# 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.

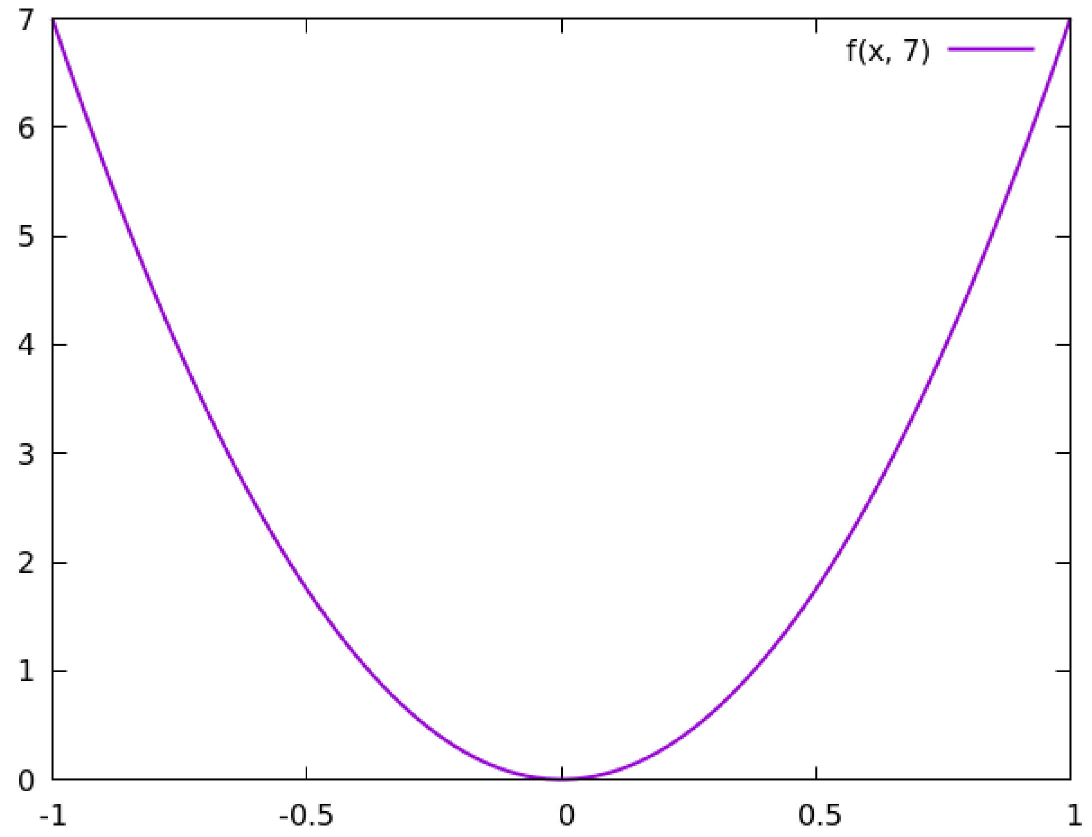
When an example script creates an animation, we'll display the first frame of the animation, in the place where we normally display the example graph, overlayed with an icon resembling a movie camera. The movie file is embedded in the book's PDF file as an attachment; not all PDF readers know how to 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

