R Programming

Graphics

Graphics

R has extensive graphics facilities. Some examples of high-level graphics functions are:

- Pie Charts, Bar Charts and Histograms
- Box-and-Whisker Plots
- Scatter plots
- Time Series Plots
- Surface Plots

The quality of the graphs produced by R is often cited as a major reason for using it in preference to other statistical software systems.

Low-Level Graphics

The high-level graphics facilities in R are built on a set of flexible low-level ones. This makes it possible to modify existing plot types or to implement completely new ones.

The low level facilities include:

- Page Layout
- Setup of Plotting Coordinates
- Drawing Points and Lines
- Drawing Polygons and Rectangles
- Color Management

Simple Plotting

The `plot` function provides simple graphics facilities. The function call

```
plot(x, y)
```

produces a scatter plot, with the points located at the coordinates in the variables `x`, `y`.

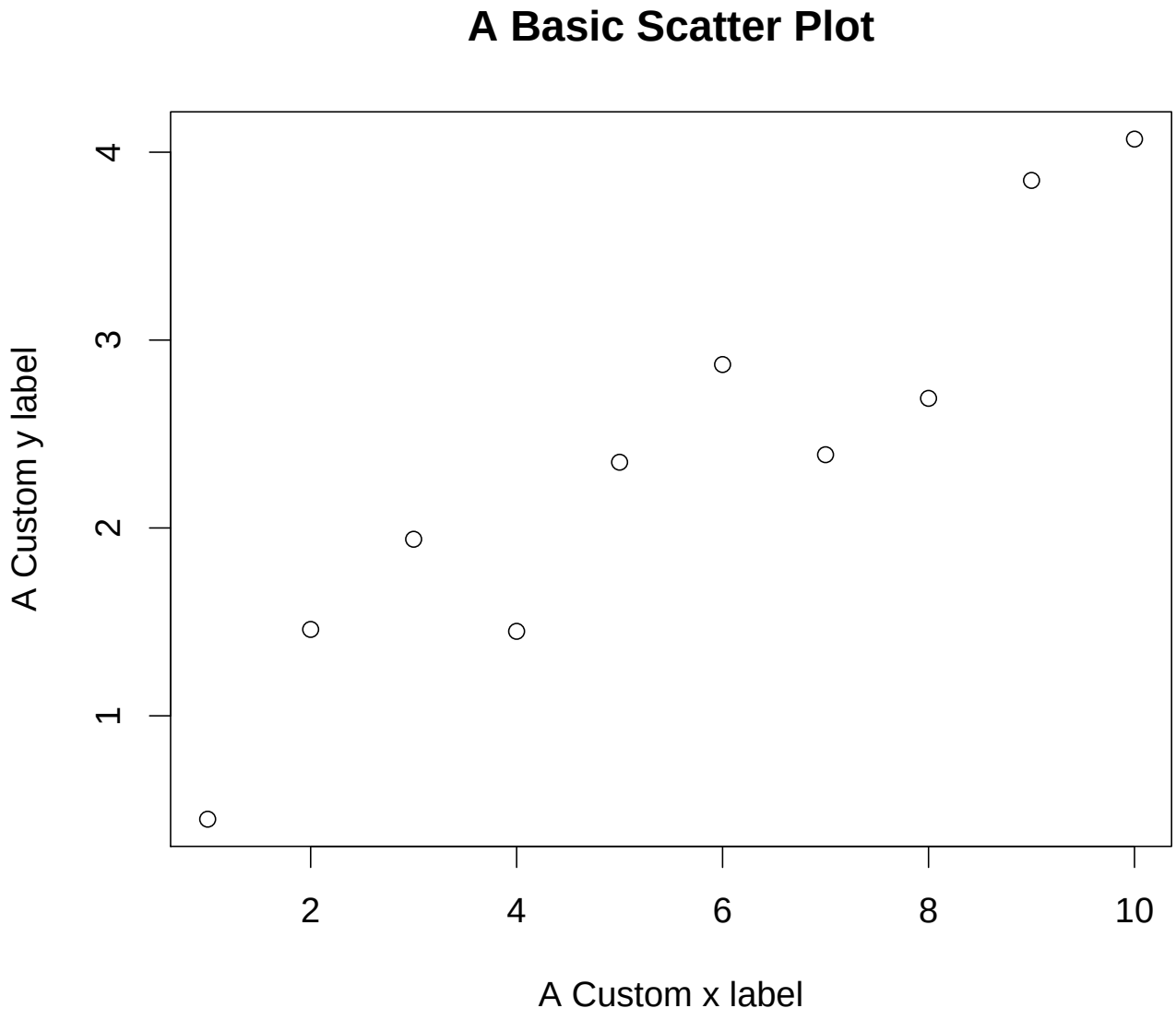
- If only one argument is provided it is used as the `y` variable. The `x` variable is assumed to have the value `1:length(y)`.
- If an additional argument, `type="l"`, is provided in the call, the points are joined with straight lines.

Example

This example show a basic scatterplot with some basic annotation.

```
> x = 1:10
> y = c(0.45, 1.46, 1.94, 1.45, 2.35,
        2.87, 2.39, 2.69, 3.85, 4.07)

> plot(x, y,
      xlab = "A Custom x label",
      ylab = "A Custom y label",
      main = "A Basic Scatter Plot")
```


A Basic Scatter Plot
A Custom y label
A Custom x label
1
2
3
4
2
4
6
8
10

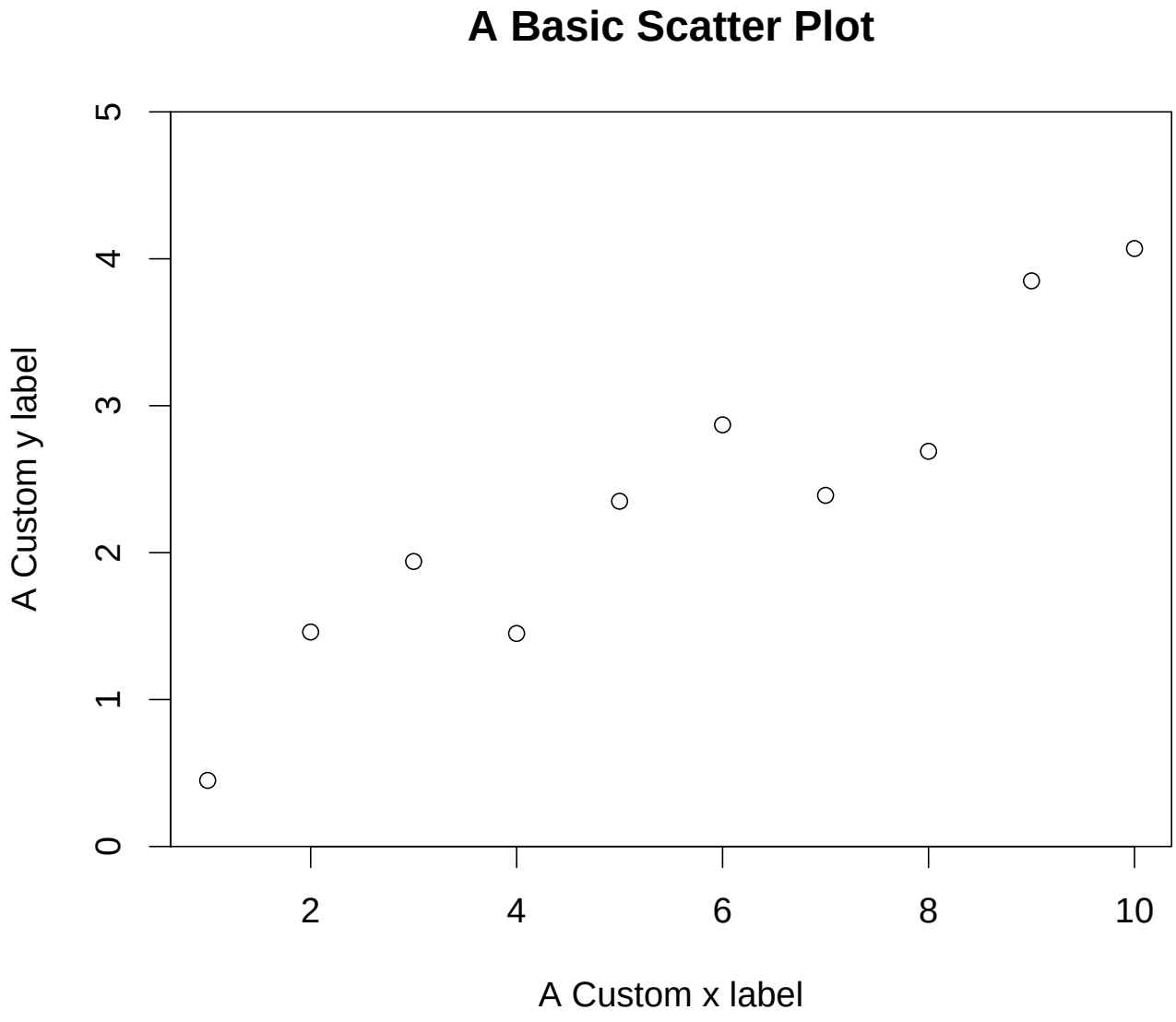
Range Customisation

A number of optional arguments to `plot` can be used to affect the scaling of the plot and the way in which axes appear.

The arguments `xlim` and `ylim` make it possible to specify the x and y ranges to appear in the plot. (The default behaviour is to use the ranges of the values in the `x` and `y` arguments.)

By default the limits of the plots are expanded by 4% at the top and bottom so that plotting symbols do not collide with the box around the plot. This expansion can be (separately) turned off by specifying `xaxs="i"` and `yaxs="i"`.

```
> plot(x, y, ylim = c(0, 5), yaxs = "i",
       xlab = "A Custom x label",
       ylab = "A Custom y label",
       main = "A Basic Scatter Plot")
```

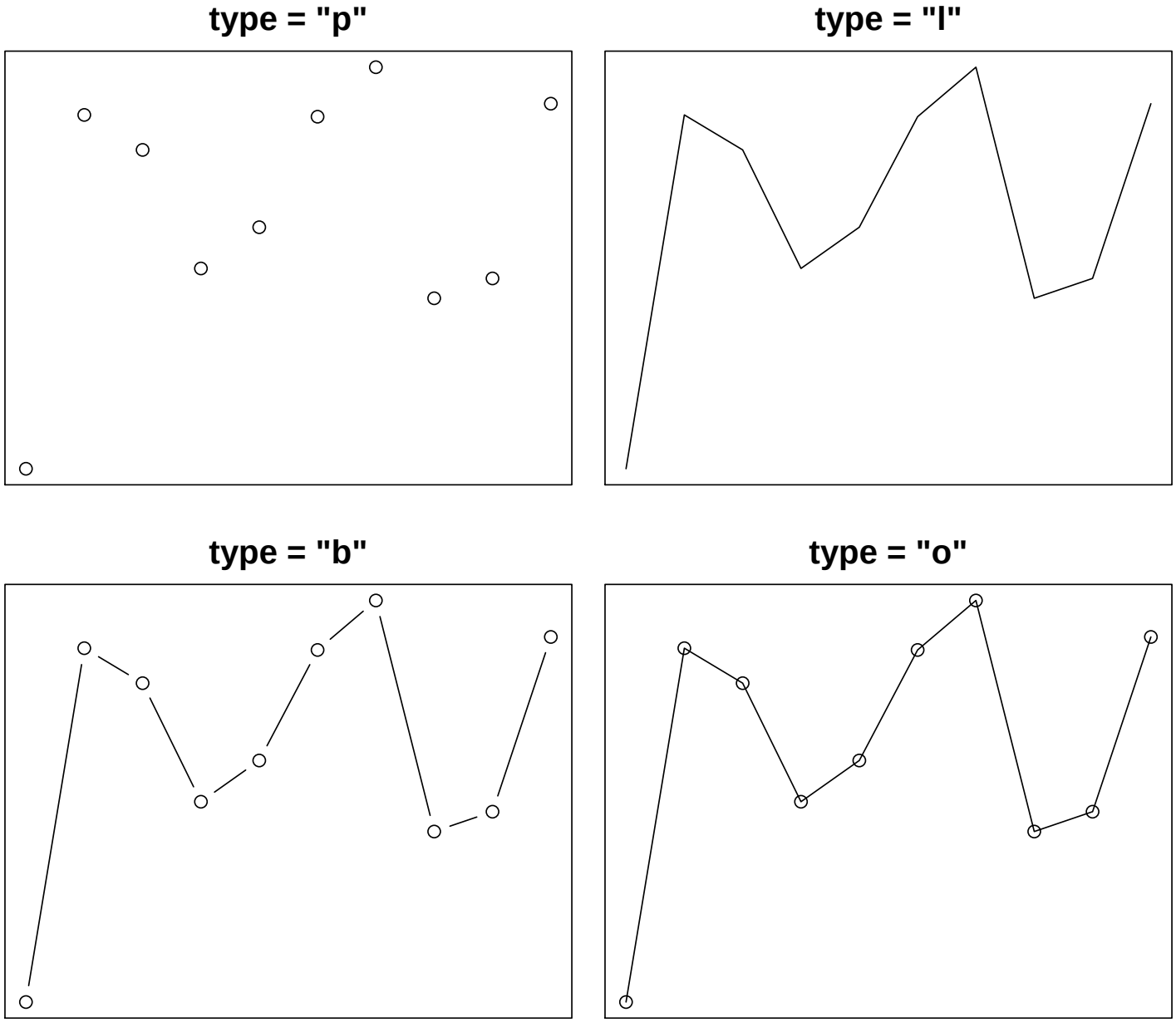

A Basic Scatter Plot
A Custom y label
A Custom x label
0
1
2
3
4
5
2
4
6
8
10

Different Types of Plot

- By default, the plot function produces a scatterplot by drawing plotting symbols at the given (x, y) coordinates.
- The optional `type` argument makes it possible to produce other types of plot by describing their content.
- The plot types are:

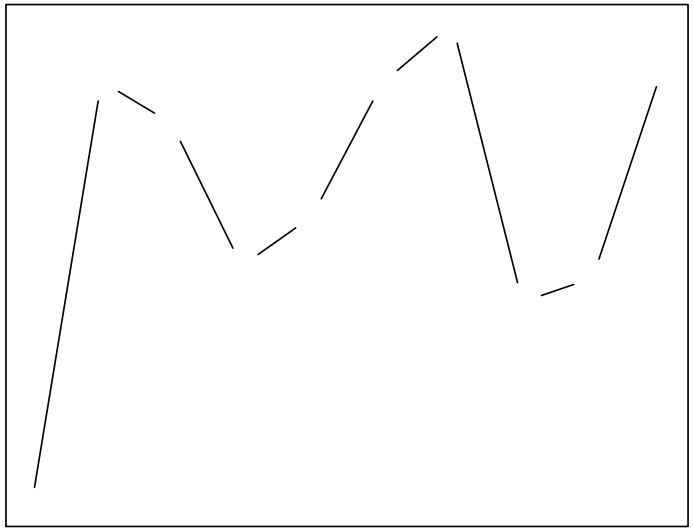
`"p"`	points (i.e. a scatterplot)
`"l"`	lines (i.e. a line plot)
`"b"`	both (points and lines)
`"c"`	just the lines from `type="b"`
`"o"`	points and lines overplotted
`"h"`	high-density needles
`"s"`	step function, horizontal step first
`"S"`	step function, vertical step first
`"n"`	nothing (i.e. no plot contents)

Some plot Types

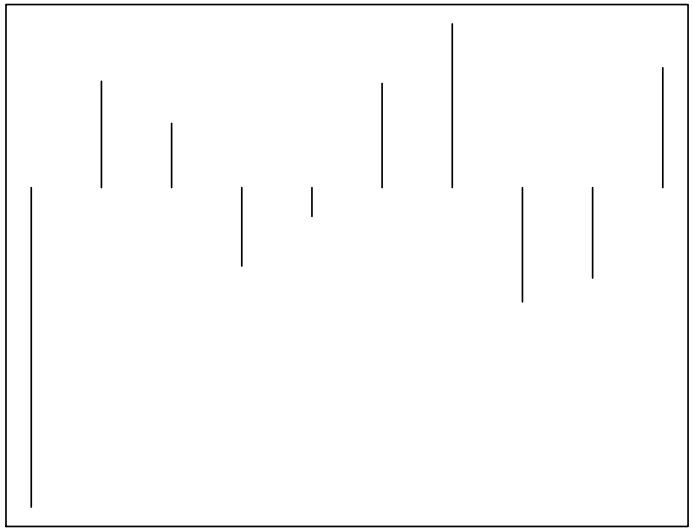


More Plot Types

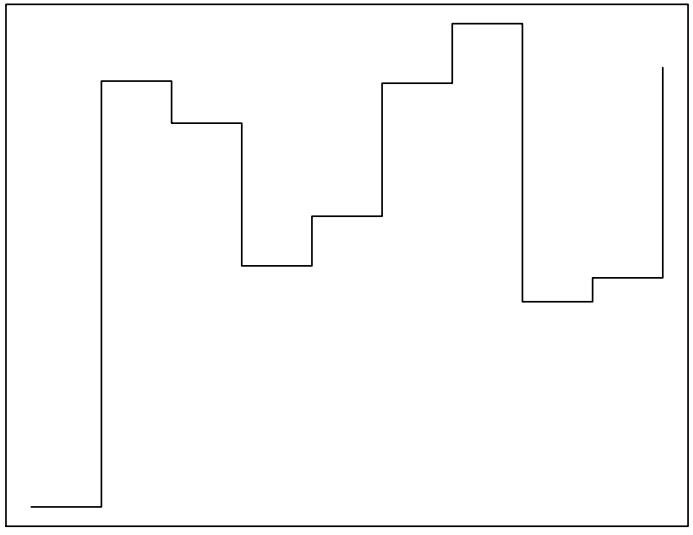
type = "c"



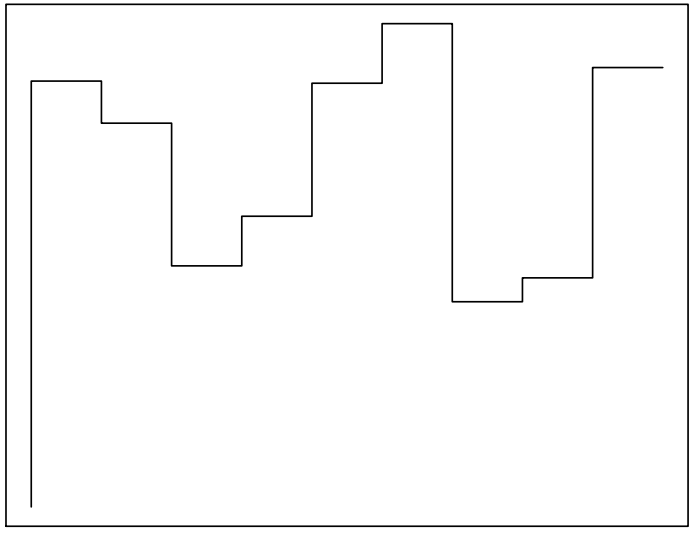
type = "h"



type = "s"



type = "S"



Plotting Details

The `plot` function is and all-in-one function which carries out a number of actions.

