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.

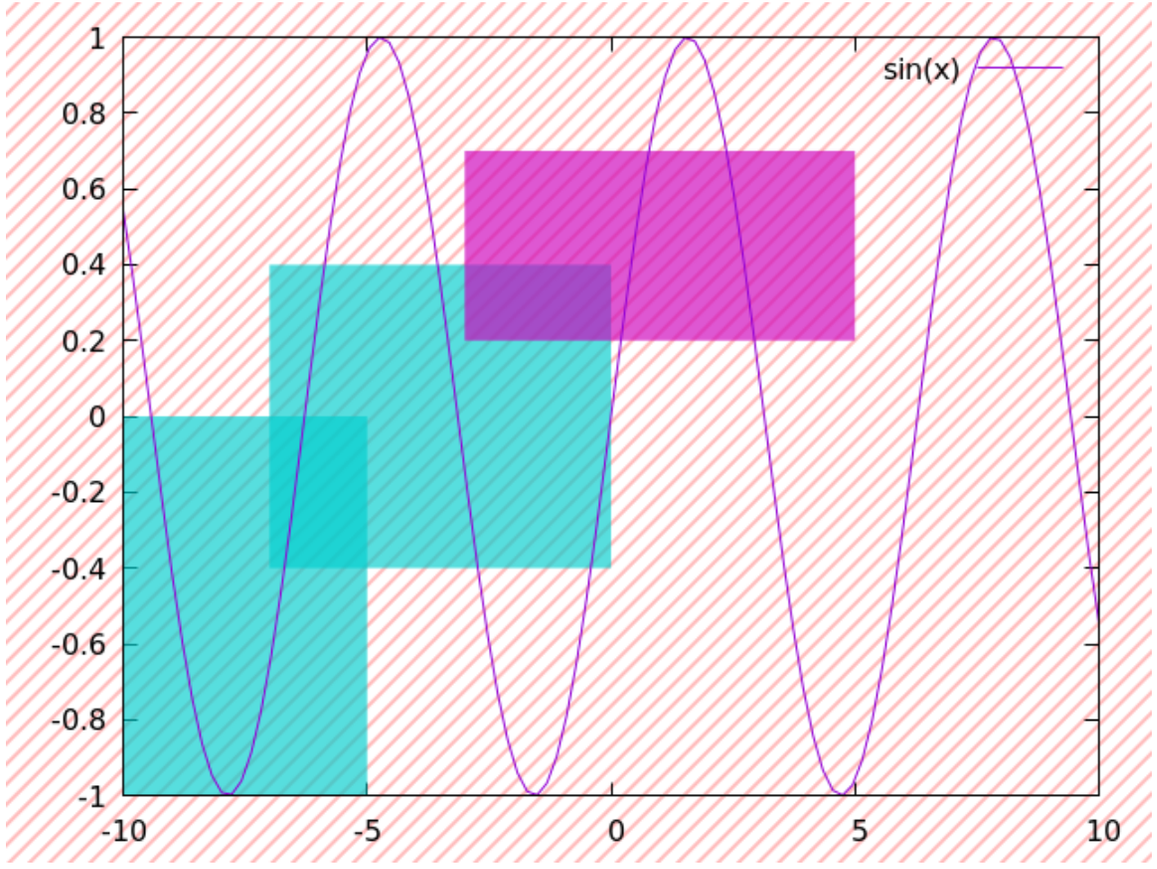
This chapter will also cover gnuplot's *arrows*. Although arrows are most often used in concert with text labels, they fit better in this chapter because they share so much syntax and behavior with objects.