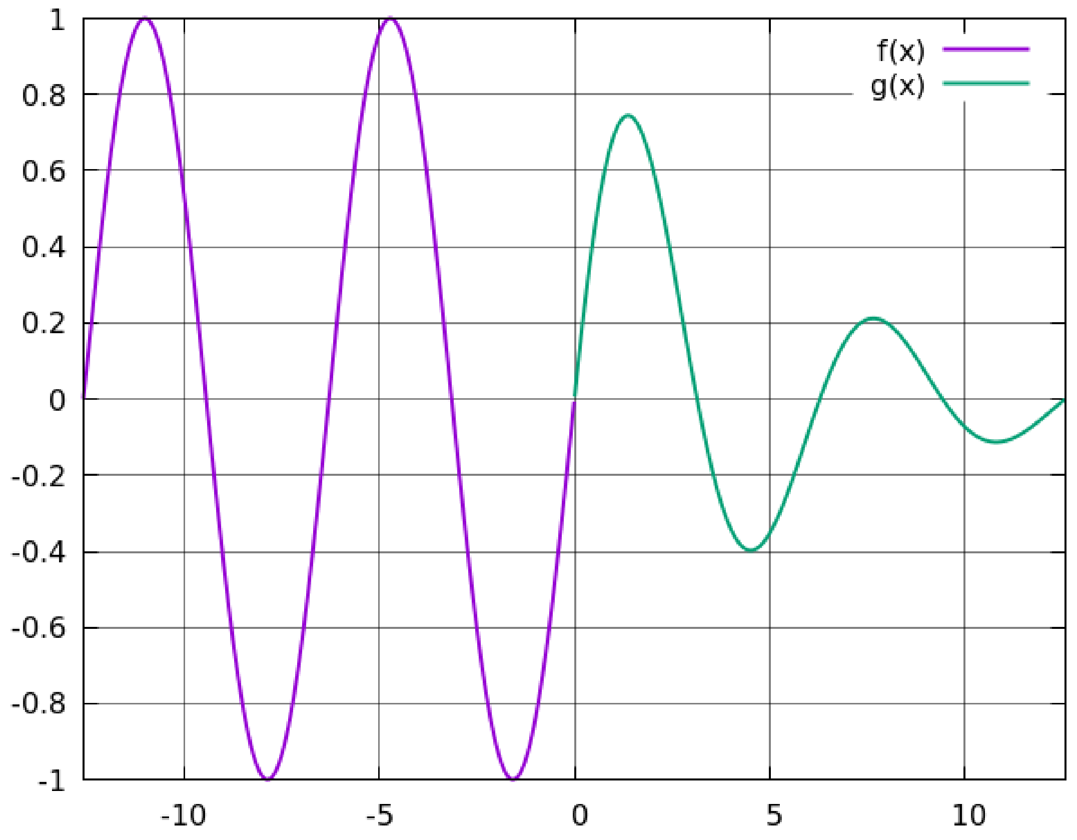


## The Ternary Operator

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.)

```
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

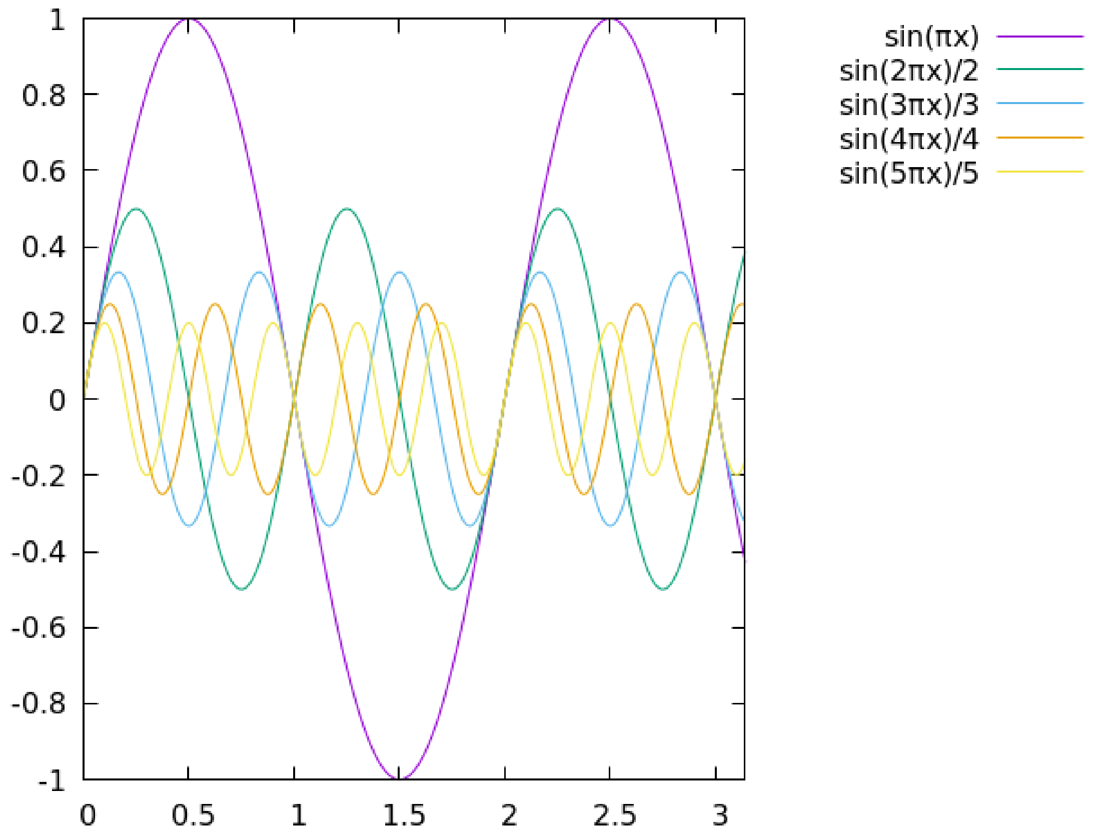


## Basic Iteration

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

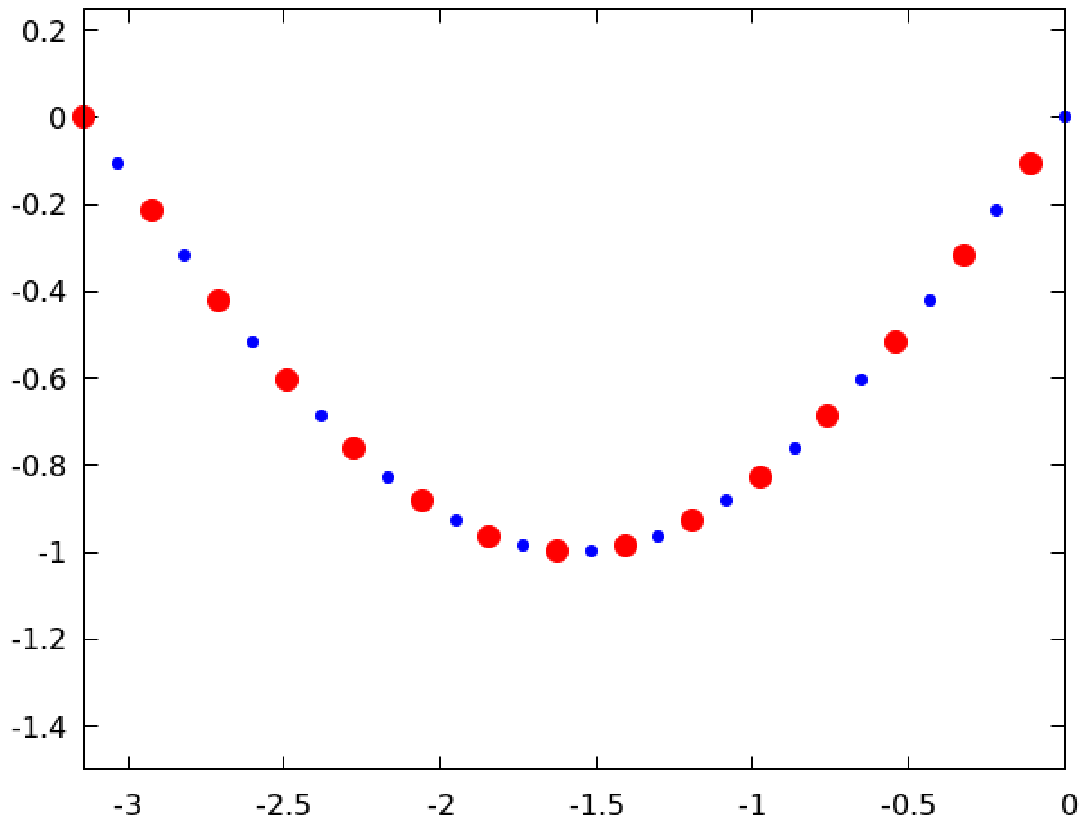


## The Special Filename "+"

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

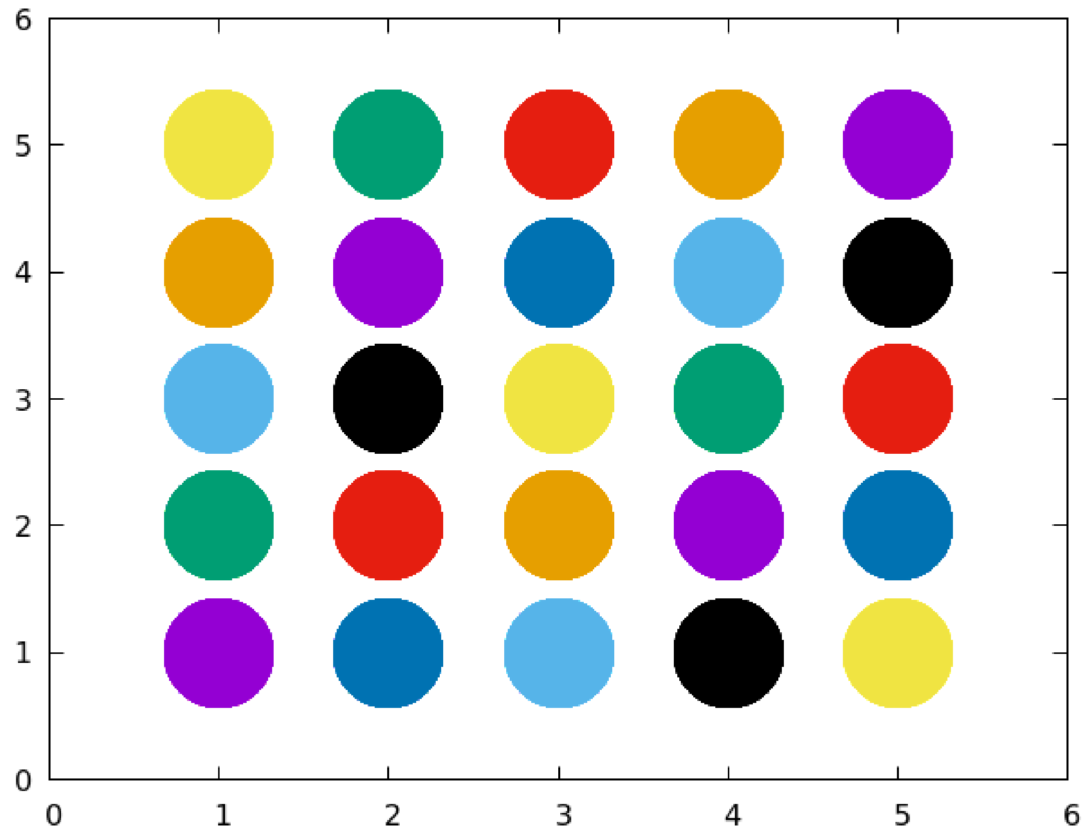