- It causes the graphics system to advance to a new "frame."
- It sets up a new plotting area using the current settings for margin size, plot type etc.
- It sets up the transformations which determine how data units are mapped to the plotting area.
- It plots the content of the graph (points, lines etc.).
- It plots x and y axes and box around the plot and produces the axis labels and title.

Low-Level Graphics

The `plot` function produces an entire graph with a single function call.

It can be useful to access graphics functionality at a lower level so that it is possible to create graphs in a much more flexible way.

The functions `plot.new` and `plot.window` are the functions that make it possible to work in this low-level way.

`plot.new` is used to begin a new plot and `plot.window` is used to set up coordinate systems.

Neither function does any actual drawing.

Assembling Plots From Components

Here is how the plot produced by the previous example can be produced using `plot.new`, `plot.window` and other low-level graphics functions.

```
> plot.new()
> plot.window(xlim = range(x), ylim = c(0, 6))
> points(x, y)
> axis(1)
> axis(2)
> box()
> title(xlab = "A Custom x label")
> title(ylab = "A Custom y label")
```

Understanding how to produce plots in this step-by-step way makes it very easy to produce custom displays.

Example

Consider how you would produce a plot of the function

$$f(x) = x(2x-3)(2x+3)$$

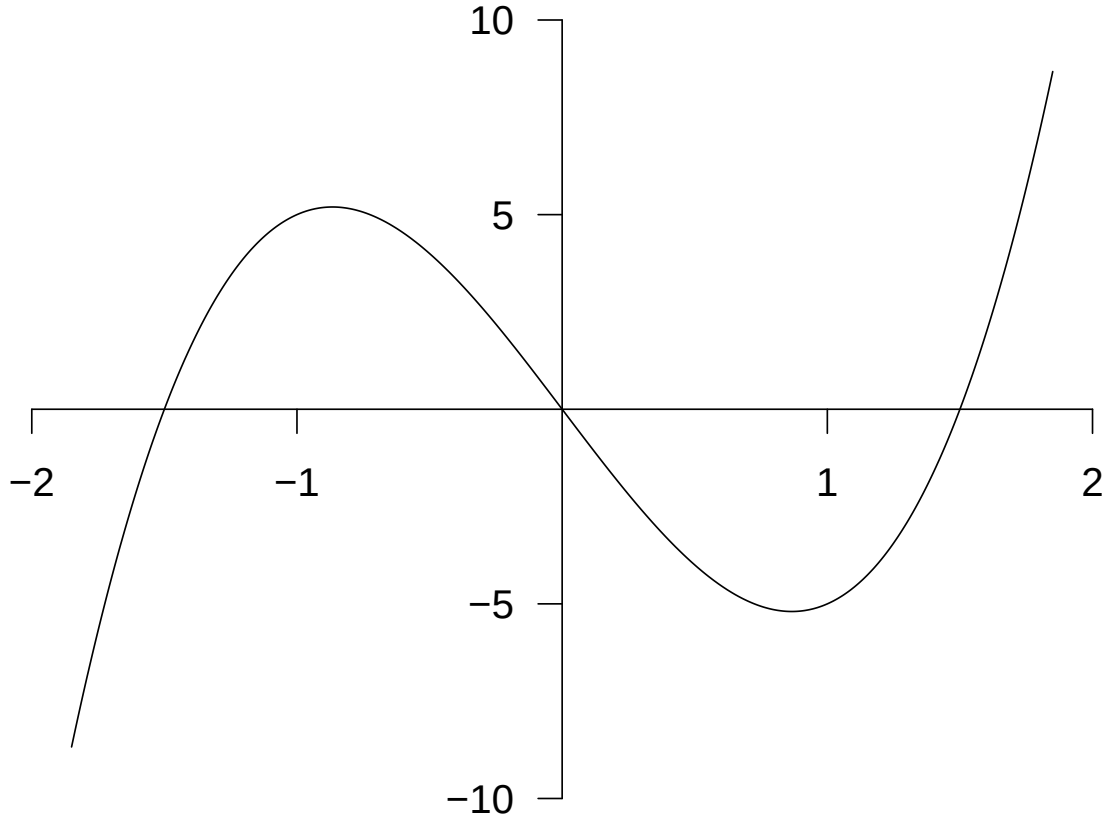
as you might find it in a a calculus book.

The requirements for such are graph are:

(i) Axes which cross at $(0,0)$ rather than lying at the sides of the plotting region.

(ii) Tickmarks chosen to avoid the path of the graph.

(iii) Tick labels which are aligned horizontally on each axis.



A Cubic Polynomial

Axis Customisation

Custom displays often require customised axes. The `axis` function has a number of arguments that enable customisation.

```
axis(side, at, labels, las)
```

`side` : Indicates which side of the graph the axis is to appear: `1` = below, `2` = left, `3` = top, `4`=right.

`at` : A vector of values giving the locations of the tickmarks. (The default is a set of "pretty" values spaced across the axis range.)

`labels` : A vector of labels to appear at the tickmarks. (Defaults to the values in `at`.)

`las` : Label "style," `0` = parallel to the axis, `1` = horizontal, `2` = perpendicular to the axis, `3` = vertical.

Axis Positioning

By default an axis is positioned at the edge of the box that surrounds a plot (on the side specified by the `side` argument).

As an alternative it is possible to place it through the actual graph at a specified coordinate (y coordinate for sides 1 and 3, x coordinate for sides 2 and 4).

This is done by specifying a value for the `pos` argument.

R Code To Produce The Graph

We start by producing the the data for the curve. (1001 points may be overkill here.)

```
> x = seq(-1.85, 1.85, length = 1001)
> y = x * (2 * x - 3) * (2 *  x + 3)
```

Now we set up the plot and draw the actual curve.

```
> plot.new()
> plot.window(xlim=c(-2, 2), ylim = c(-10, 10))
> lines(x, y)
```

Finally, we draw the custom axes and annotation.

```
> axis(1, pos = 0, at = c(-2, -1, 1, 2))
> axis(2, pos = 0, at = c(-10, -5, 5, 10),
       las = 1)
> title(xlab = "A Cubic Polynomial")
```

Low-Level Drawing Functions

The following functions provide R's basic drawing facilities.

`points`	Draws points at specified locations in a plot.
`lines`	Draws a connected set of line segments.
`segments`	Draws (multiple) disconnected line segments.
`arrows`	Draws (multiple) arrows.
`rect`	Draws (multiple) rectangles.
`polygon`	Draws a polygon.
`abline`	Draws lines in slope/intercept form.
`text`	Draws text in a plot.

Many statistical displays can be created using just these few primitives.

Drawing Points

The basic function call is

```
> points(x, y)
```

where `x` and `y` contain the coordinates of the points to be plotted.

Other (optional) arguments make it possible to change the appearance of the points.

Things that can be changed are the plotting symbols, the size of the plotting symbols and the colors that are used to draw the symbols.

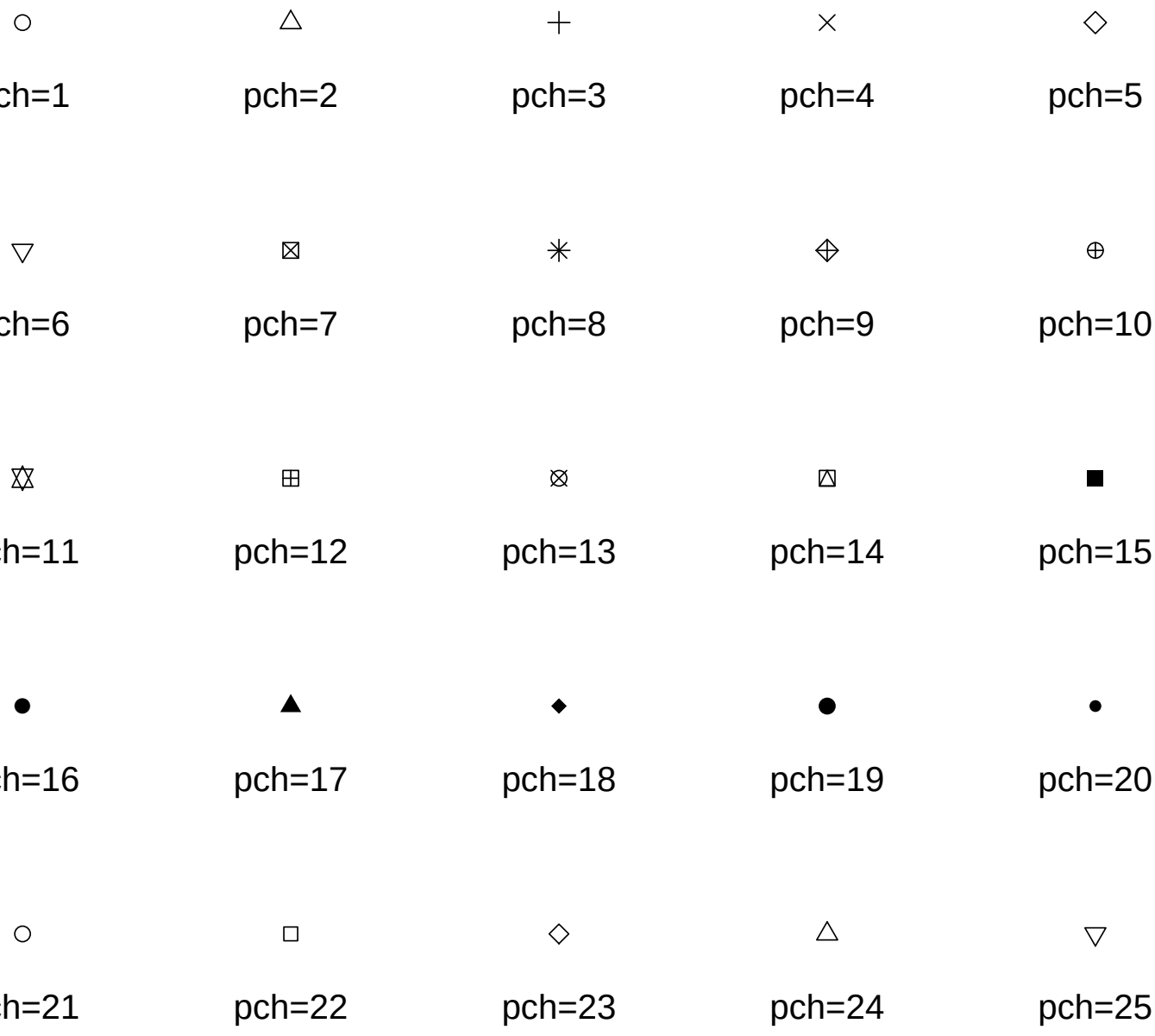
Customising Plotting Symbols

An optional argument called `pch` controls the plotting symbol(s) used by `plot`.

- The argument can be either a single character, or a numerical index into a table of symbols.
- Specifying an index of 0 produces an "invisible" symbol.
- The `pch` argument is recycled to as many values as their are points to be plotted.

The optional argument `cex` can be used to specify a value that magnifies the size of the symbols.

Plotting Symbols

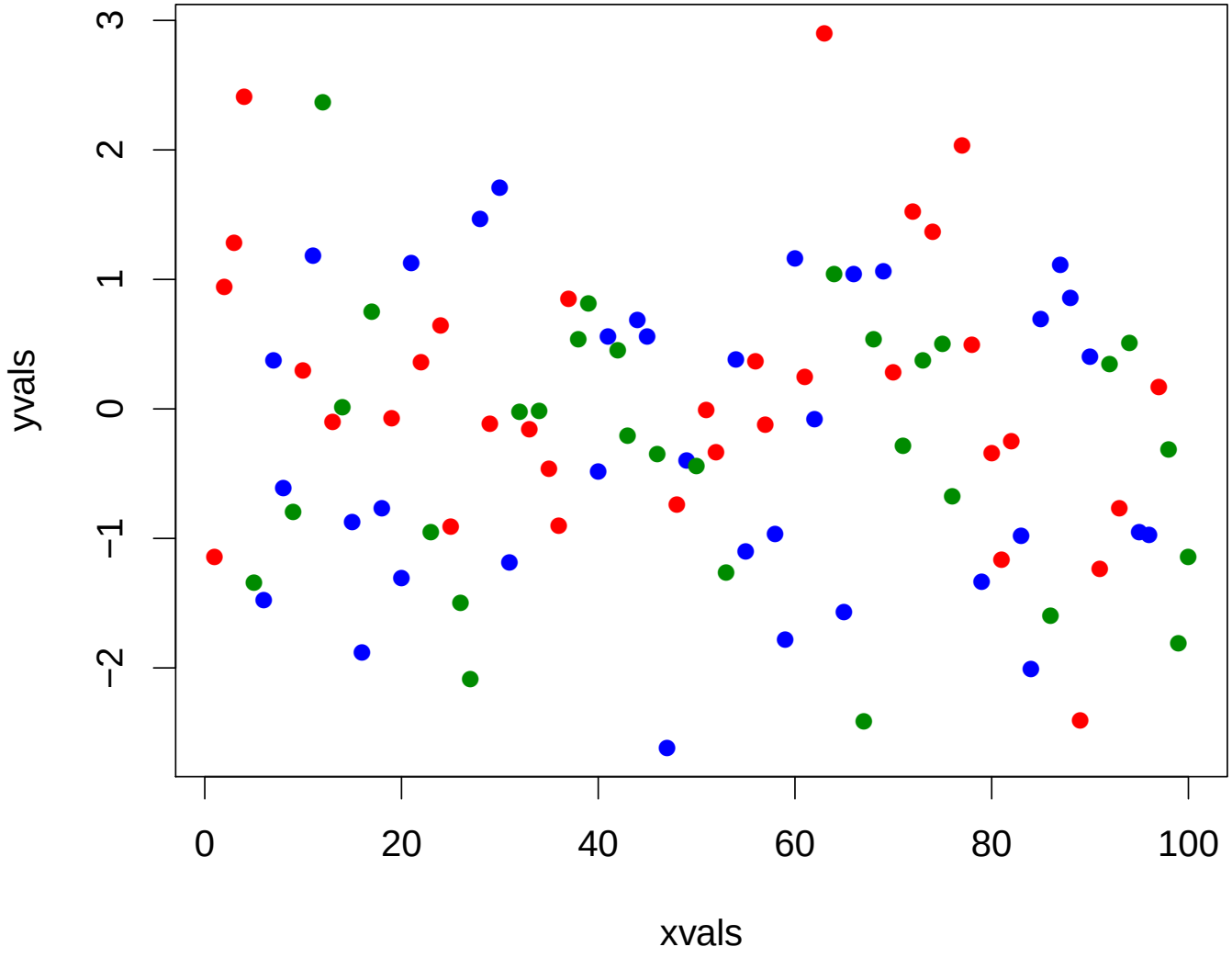


Customising Plotting Symbol Colours

Plotting symbol colours can be customised with the `col` argument.

- Colors can be specified by name with character strings; e.g. `"red"`, `"blue"`, `"brown"`, `"darkseagreen"`, etc.
- A complete list of colour names can be obtained with the expression `colors()` and viewed in the resources section of the class website.

```
> xvals = 1:100
> yvals = rnorm(100)
> cols = sample(c("red", "green4", "blue"),
                length(xvals), replace = TRUE)
> plot(xvals, yvals,
       pch = 16, col = cols)
```


yvals
3
2
1
0
−1
−2
0
20
40
60
80
100
xvals

Drawing Connected Line Segments

The function call

```
> lines(x, y)
```

connects the points whose coordinates are given in `x` and `y` with a series of straight lines.

Other optional arguments make it possible to change the appearance of the lines.

The texture, colour and width of the lines can be be changed.

Note: The `type` argument to the plot function can also be passed to `lines` to generate different plot types. (This also works for `points`. The two functions are really identical).

Line Types

The line *type* can be specified with an argument of the form `lty=`*type*. The line type can be specified by name:

> `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`,
> `"longdash"`, `"twodash"`,

or as a hexadecimal specification of the form `"DUDU"`, where `D` represents a distance drawn with "pen down" and `U` is a distance with "pen down."

`"1111"`	high density dots
`"1313"`	spaced-out dots
`"1333"`	dot-dash
`"3373"`	short-dash, longdash

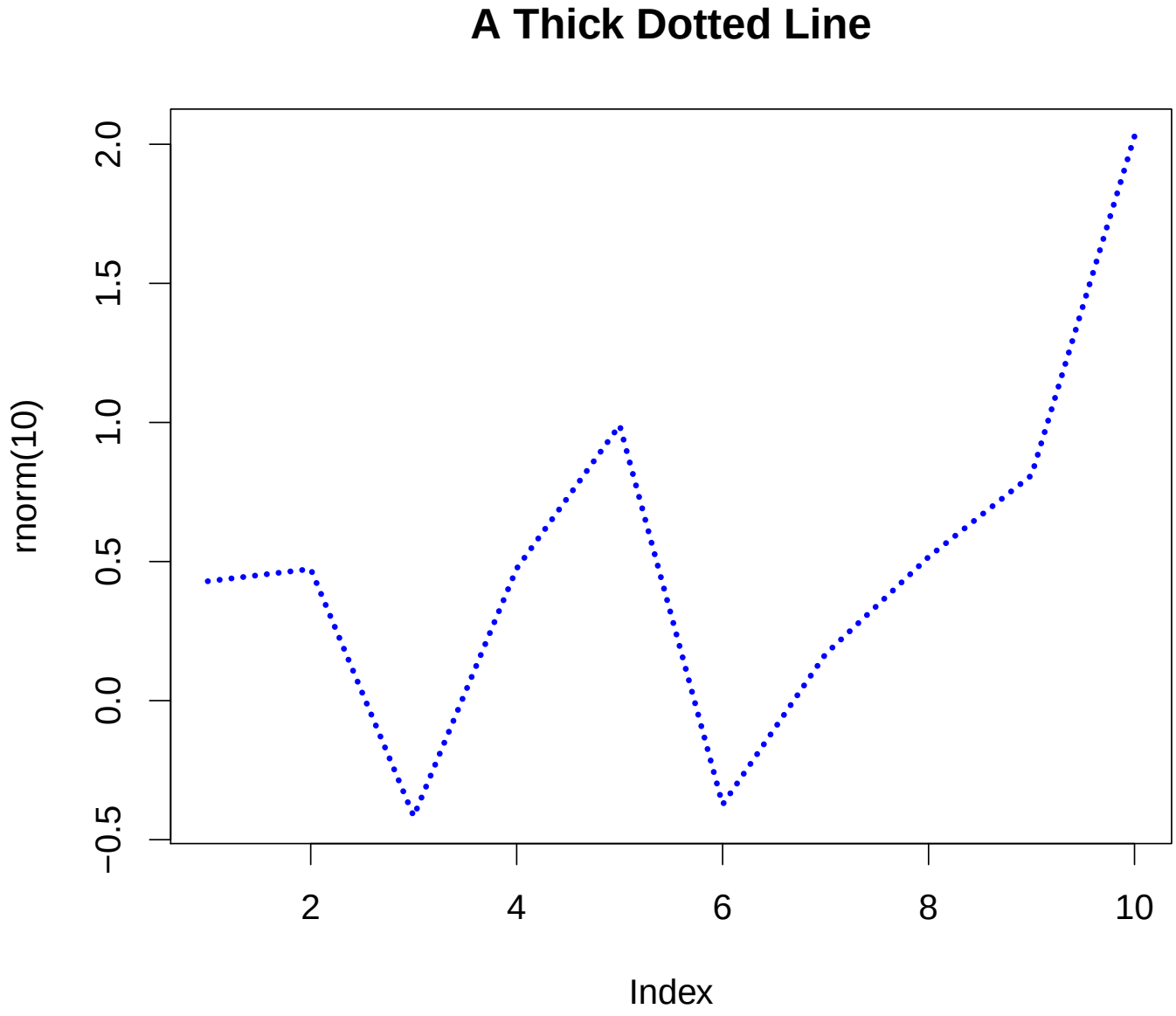
Line thickness can be set with `lwd=`*w*.