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

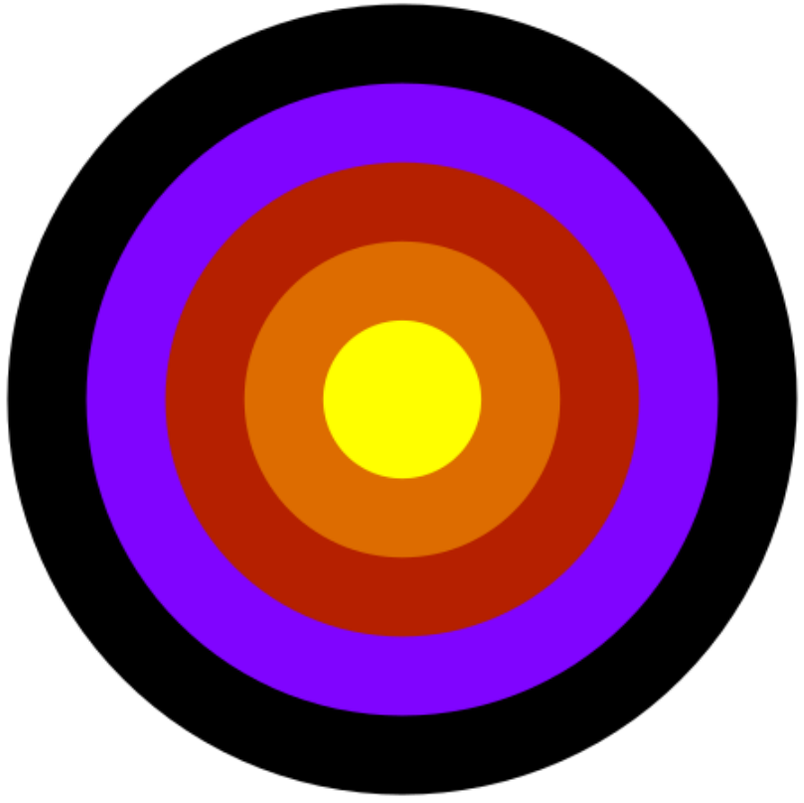


Circles

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
```

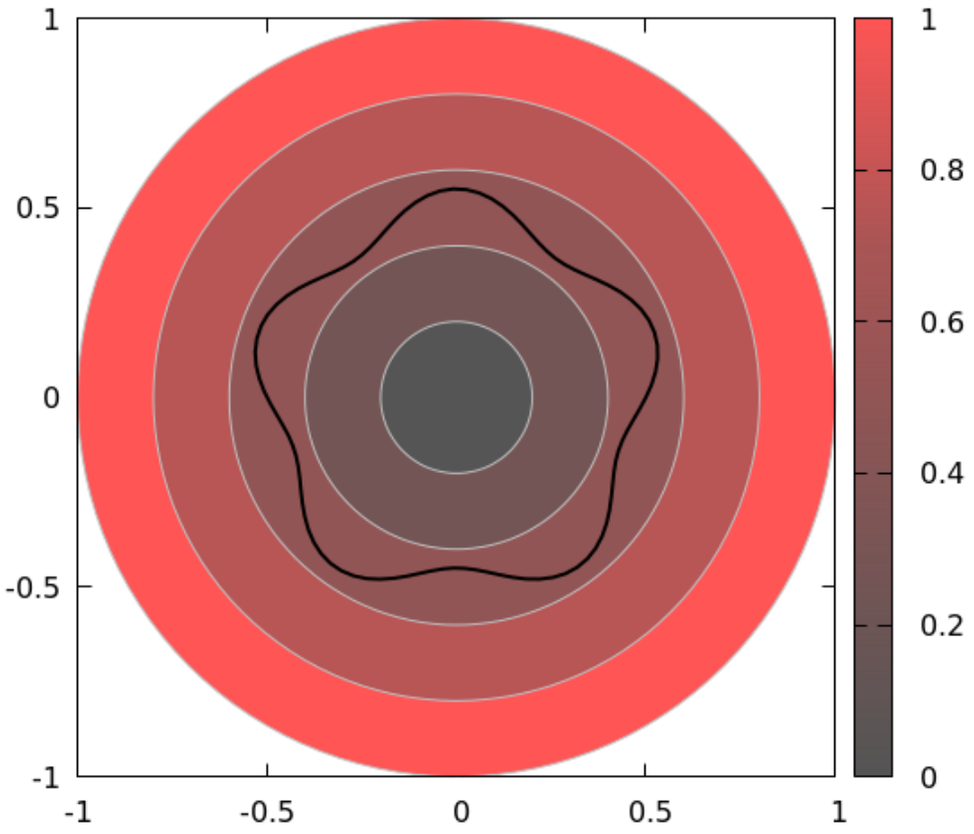
Open script



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.

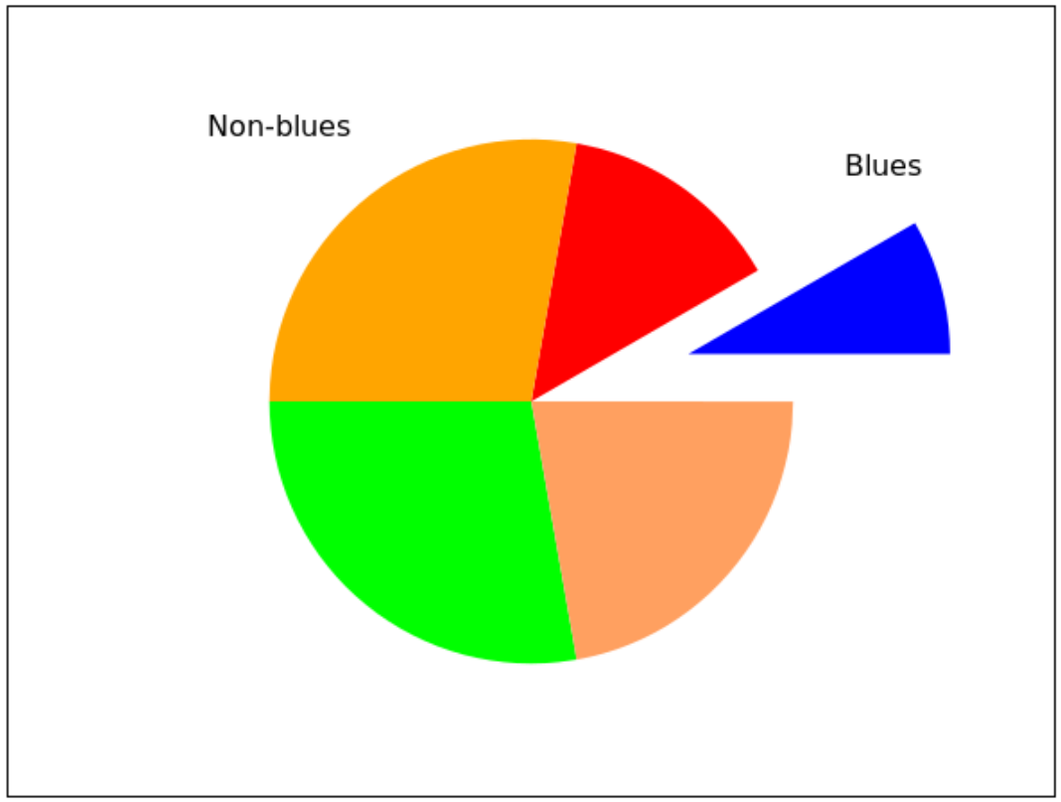
```
unset key
set size square
set lt 7 lc "grey"
set style fill solid 1.0 border lt 7
