.

6.3 Reducing Typesetting Time

PGFPLOTS does a lot of computations ranging from abstract coordinate computations to low level `.pdf` drawing commands (realized by PGF). For complex plots, this may take a considerable time – especially for 3D plots.

One possibility to reduce typesetting time is to tell PGF to generate single, temporary `.pdf` (or `.eps`) documents for a subset (or all) graphics in one run and re-use these temporary images in successive runs. For PGFPLOTS, this is the most effective way to reduce typesetting time. It can be accomplished using the `external` library described in Section 7.1.

7 Import/Export From Other Formats

This section contains information of how to single pictures into separate PDF graphics files (or EPS graphics files). Furthermore, it explains a matlab (®) script which allows to convert from matlab to PGFPLOTS.

7.1 Export to PDF/EPS

It is possible to export images to single PDF-documents using routines of PGF and/or Ti*k*Z.

7.1.1 Using the Automatic Externalization Framework of Ti*k*Z

```
\usepgfplotslibrary{external} % LaTeX and plain TeX
\usepgfplotslibrary[external] % ConTeXt
\usetikzlibrary{pgfplots.external} % LaTeX and plain TeX
\usetikzlibrary[pgfplots.external] % ConTeXt
```

The external library offers a convenient method to export every single tikzpicture into a separate .pdf (or .eps). Later runs of LaTeX will simply include these graphics, thereby reducing typesetting time considerably.

The library can also be used to submit documents to authors who do not even have PGFPLOTS or Ti*k*Z installed.

Technical foreword: The external library has been written by Christian Feuersänger (author of PGFPLOTS). It has been contributed to Ti*k*Z as general purpose library, so the reference documentation along with all tweaks can be found in [5, Section "Externalization Library"]. The command \usepgfplotslibrary{external} is actually just a wrapper which loads \usetikzlibrary{external} or, if this library does not yet exist because the installed PGF has at most version 2.00, it will load a copy which is shipped with PGFPLOTS.

The external library has been designed such that *no changes* to the document as such are necessary. The idea is as follows:

1. Every \begin{tikzpicture} ... \end{tikzpicture} gets a file name. The file name can be assigned manually with \tikzsetnextfilename{⟨*output file name*⟩} or automatically, in which case ⟨*tex file name*⟩-figure⟨*number*⟩ is used with an increasing ⟨*number*⟩.
2. The library writes the resulting images using system calls of the form pdflatex --jobname ⟨*output file name*⟩ automatically, using the write18 system call of TeX. It is the same framework which can be used to call gnuplot.

The only steps which are necessary is to use

\usepgfplotslibrary{external}

\tikzexternalize

somewhere in your document's preamble (see below for system-dependent configuration options). No further modification to the document is necessary. Suppose we have a file called test.tex:

```
\documentclass{article}

\usepackage{pgfplots}

\usepgfplotslibrary{external}
\tikzexternalize%  activate externalization!

\begin{document}
    \begin{figure}
        \begin{tikzpicture}
        \begin{axis}
            \addplot {x^2};
        \end{axis}
        \end{tikzpicture}
    \caption{Our first external graphics example}
    \end{figure}

    \begin{figure}
        \begin{tikzpicture}
        \begin{axis}
            \addplot {x^3};
        \end{axis}
        \end{tikzpicture}
    \caption{A second graphics}
    \end{figure}
\end{document}
```

To enable the system calls, we type

```
pdflatex -shell-escape test
```

and LaTeX will now generate the required graphics files `test-figure0.pdf` and `test-figure1.pdf` automatically. Any further call to `pdflatex` will simply use `\includegraphics` and the tikzpictures as such are no longer considered (you need a different command line switch for MikTeX, see the shell escape option).

If a figure shall be remade, one can simply delete all or selected graphics files and regenerate them. Alternatively, one can use the command `\tikzset{external/force remake}` somewhere in the document to remake every following picture automatically.

There are three ways to modify the file names of externalized figures:

- Changing the overall file name using a prefix,
- Changing the file name for a single figure using `\tikzsetnextfilename`,
- Changing the file name for a restricted set of figures using figure name.

`/tikz/external/prefix={`⟨*file name prefix*⟩`}` (initially `empty`)

A shortcut for `\tikzsetexternalprefix{`⟨*file name prefix*⟩`}`, see below.

`\tikzsetexternalprefix{`⟨*file name prefix*⟩`}`

Assigns a common prefix used by all file names. For example,

```
\tikzsetexternalprefix{figures/}
```

will prepend `figures/` to every external graphics file name.

`\tikzsetnextfilename{`⟨*file name*⟩`}`

Sets the file name for the *next* TikZ picture or `\tikz` short command. It will *only* be used for the next picture.

Pictures for which no explicit file name has been set will get automatically generated file names.

Please note that prefix will still be prepended to ⟨*file name*⟩.

```
\documentclass{article}
%  main document, called main.tex
\usepackage{tikz}

\usepgfplotslibrary{external}
\tikzexternalize[prefix=figures/]%  activate with a name prefix

\begin{document}

\tikzsetnextfilename{firstplot}
\begin{tikzpicture} %  will be written to 'figures/firstplot.pdf'
\begin{axis}
    \addplot {x};
\end{axis}
\end{tikzpicture}

\begin{tikzpicture} %  will be written to 'figures/main-figure0.pdf'
   \draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

```
pdflatex -shell-escape main
```

/tikz/external/figure name={⟨*name*⟩}

Same as \tikzsetfigurename{⟨*name*⟩}.

\tikzsetfigurename{⟨*name*⟩}

Changes the names of *all* following figures. It is possible to change figure name during the document using \tikzset{external/figure name={⟨*name*⟩}}. A unique counter[71] will be used for each different ⟨*name*⟩, and each counter will start at 0.

The value of prefix will be applied after figure name has been evaluated.

```
\documentclass{article}
%  main document, called main.tex
\usepackage{tikz}

\usepgfplotslibrary{external}
\tikzexternalize%  activate externalization!

\begin{document}

%  will be written to 'main-figure0.pdf'
\begin{tikzpicture}
\begin{semilogyaxis}
    \addplot {exp(x)};
\end{semilogyaxis}
\end{tikzpicture}

{
  \tikzset{external/figure name={subset_}}
  A simple image is \tikz \fill (0,0) circle(5pt);. %  will be written to 'subset_0.pdf'

  \begin{tikzpicture} %  will be written to 'subset_1.pdf'
     \begin{axis}
         \addplot {x^2};
    \end{axis}
  \end{tikzpicture}
}%  here, the old file name will be restored:

\begin{tikzpicture} %  will be written to 'main-figure1.pdf'
   \begin{axis}
           \addplot[domain=1e-3:100] {1/x};
    \end{axis}
\end{tikzpicture}
\end{document}
```

The scope of figure name ends with the next closing brace (as all values set by \tikzset do).

[71] These counters are stored into different *macros*. In other words: no TEX register will be needed.

Remark: Use `\tikzset{external/figure name/.add={⟨prefix⟩}{⟨suffix⟩}}` to prepend a ⟨*prefix*⟩ and append a ⟨*suffix*⟩ to the actual value of `figure name`. Might be useful for something like

```
\tikzset{external/figure name=main}

%  uses main_0.pdf, main_1.pdf, ...

\section{The first section}
{\tikzset{external/figure name/.add={}{_firstsection}}
    ...
    %  uses main_firstsection_0.pdf, main_firstsection_1.pdf, ...
}

\section{The second section}
{\tikzset{external/figure name/.add={}{secondsection_}}
    ...
    %  uses main_secondsection_0.pdf, main_secondsection_1.pdf, ...
    \subsection{Second subsection}
    {\tikzset{external/figure name/.add={}{sub_}}
        ...
        %  uses main_secondsection_sub_0.pdf, main_secondsection_sub_1.pdf, ...
    }
    %  uses main_secondsection_2.pdf, main_secondsection_3.pdf, ...
}
```

`\tikzappendtofigurename{`⟨*suffix*⟩`}`

Appends ⟨*suffix*⟩ to the actual value of `figure name`.

It is a shortcut for `\tikzset{external/figure name/.add={}{`⟨*suffix*⟩`}}` (a shortcut which is also supported if Ti*k*Z is not installed, see below).

Configuration option for eps output or MikTEX: Since the `external` lib works by means of system calls, it has to be modified to fit the local system. This is necessary for MikTEX since it uses a different option to enable these system calls. It is also necessary for EPS output since this involves a different set of utilities.

Note that the *most important part* is to enable system calls. This is typically done by typesetting your document with `pdflatex -shell-escape` or `pdflatex -enable-write18` (MikTEX). These options *need to be configured in your TEX editor*. Besides this step, one may want to configure the system call:

`/tikz/external/system call={`⟨*template*⟩`}`

A template string used to generate system calls. Inside of ⟨*template*⟩, the macro `\image` can be used as placeholder for the image which is about to be generated while `\texsource` contains the main file name (in truth, it contains `\input{`⟨*main file name*⟩`}`, but that doesn't matter).

The default is

```
\tikzset{external/system call={pdflatex \tikzexternalcheckshellescape -halt-on-error
    -interaction=batchmode -jobname "\image" "\texsource"}
```

where `\tikzexternalcheckshellescape` inserts the value of the configuration key `shell escape` if and only if the current document has been typeset with `-shell-escape`[72].

For `eps` output, you can (and need to) use