Line Type and Thickness Control

The `type`, `lwd`, `lty` and `col` arguments can be used together to generate a variety of line effects.

```
> plot(rnorm(10), type="l",
       lwd=4, lty="1111", col="blue",
       main = "A Thick Dotted Line")
```

Note that in this plot, the size of dot and the gap between them have both been scaled by the value of `lwd`.


A Thick Dotted Line
rnorm(10)
2.0
1.5
1.0
0.5
0.0
−0.5
2
4
6
8
10
Index

Drawing Segments and Arrows

The functions `segments` and `arrows` take four arguments which specify the start and end points of the arrows.

- The functions also accept a colour and line property specification.
- The `arrows` function also accepts arguments which affect the way the arrow heads are shown.

One of the most common uses of `arrows` is for drawing error bars.

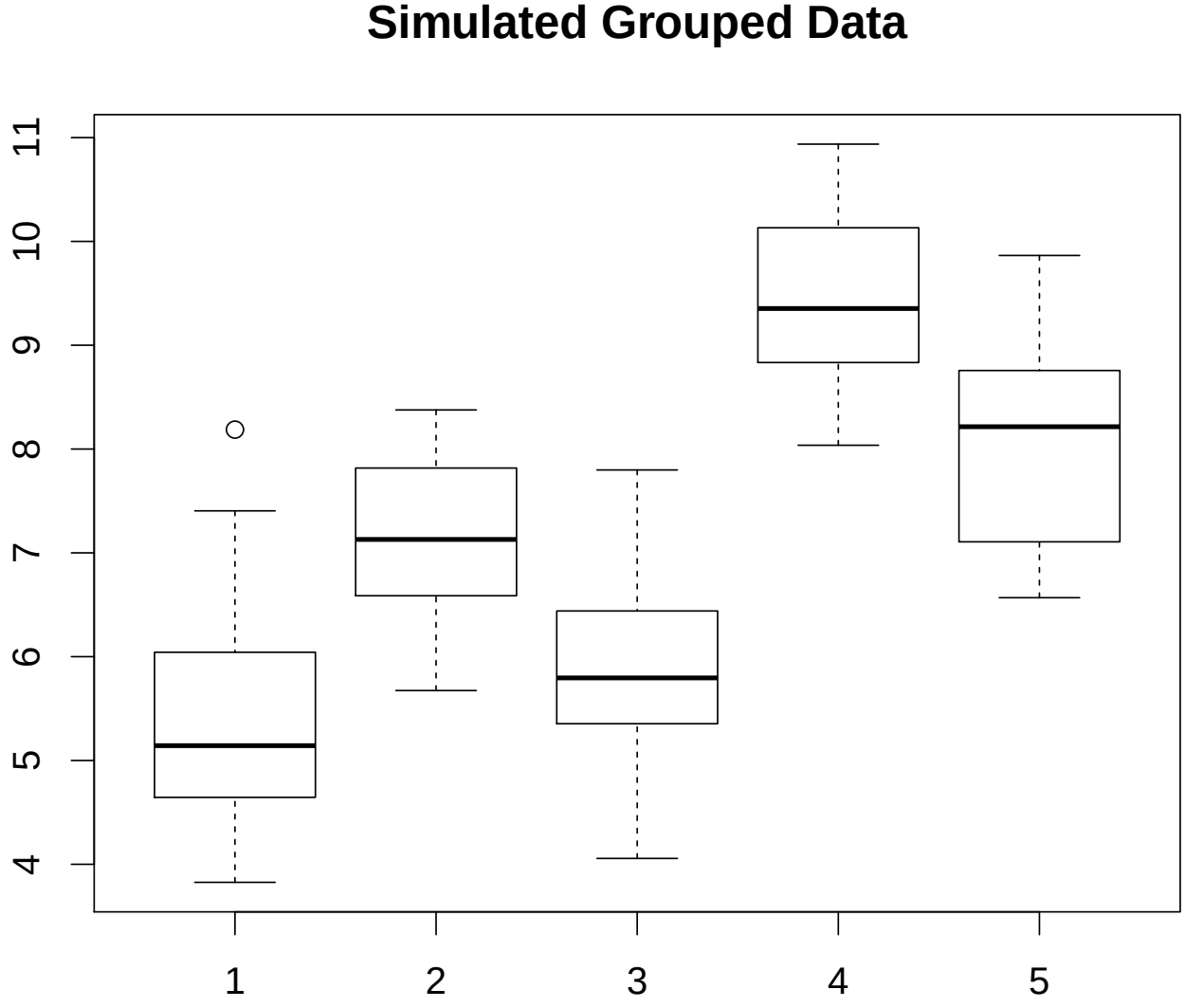
Example: Drawing Error Bars

We'll randomly generate 5 groups of 20 observations each.

```
> groups = rep(1:5, each = 20)
> means = c(5, 7, 6, 9, 8)
> y = rnorm(100) + rep(means, each = 20)
```

One way to display the data is with parallel boxplots.

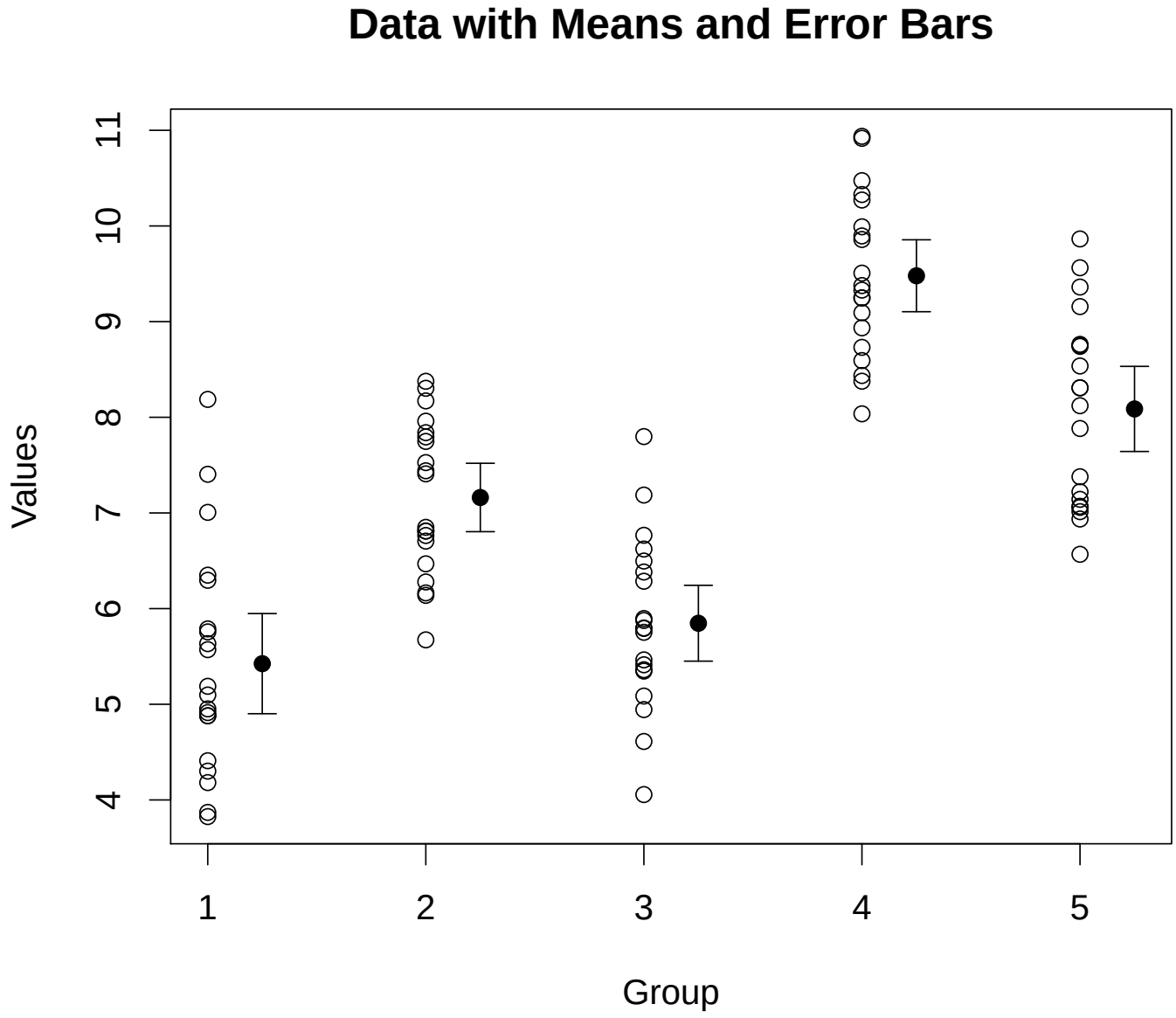
```
> boxplot(y ~ groups,
          main = "Simulated Grouped Data")
```


Simulated Grouped Data
4
5
6
7
8
9
10
11
1
2
3
4
5

An Alternative Display

Rather than using the canned `boxplot` display, we can create a new display that plots the data values and shows the mean with error bars.

```
> ybar = sapply(split(y, groups), mean)
> ysd = sapply(split(y, groups), sd)
> plot(groups, y, xlim = c(1, 5.25),
       xlab = "Group", ylab = "Values",
       main = "Data with Means and Error Bars")
> points(1:5 + .25, ybar, pch=19)
> arrows(1:5 + .25, ybar - 2 * ysd/sqrt(20),
         1:5 + .25, ybar + 2 * ysd/sqrt(20),
         code = 3, angle = 90, length = .1)
```

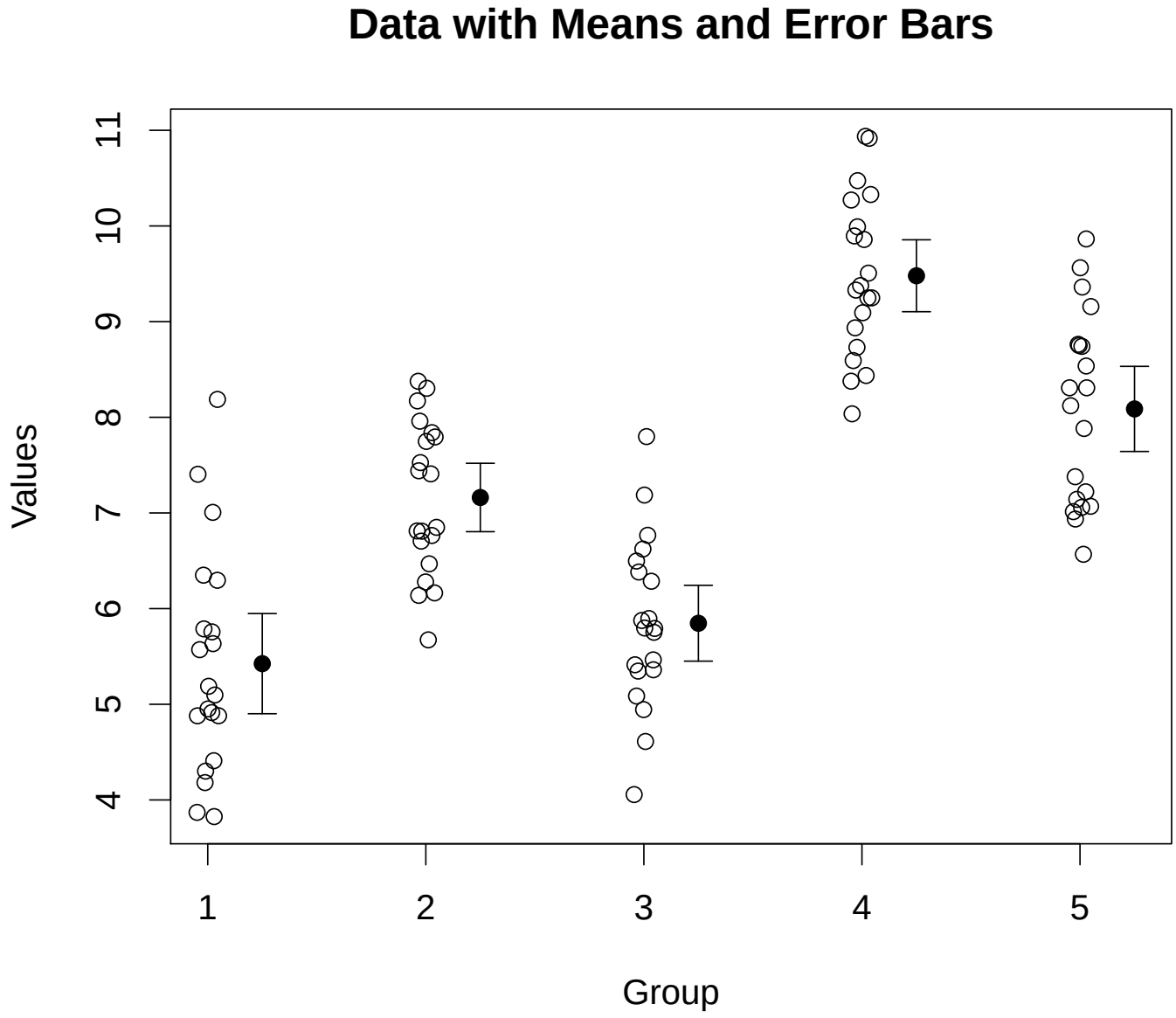

Data with Means and Error Bars
Values
4
5
6
7
8
9
10
11
1
2
3
4
5
Group

Jittering the Points

It is hard to see the individual data points because of the amount of overplotting.

This can be fixed (adding a small amount of random variation in the *x* direction).

```
> plot(groups + runif(100, -.05, .05), y,
       xlim = c(1, 5.25),
       xlab = "Group", ylab = "Values",
       main = "Data with Means and Error Bars")
> points(1:5 + .25, ybar, pch=19)
> arrows(1:5 + .25, ybar - 2 * ysd/sqrt(20),
         1:5 + .25, ybar + 2 * ysd/sqrt(20),
         code = 3, angle = 90, length = .1)
```

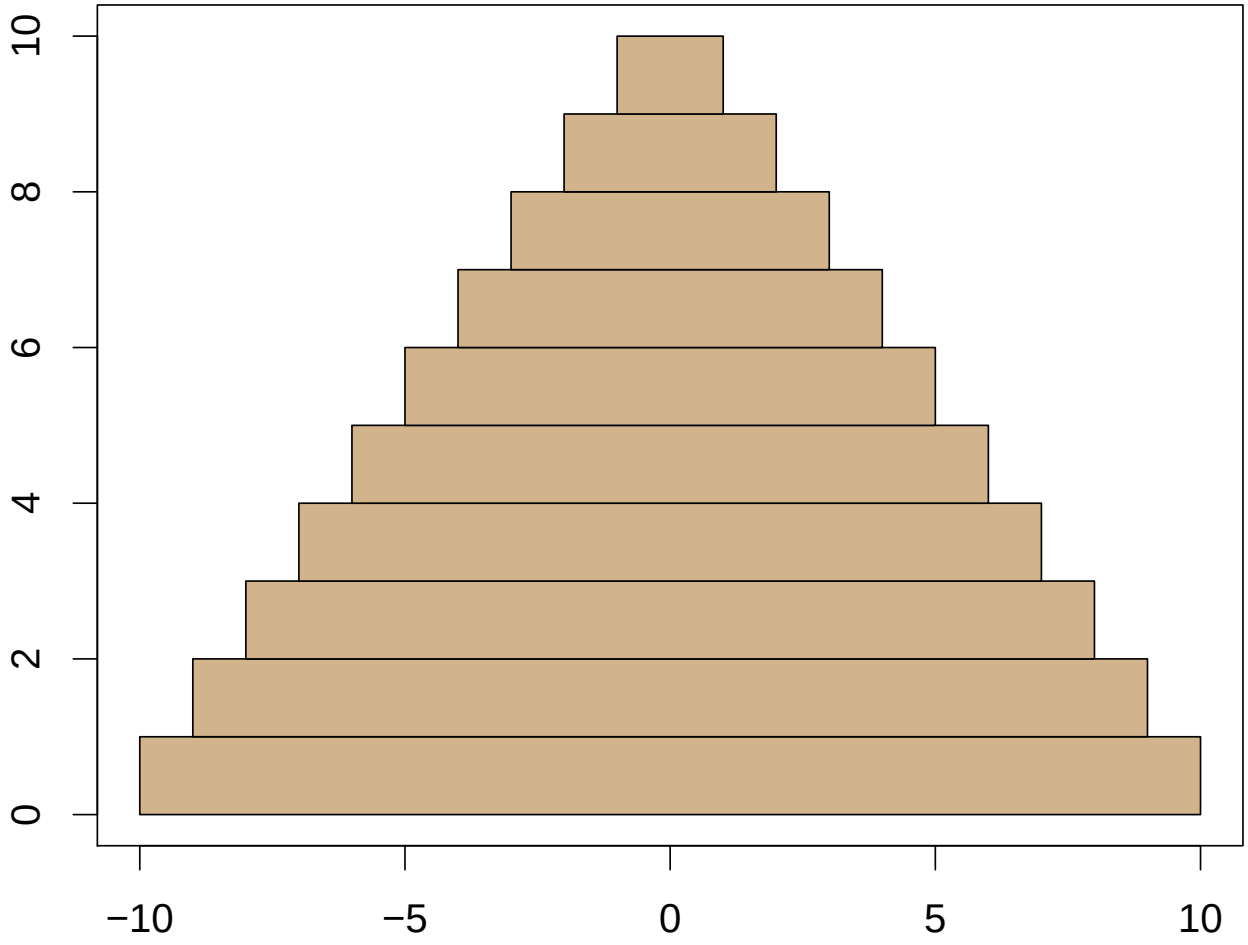

Data with Means and Error Bars
Values
4
5
6
7
8
9
10
11
1
2
3
4
5
Group

Area Fill Primitives

The graphical primitives `rect` and `polygon` can be used to draw coloured areas.