## Nested Iteration

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

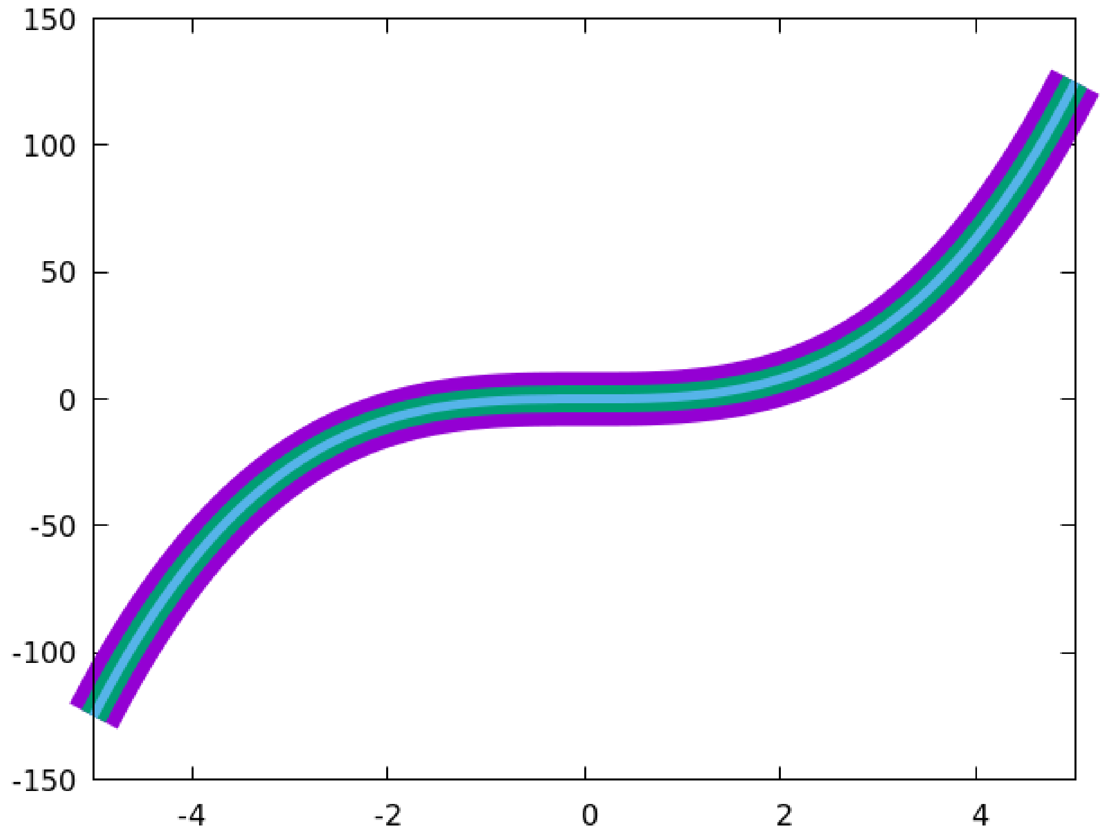


## Iteration Over Words

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

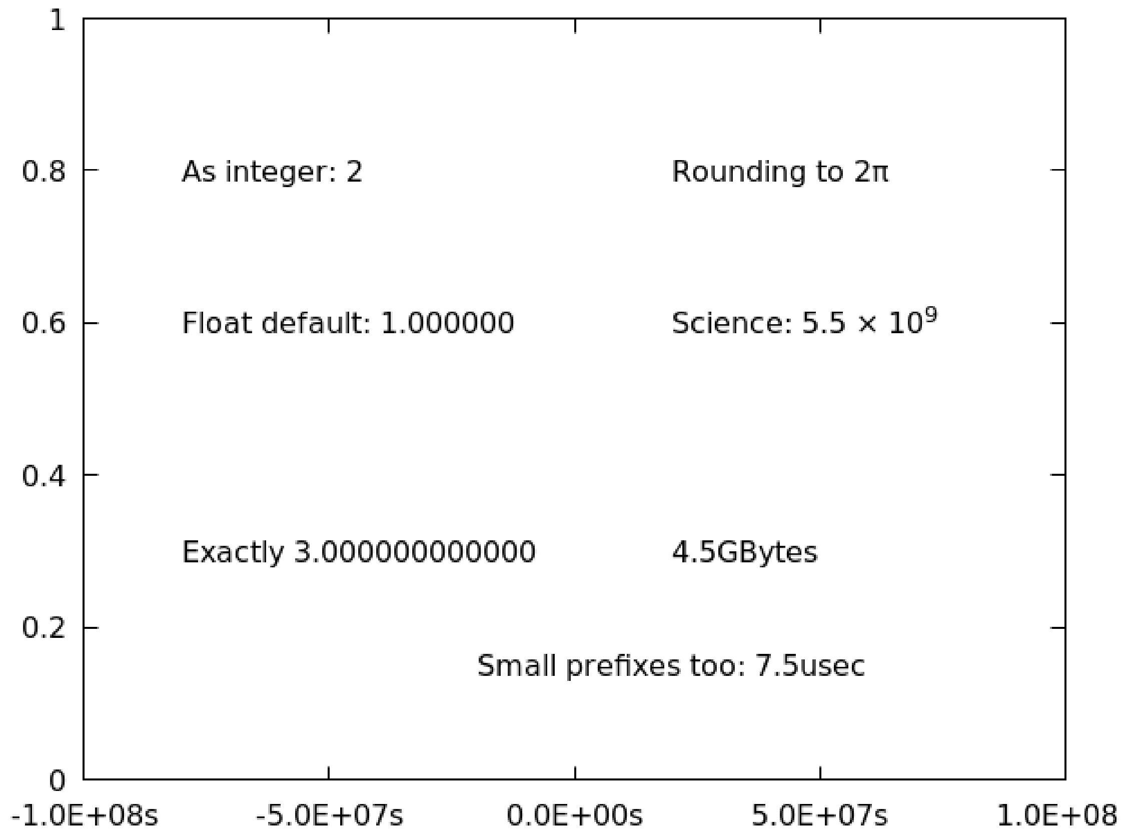


## String Formatting

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
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)
set xr [-10**8 : 10**8]
set format x "%.1Es"
plot -1
```