```
\tikzset{external/system call={latex \tikzexternalcheckshellescape -halt-on-error
    -interaction=batchmode -jobname "\image" "\texsource" &&
    dvips -o "\image".ps "\image".dvi}}
```

The argument ⟨*template*⟩ will be expanded using `\edef`, so any control sequences will be expanded. During this evaluation, '`\\`' will result in a normal backslash, '`\`'. Furthermore, double quotes '`"`', single quotes '`'`', semicolons and dashes '`-`' will be made to normal characters if any package uses them as macros. This ensures compatibility with the `german` package, for example.

[72] Note that this is always true for the default configuration. This security consideration applies mainly for `mode=list and make` which will also work *without* shell escapes.

/tikz/external/shell escape={⟨*command-line arg*⟩} (initially -shell-escape)

Contains the command line option for latex which enables the \write18 feature.

For TEX-Live, this is -shell-escape. For MikTEX, you should use \tikzexternalize[shell escape=-enable-write18].

Support for Labels and References In External Files The external library comes with extra support for \label and \ref (and other commands which usually store information in the .aux file) inside of external files.

There are, however, some points which need your attention when you try to use

a) \ref to something in the main document inside of an externalized graphics or

b) \label in the externalized graphics which is referenced in the main document.

For point a), a \ref inside of an externalized graphics works *only* if you issue the required system call *manually* or by make. The initial configuration mode=convert with system call does *not* support \ref. But you can copy–paste the system call generated by mode=convert with system call and issue it manually. The reason is that \ref information is stored in the main .aux file – but this auxiliary file is not completely written when mode=convert with system call is invoked (there is a race condition). Note that \pageref is not supported (sorry). Thus: if you have \ref inside of external graphics, consider using mode=list and make or copy–paste the system call for the image(s) and issue it manually.

Point b) is realized automatically by the external library. In detail, a \label inside of an externalized graphics causes the external library to generate separate auxiliary files for every external image. These files are called ⟨*imagename*⟩.dpth. The extension .dpth indicates that the file also contains the image's depth (the baseline key of Ti*k*Z). Furthermore, anything which would have been written to an .aux file will be redirected to the .dpth file – but only things which occur inside of the externalized tikzpicture environment. When the main document loads the image, it will copy the .dpth file into the main .aux file. Then, successive compilations of the main document contain the external \label information. In other words, a \label in an external graphics needs the following work flow:

1. The external graphics needs to be generated together with its .dpth (usually automatically by Ti*k*Z).
2. The main document includes the external graphics and copies the .dpth content into its main .aux file.
3. The main document needs to be translated once again to re-read its .aux file[73].

There is just a special case if a \label/\ref drawn as a tikzpicture. This is, for example, the case for the legend \ref images or for the \pgfplotslegendfromname feature. In such cases, you need to proceed as for case a) since mode=convert with system call can't handle that stuff on its own.

In other words: a \label in an external document works automatically, just translate the main document often enough. A \ref might need manual adjustments as described for case a) above.

Operation Modes

/tikz/external/mode=convert with system call|list and make|... (initially convert with system call)

This allows to change the default operation mode. There are a handful of choices possible, all of them are described in detail in [5, section "Externalization Library"]. The most useful ones are probably the initial configuration convert with system call and the specialized choice list and make.

The choice list and make configures the library to check if there are already external graphics and uses them. If there are no graphics, the library will *skip* the figure. However, it will also generate a makefile to generate the graphics, and a list of all required graphics files.

It is not required to use make: the library expects you to generate the images somehow and it doesn't care about the "how". Using make -f ⟨*name-of-tex-file*⟩.makefile -j 2 allows parallel execution

[73] Note that it is not possible to activate the content of an auxiliary file after \begin{document} in LATEX.

which might, indeed, be an option. Furthermore, the makefile also supports file dependencies: if one of your data tables has been updated, the external graphics will be remade automatically. PGFPLOTS tells the external library about any file dependencies (input files and tables).

The two modes have the following characteristics:

1. `convert with system call` is automatic and does everything on–the–fly. However, it *can't* work with `\ref` and/or `\label` information in external pictures.
2. `list and make` requires either manual (by issuing the system calls manually) or semi–automatic conversion (using the generated ⟨*main*⟩`.makefile`), and multiple runs of `pdflatex`. The generated Makefile can be processed in parallel. Furthermore, `list and make` provides *full support* for `\ref` and `\label`: any `\label` defined inside of an externalized graphics is still available for the main document.
   If you have legends with `legend to name` or `\label`/`\ref`, you need to generate the graphics defining the `\label` (or `legend to name`), then run `pdflatex` twice on the main document. Afterwards, you can externalize the legend graphics.

The complete reference documentation and remaining options are documented in [5, "Externalization Library"]. This reference also contains information about

- how to use `\tikzset{external/force remake}` and `\tikzset{external/remake next}` to remake selected figures,
- how to disable the externalization partially with `\tikzset{external/export=false}` or completely with `\tikzexternaldisable`,
- how to optimize the speed of the conversion process using `\tikzset{external/optimize command away=\myExpensiveMacro}`,
- how to add further remake-dependencies with `\tikzpicturedependsonfile{`⟨*name*⟩`}` and/or `\tikzexternalfiledependsonfile{`⟨*external file*⟩`}{`⟨*name*⟩`}`,
- examples how to enable `png` export,
- how to typeset such a document without PGF installed or
- how to provide work-arounds with `.pdf` images and bounding box restrictions.

Using the Library Without PGF or PGFPLOTS Installed There is a small replacement package `tikzexternal.sty` which can be used once every figure has been exported. The idea is to uncomment `\usepackage{tikz}` and `\usepackage{pgfplots}` and write `\usepackage{tikzexternal}` instead:

```
%  \usepackage{tikz}
%  \usepackage{pgfplots}
\usepackage{tikzexternal}
\tikzexternalize%  activate externalization

\begin{document}
\begin{tikzpicture}
    ...
\end{tikzpicture}
...
\end{document}
```

You do not need PGF, Ti*k*Z or PGFPLOTS installed. What you need is `tikzexternal.sty` and all generated figures (consisting of the image files, '`.pdf`' and the '`.dpth`' files containing information of the `baseline` option). The file `tikzexternal.sty` is shipped with PGF in the directory

```
latex/pgf/utilities/tikzexternal.sty
```

and a copy is shipped with PGFPLOTS in

```
tex/generic/pgfplots/oldpgfcompatib/pgfplotsoldpgfsupp_tikzexternal.sty
```

Just copy the file into your directory and rename it to `tikzexternal.sty`.

Attention: The small replacement package doesn't support key–value interfaces. Thus, it is necessary to use `\tikzsetexternalprefix` instead of the `prefix` option and `\tikzsetfigurename` instead of the `figure name` option since `\tikzset` is not available in such a context. Also, you may want to define a dummy–macro `\pgfplotsset` if you have used `\pgfplotsset`.

7.1.2 Using the Externalization Framework of PGF "By Hand"

Another way to export TEX-pictures to single graphics files is to use the externalization framework of PGF, which requires more work but works more generally than the `external` library. The basic idea is to encapsulate the desired parts with

`\beginpgfgraphicnamed{`⟨*output file name*⟩`}`
⟨*picture contents*⟩
`\endpgfgraphicnamed`.

Furthermore, one needs to tell PGF the name of the main document using

`\pgfrealjobname{`⟨*the real job's name*⟩`}`

in the preamble. This enables two different modes:

1. The first is the normal typesetting mode. LATEX checks whether a file named ⟨*output file name*⟩ with one of the accepted file extensions exists – if that is the case, the graphics file is included with `\pgfimage` and the ⟨*picture contents*⟩ is skipped. If no such file exists, the ⟨*picture contents*⟩ is typeset normally. This mode is applied if `\jobname` equals ⟨*the real job's name*⟩.

2. The second mode applies if `\jobname` equals ⟨*output file name*⟩, it initiates the "conversion mode" which is used to write the graphics file ⟨*output file name*⟩. In this case, *only* ⟨*picture contents*⟩ is written to `\jobname`, the complete rest of the LATEX is processed as normal, but it is silently discarded.

   This mode needs to be started manually with `pdflatex --jobname` ⟨*output file name*⟩ for every externalized graphics file.

A complete example may look as follows.

```
\documentclass{article}

\usepackage{pgfplots}

\pgfrealjobname{test}

\begin{document}
    \begin{figure}
        \beginpgfgraphicnamed{testfigure}
        \begin{tikzpicture}
        \begin{axis}
            \addplot {x^2};
        \end{axis}
        \end{tikzpicture}
        \endpgfgraphicnamed
    \caption{Our first external graphics example}
    \end{figure}

    \begin{figure}
        \beginpgfgraphicnamed{testfigure2}
        \begin{tikzpicture}
        \begin{axis}
            \addplot {x^3};
        \end{axis}
        \end{tikzpicture}
        \endpgfgraphicnamed
    \caption{A second graphics}
    \end{figure}
\end{document}
```

The file is named `test.tex`, and it is processed (for example) with

```
pdflatex test
```

Now, we type

```
pdflatex --jobname testfigure test
pdflatex --jobname testfigure2 test
```

to enter conversion mode. These last calls will *only* write the contents of our named graphics environments, one for ⟨*testfigure*⟩ and one for ⟨*testfigure2*⟩ into the respective output files `testfigure.pdf` and `testfigure2.pdf`.

In summary, one needs `\pgfrealjobname` and calls `pdflatex --jobname` ⟨*graphics file*⟩ for every externalized graphics environment. Please note that it is absolutely necessary to use the syntax above, *not* `\begin{pgfgraphicnamed}`.

These steps are explained in much more detail in Section "Externalizing Graphics" of [5].

Attention: Do not forget a correct `\pgfrealjobname` statement! If it is missing, externalization simply won't work. If it is wrong, any call to LaTeX will produce empty output files.

It should be noted that this approach of image externalization is not limited to Ti*k*Z picture environments. In fact, it collects everything between the begin and end statements into the external file. It is implicitly assumed that the encapsulated stuff is one box, but you can also encapsulate complete paragraphs using something like the LaTeX minipage (or a `\vbox` which is not as powerful but does not affect the remaining document that much).

`/pgf/images/aux in dpth=true|false` (initially `false`)

If this boolean is set to `true`, any `\label` information generated inside of the external image is stored into the already mentioned `.dpth` file. The main document can thus reference label information of externalized parts of the document (although you may need to run `latex` several times).

Label support is provided for `\ref`, and probably `\cite`. The `\pageref` command is only partially supported.

Using the Library Without PGF Installed Simply uncomment the packages `\usepackage{tikz}` and `\usepackage{pgfplots}` and use

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{%
    \begingroup
    \setbox1=\hbox{\includegraphics{#1}}%
    \openin1=#1.dpth
    \ifeof1 \box1
    \else
        \read1 to\pgfincludeexternalgraphicsdp \closein1
        \dimen0=\pgfincludeexternalgraphicsdp\relax
        \hbox{\lower\dimen0 \box1 }%
    \fi
    \endgroup
}
```

instead. This will include the generated graphics files (and it will respect the `baseline` information stored in `.dpth` files). Consequently, you won't need PGF or PGFPLOTS installed. See Section "Externalizing Graphics" of [5] for details.

7.2 Importing From Matlab

7.2.1 Importing Mesh Data From Matlab To PGFPlots

While it is easy to write Matlab vectors to files (using `save P.dat data -ASCII`), it is more involved to export mesh data.

The main problem is to communicate the mesh structure to PGFPLOTS.

Here is an example how to realize this task: in Matlab, we have mesh data `X`, `Y` and `Z` which are matrices of the same size. For example, suppose we have

```
[X,Y] = meshgrid( linspace(-1,1,5), linspace(4,5,10) );
Z = X + Y;
surf(X,Y,Z)
```

as data. Then, we can generate an $N \times 3$ table containing all single elements in column–wise ordering with

```
data = [ X(:) Y(:) Z(:) ]
save P.dat data -ASCII
```

where the second command stores the $N \times 3$ table into `P.dat`. Finally, we can use

```
\addplot3[surf,mesh/rows=10,mesh/ordering=colwise,shader=interp] file {P.dat};
```

in PGFPLOTS to read this data. We need to provide either the number of rows (10 here) or the number of columns – and the ordering (which is `colwise` for Matlab matrices).

An alternative which is faster in PGFPLOTS would be to transpose the matrices in Matlab and tell PGFPLOTS they are in `rowwise` ordering. So, the last step becomes

```
XX=X'; YY=Y'; ZZ=Z';
data = [ XX(:) YY(:) ZZ(:) ]
save P.dat data -ASCII
```

with PGFPLOTS command

`\addplot3[surf,mesh/cols=10,mesh/ordering=rowwise,shader=interp] file {P.dat};`.

7.2.2 matlab2pgfplots.m

This is a Matlab (®) script which attempts to convert a Matlab figure to PGFPLOTS. It requires Matlab version 7.4 (or higher).