- The functions accept arguments `col` and `border` which can be used to specify the interior and border colours of the regions.
- The functions also accept texture specifications for the border lines via `lty` and `lwd` arguments.
- The `rect` function expects arguments giving the locations of diagonally opposite sides of the rectangle (i.e there are 4 arguments, x_0, y_0 and x_1, y_1).
- The `polygon` function expects two arguments giving the x and y coordinates of the polygon vertices.

A Display Created With Rectangles



Creating the Display

The figure at the center of the array is drawn with a single call to `rect`. It creates 10 rectangles, going bottom to top.

The full set of rectangles is drawn by the code

```
xl = -(10:1)
xr =   10:1
yb =    0:9
yt =    1:10
rect(xl, yb, xr, yt, col = "tan")
```

To draw the complete plot, all that needs to be done is to set up the plot with `plot` and then to add the rectangles.

Drawing Rectangles

The following code draws the full display.

```
> plot(c(-10, 10), c(0, 10), type = "n",
       ylab = "", xlab = "")
> xl = -(10:1)
> xr =   10:1
> yb =    0:9
> yt =    1:10
> rect(xl, yb, xr, yt, col = "tan")
```

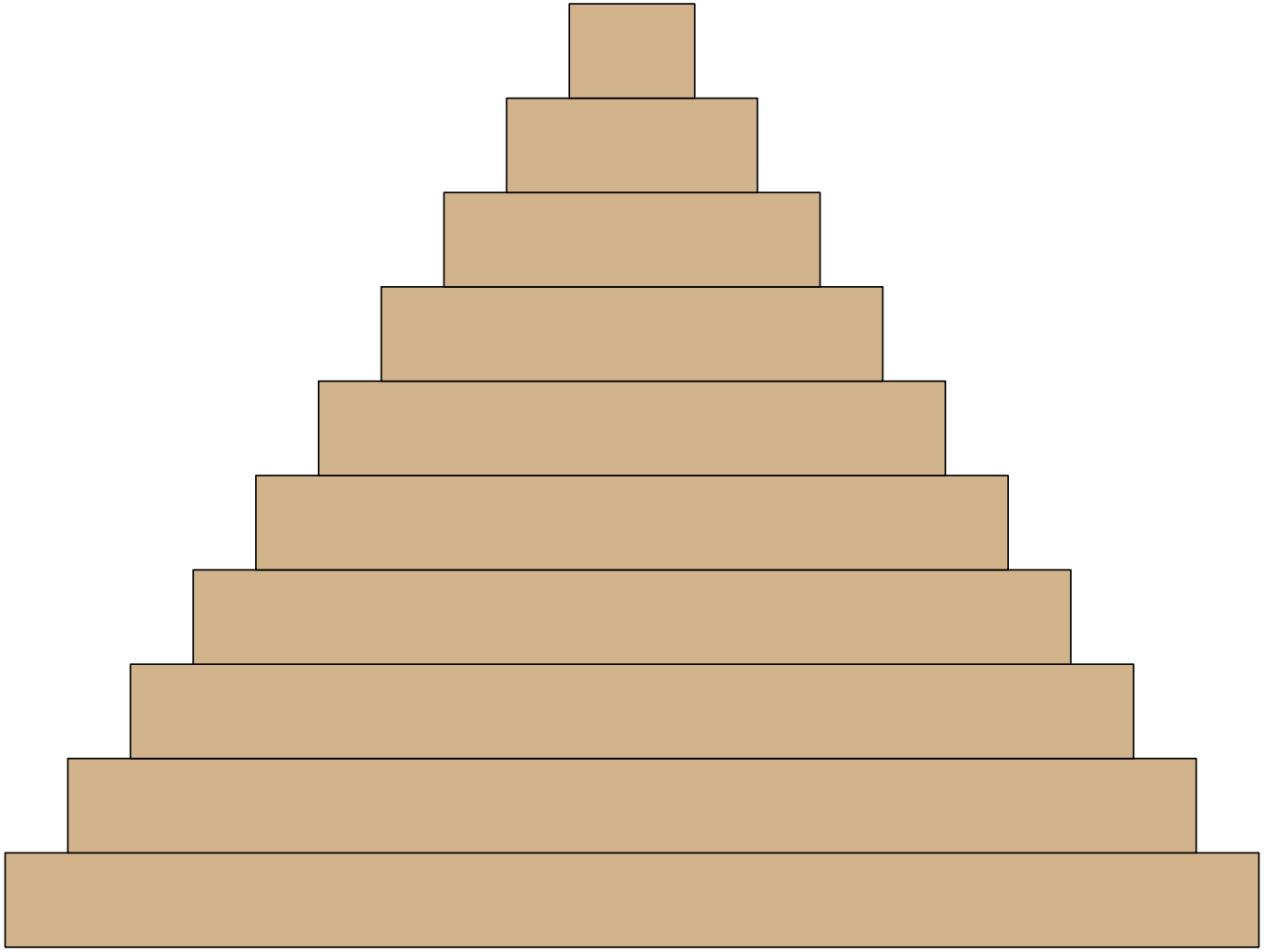
Dropping Axes from a Plot

The plot can be improved by dropping the axes by specifying `axes=FALSE` and adding a title.

The code can also be shortened by inlining the corner coordinates.

```
> par(mar = c(0, 0, 4.1, 0))
> plot(c(-10, 10), c(0, 10), type = "n",
       axes = FALSE,
       ylab = "", xlab = "",
       main = "The Great Ziggurat of Ahr")
> rect(-(10:1), 0:9, 10:1, 1:10, col = "tan")
```

The Great Ziggurat of Ahr



Polygons

Polygons provide a useful component in many graphical displays.

Examples:

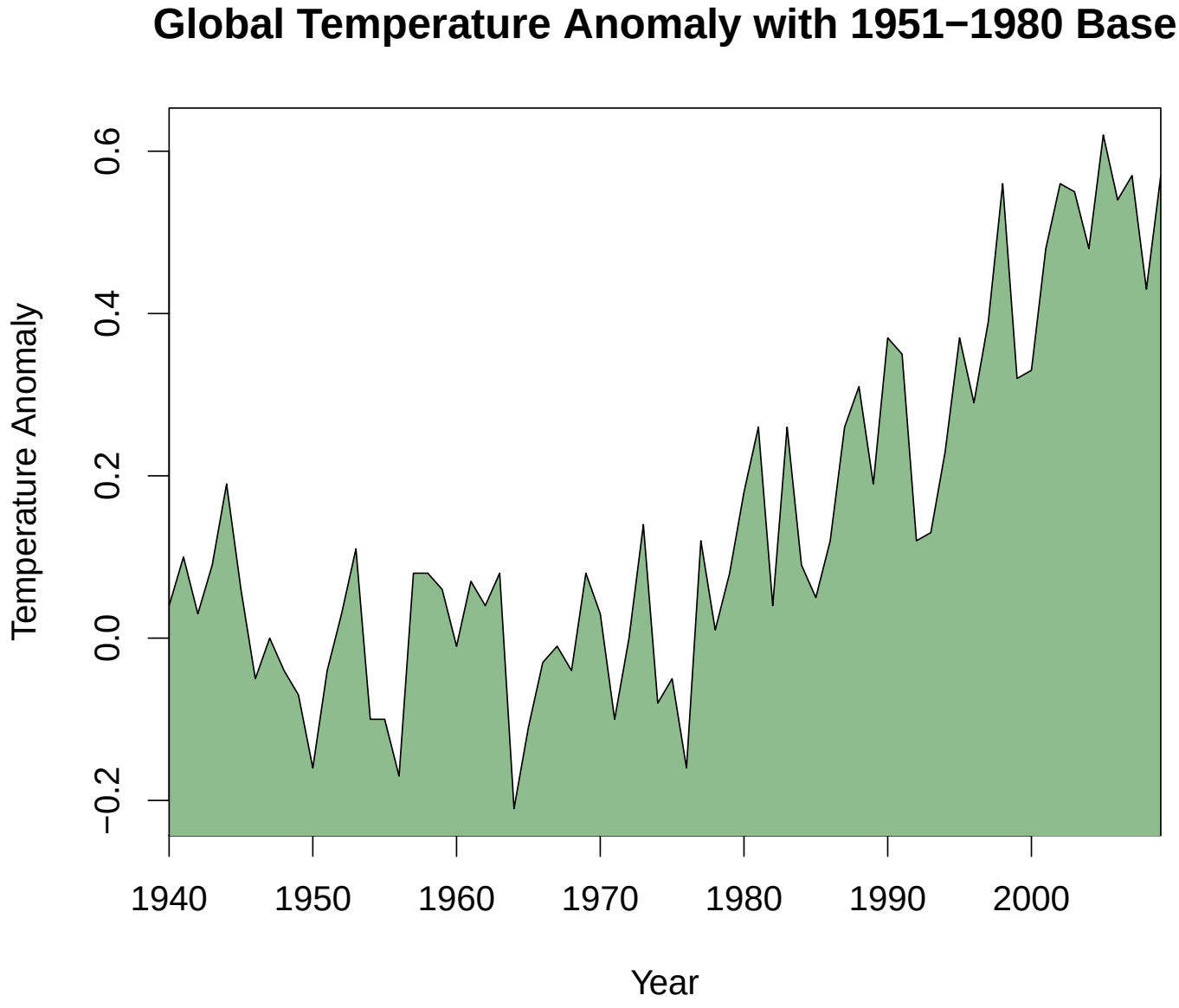
- As an approximate way of drawing (filled) circles and ellipses.
- For drawing rotated squares and rectangles.
- For drawing slices in pie graphs.
- As a way of showing areas under curves.
- Etc.

A Custom Time Series Display

The following display shows a time series display where the area below the series is filled with a solid colour.

There is no built-in way of creating this kind of display in R.

Using the ability to add filled polygons to a plot makes it possible.


Global Temperature Anomaly with 1951−1980 Base
Temperature Anomaly
−0.2
0.0
0.2
0.4
0.6
1940
1950
1960
1970
1980
1990
2000
Year

Producing the Display

The graph is produced in two stages:

- Setting up the plot and drawing the axes and annotation.
- Adding the filled polygon area.

The polygon is drawn going below the x axis and the part below the axis is "clipped" out.

```
> plot(year, temp, xaxs = "i", type = "n",
       xlab = "Year", ylab = "Temperature Anomaly",
       main = "Global Temperature Anomaly with 1951-19
> polygon(c(min(year), year, max(year)),
         c(-1, temp, -1), col = "darkseagreen")
```

Choosing Colours to Fill Areas

Choosing a set of colours to fill large areas in a plot is an important task in statistical graphics.

Choosing colours by hand from a colour chart is a very hard task unless the person choosing has a real flair for the task (most combinations look ghastly).

R has a special function (called `hcl`) which is designed to make this task easier.

The function allows a user to specify colours by *hue* (dominant colour wavelength), *chroma* (colourfulness), and *lightness* (brightness).

Six Reasonable Colours

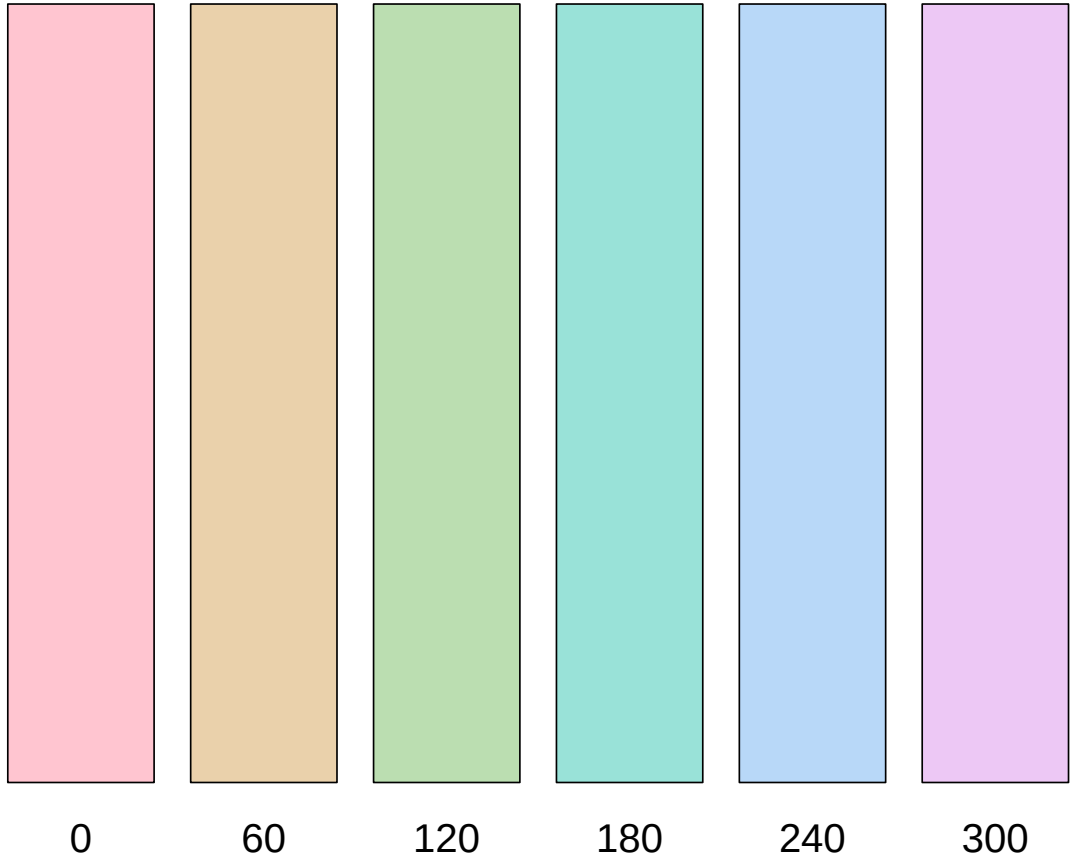
In the function call

```
hcl(h)
```

the parameter `h` specifies hue as an angle (in degrees) around the colour circle (0 = red, 60 = yellow, 120 = green, 180 = cyan, 240 = blue, 300 = magenta).

These equally spaces colours work harmoniously together and the default chroma and lightness values produce results that are easy to look at.

It is also possible to specify transparency with an optional parameter `alpha`. This makes it possible to "look though" coloured areas and see the objects behind them. (Here 0 means fully transparent and 1 means opaque.)


0
60
120
180
240
300
hue

Other Ways to Specify Colour

The functions `rgb` and `hc` also make it possible to specify colour programmatically. The calls are

```
rgb(r, g, b)
hsv(h, s, v)
```

with all the arguments in the range $[0, 1]$.

In the case of `rgb` the arguments specify the intensity of *red*, *green* and *blue* making up the colour.

In the case of `hsv` the arguments specify the *hue*, *saturation* and *value* of the colour.

Both of these functions allow the specification of transparency with an optional `alpha` argument.

Circles

The equation of a circle with radius R centered at $(0,0)$ is

$$(R\cos\theta, R\sin\theta), \quad 0 \leq \theta \leq 2\pi.$$

This circle can be approximated by a polygon which has vertexes equally spaced around the circle.

By using enough vertexes it is possible to produce a polygon which is visually indistinguishable from a circle.

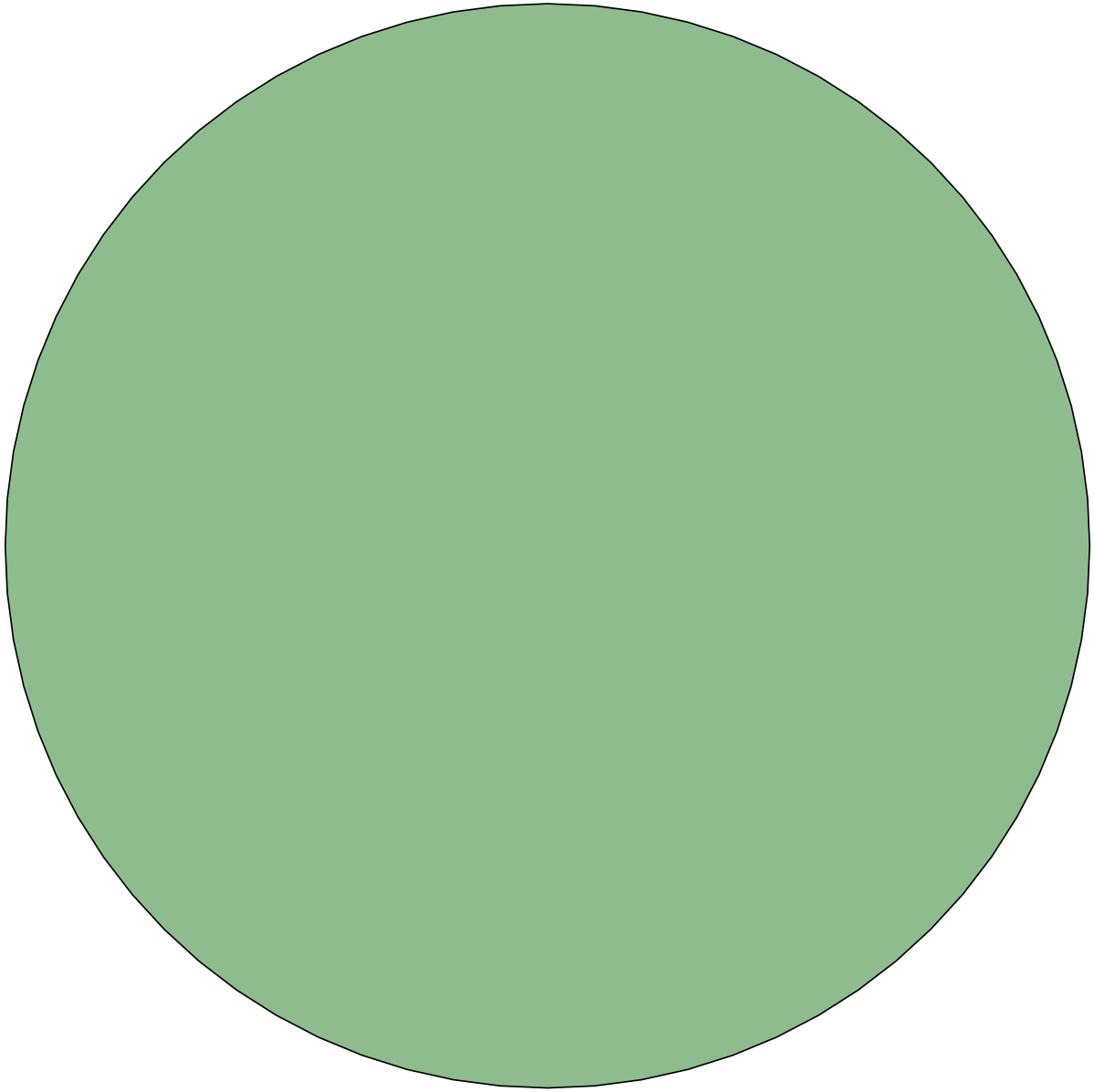
Changes of direction greater that 5° are visually detectable, so using at least 72 equally-spaced vertexes is necessary.