set xr [-1 : 1]
set yr [-1 : 1]
set pal def (0 "#555555", 0.7 "#aa5555", 1 "#ff5555")
set cbrange [0 : 1]
do for [r = 1 : 5]{
  set obj 6 - r circle center 0, 0 size r/5.0 fc pal frac (r-1)/4.0
}
set polar
unset raxis
plot .5 + .05*sin(5.*t) lc "black" lw 2
```

Open script



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.



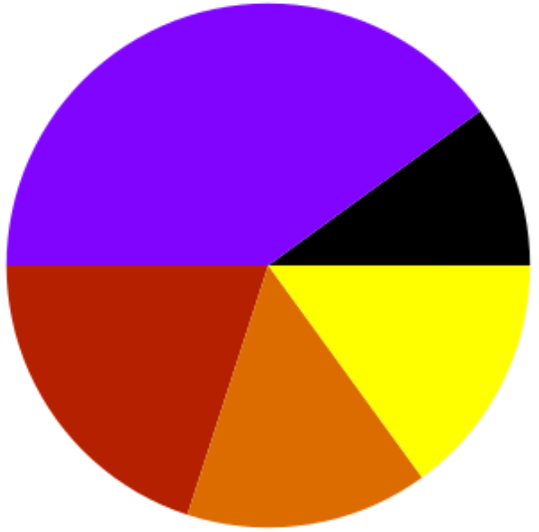