Attention: This script is largely outdated and supports only a very small subset of PGFPLOTS. You may want to look at `matlab2tikz`, a conversion script of Nico Schlömer available at

http://www.mathworks.com/matlabcentral/fileexchange/22022-matlab2tikz

which also uses PGFPLOTS for the LaTeX conversion.

The idea of `matlab2pgfplots.m` is to

- use a complete matlab figure as input,
- acquire axis labels, axis scaling (log or normal) and legend entries,
- acquire all plot coordinates

and write an equivalent `.pgf` file which typesets the plot with PGFPLOTS.

The intention is *not* to simulate matlab. It is a first step for a conversion. Type

```
> help matlab2pgfplots
```

on your matlab prompt for more information about its features and its limitations.

This script is experimental.

7.2.3 matlab2pgfplots.sh

A `bash`-script which simply starts matlab and runs

```
f=hgload( 'somefigure.fig' );
matlab2pgfplots( 'outputfile.pgf', 'fig', f );
```

See matlab2pgfplots.m above.

7.2.4 Importing Colormaps From Matlab

Occasionally, you may want to reuse your matlab `colormap` in PGFPLOTS. Here is a small Matlab script which converts it to PGFPLOTS:

```
C = colormap;  %  gets data of the current colormap.
%  C = colormap(jet) % gets data of "jet"
eachnth = 1;
I = 1:eachnth:size(C,1); %  this is nonsense for eachnth=1 -- but perhaps you don't want each color.
CC = C(I,:);
TeXstring = [ ...
 sprintf('\\pgfplotsset{\n\tcolormap={matlab}{\n') ...
 sprintf('\t\trgb=(% f,%f,%f)\n',CC') ...
 sprintf('\t}\n}\n') ]
```

7.3 SVG Output

It is possible to write every single Ti*k*Z picture into a scalable vector graphics (`.svg`) file. This has nothing to do with PGFPLOTS, it is a separate driver of PGF. Please refer to [5, Section "Producing HTML / SVG Output"].

7.4 Generate pgfplots Graphics Within Python

Mario Orne DÍAZ ANADÓN contributed a small python script `pgfplots.py` which provides a simple interface to generate PGFPLOTS figures from within python. It can be found in the PGFPLOTS installation directory, in `pgfplots/scripts/pgfplots/pgfplots.py`; documentation can be found in the file.

8 Utilities and Basic Level Commands

This section documents commands which provide access to more basic elements of PGFPLOTS. Most of them are closely related to the basic level of PGF, especially various point commands which are specific to an axis. Some of them are general purpose utilities like loops.

However, most elements in this section are only interesting for advanced users – and perhaps only for special cases.

8.1 Utility Commands

`\foreach`⟨*variables*⟩ `in` ⟨*list*⟩ `{`⟨*commands*⟩`}`

A powerful loop command provided by Ti*k*Z, see [5, Section Utilities].

Iterating 1. Iterating 2. Iterating 3. Iterating 4.

```
\foreach \x in {1,2,...,4} {Iterating \x. }%
```

A PGFPLOTS related example could be

```
\foreach \i in {1,2,...,10} {\addplot table {datafile\i}; }%
```

`\pgfplotsforeachungrouped`⟨*variable*⟩ `in` ⟨*list*⟩ `{`⟨*command*⟩`}`

A specialised variant of `\foreach` which can do two things: it does not introduce extra groups while executing ⟨*command*⟩ and it allows to invoke the math parser for (simple!) $\langle x_0\rangle,\langle x_1\rangle,\ldots,\langle x_n\rangle$ expressions.

Iterating 1. Iterating 2. Iterating 3. Iterating 4. All collected = , 1, 2, 3, 4.

```
\def\allcollected{}
\pgfplotsforeachungrouped \x in {1,2,...,4} {Iterating \x. \edef\allcollected{\allcollected, \x}}%
All collected = \allcollected.
```

A more useful example might be to work with tables. The following example is taken from PGFPLOTSTABLE:

```
\pgfplotsforeachungrouped \i in {1,2,...,10} {%
    \pgfplotstablevertcat{\output}{datafile\i} %  appends ‘datafile\i’ -> ‘\output’
}%
%  since it was ungrouped, \output is still defined (would not work
%  with \foreach)
```

Remark: The special syntax ⟨*list*⟩=$\langle x_0\rangle,\langle x_1\rangle,\ldots,\langle x_n\rangle$, i.e. with two leading elements, followed by dots and a final element, invokes the math parser for the loop. Thus, it allows larger number ranges than any other syntax if `/pgf/fpu` is active. In all other cases, `\pgfplotsforeachungrouped` invokes `\foreach` and provides the results without TEX groups.

Keep in mind that inside of an axis environment, all loop constructions (including custom loops, `\foreach` and `\pgfplotsforeachungrouped`) need to be handled with care: loop arguments can only be used in places where they are immediately evaluated; but PGFPLOTS postpones the evaluation of many macros. For example, to loop over something and to generate axis descriptions of the form `\node at (axis cs:\i,0.5)...`, the loop macro `\i` will be evaluated in `\end{axis}` – but at that time, the loop is over and its value is lost. The correct way to handle such an application is to *expand* the loop variable *explicitly*. For example:

```
\pgfplotsforeachungrouped \i/\j in {
    1 / a,
    2 / b,
    3 / c
}{
    \edef\temp{\noexpand\node at (axis cs: \i,0.5) {\j};}
    %  \show\temp % lets TeX show you what \temp contains
    \temp
}
```

The example generates three loop iterations: `\i=1`, `\j=a`; then `\i=2`, `j=b`; then `\i=3`, `\j=c`. Inside of the loop body, it expands them and assigns the result to a macro using an “expanded definition”, `\edef`.

The result no longer contains either `\i` or `\j` (since these have been expanded). Then, it invokes the resulting macro. Details about the TEX command `\edef` and expansion control can be found in the document TeX-programming-notes.pdf which comes with PGFPLOTS.

`\pgfplotsinvokeforeach{`⟨*list*⟩`} {`⟨*command*⟩`}`

A variant of `\pgfplotsforeachungrouped` (and such also of `\foreach`) which replaces any occurrence of `#1` inside of ⟨*command*⟩ once for every element in ⟨*list*⟩. Thus, it actually assumes that `{`⟨*command*⟩`}` is like a `\newcommand` body.

In other words, ⟨*command*⟩ is invoked for every element of ⟨*list*⟩. The actual element of ⟨*list*⟩ is available as `#1`.

As `\pgfplotsforeachungrouped`, this command does *not* introduce extra scopes (i.e. it is ungrouped as well).

The difference to `\foreach \x in` ⟨*list*⟩`{`⟨*command*⟩`}` is subtle: the `\x` would *not* be expanded whereas `#1` is.

Invoke them: [a] [b] [c] [d]

```
\pgfkeys{
  otherstyle a/.code={[a]},
  otherstyle b/.code={[b]},
  otherstyle c/.code={[c]},
  otherstyle d/.code={[d]}}
\pgfplotsinvokeforeach{a,b,c,d}
    {\pgfkeys{key #1/.style={otherstyle #1}}}
Invoke them:
\pgfkeys{key a} \pgfkeys{key b}
\pgfkeys{key c} \pgfkeys{key d}
```

The counter example would use a macro (here `\x`) as loop argument:

Invoke them: [d] [d] [d] [d]

```
\pgfkeys{
  otherstyle a/.code={[a]},
  otherstyle b/.code={[b]},
  otherstyle c/.code={[c]},
  otherstyle d/.code={[d]}}
\pgfplotsforeachungrouped \x in {a,b,c,d}
    {\pgfkeys{key \x/.style={otherstyle \x}}}
Invoke them:
\pgfkeys{key a} \pgfkeys{key b}
\pgfkeys{key c} \pgfkeys{key d}
```

Restrictions: you can't nest this command yet (since it does not introduce protection by scopes).

`\pgfmathparse{`⟨*expression*⟩`}`

Invokes the PGF math parser for ⟨*expression*⟩ and defines `\pgfmathresult` to be the result.

The result is '42.0'.

```
\pgfmathparse{1+41}

The result is '\pgfmathresult'.
```

The math engine in PGF typically uses TEX's internal arithmetics. That means: it is well suited for numbers in the range $[-16384, 16384]$ and has a precision of 5 digits.

The number range is typically too small for plotting applications. PGFPLOTS improves the number range by means of `\pgfkeys{/pgf/fpu}\pgfmathparse{1+41}` to activate the "floating point unit" (fpu) and to apply all following operations in floating point.

In PGFPLOTS, the key `/pgfplots/use fpu` is typically on, which means that any coordinate arithmetics are carried out with the `fpu`. However, all PGF related drawing operations still use the standard math engine.

In case you ever need to process numbers of extended precision, you may want to use

The result is '$1 \cdot 10^6$'.

```
\pgfkeys{/pgf/fpu}%
\pgfmathparse{1000*1000}

The result is '\pgfmathprintnumber{\pgfmathresult}'.
```

Note that results of the fpu are typically not in human-readable format, so \pgfmathprintnumber is the preferred way to typeset such numbers.

Please refer to [5] for more details.

\pgfplotstableread{⟨*file*⟩}

Please refer to the manual of PgfplotsTable, pgfplotstable.pdf, which is part of the pgfplots-bundle.

\pgfplotstabletypeset{⟨*macro*⟩}

Please refer to the manual of PgfplotsTable, pgfplotstable.pdf, which is part of the pgfplots-bundle.

\pgfplotsiffileexists{⟨*filename*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

Invokes ⟨*true code*⟩ if ⟨*filename*⟩ exists and ⟨*false code*⟩ if not. Can be used in looping macros, for example to plot every data file until there are no more of them.

\pgfplotsutilifstringequal{⟨*first*⟩}{⟨*second*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

A simple "strcmp" tool which invokes ⟨*true code*⟩ if ⟨*first*⟩ =⟨*second*⟩ and ⟨*false code*⟩ otherwise. This does not expand macros.

\pgfkeys
\pgfeov
\pgfkeysvalueof
\pgfkeysgetvalue

These commands are part of the Ti*k*Z way of specifying options, its sub-package pgfkeys. The \pgfplotsset command is actually nothing but a wrapper around \pgfkeys.

A short introduction into \pgfkeys can be found in [7] whereas the complete reference is, of course, the Ti*k*Z manual [5].

The key \pgfkeysvalueof{⟨*key name*⟩} expands to the value of a key; \pgfkeysgetvalue{⟨*key name*⟩}{⟨*macro*⟩} stores the value of ⟨*key name*⟩ into ⟨*macro*⟩. The \pgfeov macro is used to delimit arguments for code keys in \pgfkeys, please refer to the references mentioned above.

8.2 Commands Inside Of pgfplots Axes

\autoplotspeclist

This command should no longer be used, although it will be kept as technical implementation detail. Please use the 'cycle list' option, Section 4.6.7.

\logten

Expands to the constant log(10). Useful for logplots because $\log(10^i) = i \log(10)$. This command is only available inside of a Ti*k*Z-picture.

\pgfmathprintnumber{⟨*number*⟩}

Generates pretty–printed output[74] for ⟨*number*⟩. This method is used for every tick label.

The number is printed using the current number printing options, see the manual of PgfplotsTable which comes with this package for the different number styles, rounding precision and rounding methods.