Drawing Circles with R

Here is some simple code to draw a coloured circle.

```
> theta = seq(0, 2 * pi, length = 73)[1:72]
> x = cos(theta)
> y = sin(theta)
> plot.new()
> plot.window(xlim = c(-1, 1),
              ylim = c(-1, 1),
              asp = 1)
> polygon(x, y, col = "darkseagreen")
```

The specification `asp = 1` is is important. It ensures that equal coordinate changes in the x and y directions are represented as equal distances on the page.

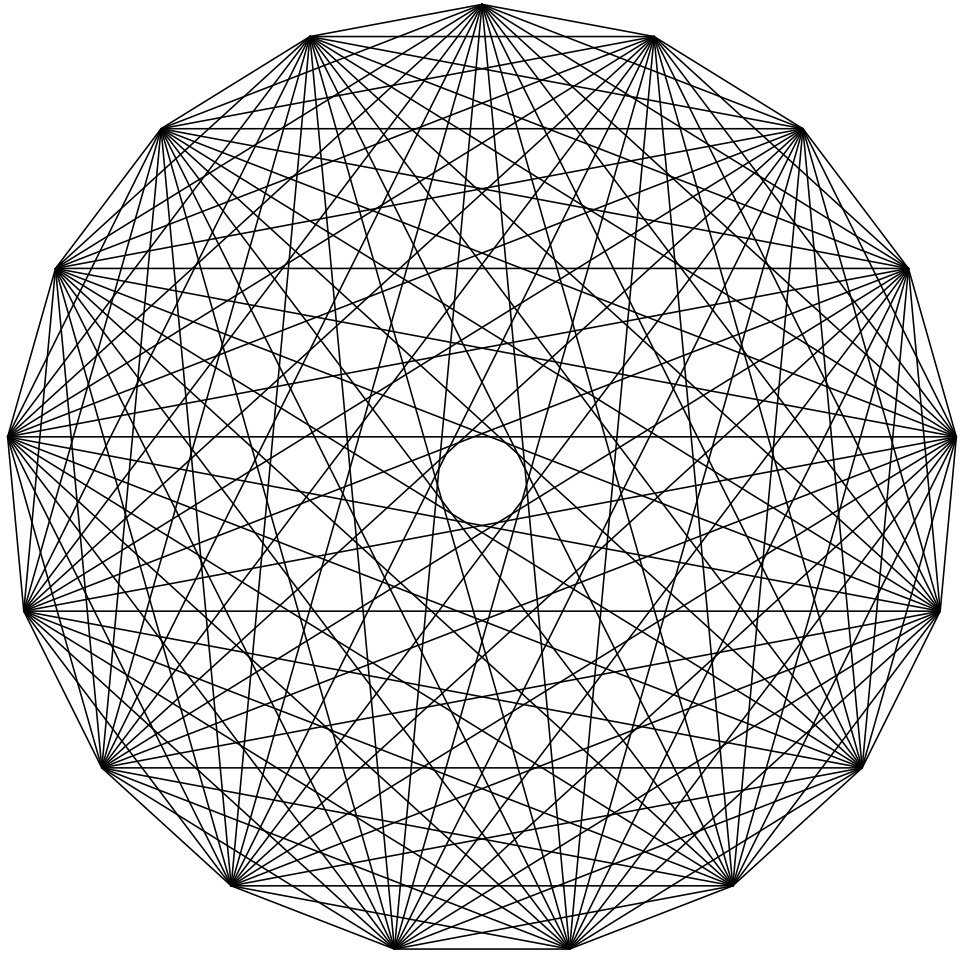


Rosettes

A rosette is a figure obtained by taking n points equally spaced around the circumference of circle, and joining every point to every other point.

```
> n = 17
> theta = seq(0, 2 * pi, length = n + 1)[1:n]
> x = sin(theta)
> y = cos(theta)
> plot.new()
> plot.window(xlim = c(-1, 1),
              ylim = c(-1, 1),
              asp = 1)
> for(i in 2:n)
    for(j in 1:(i - 1))
      segments(x[i], y[i], x[j], y[j])
> title(main="A 17-Rosette")
```

A 17−Rosette

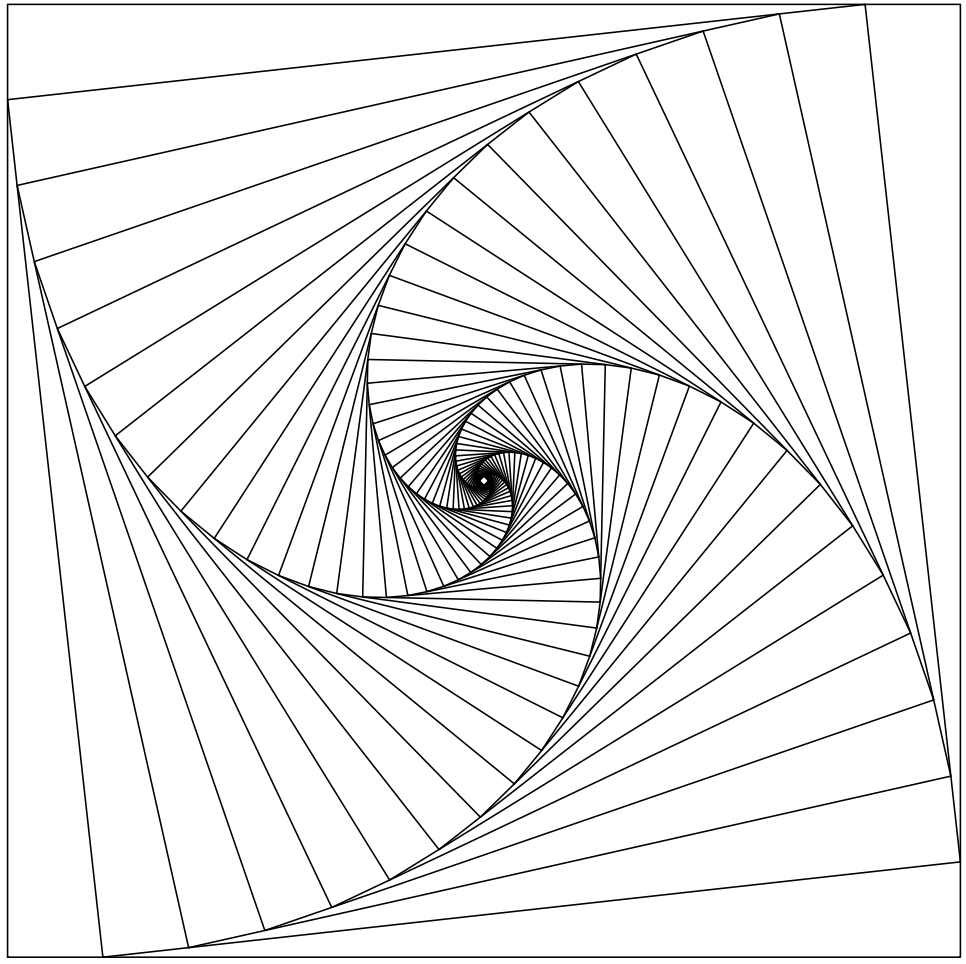


Spiral Squares

Here is how to create an interesting figure by rotating and shrinking a square multiple times.

```
> plot.new()
> plot.window(xlim = c(-1, 1),
                ylim = c(-1, 1), asp = 1)
> x = c(-1, 1, 1, -1)
> y = c(-1, -1, 1, 1)
> n = 51
> i1 = 1:4
> i2 = c(2:4, 1)
> for(i in 1:n) {
    polygon(x, y)
    x = .9 * x[i1] + .1 * x[i2]
    y = .9 * y[i1] + .1 * y[i2]
  }
> title(main="A Spiral Square")
```

A Spiral Square



Adding Color

Using a colour gradient can provide more "visual impact." In this case we can use a gradient from red to cyan generated by `hsv` function with

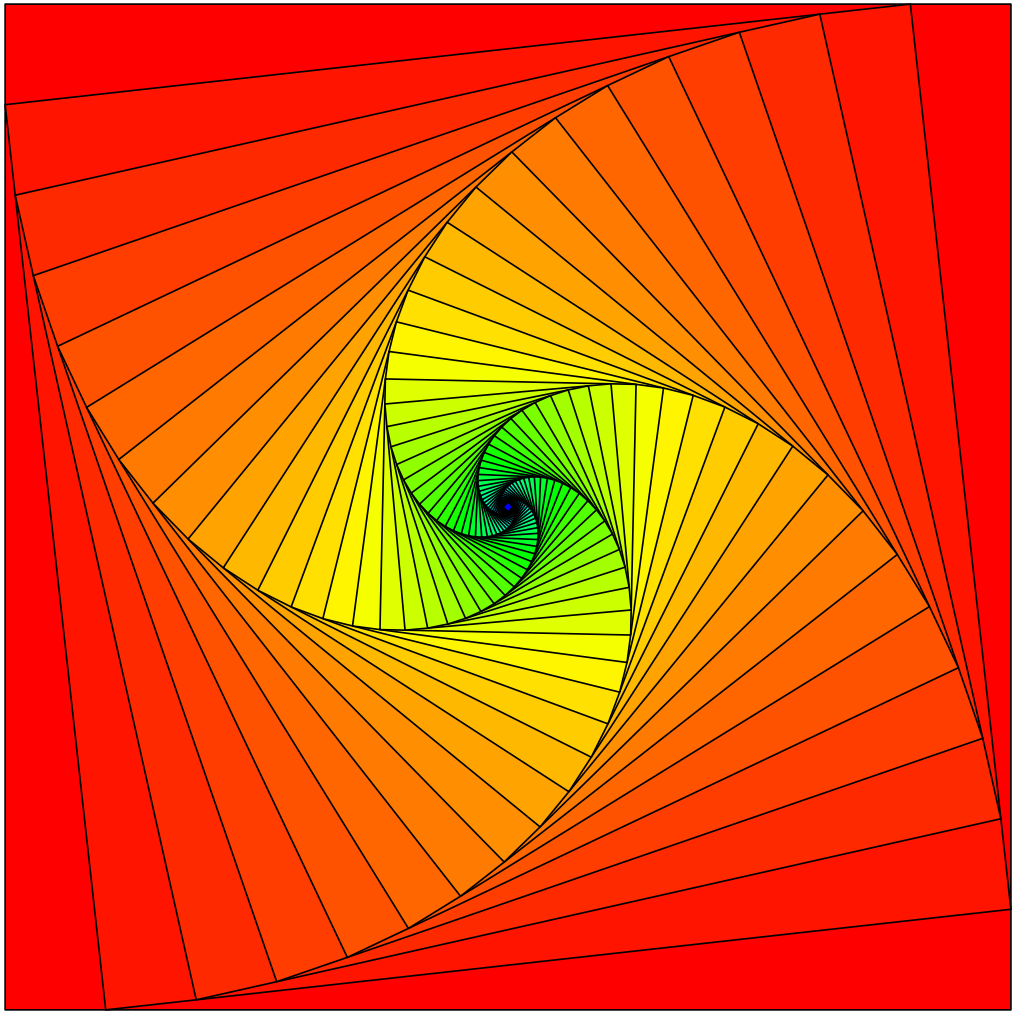
```
col = hsv(seq(0, 2/3, length = n))
```

and the passing the `i`-th colour to the `polygon` function with

```
polygon(x, y, col = col[i])
```

(In this example the `s` and `v` arguments take their default value of 1.)

A Spiral Square



Drawing Text in Plots

The `text` function can be used to draw character strings within a plot. The basic call to `text` has the form:

```
text(x, y, labels)
```

where `x` and `y` specify the locations the strings are to be drawn at and `labels` contains the actual strings.

Other optional arguments make it possible to specify how the strings are drawn; `adj` controls the string "adjustment," `srt` controls rotation of the string, `col` controls colour, `cex` controls magnification and `font` specifies which font to use.

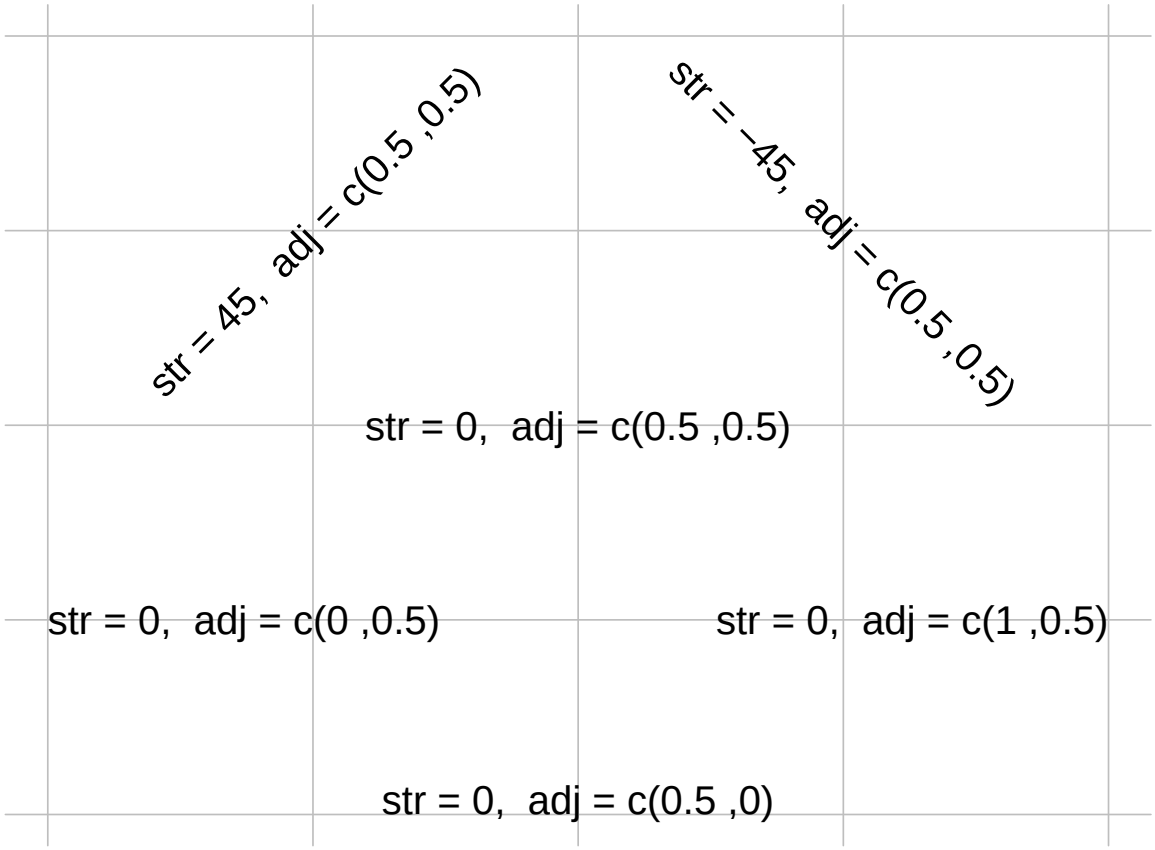
String Adjustment and Rotation

The `adj` parameter is a vector containing one or two values. The first value specified the *x* justification and the second the *y* justification.

A value of 0 indicates left-justification, a value of .5 indicates centred text and a value of 1 indicates right justification. (Intermediate values are also possible, but less used.)

The value of `srt` specifies the string rotation in degrees counter-clockwise from horizontal.

String rotation occurs before justification.


str = 45, adj = c(0.5 ,0.5)
str = –45, adj = c(0.5 ,0.5)
str = 0, adj = c(0.5 ,0.5)
str = 0, adj = c(0 ,0.5)
str = 0, adj = c(1 ,0.5)
str = 0, adj = c(0.5 ,0)

Drawing Text in Plot Margins

The function `mtext` can be used to draw text in the margins of a plot. The basic call has the form

```
mtext(text, side, line, at)
```

The argument `text` contains the strings to be written, `side` specifies which margin the text is to appear in (default 3), `line` specifies how many lines out form the plot the text is to appear and `at` contains the locations that the text is to be drawn at.

Other optional arguments affect how the text is drawn. These include `cex`, `col` and `font`.

Other arguments have effects specfic to `mtext`.

Some Margin-Text Specific Parameters

Several parameters have a effect specific to `mtext`.

`las` : This specifies the label orientation. A value of 0 means always parallel to the axis, 1 means always horizontal, 2 means always perpendicular to the axis and 3 means always vertical.

`adj` : the adjustment for each string in reading direction.

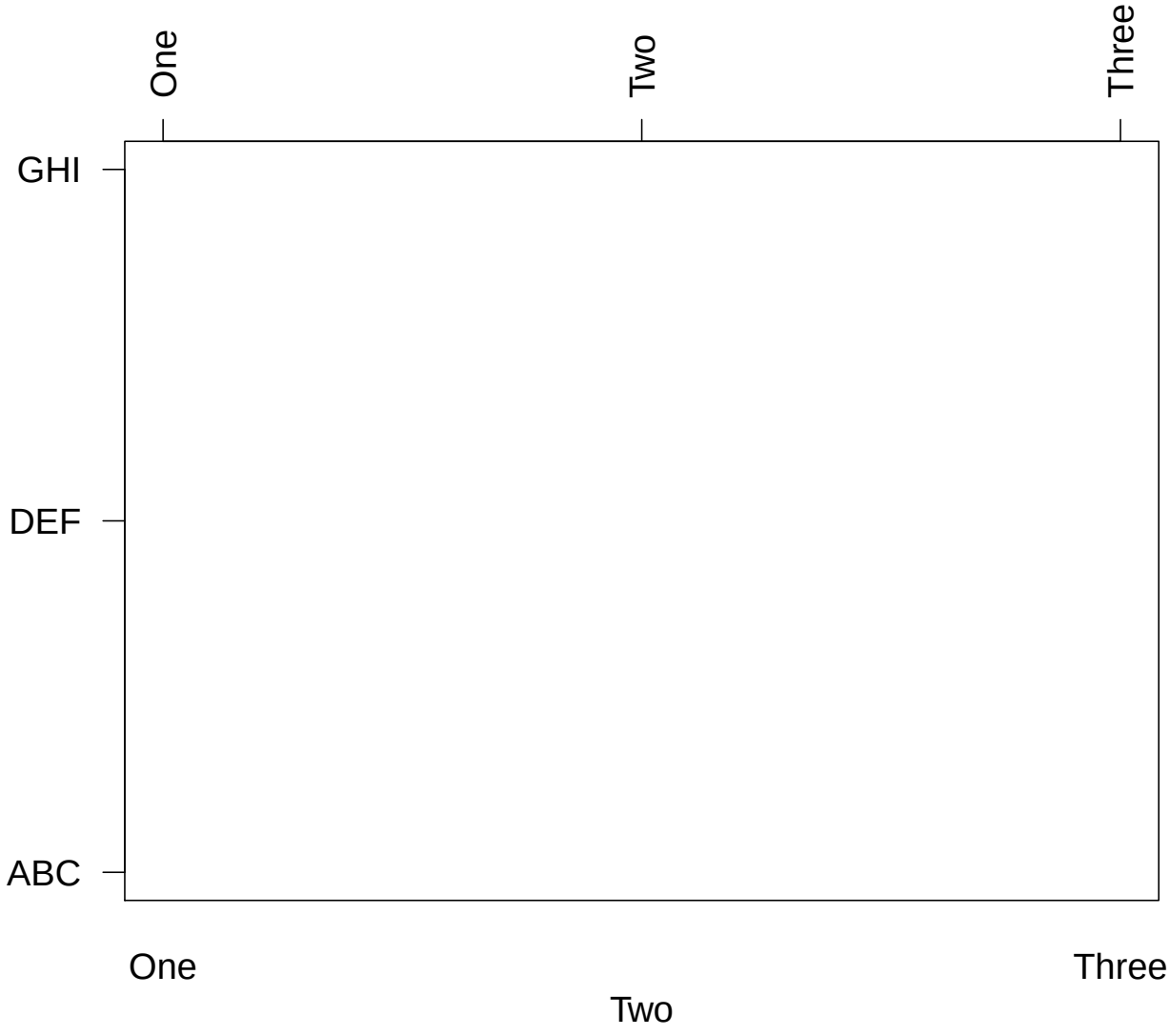
`padj` : the adjustment for each string perpendicular to the reading direction.