Open script

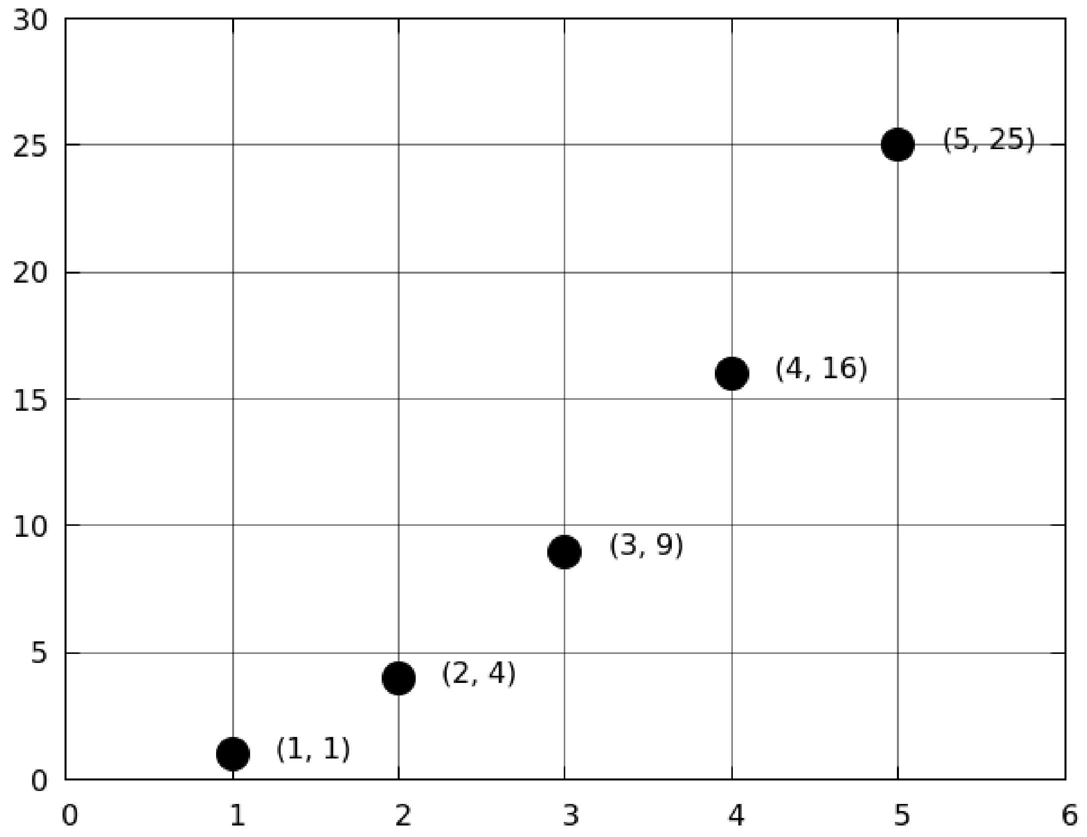


## Iteration Over Blocks

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



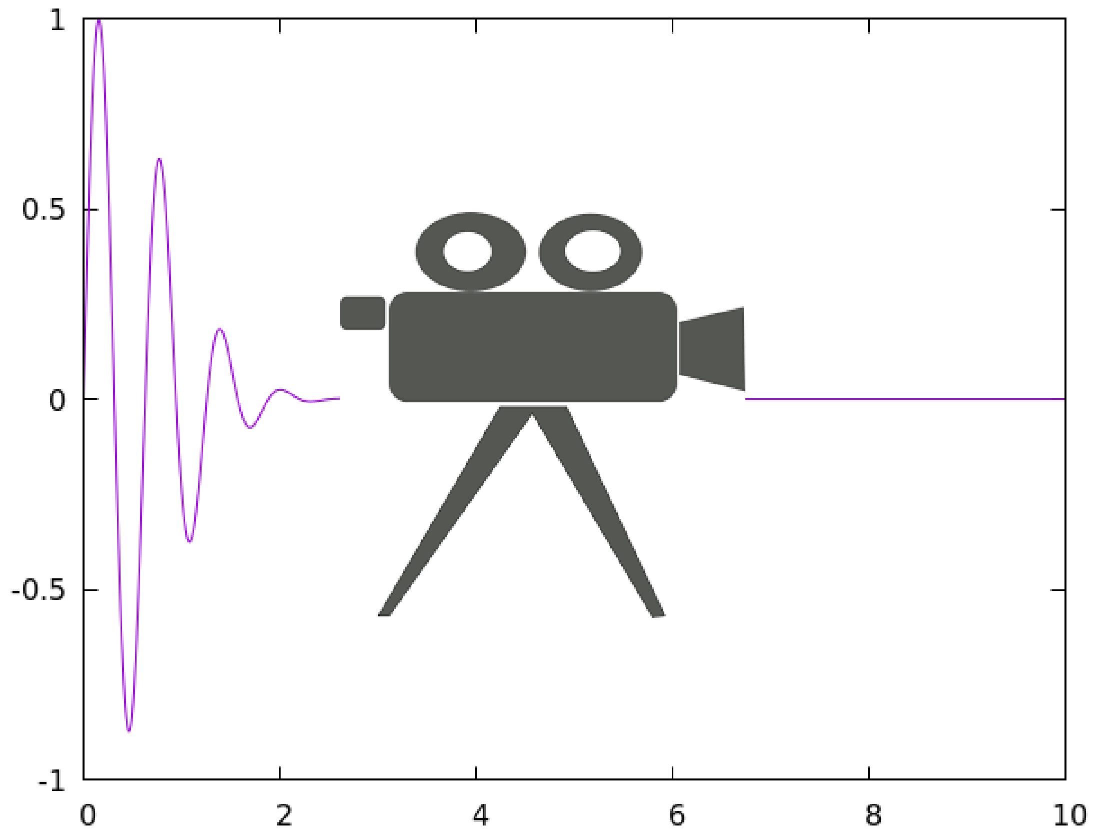
## Animations

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



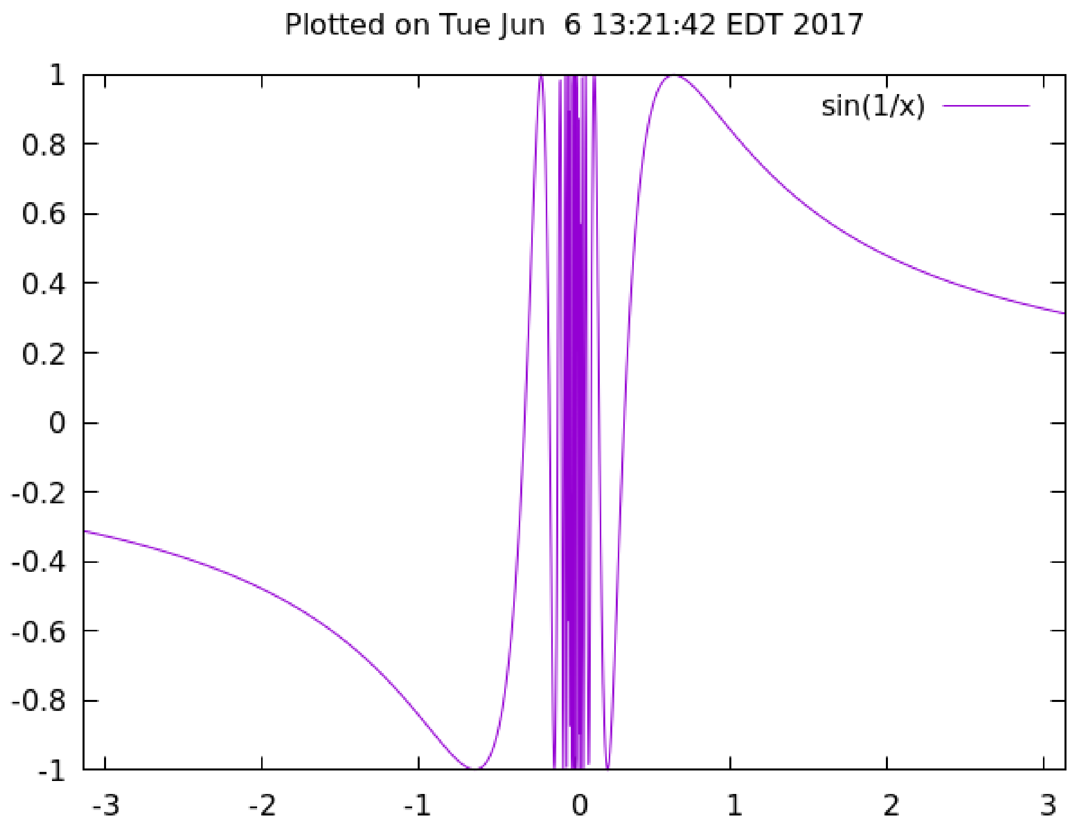
Click or double-click the image to open, or go here.