\numplots

Inside of any of the axis environments, associated style, option or command, \numplots expands to the total number of plots.

\numplotsofactualtype

Like \numplots, this macro returns the total number of plots which have the same plot handler. Thus, if you have sharp plot active, it returns the number of all sharp plots. If you have ybar active, it returns the number of ybar plots and so on.

[74] This method was previously \prettyprintnumber. Its functionality has been included into pgf and the old command is now deprecated.

\plotnum

Inside of \addplot or any associated style, option or command, \plotnum expands to the current plot's number, starting with 0.

\plotnumofactualtype

Like \plotnum, but it returns the number among all plots of the same type. The number of all such plots is available using \numplotsofactualtype.

\coordindex

Inside of an \addplot command, this macro expands to the number of the actual coordinate (starting with 0).

It is useful together with x filter or y filter to (de)select coordinates.

8.3 Path Operations

\path
\draw
\fill
\node
\matrix

These commands are Ti*k*Z drawing commands all of which are documented in [5]. They are used to draw or fill paths, generate text nodes or aligned text matrices. They are equivalent to \path[draw], \path[fill], \path[node], \path[matrix], respectively.

\path ... --⟨*coordinate*⟩ ... ;

A Ti*k*Z path operation which connects the current point (the last one before --) and ⟨*coordinate*⟩ with a straight line.

\path ... |-⟨*coordinate*⟩ ... ;

A Ti*k*Z path operation which connects the current point and ⟨*coordinate*⟩ with *two* straight lines: first vertical, then horizontal.

\path ... -|⟨*coordinate*⟩ ... ;

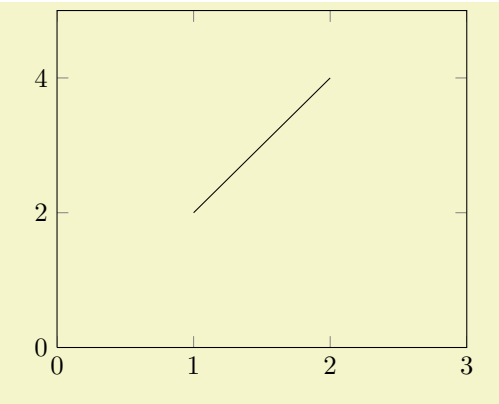
A Ti*k*Z path operation which connects the current point and ⟨*coordinate*⟩ with *two* straight lines: first horizontal, then vertical.

/tikz/xshift={⟨*dimension*⟩}
/tikz/yshift={⟨*dimension*⟩}

These Ti*k*Z keys allow to shift something by ⟨*dimension*⟩ which is any TEX size (or expression).

\pgfplotsextra{⟨*low-level path commands*⟩}

A command to execute ⟨*low-level path commands*⟩ in a PGFPLOTS axis. Since any drawing commands inside of an axis need to be postponed until the axis is complete and the scaling has been initialised, it is not possible to simply draw any paths. Instead, it is necessary to draw them as soon as the axis is finished. This is done automatically for every Ti*k*Z path – and it is also done manually if you write \pgfplotsextra{⟨*commands*⟩}.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[xmin=0,xmax=3,ymin=0,ymax=5]
    \pgfplotsextra{%
        \pgfpathmoveto{\pgfplotspointaxisxy{1}{2}}%
        \pgfpathlineto{\pgfplotspointaxisxy{2}{4}}%
        \pgfusepath{stroke}%
    }
    \end{axis}
\end{tikzpicture}
```

The example above initializes an axis and executes the basic level path commands as soon as the axis is ready. The execution of multiple \path, \addplot and \pgfplotsextra commands is in the same sequence as they occur in the environment[75].

8.4 Specifying Basic Coordinates

\pgfplotspointaxisxy{⟨*x coordinate*⟩}{⟨*y coordinate*⟩}
\pgfplotspointaxisxyz{⟨*x coordinate*⟩}{⟨*y coordinate*⟩}{⟨*z coordinate*⟩}

Point commands like \pgfpointxy which take logical, absolute coordinates and return a low–level point. Every transformation from user transformations to logarithms is applied.

Since the transformations are initialized after the axis is complete, this command needs to be postponed (see \pgfplotsextra).

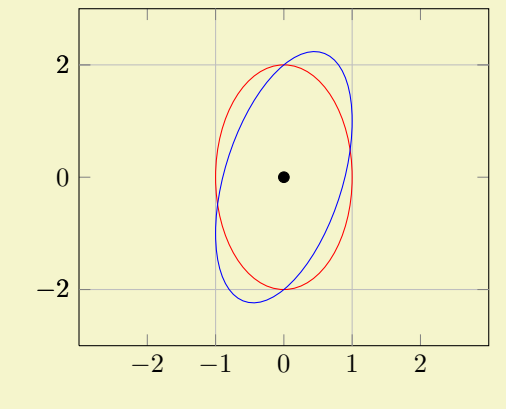
This command is the basic–level variant of `axis cs:`⟨*x coordinate*⟩`,`⟨*y coordinate*⟩`,`⟨*z coordinate*⟩.

\pgfplotspointaxisdirectionxy{⟨*x coordinate*⟩}{⟨*y coordinate*⟩}
\pgfplotspointaxisdirectionxyz{⟨*x coordinate*⟩}{⟨*y coordinate*⟩}{⟨*z coordinate*⟩}

Point commands like \pgfpointxy which take logical, *relative* coordinates and return a low–level point. Every transformation from user transformations to logarithms is applied. The difference to \pgfplotspointaxisxy is that the shift of the linear transformation is skipped here (compare disabledatascaling).

This command is the basic–level variant of `axis direction cs:`⟨*x coordinate*⟩`,`⟨*y coordinate*⟩`,`⟨*z coordinate*⟩. Please refer to the documentation of `axis direction cs` for more details.

Use this command whenever something of *relative* character like directions or lengths need to be supplied. One use-case is to draw ellipses:




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
\begin{axis}[
    xmin=-3,   xmax=3,
    ymin=-3,   ymax=3,
    extra x ticks={-1,1},
    extra y ticks={-2,2},
    extra tick style={grid=major},
]
    \draw[red] \pgfextra{
      \pgfpathellipse{\pgfplotspointaxisxy{0}{0}}
        {\pgfplotspointaxisdirectionxy{1}{0}}
        {\pgfplotspointaxisdirectionxy{0}{2}}
      %  see also the documentation of
      %  'axis direction cs' which
      %  allows a simpler way to draw this ellipse
    };
    \draw[blue] \pgfextra{
      \pgfpathellipse{\pgfplotspointaxisxy{0}{0}}
        {\pgfplotspointaxisdirectionxy{1}{1}}
        {\pgfplotspointaxisdirectionxy{0}{2}}
    };
    \addplot [only marks,mark=*] coordinates { (0,0) };
\end{axis}
\end{tikzpicture}
```

Since the transformations are initialized after the axis is complete, this command needs to be provided either inside of a Ti*k*Z \path command (like \draw in the example above) or inside of \pgfplotsextra.

\pgfplotspointrelaxisxy{⟨*rel x coordinate*⟩}{⟨*rel y coordinate*⟩}
\pgfplotspointrelaxisxyz{⟨*rel x coordinate*⟩}{⟨*rel y coordinate*⟩}{⟨*rel z coordinate*⟩}

Point commands which take *relative* coordinates such that $x = 0$ is the *lower* x axis limit and $x = 1$ the *upper* x axis limit.

These commands are used for `rel axis cs`.

Please note that the transformations are only initialised if the axis is complete! This means you need to provide \pgfplotsextra.

[75] Except for stacked plots where the sequence may be reverse, see the key `reverse stack plots`.

\pgfplotspointdescriptionxy{⟨*x fraction*⟩}{⟨*y fraction*⟩}
\pgfplotsqpointdescriptionxy{⟨*x fraction*⟩}{⟨*y fraction*⟩}

Point commands such that `{0}{0}` is the lower left corner of the axis' bounding box and `{1}{1}` the upper right one; everything else is in between. The '`q`' variant is quicker as it doesn't invoke the math parser on its arguments.

They are used for `axis description cs`, see Section 4.8.1.

\pgfplotspointaxisorigin

A point coordinate at the origin, $(0, 0, 0)$. If the origin is not part of the axis limits, the nearest point on the boundary is returned instead.

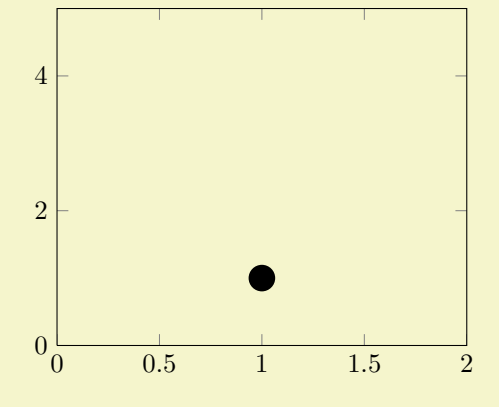
This is the same coordinate as returned by the `origin` anchor.

\pgfplotstransformcoordinatex{⟨*x coordinate of an axis*⟩}
\pgfplotstransformcoordinatey{⟨*y coordinate of an axis*⟩}
\pgfplotstransformcoordinatey{⟨*z coordinate of an axis*⟩}

Defines \pgfmathresult to be the low-level PGF coordinate corresponding to the input argument.

The command applies any `[xyz] coord trafo` keys, data scalings and/or logarithms or whatever PGFPLOTS does to map input coordinates to internal coordinates.

The result can be used inside of a `\pgfpointxy` statement (i.e. it still needs to be scaled with the respective PGF unit vector).




```
% Preamble: \pgfplotsset{width=7cm,compat=1.5.1}
\begin{tikzpicture}
    \begin{axis}[xmin=0,xmax=2,ymin=0,ymax=5]
    \pgfplotsextra{%
        \pgfplotstransformcoordinatex{1}%
        \let\xcoord=\pgfmathresult
        \pgfplotstransformcoordinatey{1}%
        \let\ycoord=\pgfmathresult
        \pgfpathcircle
            {\pgfqpointxy{\xcoord}{\ycoord}}
            {5pt}%
        \pgfusepath{fill}%
    }%
    \end{axis}
\end{tikzpicture}
```

The result of this command is also available as math method `transformcoordinatex` (see the documentation for `axis cs`).

Please note that the transformations are only initialised if the axis is complete. This means you need to provide \pgfplotsextra as is shown in the example above.

\pgfplotstransformdirectionx{⟨*x direction of an axis*⟩}
\pgfplotstransformdirectiony{⟨*y direction of an axis*⟩}
\pgfplotstransformdirectiony{⟨*z direction of an axis*⟩}

Defines \pgfmathresult to be a low-level PGF *direction vector component*.

A direction vector needs to be *added* to some coordinate in order to get a coordinate, compare the documentation for \pgfplotspointaxisdirectionxy and `axis direction cs`.

The argument ⟨*x direction of an axis*⟩ is processed in (almost) the same way as for \pgfplotstransformcoordinatex. The only difference is that *directions* need no shifting transformation.

The result of this command is also available as math method `transformdirectionx` (see the documentation for `axis direction cs`).

See `axis direction cs` for details and examples about this command.