`outer` : If true, the string is plotted in the outer margin of the plot.

Marginal Text Examples

Here are a few examples showing what is possible with `mtext`.

```
> nums = c("One", "Two", "Three")
> lets = c("ABC", "DEF", "GHI")
> plot.new()
> plot.window(xlim = c(1, 3), ylim = c(1, 3))
> box()
> mtext(nums, side = 1, line = c(1, 2, 1),
        at = 1:3)
> axis(2, at = 1:3, labels = FALSE)
> mtext(lets, side = 2, line = 1,
        at = 1:3, las = 2)
> axis(3, at = 1:3, labels = FALSE)
> mtext(nums, side = 3, at = 1:3,
        line = 1, las = 3)
```


One
Two
Three
GHI
DEF
ABC
One
Three
Two

Eye-Catching Visual "Tricks"

In commercial presentation graphics it is common to add a variety of effects to make the the graphics more eye-catching.

Most R graphics do not use these tricks, but by building plots incrementally it is possible to liven up plots.

We'll look at just a few such effects.

Filling the Plot Background

It is simple to fill the rectangular background region of a plot.

This is done by drawing a rectangle of the appropriate size.

The function `par` makes it possible to get (and set) many properties associated with a plot.

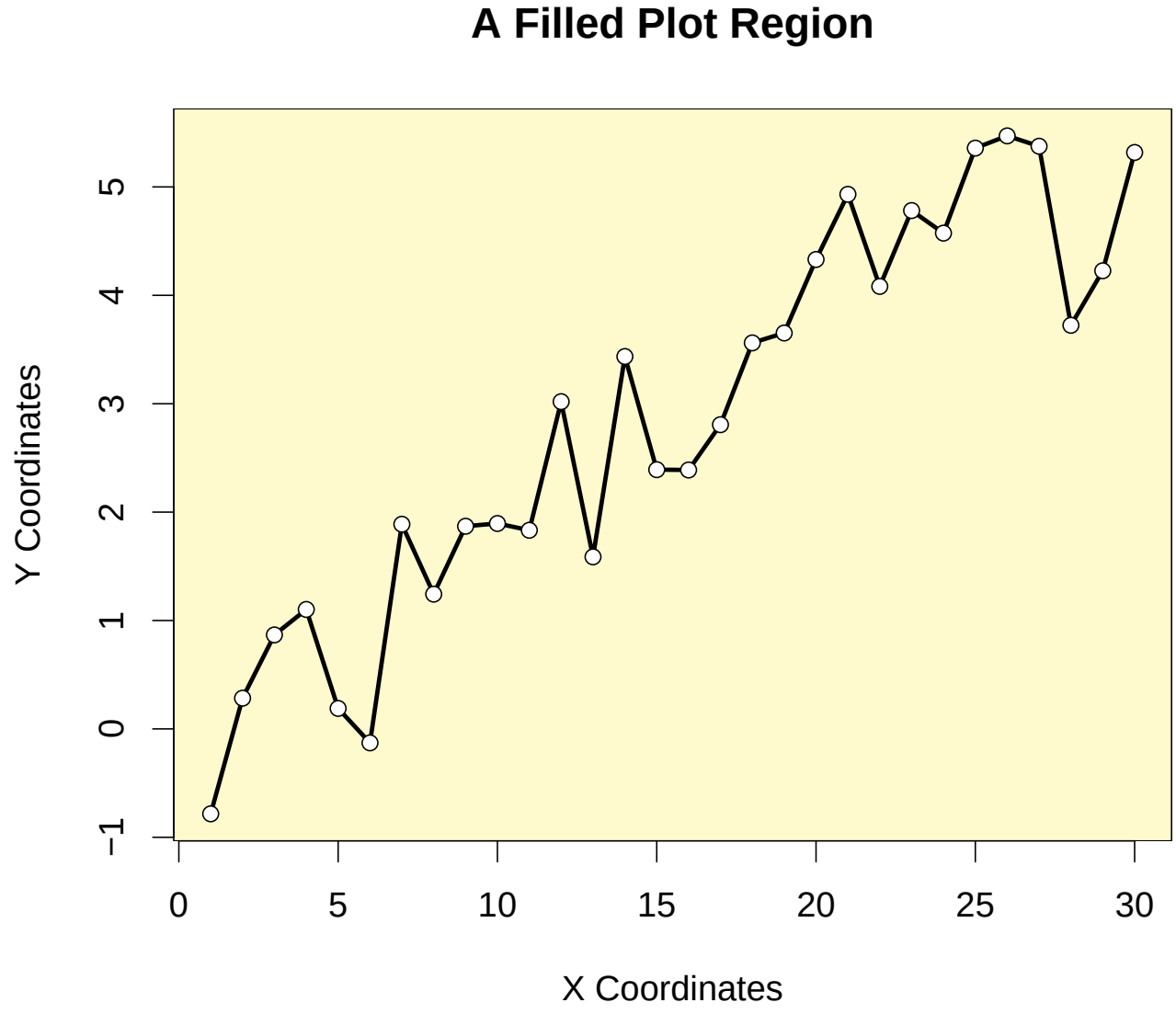
The call `par("usr")` returns the coordinates of the central region of a plot as vector of values

$$x_{\text{left}}, x_{\text{right}}, y_{\text{bottom}}, y_{\text{top}}.$$

Example

A simple filled background.

```
> x = 1:30
> y = rnorm(30) + x/5
> plot.new()
> plot.window(xlim = range(x),
                ylim = range(y))
> usr = par("usr")
> rect(usr[1], usr[3],
       usr[2], usr[4], col = "lemonchiffon")
> lines(x, y, lwd = 3)
> points(x, y, pch = 21, bg = "white")
> axis(1)
> axis(2)
> title(main = "A Filled Plot Region")
> title(xlab = "X Coordinates")
> title(ylab = "Y Coordinates")
```


A Filled Plot Region
Y Coordinates
X Coordinates
−1
0
1
2
3
4
5
0
5
10
15
20
25
30

Drop Shadows

One simple effect is the addition of *drop shadows* for objects that appear in the graph.

To create a shadow effect the object casting the shadow is drawn twice; first as a shadow and then as the real thing.

The shadow is usually drawn in gray, offset down and to the right.

The functions `xinch` and `yinch` can be used to produce appropriate offsets in the x and y directions.

R Code

As an example, consider drawing a smooth line through a set of points.

The coordinates of the smooth can be obtained with the *lowess* scatterplot smoother.

```
> smo = lowess(x, y)
```

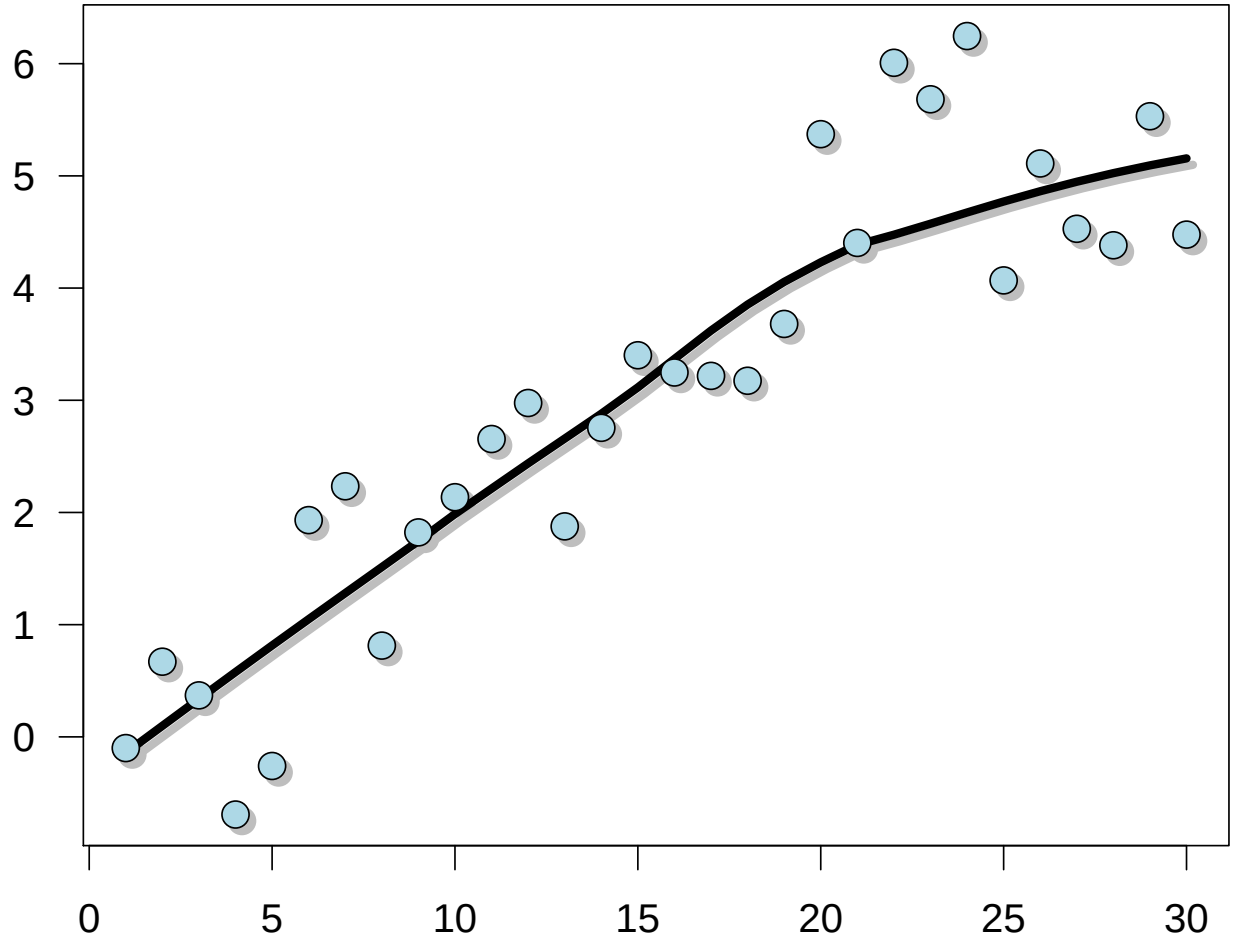
The shadow for the line can be drawn with the expression

```
> lines(smo$x + xinch(.4),
        smo$y - yinch(.4),
        col="gray", lwd = 5)
```

and the line itself with

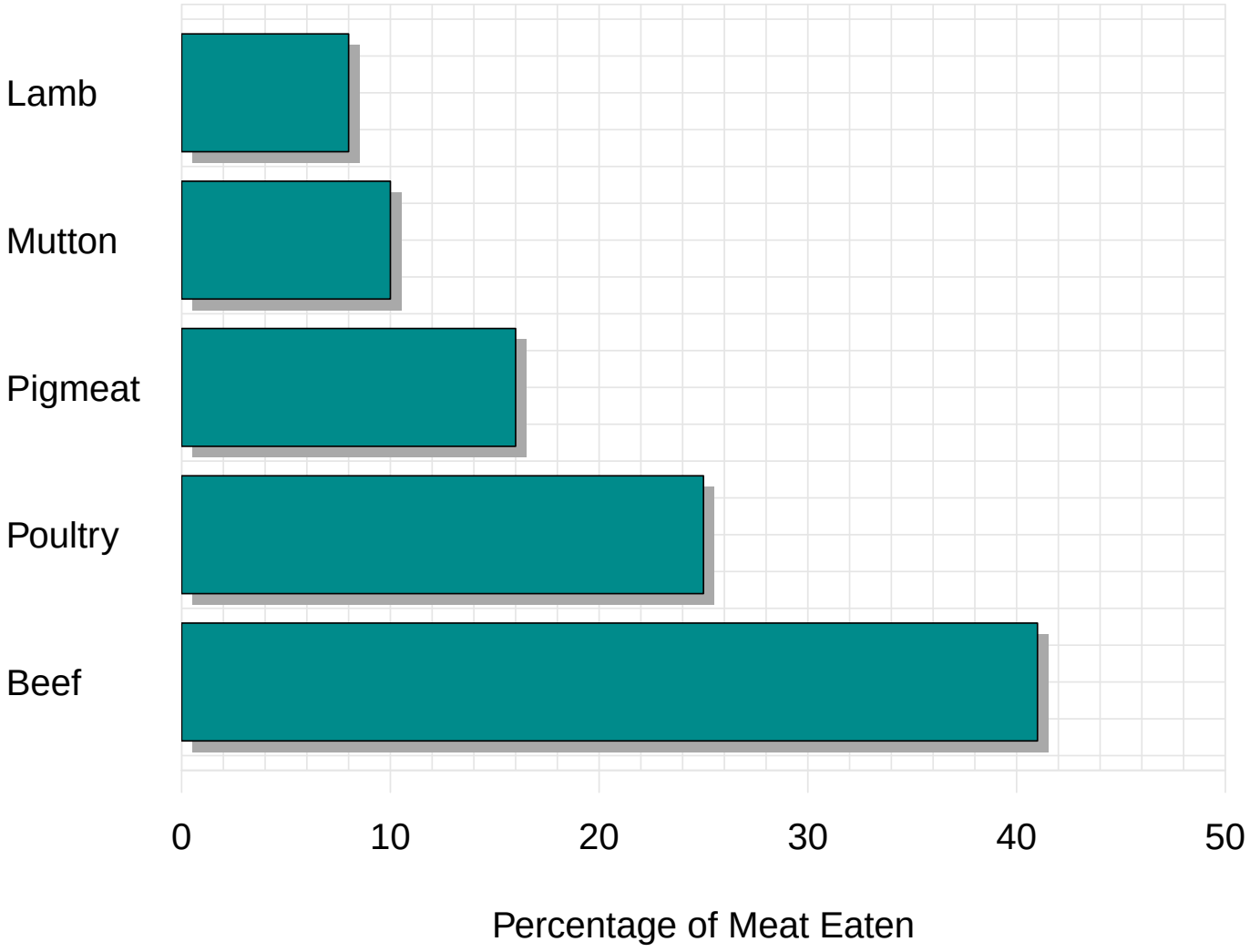
```
> lines(smo$x, smo$y, lwd = 5)
```

A Plot with Drop Shadows



The points have been magnified and given shadows too.

New Zealand Meat Consumption

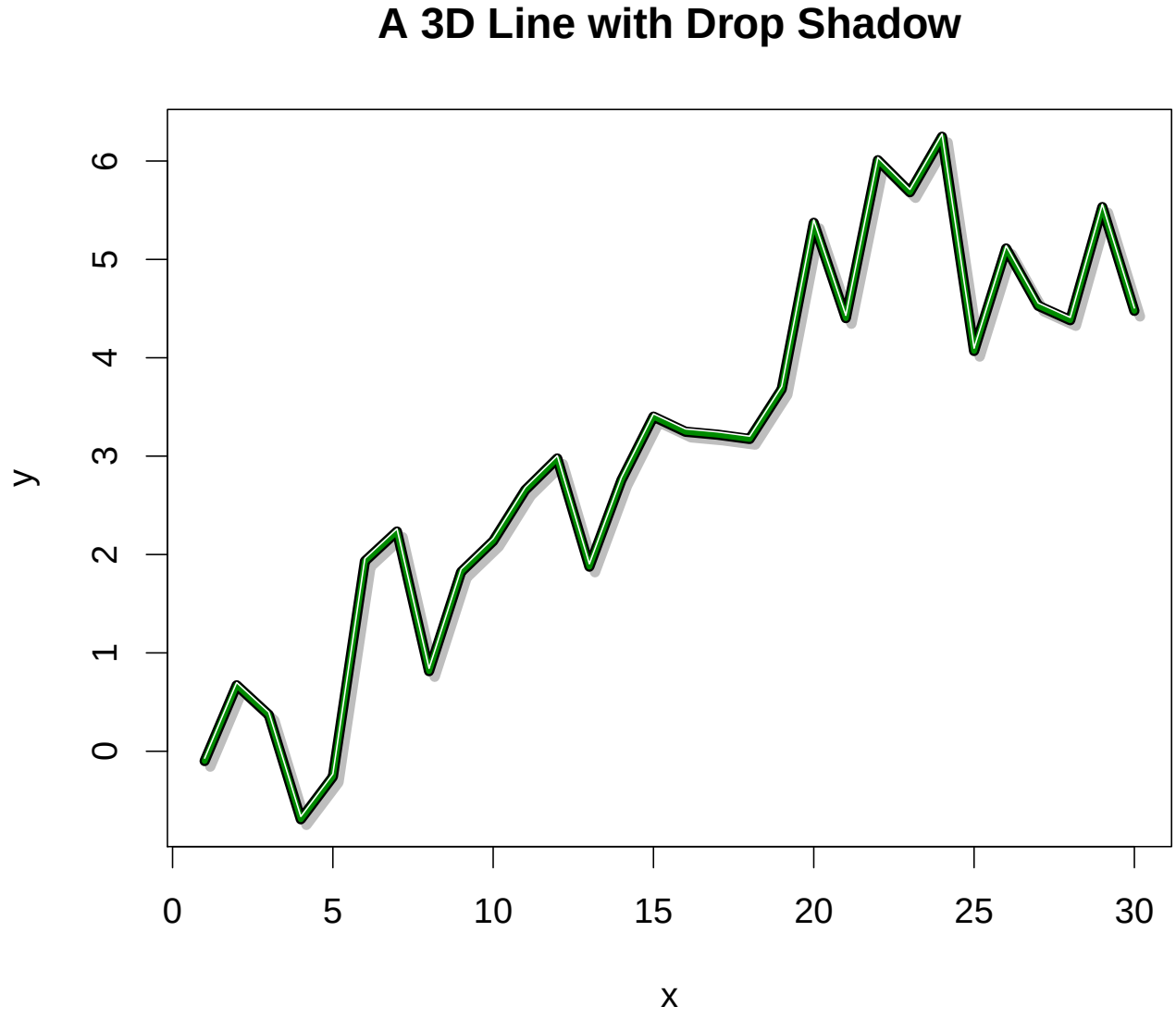


Line Thickness and Colour Manipulation

By manipulating the properties of lines it is possible to create interesting effects.