```
unset key
unset tics
set style fill solid 1 noborder
set obj 1 circle at graph .65,.56 size graph .25 fc "blue" arc [0:30]
set obj 2 circle at graph .5,.5 size graph .25 fc "red" arc [30:80]
set obj 3 circle at graph .5,.5 size graph .25 fc "orange" arc [80:180]
set obj 4 circle at graph .5,.5 size 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

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
```

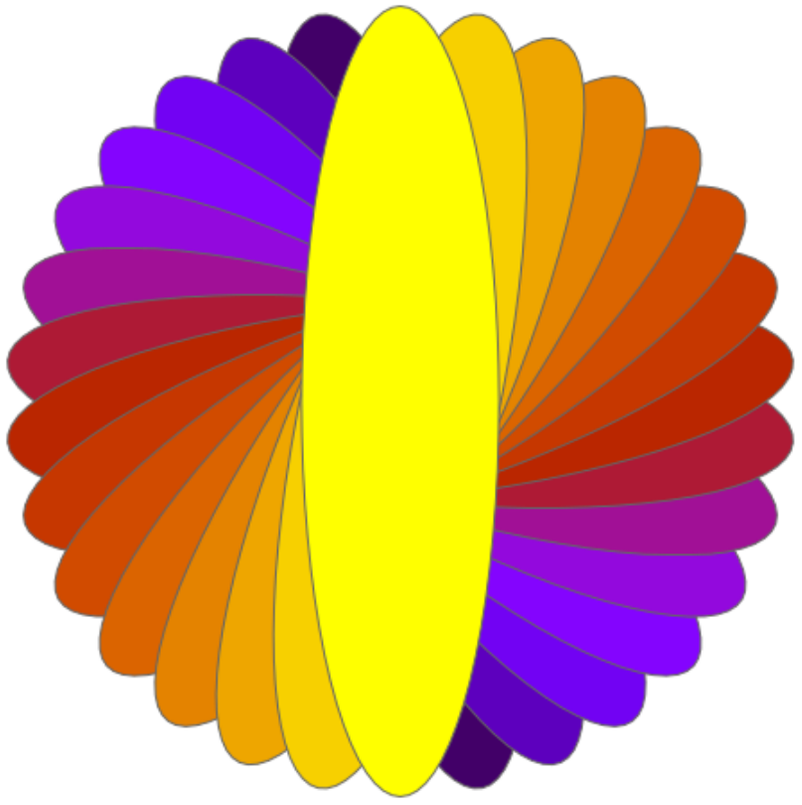
Open file

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

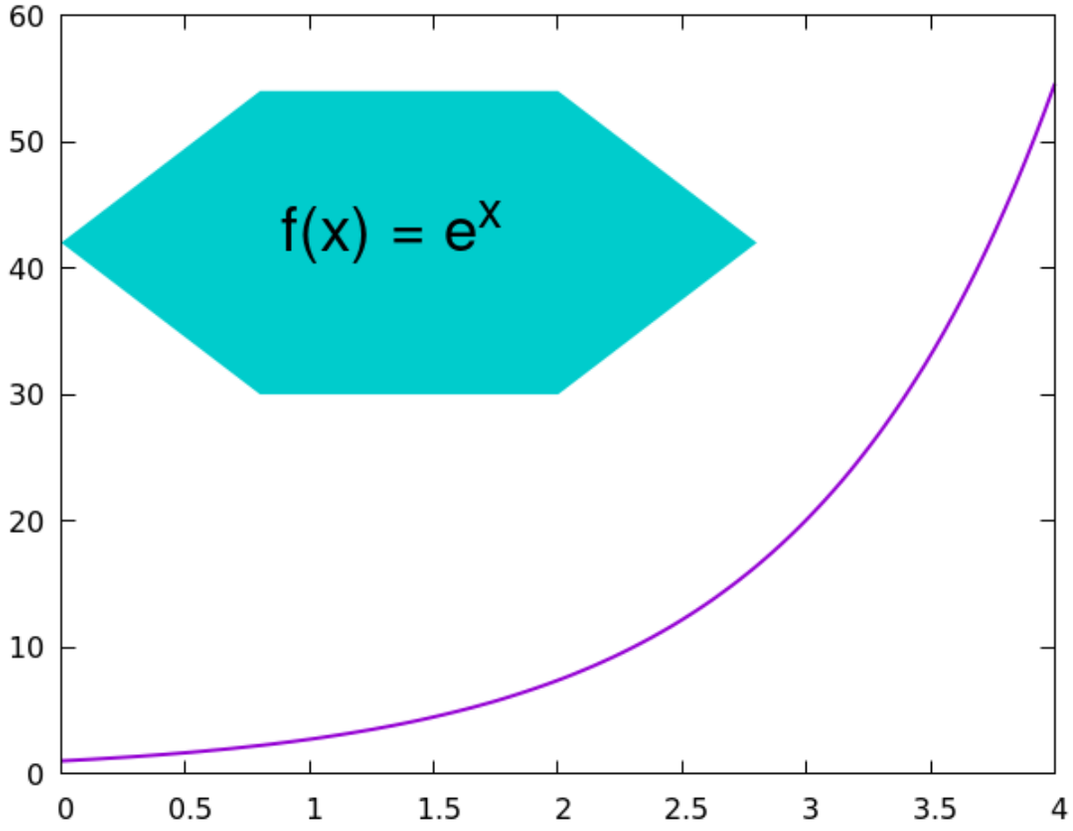


Polygons

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



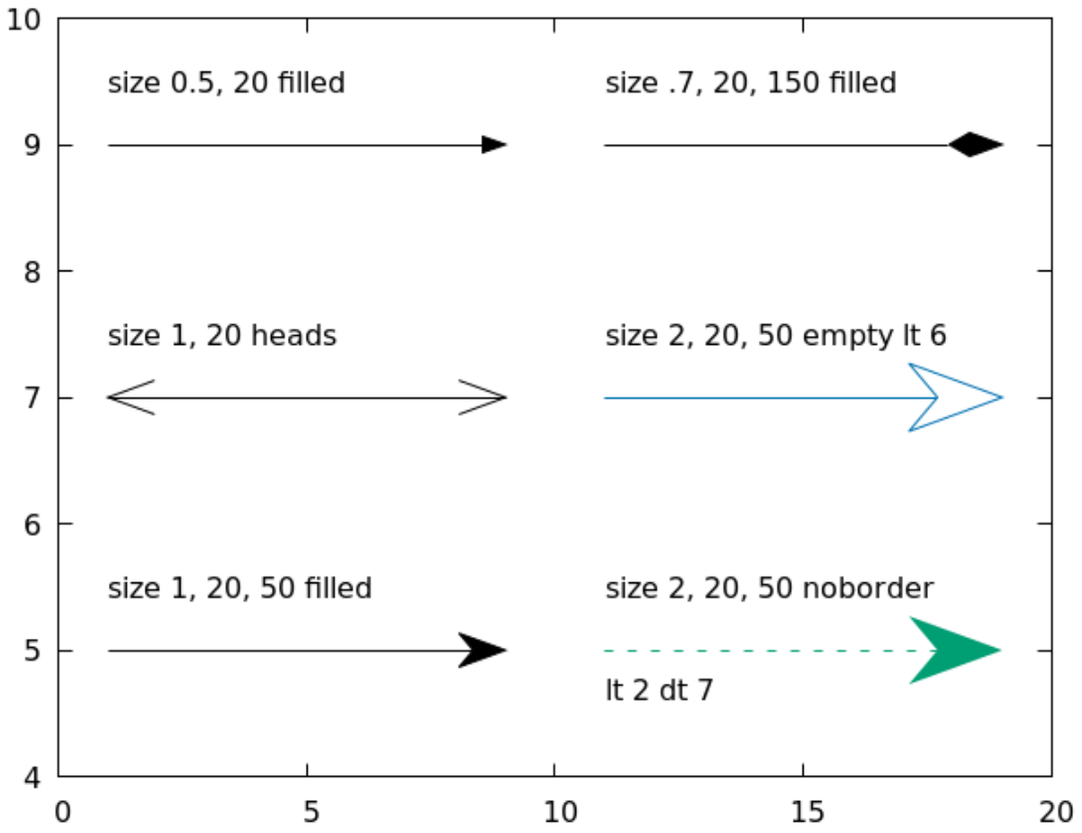
Arrows

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
```