\pgfplotsconvertunittocoordinate{⟨*x, y or z*⟩}{⟨*dimension*⟩}

Converts a dimension (with unit!) to a corresponding x, y or z coordinate. The result will be written to \pgfmathresult (without units).

It is possible to use the result as arguments for the `\pgfpointxyz` commands.

The effect is to multiply ⟨*dimension*⟩ with the inverse length of the unit vector for the specified axis. These lengths are precomputed in PGFPLOTS so the operation is fast.

```
\pgfplotsconvertunittocoordinate{x}{5pt}
%  now, the command uses exactly 5pt in x direction:
\pgfqpointxyz{\pgfmathresult}{4}{3}
```

\pgfplotspointunitx
\pgfplotspointunity
\pgfplotspointunitz

Low–level point commands which return the canvas x, y or z unit vectors.

The `\pgfplotspointunitx` is the PGF unit vector in x direction.

These vectors are essentially the same as `\pgfqpointxyz{1}{0}{0}`, `\pgfqpointxyz{0}{1}{0}`, and `\pgfqpointxyz{0}{0}{1}`, respectively.

The unit z vector is only defined for three dimensional axes.

\pgfplotsunitxlength
\pgfplotsunitylength
\pgfplotsunitzlength
\pgfplotsunitxinvlength
\pgfplotsunityinvlength
\pgfplotsunitzinvlength

Macros which expand to the vector length $\|x_i\|$ of the respective unit vector x_i or the inverse vector length, $1/\|x_i\|$. These macros can be used inside of `\pgfmathparse`, for example.

The x_i are the `\pgfplotspointunitx` variants.

\pgfplotsqpointoutsideofaxis{⟨*three-char-string*⟩}{⟨*coordinate*⟩}{⟨*normal distance*⟩}

Provides a point coordinate on one of the available four axes in case of a two dimensional figure or on one of the available twelve axes in case of a three dimensional figure.

The desired axis is uniquely identified by a three character string, provided as first argument to the command. The first of the three characters is '`0`' if the x coordinate of the specified axis passes through the lower axis limit. It is '`1`', if the x coordinate of the specified axis passes through the upper axis limit. Furthermore, it is '`2`' if it passes through the origin. The second character is also either `0`, `1` or `2` and it characterizes the position on the y axis. The third character is for the third dimension, the z axis. It should be left at '`0`' for two dimensional plots. However, *one* of the three characters should be '`v`', meaning the axis varies. For example, `v01` denotes $\{(x, y_{\min}, z_{\max}) | x \in \mathbb{R}\}$.

The second argument, ⟨*coordinate*⟩ is the logical coordinate on that axis. Since two coordinates of the axis are fixed, ⟨*coordinate*⟩ refers to the varying component of the axis. It must be a number without unit; no math expressions are supported here.

The third argument ⟨*normal distance*⟩ is a dimension like `10pt`. It shifts the coordinate away from the designated axis in direction of the outer normal vector. The outer normal vector always points away from the axis. It is computed using `\pgfplotspointouternormalvectorofaxis`.

There are several variants of this command which are documented in the source code. One of them is particularly useful:

\pgfplotsqpointoutsideofaxisrel{⟨*three-char-string*⟩}{⟨*axis fraction*⟩}{⟨*normal distance*⟩}

This point coordinate is a variant of `\pgfplotsqpointoutsideofaxis` which allows to provide an ⟨*axis fraction*⟩ instead of an absolute coordinate. The fraction is a number between 0 (lower axis limit) and 1 (upper axis limit), i.e. it is given in percent of the total axis. It is possible to provide negative values or values larger than one.

The `\pgfplotsqpointoutsideofaxisrel` command is similar in spirit to `rel axis cs`.

There is one speciality in conjunction with reversed axes: if the axis has been reversed by `x dir=reverse` and, in addition, `allow reversal of rel axis cs` is true, the value 0 denotes the *upper* limit while 1 denotes the *lower* limit. The effect is that coordinates won't change just because of axis reversal.

\pgfplotspointouternormalvectorofaxis{⟨*three-char-string*⟩}

A point command which yields the outer normal vector of the respective axis. The normal vector has length 1 (computed with \pgfpointnormalised). It is the same normal vector used inside of \pgfplotsqpointoutsideofaxis and its variants.

The output of this command will be cached and re-used during the lifetime of an axis.

\pgfplotsticklabelaxisspec{⟨*x, y or z*⟩}

Expands to the three-character-identification for the axis containing tick labels for the chosen axis, either ⟨*x*⟩, ⟨*y*⟩ or ⟨*z*⟩.

\pgfplotsvalueoflargesttickdimen{⟨*x, y or z*⟩}

Expands to the largest distance of a tick position to its tick label bounding box in direction of the outer unit normal vector. It does also include the value of the ticklabel shift key.

This value is used for ticklabel cs.

\pgfplotsmathfloatviewdepthxyz{⟨*x*⟩}{⟨*y*⟩}{⟨*z*⟩}
\pgfplotsmathviewdepthxyz{⟨*x*⟩}{⟨*y*⟩}{⟨*z*⟩}

Both macros define \pgfmathresult to be the "depth" of a three dimensional point $\bar{x} = (x, y, z)$. The depth is defined to be the scalar product of $\bar{x}$ with $\vec{d}$, the view direction of the current axis.

For \pgfplotsmathfloatviewdepthxyz, the arguments are parsed as floating point numbers and the result is encoded in floating point. A fixed point representation can be generated with \pgfmathfloattofixed{\pgfmathresult}.

For \pgfplotsmathviewdepthxyz, TEX arithmetics is employed for the inner product and the result is assigned in fixed point. This is slightly faster, but has considerably smaller data range.

Both commands can only be used *inside* of a three dimensional PGFPLOTS axis (as soon as the axis is initialised, see \pgfplotsextra).

\ifpgfplotsthreedim⟨*true code*⟩\else⟨*else code*⟩\fi

A TEX \if which evaluates the ⟨*true code*⟩ if the axis is three dimensional and the ⟨*else code*⟩ if not.

8.5 Accessing Axis Limits

It is also possible to access axis limits during the visualization phase, i.e. during \end{axis}. Please refer to the reference documentation for xmin on page 214.

Index

— Symbols —

plot (⟨x expression⟩,⟨y expression⟩) . . . 33
(⟨x expression⟩,⟨y expression⟩,⟨z expression⟩) . . 90
-- path operation . . . 362
.style key . . . 302, 304
plot {⟨math expression⟩} . . . 31
{⟨math expression⟩} . . . 89
3d box key . . . 202
3d box foreground style key . . . 249

— A —

Accuracy
  Data Transformation . . . 278
  Floating Point in PGFPLOTS . . . 23
  High Precision for Plot Expression . . . 32
\addlegendentry . . . 152
\addlegendentryexpanded . . . 153
\addlegendimage . . . 167
\addplot . . . 22
\addplot3 . . . 86
after arrow key . . . 68
after end axis key . . . 281
Alignment
  Array . . . 258
  Subplots . . . 258
allow reversal of rel axis cs . . . 365
allow reversal of rel axis cs key . . . 240
allow upside down key . . . 149, 242
anchor key . . . 253
Anchors
  near ticklabel . . . 147
  near xticklabel . . . 147
  near yticklabel . . . 147
  near zticklabel . . . 147
annot/
  collected plots . . . 294
  font . . . 293
  height . . . 294
  js fillColor . . . 292
  jsname . . . 294
  point format . . . 292
  point format 3d . . . 292
  popup size . . . 293
  popup size generic . . . 293
  popup size snap . . . 293
  printable . . . 293
  richtext . . . 293
  slope format . . . 293
  snap dist . . . 293
  textSize . . . 293
  width . . . 294
  xmax . . . 294
  xmin . . . 294
  ymax . . . 294
  ymin . . . 294
.append style handler . . . 246
area cycle list key . . . 71
area legend key . . . 160
area style key . . . 71
array
  Array Alignment . . . 258
at key . . . 253
\autoplotspeclist . . . 361
autumn key . . . 295
aux in dpth key . . . 356
axis environment . . . 20
axis background key . . . 137
axis base prefix key . . . 345
axis cs coordinate system . . . 237
axis description cs coordinate system . . . 142
axis direction cs coordinate system . . . 238
axis equal key . . . 193
axis equal image key . . . 194
axis line style key . . . 171, 248
axis lines key . . . 168
axis lines* key . . . 168
axis on top key . . . 282
axis x discontinuity key . . . 173
axis x line key . . . 168
axis x line* key . . . 168
axis y discontinuity key . . . 174
axis y line key . . . 168
axis y line* key . . . 168
axis z discontinuity key . . . 174
az key . . . 199

— B —

bar cycle list key . . . 58
Bar Plots
  Skewed axes problems . . . 192
bar shift key . . . 59
bar width key . . . 59
baseline key . . . 256
before arrow key . . . 68
before end axis key . . . 281
Behavior Options . . . 22
bins key . . . 63
blackwhite key . . . 127
bled key . . . 295
bluered key . . . 127
bone key . . . 295
Bounding Box Control . . . 263
  Disable *data* bounding box modifications . . . 218
  Excluding Image Parts . . . 262
  Image Externalization Problems . . . 354
  pgfinterruptboundingbox . . . 263
bright key . . . 295

— C —

cartesian cs coordinate system . . . 339
.cd handler . . . 50
change x base key . . . 344
change y base key . . . 344
change z base key . . . 344
check key . . . 95
classes key . . . 77
clickable key . . . 292
clickable library . . . 288
clickable coords key . . . 289
clickable coords code key . . . 290
clickable coords size key . . . 291