In the following plot, a green line is drawn with a black outline and a white "specular highlight."

```
> plot(x, y, type = "n")
> lines(x + xinch(.04), y - yinch(.04), lwd = 7,
        col = "gray")
> lines(x, y, lwd = 7)
> lines(x, y, lwd = 4, col = "green4")
> lines(x, y+yinch(.02), lwd = 1, col = "white")
> title(main = "A 3D Line with Drop Shadow")
```

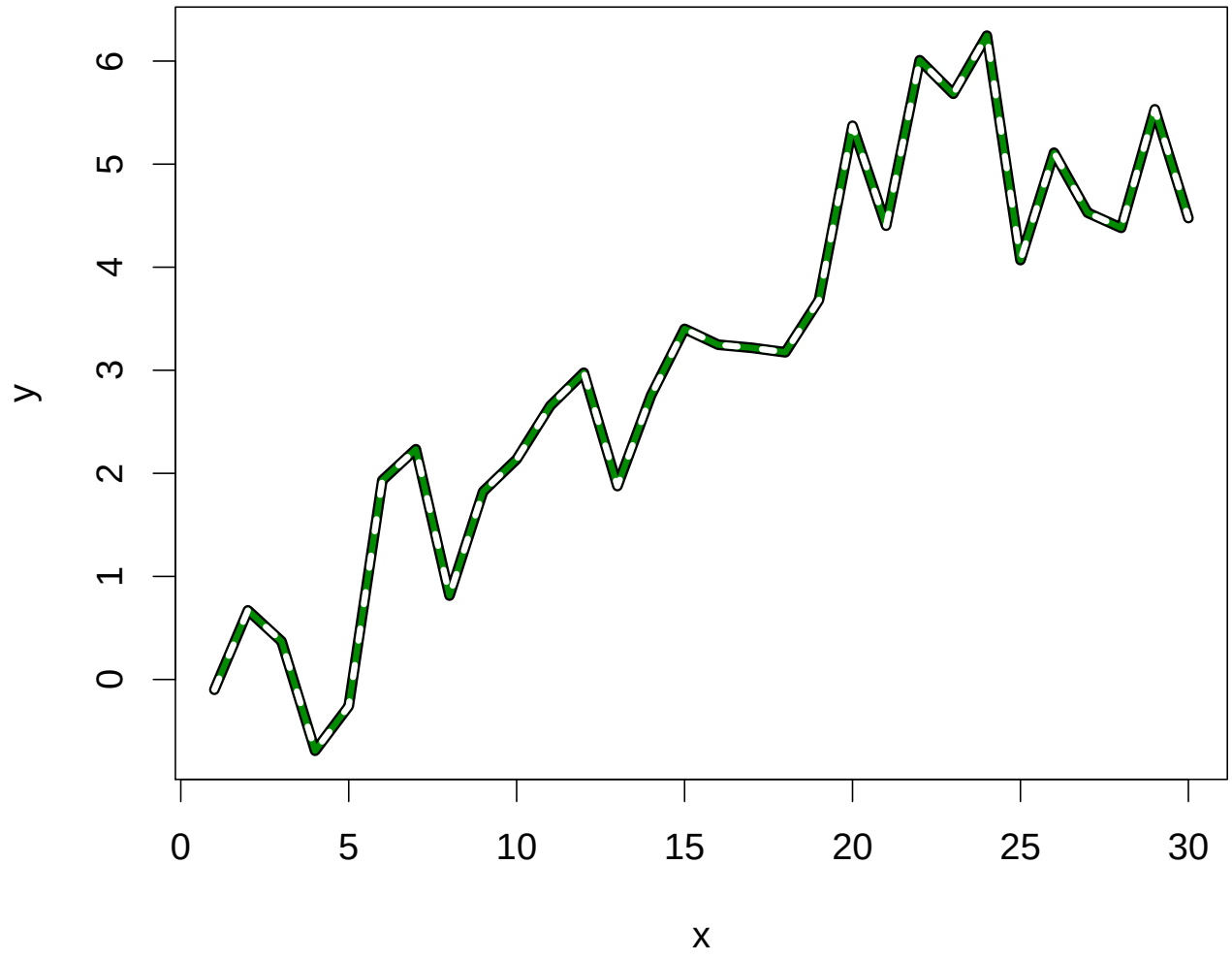

A 3D Line with Drop Shadow
y
0
1
2
3
4
5
6
0
5
10
15
20
25
30
x

Textured Line Example

By manipulating the properties of lines it is possible to create interesting effects.

In the following plot, a green line is drawn with a black outline and a white dashed line is drawn over the top.

```
> plot(x, y, type = "n")
> lines(x, y, lwd = 7)
> lines(x, y, lwd = 4, col = "green4")
> lines(x, y, lwd = 4, lty = "33", col = "white")
```


y
0
1
2
3
4
5
6
0
5
10
15
20
25
30
x

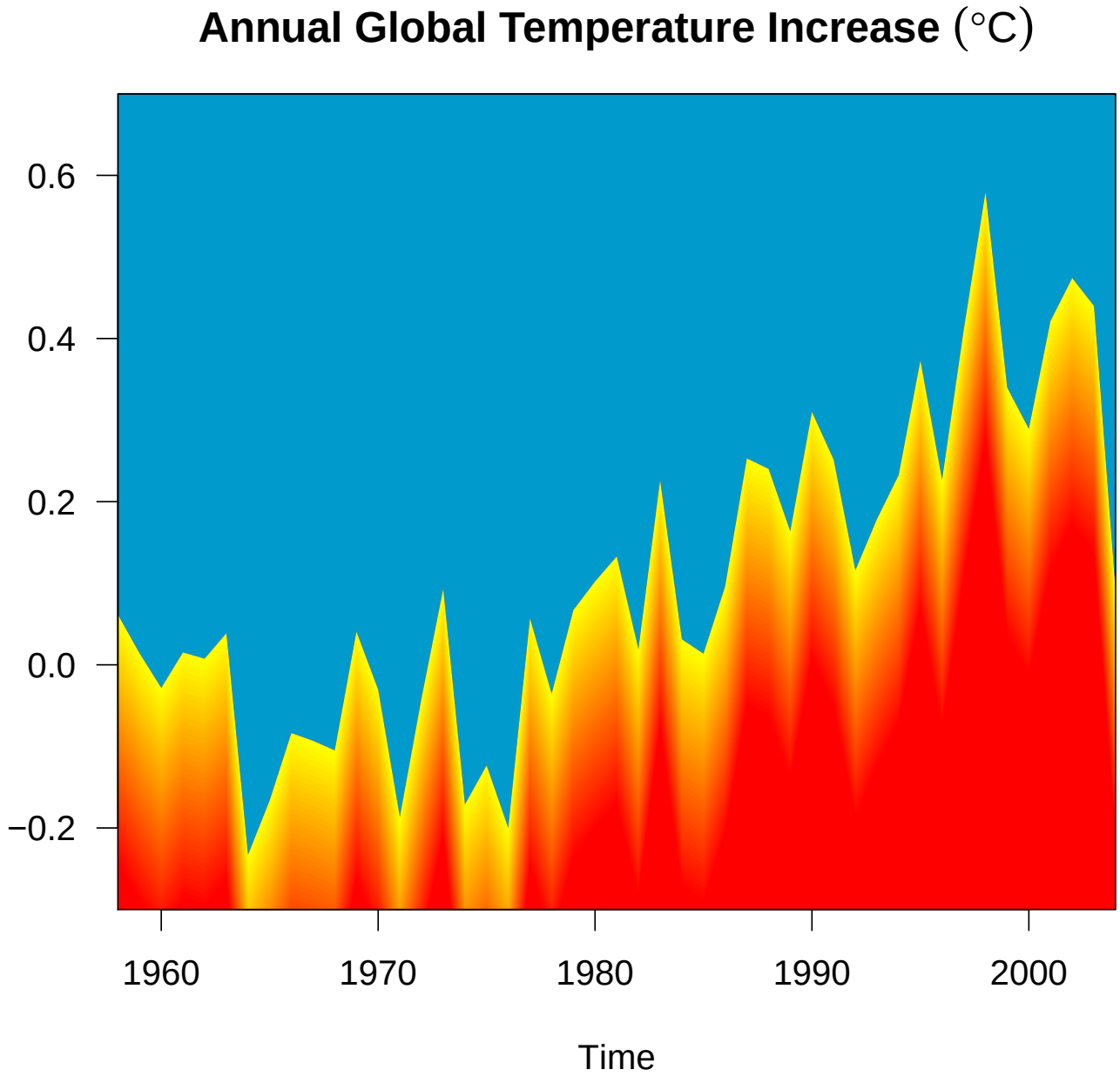
Colour Gradients

Colour gradients can be produced in R by superimposing slightly offset polygons with small colour changes.

In the following example the flame effect is obtained by drawing a series of identical polygons. Each polygon is slightly lower and slightly redder than the one before.

In this case the colours were generated using `hsv`.

This example should be filed under “propaganda.”


Annual Global Temperature Increase (°C)
0.6
0.4
0.2
0.0
−0.2
1960
1970
1980
1990
2000
Time

The `par` Function

R graphics are controlled by use of the `par` function.

`par` makes it possible to control low-level graphics by querying and setting a large set of *graphical parameters*. Graphical parameters control many features such as:

- the layout of figures on the device
- the size of the margins around plots
- the colours, sizes and typefaces of text
- the colour and texture of lines
- the style of axis to be used
- the orientation of axis labels

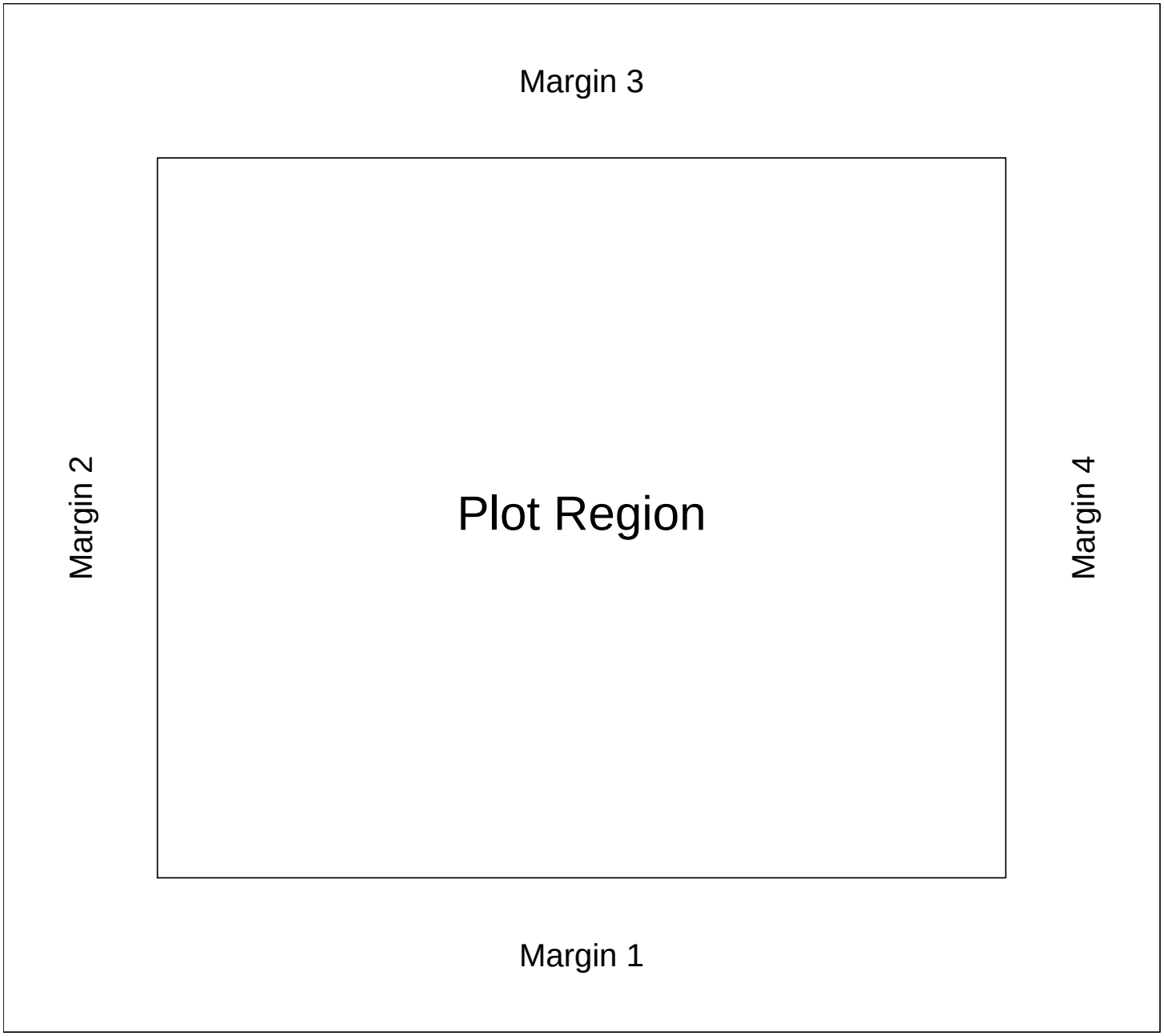
The Layout of Graphics

Graphs appear on R graphics devices as a series of rectangular graphical *figures*.

Each figure consists of a rectangular *plot region* surrounded by four *margins*.

Each element of a figure can be described as being in either the plot region or the margins.

The sides of a plot are numbered 1 through 4.


Margin 3
Margin 2
Plot Region
Margin 4
Margin 1

Controlling The Margins

The sizes of the margins can be changed by making a call to the function `par` before calling `plot.new`.

There are several possibilities:

1. Set the margin sizes in inches.

```
> par(mai=c(2, 2, 1, 1))
```

2. Set the margin sizes in lines of text.

```
> par(mar=c(4, 4, 2, 2))
```

3. Set the plot width and height in inches.

```
> par(pin=c(5, 4))
```

Querying the Margin Sizes

The par function can be used for querying the margin sizes.

The margin sizes in lines and inches can be obtained as follows:

```
> par("mar")
[1] 5.1 4.1 4.1 2.1
```

```
> par("mai")
[1] 1.02 0.82 0.82 0.42
```

and the the plot dimensions in inches as follows:

```
> par("pin")
[1] 5.76 5.16
```

Multifigure Layouts

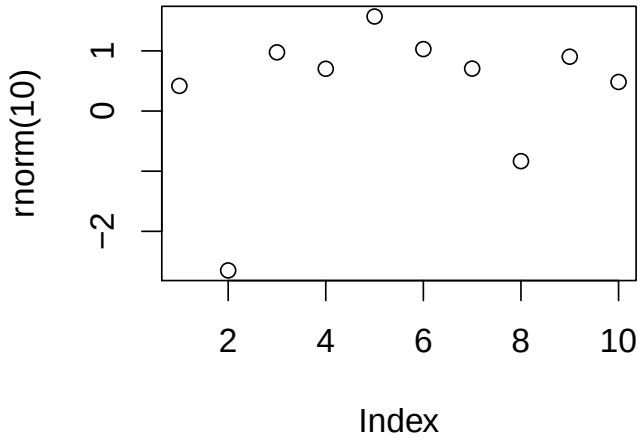
`par` can be used to set up arrays of figures on the page. These arrays are then filled row-by-row or column-by-column.

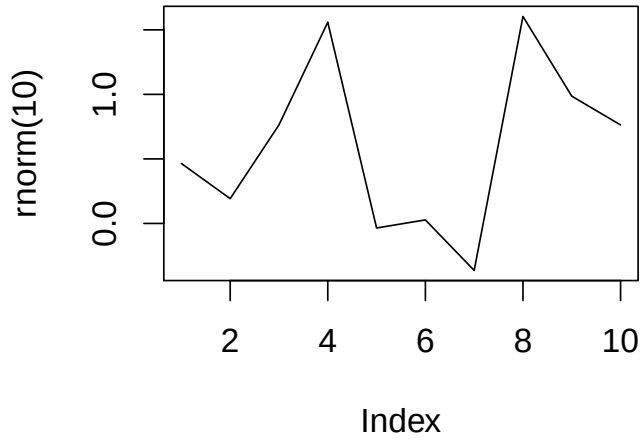
The following example declares a 2×2 array to be filled by rows and then produces the plots.