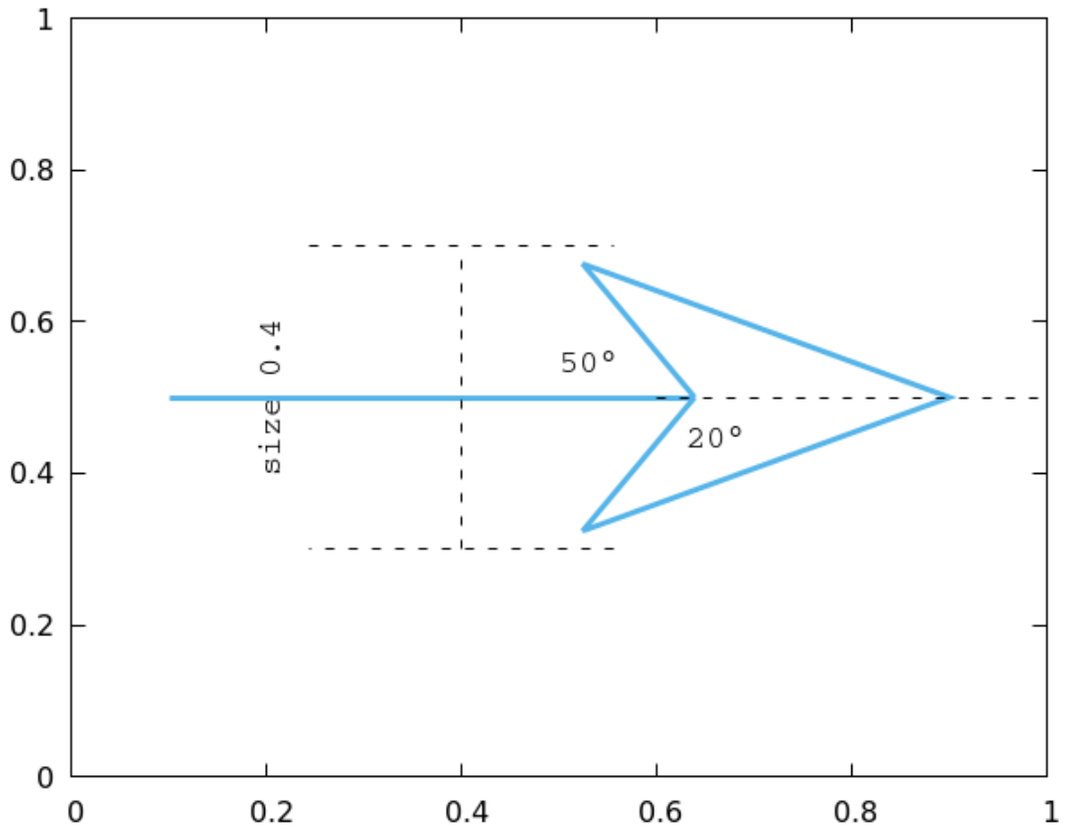
Open script



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

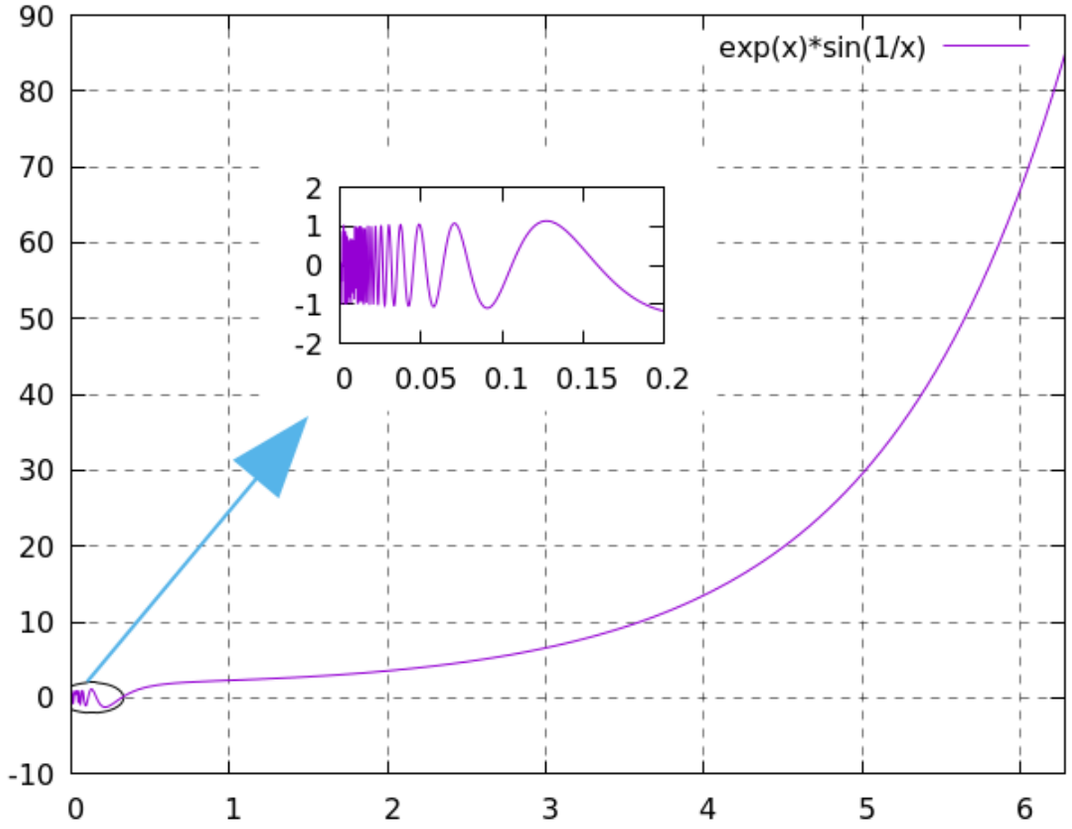


A Better Inset Plot

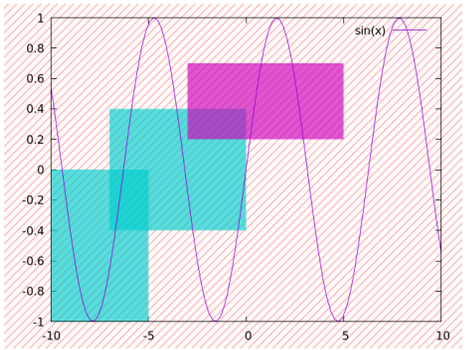
Here we show how to use an object and an arrow to make a better version of the inset plot that we created a couple of chapters ago. This example uses an ellipse to demarcate the area of the plot that we intend to magnify, and an arrow to point from there to the magnified inset.