clip key . . . 282
clip limits key . . . 216
clip marker paths key . . . 282
\closedcycle . . . 265
cmd key . . . 107
.code handler . . . 50
.code 2 args handler . . . 50
col sep key . . . 30
cold key . . . 295
collected plots key . . . 294
color key . . . 124
colorbar/
    draw . . . 183
    width . . . 182
colorbar key . . . 175
colorbar horizontal key . . . 179
colorbar left key . . . 178
colorbar right key . . . 177
colorbar sampled key . . . 183
colorbar sampled line key . . . 184
colorbar sampled line style key . . . 249
colorbar shift key . . . 183
colorbar source key . . . 181
colorbar style key . . . 182, 248
colorbar to name key . . . 185
colored key . . . 68
colormap/
    autumn . . . 295
    blackwhite . . . 127
    bled . . . 295
    bluered . . . 127
    bone . . . 295
    bright . . . 295
    cold . . . 295
    cool . . . 128
    copper . . . 296
    copper2 . . . 296
    earth . . . 296
    gray . . . 296
    greenyellow . . . 128
    hot . . . 126
    hot2 . . . 126, 296
    hsv . . . 297
    hsv2 . . . 297
    jet . . . 127, 297
    pastel . . . 297
    pink . . . 297
    redyellow . . . 128
    sepia . . . 298
    spring . . . 298
    summer . . . 298
    temp . . . 298
    thermal . . . 299
    violet . . . 128
    winter . . . 299
colormap key . . . 124
colormap access key . . . 141
colormap default colorspace key . . . 126
colormap name key . . . 124
colormaps library . . . 294
cols key . . . 88
columns key . . . 302
comment chars key . . . 30
compat/
    general . . . 7
    labels . . . 6
    path replacement . . . 6
    scaling . . . 6
compat key . . . 6, 150f.
const plot key . . . 52
const plot mark left key . . . 52
const plot mark mid key . . . 53
const plot mark right key . . . 53
contour/
    contour label style . . . 106
    draw color . . . 105
    every contour label . . . 105
    handler . . . 106
    label distance . . . 105
    label node code . . . 106
    labels . . . 105
    labels over line . . . 106
    number . . . 102
contour external/
    cmd . . . 107
    file . . . 107
    scanline marks . . . 107
    script . . . 107
    script extension . . . 107
contour external key . . . 106
contour gnuplot key . . . 101, 107
contour label style key . . . 106
contour prepared key . . . 103
contour prepared format key . . . 103
cool key . . . 128
Coordinate systems
    axis cs . . . 237
    axis description cs . . . 142
    axis direction cs . . . 238
    cartesian cs . . . 339
    rel axis cs . . . 240
    ticklabel cs . . . 144
    xticklabel cs . . . 144
    yticklabel cs . . . 144
    zticklabel cs . . . 144
plot coordinates . . . 24
coordinates . . . 86
\coordindex . . . 36, 362
copper key . . . 296
copper2 key . . . 296
crossref file suffix . . . 165
cube/
    size x . . . 118
    size y . . . 118
    size z . . . 118
cumulative key . . . 63
current axis node . . . 262
current colorbar axis node . . . 183
current plot begin node . . . 240
current plot end node . . . 240
curve style key . . . 342
cycle list key . . . 129
cycle list name key . . . 129
cycle list shift key . . . 136
cycle multi list key . . . 134

— D —

dashdotdotted key . . . 121
dashdotted key . . . 121
dashed key . . . 120
data key . . . 63
data coord inv trafo key . . . 64
data coord trafo key . . . 64
data cs key . . . 274
data filter key . . . 64
data max key . . . 63
data min key . . . 63
/data point/
x . . . 243
y . . . 243
z . . . 243
date coordinates in key . . . 268
date ZERO key . . . 270
dateplot library . . . 268, 299
debug key . . . 48
default smithchart xtick key . . . 327
default smithchart xytick key . . . 327
default smithchart ytick key . . . 327
dense smithchart ticks key . . . 325
densely dashdotdotted key . . . 121
densely dashdotted key . . . 121
densely dashed key . . . 120
densely dotted key . . . 120
disabledatascaling key . . . 278
disablelogfilter key . . . 278
domain key . . . 33
domain y key . . . 33
dotted key . . . 120
Downsampling . . . 272
\draw . . . 362
draw key . . . 124, 183
draw color key . . . 105
draw error bar key . . . 207

— E —

each nth point key . . . 272
each nth tie key . . . 341
earth key . . . 296
el key . . . 200
empty legend key . . . 160
enlarge x limits key . . . 216
enlarge y limits key . . . 216
enlarge z limits key . . . 216
enlargelimits key . . . 216
Environments
axis . . . 20
groupplot . . . 300
loglogaxis . . . 20
pgfinterruptboundingbox . . . 263
pgfplotsinterruptdatabb . . . 218
polaraxis . . . 316
semilogxaxis . . . 20
semilogyaxis . . . 20
smithchart . . . 322
ternaryaxis . . . 331
tikzpicture . . . 20
error bar style key . . . 207, 253
error bars/
draw error bar . . . 207
error bar style . . . 207, 253
error mark . . . 206
error mark options . . . 206
x dir . . . 206
x explicit . . . 206
x explicit relative . . . 206
x fixed . . . 206
x fixed relative . . . 206
y dir . . . 206
y explicit . . . 206
y explicit relative . . . 206
y fixed . . . 206
y fixed relative . . . 206
z dir . . . 206
z explicit . . . 206
z explicit relative . . . 206
z fixed . . . 206
z fixed relative . . . 206
error mark key . . . 206
error mark options key . . . 206
Error Messages
No room for a new dimen . . . 9
Errors
dimension too large . . . 32, 278
Patch Input and Memory Problems . . . 112
Skewed axes and bar plots . . . 192
every 3d box foreground key . . . 249
every 3d description key . . . 200
every 3d view {⟨h⟩}{⟨v⟩} key . . . 201
every arrow key . . . 68
every axis key . . . 246
every axis grid key . . . 251
every axis label key . . . 247
every axis legend key . . . 155, 248
every axis plot key . . . 246
every axis plot no # key . . . 247
every axis plot post key . . . 120, 246
every axis title key . . . 247
every axis title shift key . . . 247
every axis x grid key . . . 252
every axis x label key . . . 247
every axis y grid key . . . 252
every axis y label key . . . 247
every axis z grid key . . . 252
every axis z label key . . . 247
every boxed x axis key . . . 171
every boxed y axis key . . . 171
every boxed z axis key . . . 172
every colorbar key . . . 180, 248
every colorbar sampled line key . . . 185, 249
every colorbar to name picture key . . . 186
every contour label key . . . 105
every crossref picture key . . . 165
every error bar key . . . 252
every extra x tick key . . . 251
every extra y tick key . . . 251
every extra z tick key . . . 251
every forget plot key . . . 247
every inner x axis line key . . . 171, 248
every inner y axis line key . . . 171, 248
every inner z axis line key . . . 171, 248
every legend image post key . . . 159, 248
every legend to name picture key . . . 167, 248
every linear axis key . . . 246
every loglog axis key . . . 246

every major grid key . . . 252
every major tick key . . . 249
every major x grid key . . . 252
every major x tick key . . . 250
every major y grid key . . . 252
every major y tick key . . . 251
every major z grid key . . . 252
every major z tick key . . . 251
every mark key . . . 118
every minor grid key . . . 252
every minor tick key . . . 249
every minor x grid key . . . 252
every minor x tick key . . . 250
every minor y grid key . . . 252
every minor y tick key . . . 250
every minor z grid key . . . 252
every minor z tick key . . . 250
every node near coord key . . . 82
every non boxed x axis key . . . 172
every non boxed y axis key . . . 172
every non boxed z axis key . . . 172
every outer x axis line key . . . 171, 248
every outer y axis line key . . . 171, 248
every outer z axis line key . . . 171, 248
every patch key . . . 116
every semilogx axis key . . . 246
every semilogy axis key . . . 246
every smithchart axis key . . . 329
every ternary axis key . . . 339
every tick key . . . 249
every tick label key . . . 249
every x tick key . . . 250
every x tick label key . . . 249
every x tick scale label key . . . 250
every y tick key . . . 250
every y tick label key . . . 249
every y tick scale label key . . . 250
every z tick key . . . 250
every z tick label key . . . 249
every z tick scale label key . . . 250
execute at begin axis key . . . 279
execute at begin plot key . . . 279
execute at begin plot visualization key . . . 279
execute at end axis key . . . 279
execute at end plot key . . . 279
execute at end plot visualization key . . . 280
execute for upside down key . . . 149
plot expression . . . 33
expression . . . 90
external library . . . 299, 349
External Graphics
  Bounding Box Issues . . . 354
extra description key . . . 152
extra tick style key . . . 251
extra x tick label key . . . 227
extra x tick labels key . . . 227
extra x tick style key . . . 251
extra x ticks key . . . 221
extra y tick label key . . . 227
extra y tick labels key . . . 227
extra y tick style key . . . 251
extra y ticks key . . . 221
extra z tick label key . . . 227
extra z tick labels key . . . 227
extra z tick style key . . . 251
extra z ticks key . . . 221

— F —

faceted color key . . . 101
few smithchart ticks key . . . 323
figure name key . . . 351
plot file . . . 26
file . . . 87
file key . . . 107
\fill . . . 362
fill key . . . 124
filter discard warning key . . . 273
filter point key . . . 270
Floating Point Unit . . . 32
font key . . . 122, 186, 293
footnotesize key . . . 188
\foreach . . . 359
forget plot key . . . 280
forget plot style key . . . 247
fpu key . . . 284
plot function . . . 37

— G —

general key . . . 7
plot gnuplot . . . 36
plot graphics . . . 39
gray key . . . 296
greenyellow key . . . 128
grid key . . . 235
grid style key . . . 251
group/
  columns . . . 302
  empty plot/
    .style . . . 304
  every plot/
    .style . . . 302
  group name . . . 304
  group size . . . 302
  horizontal sep . . . 302
  rows . . . 302
  vertical sep . . . 302
  x descriptions at . . . 303
  xlabels at . . . 302
  xticklabels at . . . 303
  y descriptions at . . . 303
  ylabels at . . . 302
  yticklabels at . . . 303
Group library
  Subplots . . . 299
group name key . . . 304
group size key . . . 302
group style key . . . 302
groupplot environment . . . 300
groupplots library . . . 299

— H —

h key . . . 199
handler key . . . 63, 106
header key . . . 29
height key . . . 189, 294
hide axis key . . . 175
hide x axis key . . . 175

hide y axis key . . . 175
hide z axis key . . . 175
hist/
bins . . . 63
cumulative . . . 63
data . . . 63
data coord inv trafo . . . 64
data coord trafo . . . 64
data filter . . . 64
data max . . . 63
data min . . . 63
handler . . . 63
intervals . . . 63
symbolic coords . . . 64
hist key . . . 61
horizontal sep key . . . 302
hot key . . . 126
hot2 key . . . 126, 296
hsv key . . . 297
hsv2 key . . . 297

— I —
id key . . . 38f.
ifpgfplotsthreedim . . . 366
ignore chars key . . . 30
ignore first key . . . 26
includegraphics key . . . 43
includegraphics cmd key . . . 43
inner axis line style key . . . 171, 248
Interrupted Plots . . . 84
intervals key . . . 63
invoke after crossref tikzpicture key . . . 165
invoke before crossref tikzpicture key . . . 165
is smithchart cs key . . . 327

— J —
jet key . . . 127, 297
js fillColor key . . . 292
jsname key . . . 294
jump mark left key . . . 53
jump mark mid key . . . 54
jump mark right key . . . 54

— K —
Key handlers
.append style . . . 246
.cd . . . 50
.code . . . 50
.code 2 args . . . 50
.style . . . 246

— L —
\label . . . 164
label distance key . . . 105
label node code key . . . 106
label shift key . . . 150
label style key . . . 247
labels key . . . 6, 105
labels over line key . . . 106
\legend . . . 153
legend cell align key . . . 158
legend columns key . . . 159
legend entries key . . . 154
legend image code key . . . 159
legend image post style key . . . 159, 248
legend plot pos key . . . 159
legend pos key . . . 157
legend reversed key . . . 162
legend style key . . . 157, 248
legend to name key . . . 165
legend transposed key . . . 162
Libraries
clickable . . . 288
colormaps . . . 294
dateplot . . . 268, 299
external . . . 299, 349
groupplots . . . 299
patchplots . . . 305
polar . . . 316
smithchart . . . 322
ternary . . . 331
units . . . 342
line cap key . . . 121
line join key . . . 121
line legend key . . . 160
line width key . . . 122
linear regression key . . . 275
\lineno . . . 36
log base 10 number format code key . . . 212
log basis x key . . . 235
log basis y key . . . 235
log basis z key . . . 235
log identify minor tick positions key . . . 211
log number format basis key . . . 212
log number format code key . . . 212
log origin key . . . 218
log origin x key . . . 217
log origin y key . . . 218
log origin z key . . . 218
log plot exponent style key . . . 210
log ticks with fixed point key . . . 209
loglogaxis environment . . . 20
\logten . . . 361
loosely dashdotdotted key . . . 121
loosely dashdotted key . . . 121
loosely dashed key . . . 121
loosely dotted key . . . 120
lowlevel draw key . . . 43