## 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

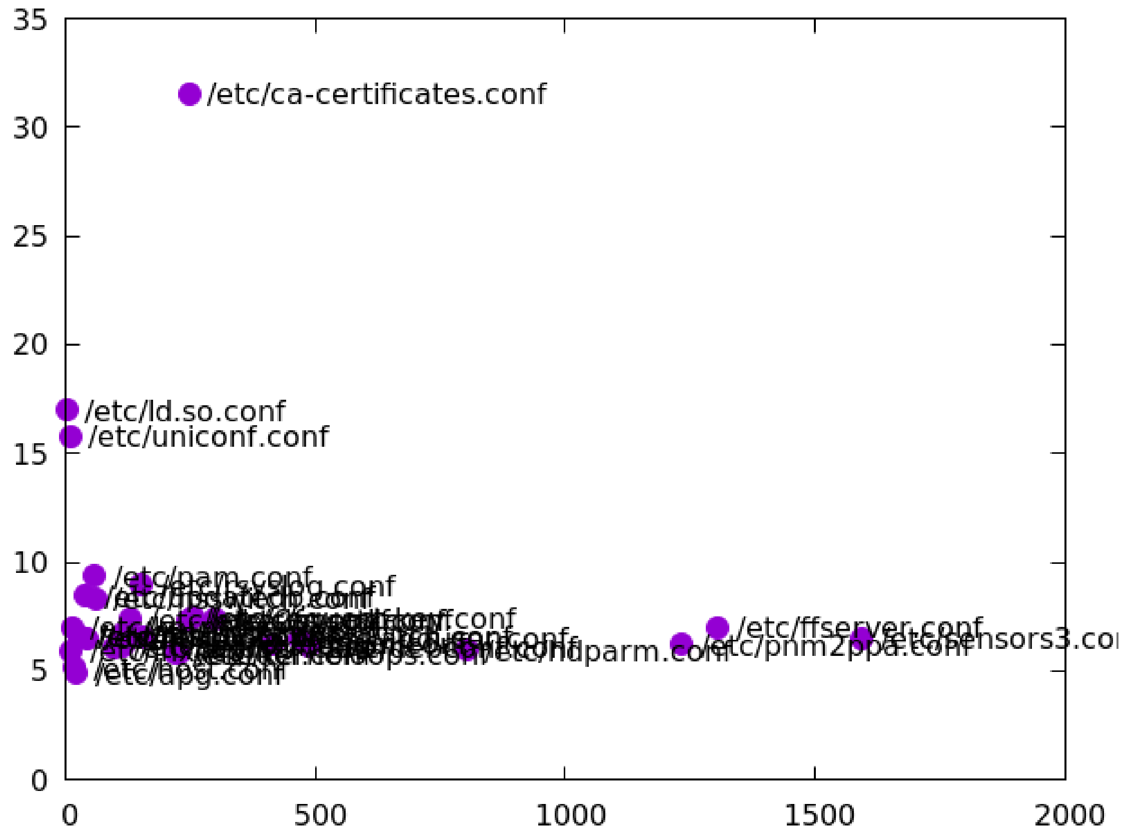


## Externally Processed Data Files

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

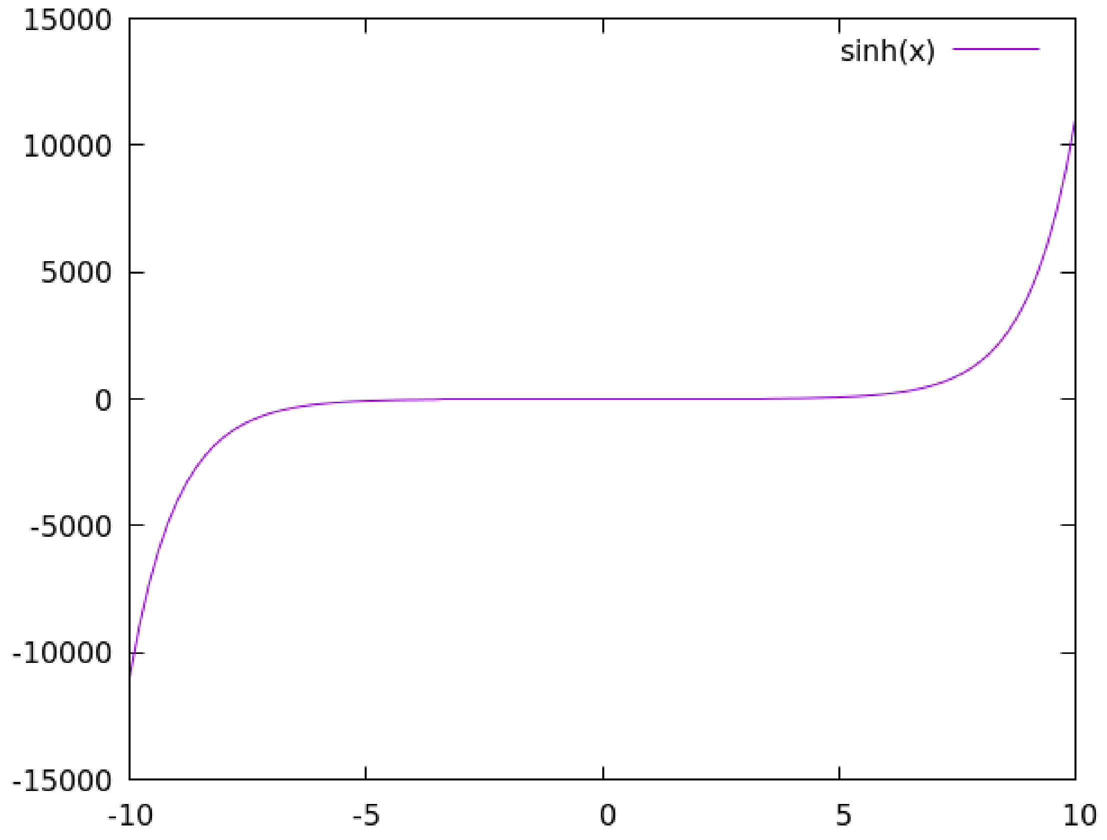


## Invocation

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)"
```

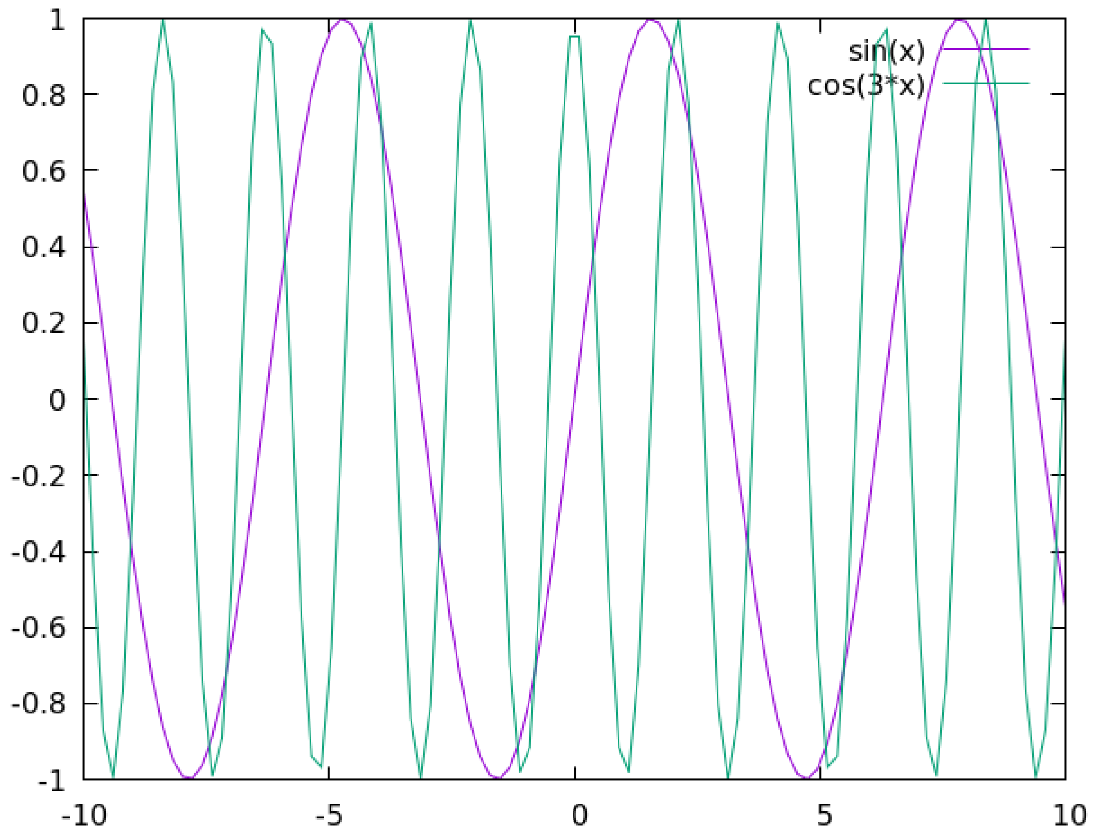