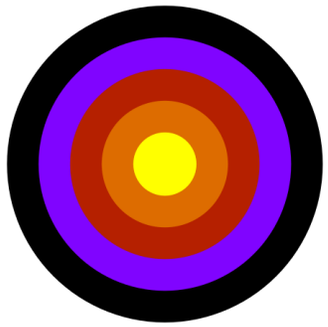
```
set multi
set object ellipse center .13, 0 size .4, 4
set arrow from .1, 2.1 to screen .29, .49 size screen 0.07, 20\
   filled front lt 3 lw 2
set samples 1000
set grid lt -1 dt "_ "
set xtics 1
set ytics 10
plot [0:2*pi] exp(x)*sin(1/x)
set origin .25, .5
set size .4, .3
clear
unset grid
unset object
unset arrow
unset key
set xtics .05
set ytics 1
plot [0:.2] exp(x)*sin(1/x)
```

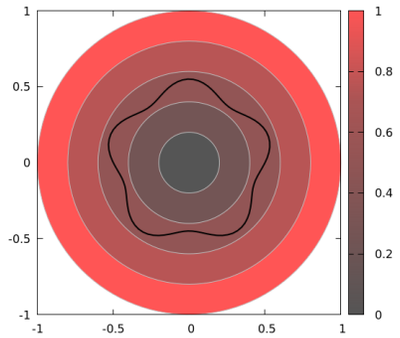
Open script

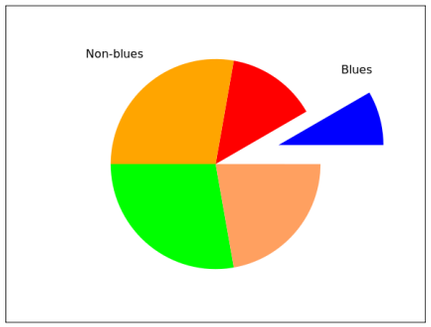


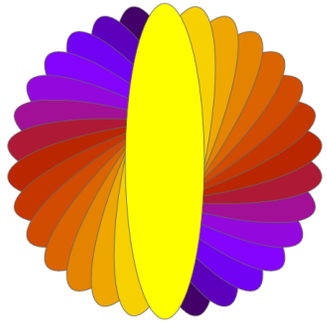
Index of Plots

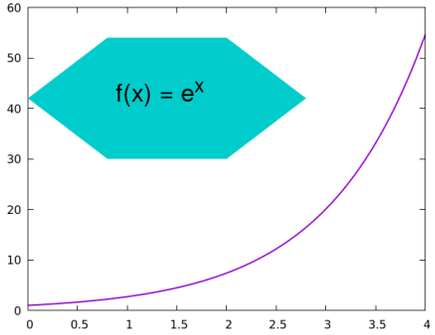


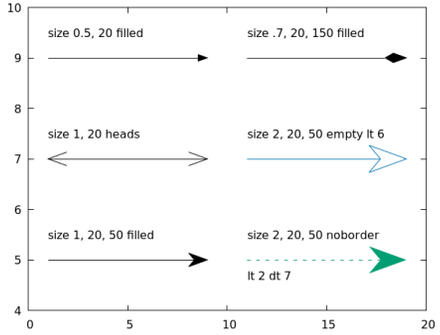


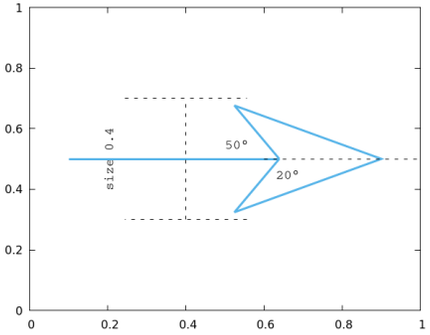


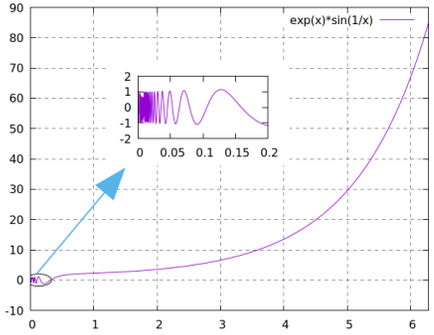












Index