— M —
major grid style key . . . 252
major tick length key . . . 234
major tick style key . . . 249
major x grid style key . . . 252
major x tick style key . . . 251
major y grid style key . . . 252
major y tick style key . . . 251
major z grid style key . . . 252
major z tick style key . . . 251
many smithchart ticks key . . . 324
mark color key . . . 119
mark list fill key . . . 132
mark options key . . . 119
mark repeat key . . . 119
mark size key . . . 118, 186
math parser key . . . 25
\matrix . . . 362
max key . . . 213

max space between ticks key 186, 234
mesh/
check 95
cols 88
ordering 88
rows 88
scanline verbose 88
mesh key 83, 93
mesh input key 114
mesh legend key 161
meta key 29
meta expr key 29
meta index key 30
min key 213
minor grid style key 252
minor tick key 221
minor tick length key 234
minor tick num key 220
minor tick style key 249
minor x grid style key 252
minor x tick num key 220
minor x tick style key 250
minor xtick key 221
minor y grid style key 252
minor y tick num key 220
minor y tick style key 250
minor ytick key 221
minor z grid style key 252
minor z tick num key 220
minor z tick style key 250
minor ztick key 221
miter limit key 121
mode key 353

— N —
near ticklabel anchor 147
near xticklabel anchor 147
near yticklabel anchor 147
near zticklabel anchor 147
\nextgroupplot 300
no markers key 119
\node 362
node key 43
nodes near coords key 80
nodes near coords align key 82
nodes near coords* key 80
normalsize key 186
number key 102
\numplots 361
\numplotsofactualtype 361

— O —
only marks key 74
Options
Distinction Behavior, Style Options 22
ordering key 88
outer axis line style key 171, 248
overlay key 262

— P —
parametric/
var 1d 38
var 2d 38
parametric key 38
parent axis node 180
parent axis height key 181
parent axis width key 181
pastel key 297
patch key 109
patch refines key 312
patch table key 111
patch table with individual point meta key 111
patch table with point meta key 111
patch to triangles key 312
patch type key 114, 305
patchplots library 305
\path 362
Path operations
-- 362
path replacement key 6
/pgf/
bar shift 59
bar width 59
fpu 284
images/
aux in dpth 356
mark color 119
text mark 119
text mark as node 120
text mark style 119
\pgfdeclareplotmark 120
\pgfeov 361
pgfinterruptboundingbox environment 263
\pgfkeys 361
\pgfkeysgetvalue 361
\pgfkeysvalueof 361
\pgfmathparse 360
\pgfmathprintnumber 209, 361
\pgfplotsaxistransformcs 275
\pgfplotsclickablecreate 294
\pgfplotscolorbarfromname 186
\pgfplotscolormaptoshadingspec 128
\pgfplotsconvertunittocoordinate 364
\pgfplotscreateplotcyclelist 133
\pgfplotsdefinecstransform 275
\pgfplotsextra 362
\pgfplotsforeachungrouped 359
\pgfplotsiffileexists 361
pgfplotsinterruptdatabb environment 218
\pgfplotsinvokeforeach 360
\pgfplotslegendfromname 166
\pgfplotsmathfloatviewdepthxyz 366
\pgfplotsmathviewdepthxyz 366
\pgfplotspointaxisdirectionxy 363
\pgfplotspointaxisdirectionxyz 363
\pgfplotspointaxisorigin 364
\pgfplotspointaxisxy 363
\pgfplotspointaxisxyz 363
\pgfplotspointdescriptionxy 364
\pgfplotspointouternormalvectorofaxis 366
\pgfplotspointplotattime 243
\pgfplotspointrelaxisxy 363
\pgfplotspointrelaxisxyz 363
\pgfplotspointunitx 365
\pgfplotspointunity 365
\pgfplotspointunitz 365
\pgfplotsqpointdescriptionxy 364

\pgfplotsqpointoutsideofaxis . . . 365
\pgfplotsqpointoutsideofaxisrel . . . 365
\pgfplotsset . . . 49
\pgfplotstableread . . . 361
\pgfplotstabletypeset . . . 361
\pgfplotsticklabelaxisspec . . . 366
\pgfplotstransformcoordinatex . . . 364
\pgfplotstransformcoordinatey . . . 364
\pgfplotstransformdirectionx . . . 364
\pgfplotstransformdirectiony . . . 364
\pgfplotsunitxinvlength . . . 365
\pgfplotsunitxlength . . . 365
\pgfplotsunityinvlength . . . 365
\pgfplotsunitylength . . . 365
\pgfplotsunitzinvlength . . . 365
\pgfplotsunitzlength . . . 365
\pgfplotsutilifstringequal . . . 361
\pgfplotsvalueoflargesttickdimen . . . 366
\pgfresetboundingbox . . . 263
pink key . . . 297
plot box ratio key . . . 201
plot coordinates/
math parser . . . 25
plot file/
ignore first . . . 26
skip first . . . 26
plot graphics/
debug . . . 48
includegraphics . . . 43
includegraphics cmd . . . 43
lowlevel draw . . . 43
node . . . 43
points . . . 42
xmax . . . 42
xmin . . . 42
ymax . . . 42
ymin . . . 42
zmax . . . 42
zmin . . . 42
plot graphics key . . . 43
Plot operations
(⟨*x expression*⟩,⟨*y expression*⟩,⟨*z expression*⟩) 90
coordinates . . . 86
expression . . . 90
file . . . 87
table . . . 87
{⟨*math expression*⟩} . . . 89
plot (⟨*x expression*⟩,⟨*y expression*⟩) . . . 33
plot coordinates . . . 24
plot expression . . . 33
plot file . . . 26
plot function . . . 37
plot gnuplot . . . 36
plot graphics . . . 39
plot shell . . . 38
plot table . . . 26, 35
plot {⟨*math expression*⟩} . . . 31
\plotnum . . . 362
\plotnumofactualtype . . . 362
point format key . . . 292
point format 3d key . . . 292
point meta key . . . 138
point meta max key . . . 141, 181
point meta min key . . . 141, 181
point meta rel key . . . 140
points key . . . 42
polar library . . . 316
polar comb key . . . 319
polaraxis environment . . . 316
popup size key . . . 293
popup size generic key . . . 293
popup size snap key . . . 293
pos key . . . 241
pos segment key . . . 242
Precision . . . 32, 284
precision key . . . 101
Predefined node
current axis . . . 262
current colorbar axis . . . 183
current plot begin . . . 240
current plot end . . . 240
parent axis . . . 180
prefix key . . . 38f., 350
prefixes
unit . . . 344
printable key . . . 293

— Q —

quiver/
after arrow . . . 68
before arrow . . . 68
colored . . . 68
every arrow . . . 68
scale arrows . . . 68
u . . . 66
u value . . . 68
update limits . . . 68
v . . . 67
v value . . . 68
w . . . 67
w value . . . 68
quiver key . . . 65

— R —

raw gnuplot key . . . 38
read completely key . . . 30
redyellow key . . . 128
\ref . . . 164
refstyle key . . . 164
rel axis cs coordinate system . . . 240
reset nontranslations key . . . 149
restrict expr to domain key . . . 273
restrict expr to domain* key . . . 273
restrict x to domain key . . . 272
restrict x to domain* key . . . 272
restrict y to domain key . . . 272
restrict y to domain* key . . . 272
restrict z to domain key . . . 272
restrict z to domain* key . . . 272
reverse legend key . . . 162
reverse stacked plots key . . . 70
richtext key . . . 293
row sep key . . . 30
rows key . . . 88, 302

— S —

samples key . . . 34
samples at key . . . 34
samples y key . . . 34
scale key . . . 196
scale arrows key . . . 68
scale mode key . . . 192
scale only axis key . . . 189
scale ticks above key . . . 234
scale ticks below key . . . 234
scaled ticks key . . . 230
scaled x ticks key . . . 230
scaled y ticks key . . . 230
scaled z ticks key . . . 230
scaling key . . . 6
scanline marks key . . . 107
scanline verbose key . . . 88
scatter/
    classes . . . 77
    use mapped color . . . 76
scatter key . . . 75
scatter src key . . . 75
script key . . . 107
script extension key . . . 107
semilogxaxis environment . . . 20
semilogyaxis environment . . . 20
semithick key . . . 122
separate axis lines key . . . 172
sepia key . . . 298
set point meta if empty key . . . 140
shader key . . . 98
sharp plot key . . . 51
plot shell . . . 38
shell escape key . . . 353
show origin key . . . 328
show origin code key . . . 328
size x key . . . 118
size y key . . . 118
size z key . . . 118
skip coords between index key . . . 272
skip first key . . . 26
skip first n key . . . 31
slope format key . . . 293
sloped/
    allow upside down . . . 149
    execute for upside down . . . 149
    reset nontranslations . . . 149
sloped key . . . 242
sloped like x axis key . . . 149
sloped like y axis key . . . 149
sloped like z axis key . . . 149
small key . . . 187
smithchart environment . . . 322
smithchart library . . . 322
smooth key . . . 51
snap dist key . . . 293
solid key . . . 120
spring key . . . 298
stack dir key . . . 70
stack plots key . . . 68
.style handler . . . 246
Subplots . . . 258
subtickwidth key . . . 234
summer key . . . 298
surf key . . . 96
surf shading/
    precision . . . 101
symbolic coords key . . . 64
symbolic x coords key . . . 267
symbolic y coords key . . . 267
symbolic z coords key . . . 267
system call key . . . 352

— T —

plot table . . . 26, 35
    Unbalanced Columns . . . 28
table/
    col sep . . . 30
    comment chars . . . 30
    create col/
      linear regression . . . 275
    header . . . 29
    ignore chars . . . 30
    meta . . . 29
    meta expr . . . 29
    meta index . . . 30
    read completely . . . 30
    row sep . . . 30
    skip first n . . . 31
    tie end x . . . 341
    tie end x index . . . 341
    tie end y . . . 341
    tie end y index . . . 341
    tie end z . . . 341
    tie end z index . . . 341
    white space chars . . . 30
    x . . . 29
    x error . . . 29
    x error expr . . . 29
    x error index . . . 29
    x expr . . . 29
    x index . . . 29
    y . . . 29
    y error . . . 29
    y error expr . . . 29
    y error index . . . 29
    y expr . . . 29
    y index . . . 29
    z . . . 29
    z error . . . 29
    z error expr . . . 29
    z error index . . . 29
    z expr . . . 29
    z index . . . 29
table . . . 87
table key . . . 276
temp key . . . 298
ternary library . . . 331
ternary limits relative key . . . 336
ternary relative limits key . . . 336
ternaryaxis environment . . . 331
text mark key . . . 119
text mark as node key . . . 120
text mark style key . . . 119
textSize key . . . 293
thermal key . . . 299
thick key . . . 122
thin key . . . 122