This will immediately display a window with the hyperbolic sine function, without leaving gnuplot running. The `-e` flag is a convenient way to get a quick look at some data or function.



## 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)
```

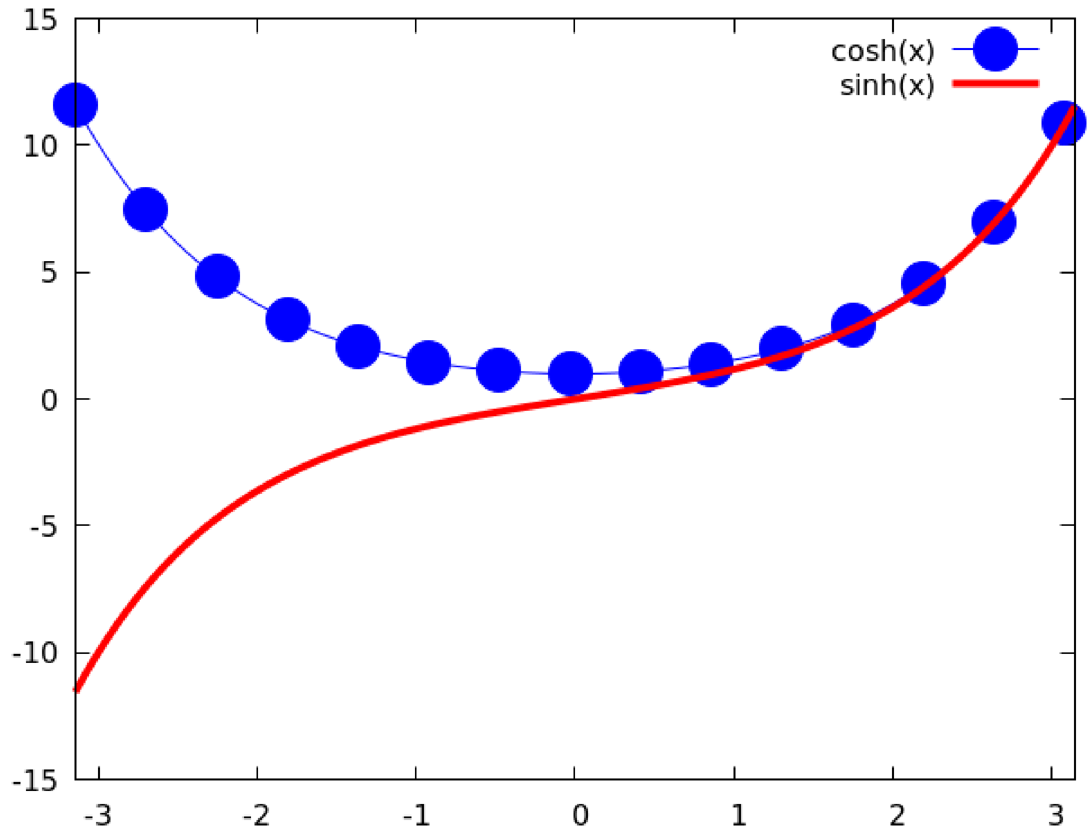


## Macros

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.

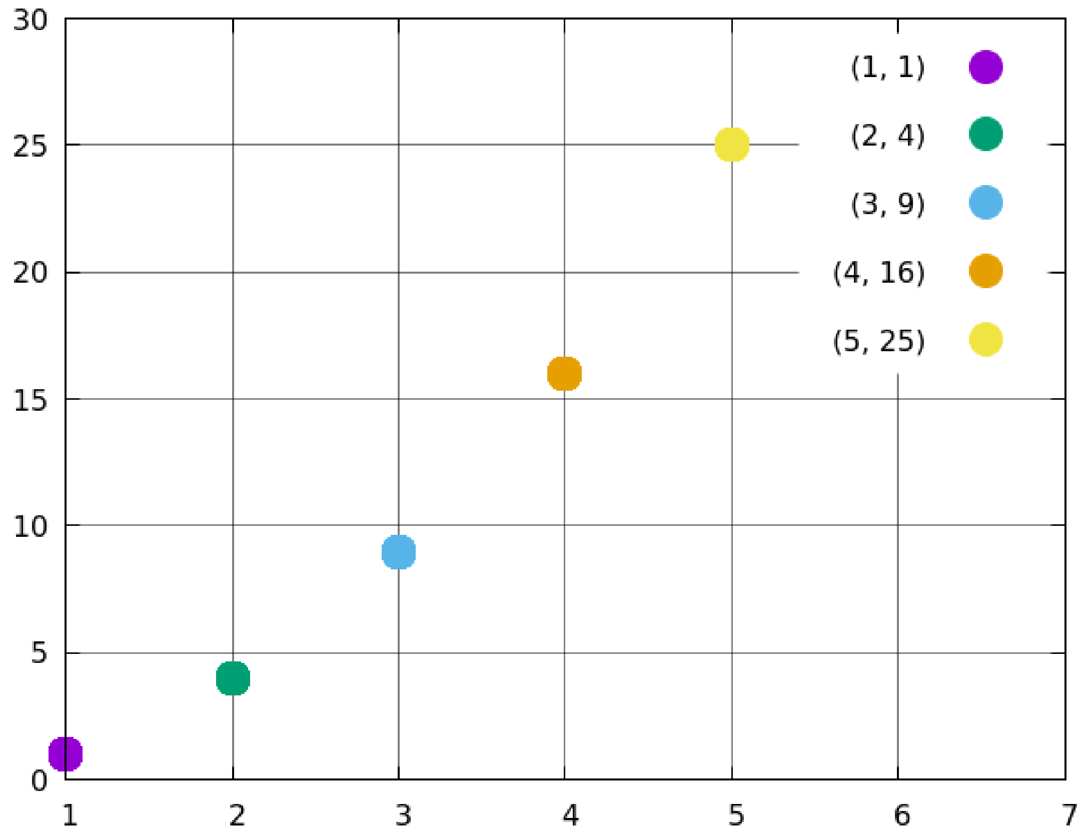


## Arrays

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

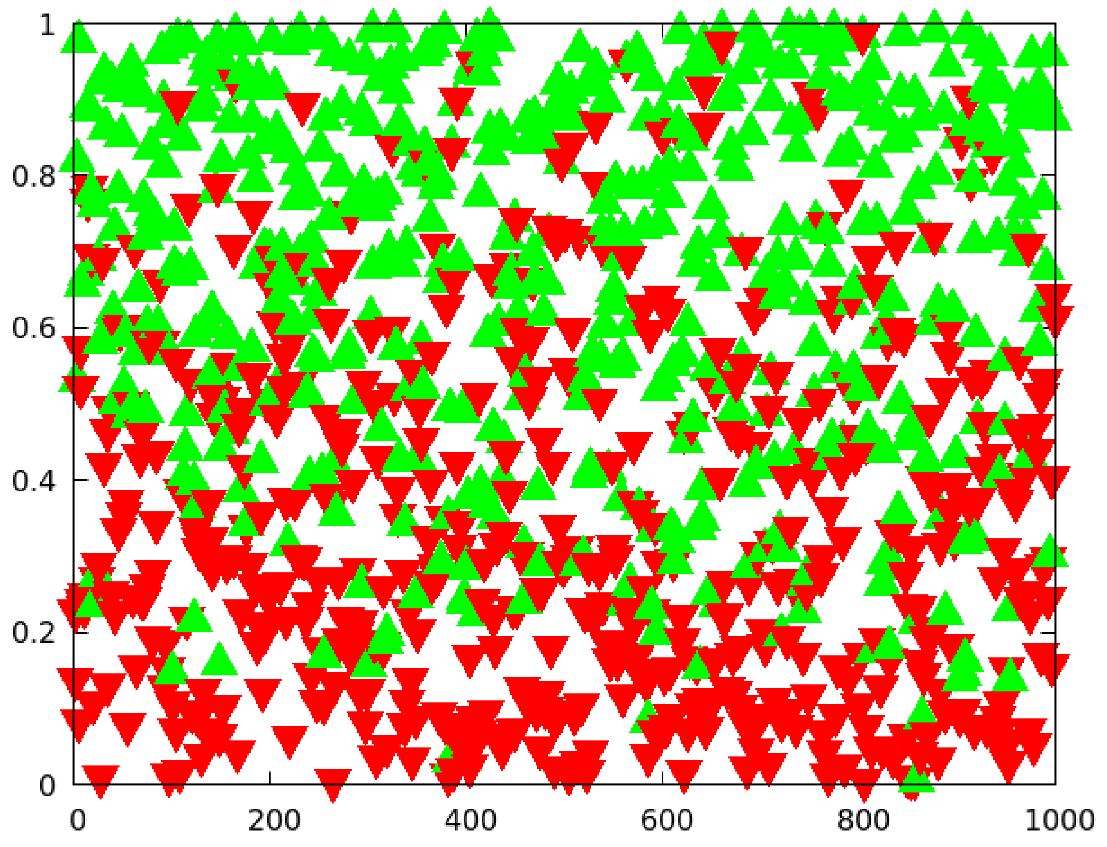


## `if` and `else`

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

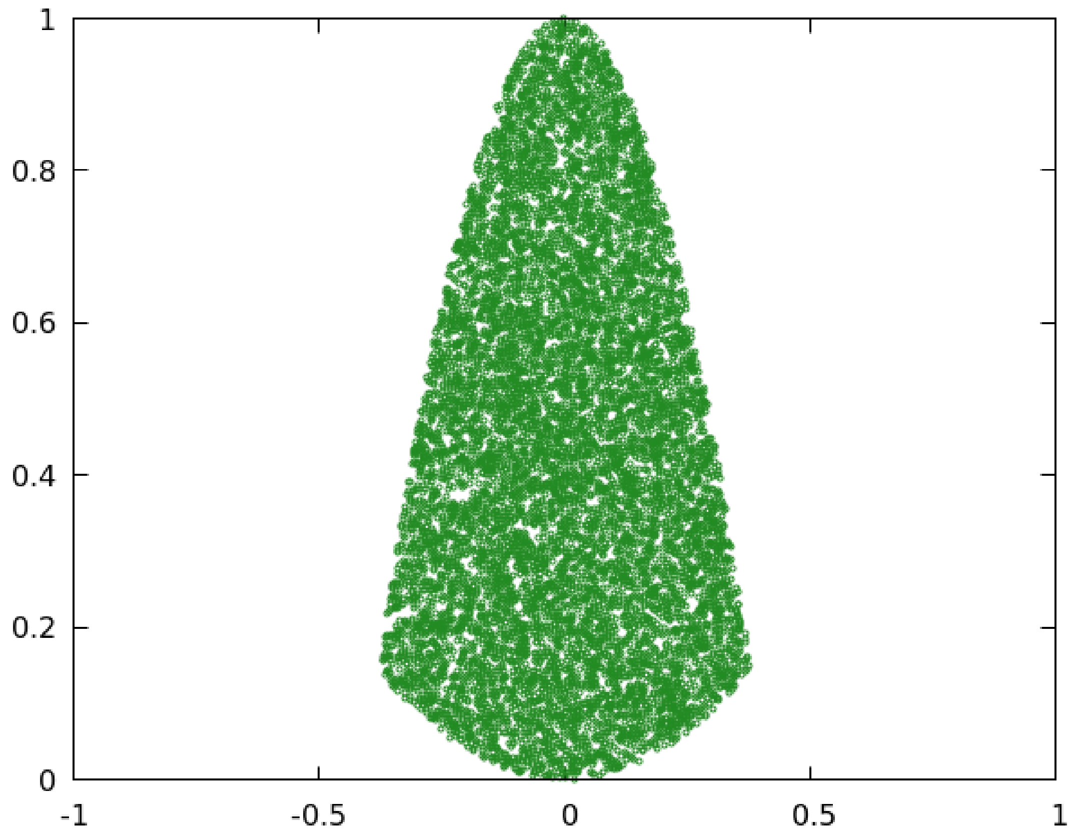


## **while, break, and continue**

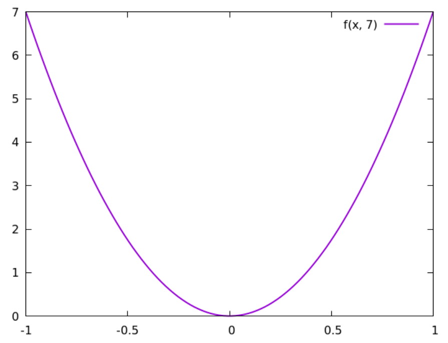
These keywords work in gnuplot scripts just as they do in other programming languages that use them. `while` repeats a block while a condition remains true; `break` stops an iteration immediately, leaving the block; and `continue` skips to the end of the block and continues with the next iteration, if 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 `&&`.