\thisrow . . . 35
\thisrowno . . . 36
tick align key . . . 229
tick label style key . . . 249
tick pos key . . . 229
tick scale binop key . . . 233
tick scale label code key . . . 233
tick style key . . . 249
ticklabel cs coordinate system . . . 144
ticklabel pos key . . . 229
ticklabel shift key . . . 230
ticklabel style key . . . 249
ticks key . . . 228
tickwidth key . . . 234
tie end x key . . . 341
tie end x index key . . . 341
tie end y key . . . 341
tie end y index key . . . 341
tie end z key . . . 341
tie end z index key . . . 341
tieline/
  curve style . . . 342
  each nth tie . . . 341
  tieline style . . . 341
tieline key . . . 340
tieline style key . . . 341
/tikz/
  allow upside down . . . 242
  baseline . . . 256
  color . . . 124
  const plot . . . 52
  const plot mark left . . . 52
  const plot mark mid . . . 53
  const plot mark right . . . 53
  dashdotdotted . . . 121
  dashdotted . . . 121
  dashed . . . 120
  densely dashdotdotted . . . 121
  densely dashdotted . . . 121
  densely dashed . . . 120
  densely dotted . . . 120
  dotted . . . 120
  draw . . . 124
  every mark . . . 118
  external/
    figure name . . . 351
    mode . . . 353
    prefix . . . 350
    shell escape . . . 353
    system call . . . 352
  fill . . . 124
  font . . . 122, 186
  id . . . 38f.
  jump mark left . . . 53
  jump mark mid . . . 54
  jump mark right . . . 54
  line cap . . . 121
  line join . . . 121
  line width . . . 122
  loosely dashdotdotted . . . 121
  loosely dashdotted . . . 121
  loosely dashed . . . 121
  loosely dotted . . . 120
  mark options . . . 119
  mark repeat . . . 119
  mark size . . . 118, 186
  miter limit . . . 121
  only marks . . . 74
  overlay . . . 262
  polar comb . . . 319
  pos . . . 241
  pos segment . . . 242
  prefix . . . 38f.
  raw gnuplot . . . 38
  semithick . . . 122
  sharp plot . . . 51
  sloped . . . 242
  sloped like x axis . . . 149
  sloped like y axis . . . 149
  sloped like z axis . . . 149
  smooth . . . 51
  solid . . . 120
  thick . . . 122
  thin . . . 122
  tieline . . . 340
  trim axis group left . . . 264
  trim axis group right . . . 264
  trim axis left . . . 264
  trim axis right . . . 264
  trim left . . . 264
  trim right . . . 264
  ultra thick . . . 122
  ultra thin . . . 122
  very thick . . . 122
  very thin . . . 122
  xbar . . . 54
  xbar interval . . . 60
  xcomb . . . 65
  xshift . . . 362
  ybar . . . 56
  ybar interval . . . 59
  ycomb . . . 65
  yshift . . . 362
\tikzappendtofigurename . . . 352
tikzpicture environment . . . 20
\tikzsetexternalprefix . . . 350
\tikzsetfigurename . . . 351
\tikzsetnextfilename . . . 350
tiny key . . . 188
title key . . . 151
title style key . . . 247
translate gnuplot key . . . 38
transpose legend key . . . 162
trim axis group left key . . . 264
trim axis group right key . . . 264
trim axis left key . . . 264
trim axis right key . . . 264
trim left key . . . 264
trim right key . . . 264
try min ticks key . . . 186, 234
try min ticks log key . . . 234

— **U** —

u key . . . 66
u value key . . . 68
ultra thick key . . . 122
ultra thin key . . . 122

Unbalanced Columns 28
unbounded coords key 84
unit code key 344
unit marking post key 343
unit marking pre key 343
unit markings key 343
unit rescale keep size key 195
unit vector ratio key 194
unit vector ratio* key 195
units library 342
update limits key 68, 218
use mapped color key 76
use units key 342

— V —
v key 67, 200
v value key 68
var 1d key 38
var 2d key 38
variable key 34
variable y key 34
variance key 277
variance list key 277
variance src key 278
vertex count key 311
vertical sep key 302
very thick key 122
very thin key 122
view/
    az 199
    el 200
    h 199
    v 200
view key 198
violet key 128
visualization depends on key 283

— W —
w key 67
w value key 68
white space chars key 30
width key 182, 189, 294
winter key 299

— X —
x key 29, 190, 243, 277
\x In Coordinate Expressions 32
x axis line style key 171, 248
x coord inv trafo key 267
x coord trafo key 266
x descriptions at key 303
x dir key 193, 206, 214
x error key 29
x error expr key 29
x error index key 29
x explicit key 206
x explicit relative key 206
x expr key 29
x filter key 270
x fixed key 206
x fixed relative key 206
x grid style key 252
x index key 29
x label style key 247
x post scale key 196
x SI prefix key 344
x tick label as interval key 227
x tick label style key 249
x tick scale label style key 250
x unit key 342
x unit prefix key 343
xbar key 54, 56
xbar interval key . . . . . 60f.
xbar interval legend key 160
xbar interval stacked key 70
xbar legend key 160
xbar stacked key 70
xcomb key 65
xgrid each nth passes y key 329
xgrid each nth passes y start key 330
xgrid stop at y key 330
xlabel
    Line break 150
    Multiline 150
xlabel key 149
xlabel absolute key 151
xlabel near ticks key 150
xlabel shift key 150
xlabel style key 247
xlabels at key 302
xmajorgrids key 235
xmajorticks key 228
xmax key 42, 212, 294
xmin key 42, 212, 294
xminorgrids key 235
xminorticks key 228
xmode key 192, 214, 277
xshift key 362
xtick key 218
xtick align key 229
xtick placement tolerance key 234
xtick pos key 229
xtick scale label code key 233
xtick style key 250
xticklabel key 225
xticklabel cs coordinate system 144
xticklabel interval boundaries key 61
xticklabel pos key 229
xticklabel shift key 230
xticklabel style key 250
xticklabels key 224
xticklabels at key 303
xticklabels from table key 227
xtickmax key 174, 229
xtickmin key 174, 228
xtickten key 223

— Y —
y key 29, 190, 243, 277
y axis line style key 171, 248
y coord inv trafo key 267
y coord trafo key 266
y descriptions at key 303
y dir key 193, 206, 214
y domain key 33
y error key 29
y error expr key 29
y error index key 29

y explicit key . . . 206
y explicit relative key . . . 206
y expr key . . . 29
y filter key . . . 270
y fixed key . . . 206
y fixed relative key . . . 206
y grid style key . . . 252
y index key . . . 29
y label style key . . . 247
y post scale key . . . 196
y SI prefix key . . . 344
y tick label as interval key . . . 227
y tick label style key . . . 249
y tick scale label style key . . . 250
y unit key . . . 343
y unit prefix key . . . 343
ybar key . . . 56, 58
ybar interval key . . . 59f.
ybar interval legend key . . . 160
ybar interval stacked key . . . 70
ybar legend key . . . 160
ybar stacked key . . . 70
ycomb key . . . 65
ygrid each nth passes x key . . . 330
ygrid each nth passes x start key . . . 330
ygrid stop at x key . . . 330
ylabel key . . . 149
ylabel absolute key . . . 151
ylabel near ticks key . . . 150
ylabel shift key . . . 150
ylabel style key . . . 247
ylabels at key . . . 302
ymajorgrids key . . . 235
ymajorticks key . . . 228
ymax key . . . 42, 213, 294
ymin key . . . 42, 212, 294
yminorgrids key . . . 235
yminorticks key . . . 228
ymode key . . . 192, 214, 277
yshift key . . . 362
ytick key . . . 218
ytick align key . . . 229
ytick placement tolerance key . . . 234
ytick pos key . . . 229
ytick scale label code key . . . 233
ytick style key . . . 250
yticklabel key . . . 225
yticklabel cs coordinate system . . . 144
yticklabel in circle key . . . 328
yticklabel interval boundaries key . . . 61
yticklabel pos key . . . 229
yticklabel shift key . . . 230
yticklabel style key . . . 250
yticklabels key . . . 224
yticklabels at key . . . 303
yticklabels from table key . . . 227
ytickmax key . . . 174, 229
ytickmin key . . . 174, 228
ytickten key . . . 223

— Z —

z key . . . 29, 190, 243
z axis line style key . . . 171, 248
z buffer key . . . 96
z coord inv trafo key . . . 267
z coord trafo key . . . 267
z dir key . . . 193, 206, 214
z error key . . . 29
z error expr key . . . 29
z error index key . . . 29
z explicit key . . . 206
z explicit relative key . . . 206
z expr key . . . 29
z filter key . . . 270
z fixed key . . . 206
z fixed relative key . . . 206
z grid style key . . . 252
z index key . . . 29
z label style key . . . 247
z post scale key . . . 196
z SI prefix key . . . 344
z tick label as interval key . . . 227
z tick label style key . . . 250
z tick scale label style key . . . 250
z unit key . . . 343
z unit prefix key . . . 343
zbar interval legend key . . . 160
zbar legend key . . . 160
zlabel key . . . 149
zlabel absolute key . . . 151
zlabel near ticks key . . . 150
zlabel shift key . . . 150
zlabel style key . . . 247
zmajorgrids key . . . 235
zmajorticks key . . . 228
zmax key . . . 42, 213
zmin key . . . 42, 212
zminorgrids key . . . 235
zminorticks key . . . 228
zmode key . . . 192, 214
ztick key . . . 218
ztick align key . . . 229
ztick placement tolerance key . . . 234
ztick pos key . . . 229
ztick scale label code key . . . 233
ztick style key . . . 250
zticklabel key . . . 225
zticklabel cs coordinate system . . . 144
zticklabel interval boundaries key . . . 61
zticklabel pos key . . . 229
zticklabel shift key . . . 230
zticklabel style key . . . 250
zticklabels key . . . 224
zticklabels from table key . . . 227
ztickmax key . . . 174, 229
ztickmin key . . . 174, 228
ztickten key . . . 223

References

[1] C. Feuersänger. PgfplotsTable package – Loading, rounding and formatting tables in LaTeX. Available as separate package \usepackage{pgfplotstable}, as part of pgfplots.

[2] C. Feuersänger. Programming in TEX and Library Functions from pgf and pgfplots. Available as part of pgfplots, TeX-programming-notes.pdf, December 29, 2011.

[3] U. Kern. Extending LATEX's color facilities: the xcolor package.

[4] D. P. Story. The AcroTEX eDucation Bundle. http://www.ctan.org/tex-archive/macros/latex/contrib/acrotex. Sub packages insdljs and eforms are required for the clickable library.

[5] T. Tantau. Ti*k*Z and pgf manual. http://sourceforge.net/projects/pgf. *v.* ≥ 2.00.

[6] K. van Zonneveld. PhP to javascript conversion project (GPL). http://kevin.vanzonneveld.net/techblog/article/phpjs_licensing.

[7] J. Wright and C. Feuersänger. Implementing keyval input: an introduction. http://pgfplots.sourceforge.net as .pdf, 2008.