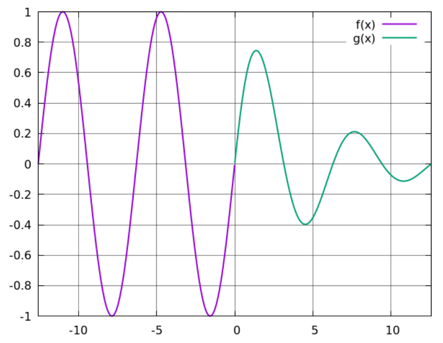
```
set xr [-1 : 1]; set yr [0 : 1]
set nokey
n = 10000; set samp n; count = 0
array inx[n]; array iny[n]
upper(x) = 1-6*x**2; lower(x) = x**2
while 1 {
   xx = 2*rand(0)-1
   yy = rand(0)
   if (yy >= upper(xx) || yy <= lower(xx)) { continue }
   else {
      count = count + 1
      if (count > n) {break}
      inx[count] = xx
      iny[count] = yy } }
plot "+" u (inx[$0+1]):(iny[$0+1]) pt 6 ps 0.5 lc "forest-green"
```

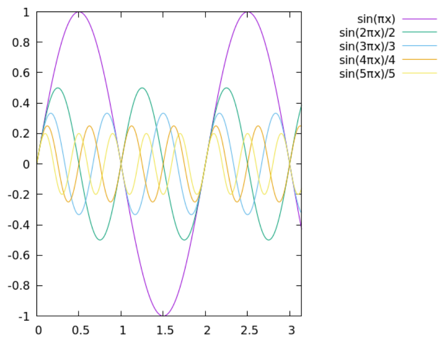
Open script

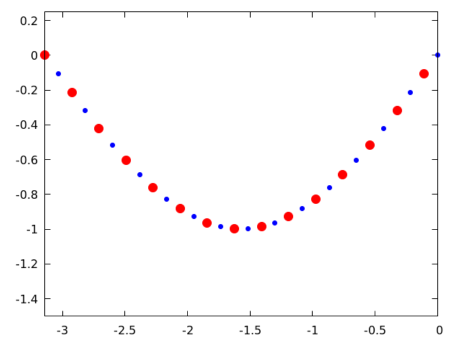


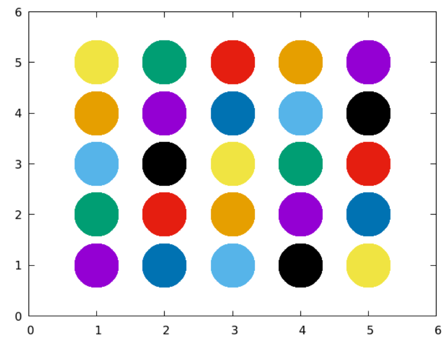
## Index of Plots

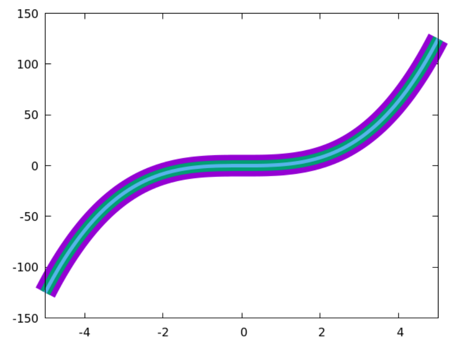


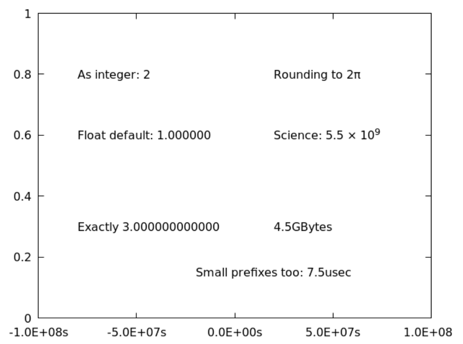


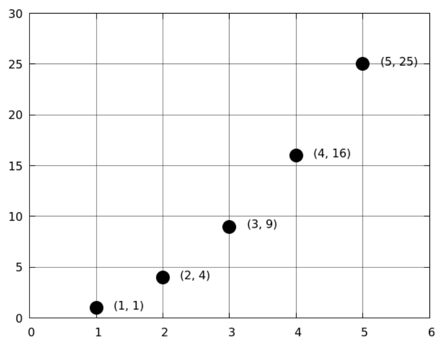


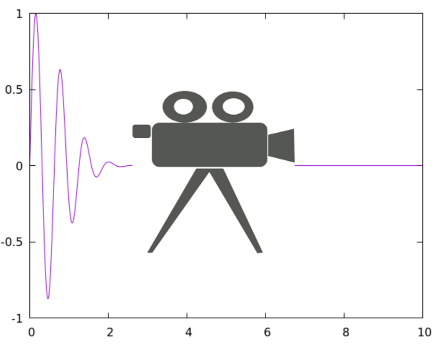


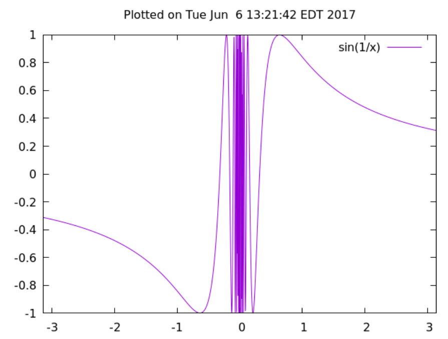


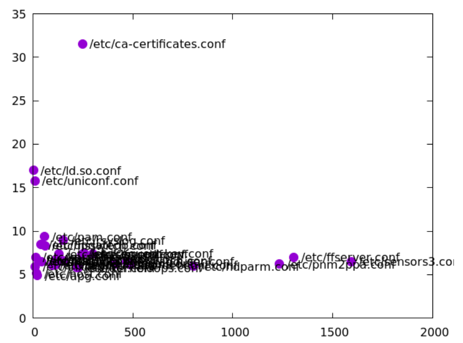


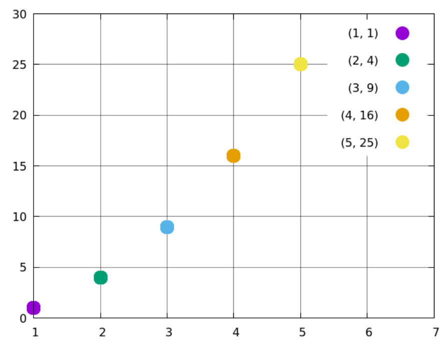


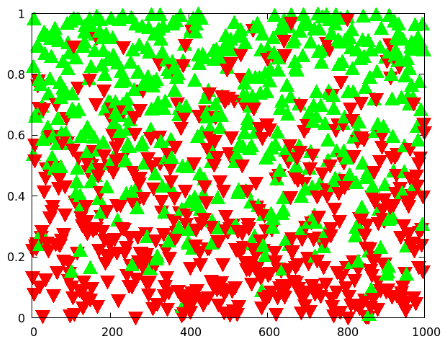


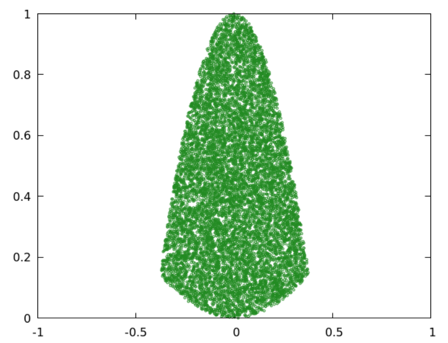












# Index