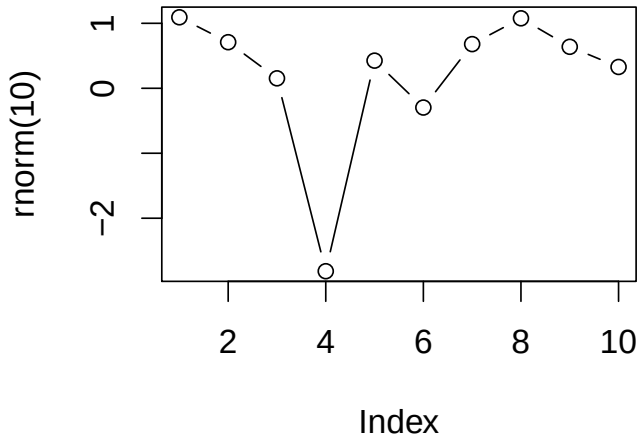
```
> par(mfrow=c(2, 2))
> plot(rnorm(10), type = "p")
> plot(rnorm(10), type = "l")
> plot(rnorm(10), type = "b")
> plot(rnorm(10), type = "o")
```

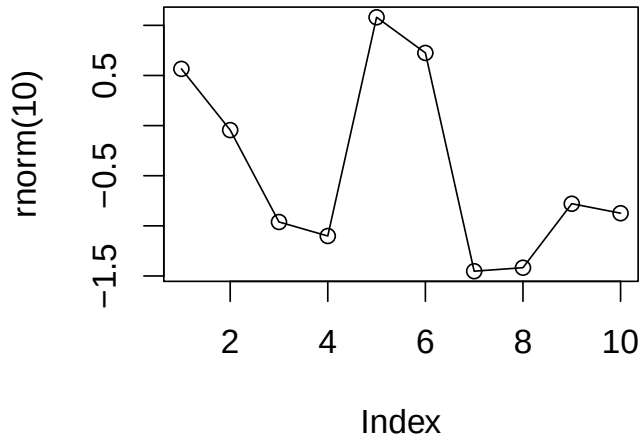
A 2×2 array to be filled by columns would be declared with

```
> par(mfcol = c(2, 2))
```


rnorm(10)
−2
0
1
2
4
6
8
10
Index


rnorm(10)
0.0
1.0
2
4
6
8
10
Index


rnorm(10)
−2
0
1
2
4
6
8
10
Index


rnorm(10)
−1.5
−0.5
0.5
2
4
6
8
10
Index

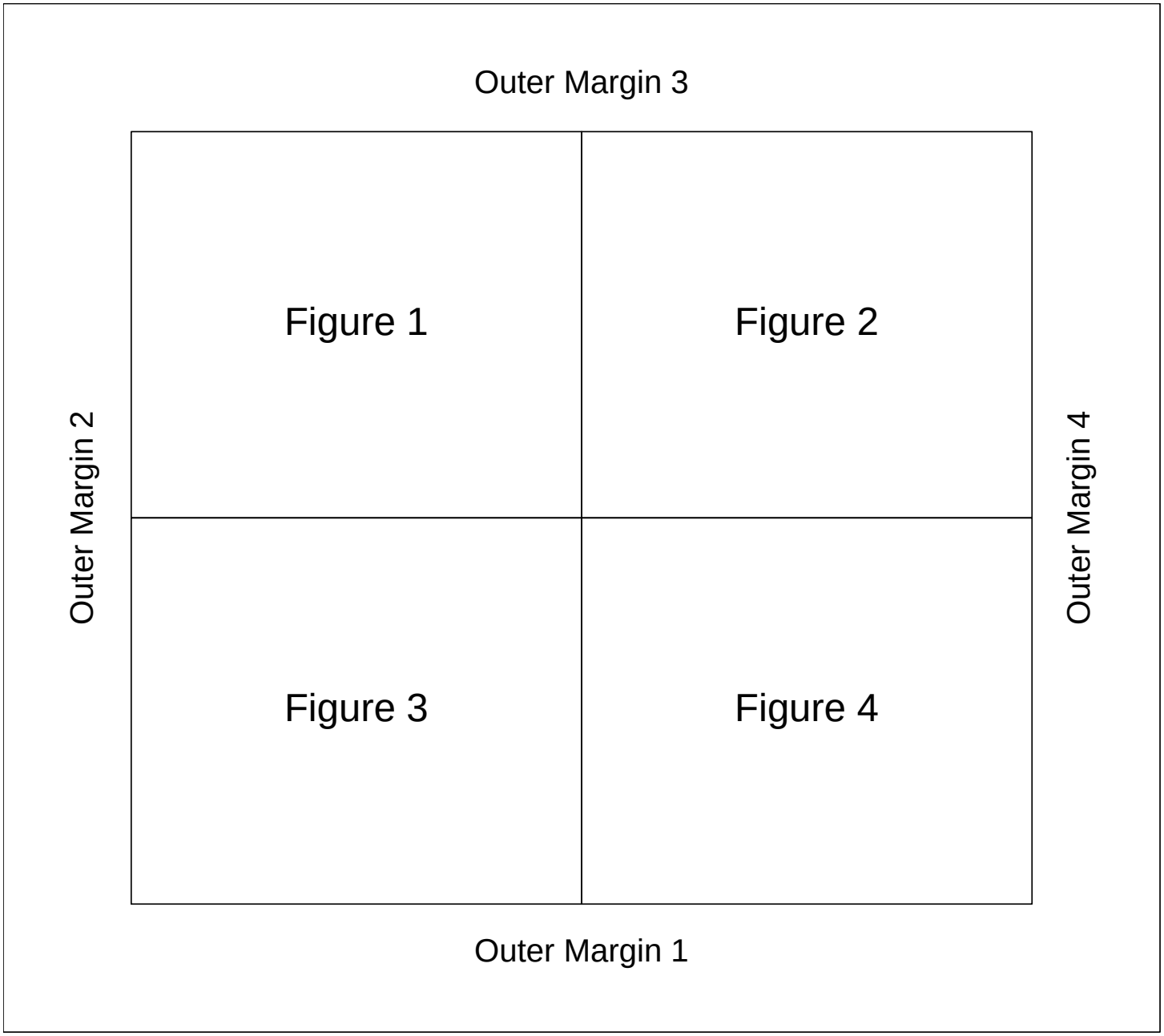
Outer Margins

Arrays of figures can also be created with a set of outer margins.

The size of outer margins can be specified with the `oma` or `omi` arguments to `par`.

Here is an example which shows a 2×2 array of figures surrounded by an outer margin with space for 4 lines of text on all sides.

```
> par(oma = c(4, 4, 4, 4),
      mfrow = c(2, 2))
```

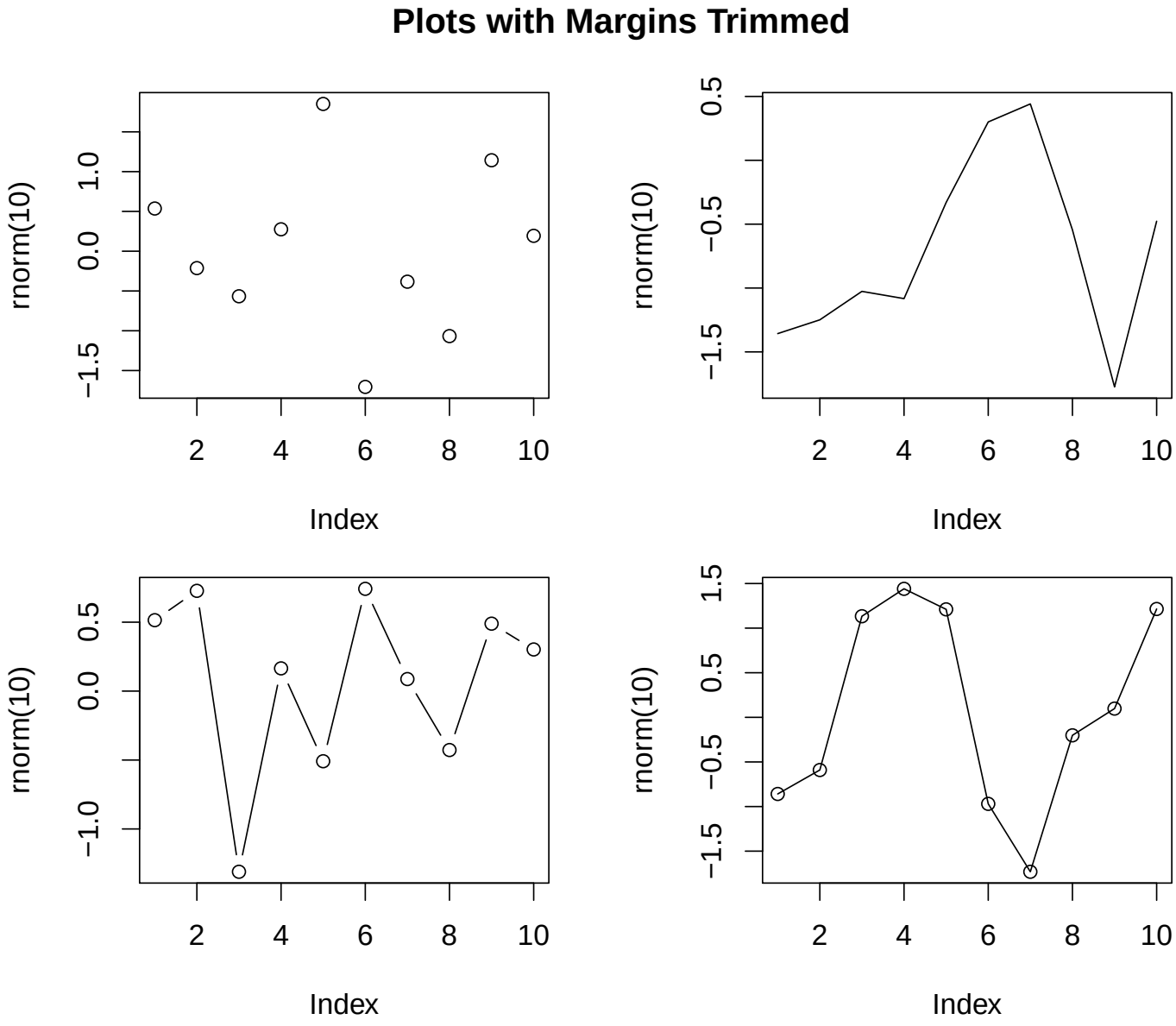

Outer Margin 3
Figure 1
Figure 2
Outer Margin 2
Outer Margin 4
Figure 3
Figure 4
Outer Margin 1

Eliminating Waste Margin Space

It can be useful to eliminate redundant space from multi-way arrays by trimming the margins a little.

```
> par(mfrow=c(2, 2))
> par(mar = c(5.1, 4.1, 0.1, 2.1))
> par(oma = c(0, 0, 4, 0))
> plot(rnorm(10), type = "p")
> plot(rnorm(10), type = "l")
> plot(rnorm(10), type = "b")
> plot(rnorm(10), type = "o")
> title(main = "Plots with Margins Trimmed",
        outer = TRUE)
```

Here we have trimmed space from the top of each plot, and placed a 4 line *outer margin* at the top of of the layout.


Plots with Margins Trimmed
rnorm(10)
Index
rnorm(10)
Index
rnorm(10)
Index
rnorm(10)
Index

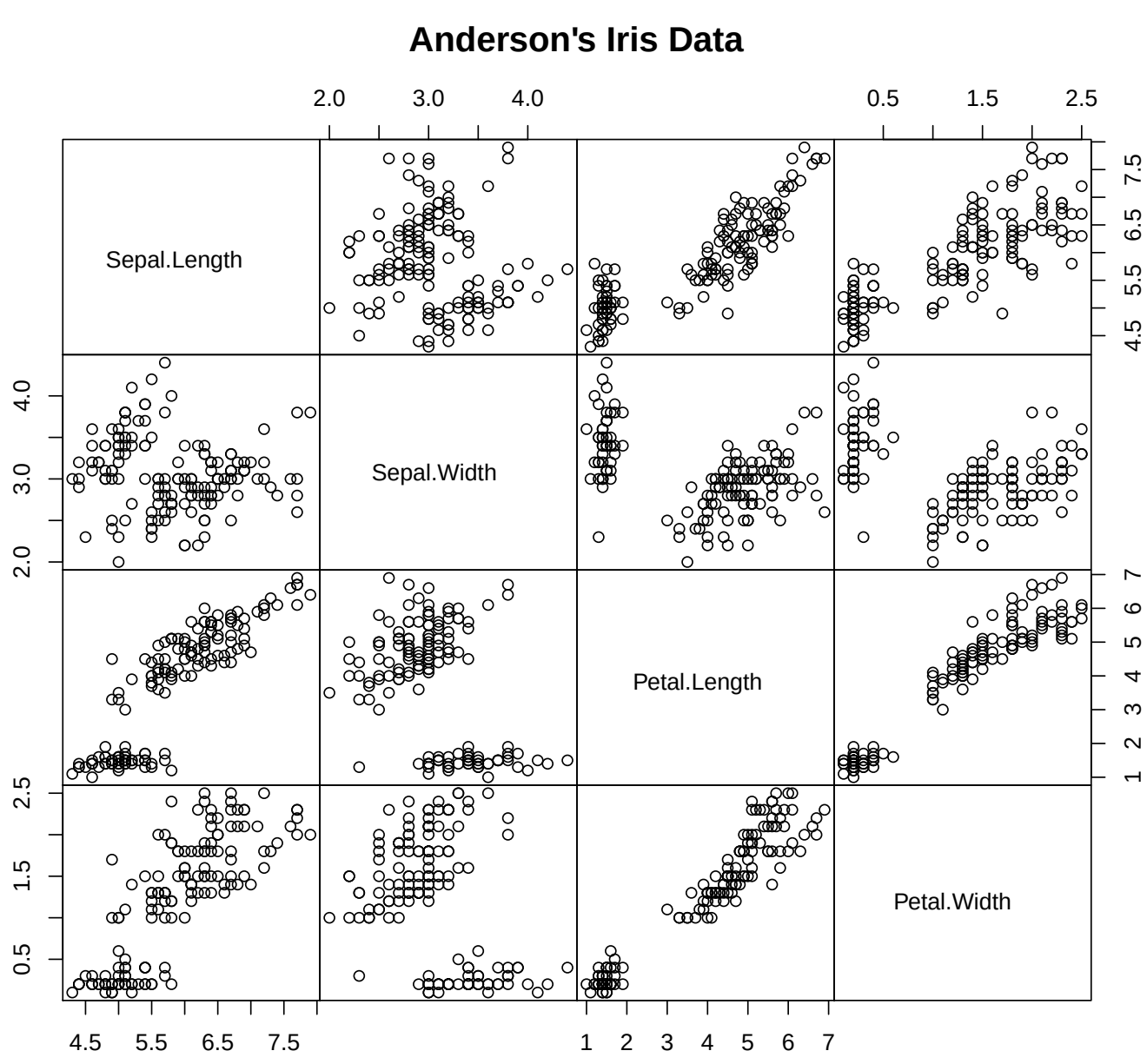
An Example: Scatterplot Matrices

A scatterplot matrix is a multidimensional generalisation of a simple scatterplot.

Given n variables, a scatterplot matrix presents scatterplots of every possible variable in the set against every other possible variable in the set.

The results are displayed in an $n \times n$ matrix, with the element in the ij-th position being the plot of variable i against variable j.

Since a plot of variable i against variable i is relatively uninteresting, something more useful can be displayed in that position.


Anderson's Iris Data
Sepal.Length
Sepal.Width
Petal.Length
Petal.Width
2.0
3.0
4.0
0.5
1.5
2.5
4.5
5.5
6.5
7.5
1
2
3
4
5
6
7

Setting Up a Scatterplot Matrix

To create a scatterplot matrix we need to arrange a series of plots on the page.

The basic arrangement is to have the n^2 plots arranged in an $n \times n$ array.

To ensure that the plots are properly juxtaposed, we need to eliminate the margins around the individual plots but to create an outer margin around the whole array to hold the axes and an overall title.

```
par(mfrow = c(ncol(x), ncol(x)),
    mar = rep(0, 4),
    oma = c(4.1, 4.1, 5.1, 4.1))
```

Saving and Restoring Parameter Values

Since the layout parameters for the scatterplot matrix only apply to that matrix, it is useful to be able to revert back to the previous set of values when the plot is complete.

The `par` function returns the previous values of any parameters which are changed by a a call to it and these can be save for later use.

```
opar = par(mfrow = c(ncol(x), ncol(x)),
           mar = rep(0, 4),
           oma = rep(4.1, 4))
```

The values can be restored later with another call to `par`.

```
par(opar)
```

Drawing The Scatterplots

Once the layout has been created it is easy to produce the individual scatterplots.

```
for(i in 1:n)
    for(j in 1:n) {
        xrange = range(x[,j])
        yrange = range(x[,i])
        plot.new()
        plot.window(xlim = xrange, ylim = yrange)
        if (i == j)
            text(mean(xrange), mean(yrange),
                 colnames(x)[j])
         else points(x[,j], x[,i])
         box()
    }
```

Drawing Axes

Clearly, not every scatterplot in the matrix has axes and we need a computation to determine which (if any) axes should be drawn.

Only axes at the edge of the plot are drawn, and then only alternate axes are drawn to avoid label collisions.

```
if (i == 1 && j %% 2 == 0)
    axis(3)
if (i == n && j %% 2 == 1)
    axis(1)
if (j == 1 && i %% 2 == 0)
    axis(2)
if (j == n && i %% 2 == 1)
    axis(4)
```

Printing a Title

If we want to have an overall title on the plot we need to ensure that there is enough space in the top outer margin.

```
opar = par(mfrow = c(ncol(x), ncol(x)),
           mar = rep(0, 4),
           oma = c(4.1, 4.1, 5.1, 4.1))
```

We can then produce the title using the `mtext` function.

```
mtext(main, side = 3, line = 3,
      font = 2, outer = TRUE)
```

This draws the text in the (middle of the) third line of the third outer margin, in bold font.

Final Cleanup

For general hygiene it is important to reset any parameters before leaving the function.

```
par(opar)
```

Customisation

The scatterplot matrix function we've described is very inflexible.

We could customise it by adding of optional arguments.

```
pairs = function(x, col = 1, pch = 1, ...)
```

and then passing these arguments on to the `points` function.

```
points(x[,j], x[,i], col = col, pch = pch)
```

Panel Functions

The heart of the scatterplot matrix function is the call

```
points(x[,j], x[,i])
```

We can make the function highly customisable by making the function which is invoked at this point be an argument to the function.

```
pairs = function(x, y, ...,
                 panel = function(x, y, ...)
                         pairs(x, y, ...))
```

and invoking this function in the body of the function.

Panel Function Example

Here is an example which shows a panel function which plots the points and then superimposes the regression line between the pair of variables.

```
pairs(x,
      panel = function(x, y, ...) {
                  points(x, y)
                  abline(lm(y ~ x))
              })
```

Using `par`

The `par` function can be used to control margins and the layout of plots on the page, using the parameters `mar`, `mai`, `pin`, `oma`, `omi`, `mfrow` and `mfcol`.

This is just the tip of the iceberg; there are 71 graphics parameters in all.

You can view the settings of the parameters by typing

```
> par()
```

and view their names by typing

```
> names(par())
```

The Full Set of Graphical Parameters

xlog	ylog	adj	ann	ask
bg	bty	cex	cex.axis	cex.lab
cex.main	cex.sub	cin	col	col.axis
col.lab	col.main	col.sub	cra	crt
csi	cxy	din	err	family
fg	fig	fin	font	font.axis
font.lab	font.main	font.sub	gamma	lab
las	lend	lheight	ljoin	lmitre
lty	lwd	mai	mar	mex
mfcol	mfg	mfrow	mgp	mkh
new	oma	omd	omi	pch
pin	plt	ps	pty	smo
srt	tck	tcl	usr	xaxp
xaxs	xaxt	xpd	yaxp	yaxs
yaxt				

Important Groups of Graphics Parameters

Graphics parameters come in groups (usually with related names). Important groups correspond to names as follows:

`cex` An abbreviation for *character expansion factor*. This provides a magnification of text relative to the default.

`col` The colour to be used for text or graphics elements.

`font` The font to be used for drawing character strings.

Controlling Font Size

Setting the value of `cex` controls the default size of text elements and symbols in graphs. A value of `cex=2` makes text twice its standard size.

More specialized `cex` values are as follows:

`cex.main`	The expansion factor for the main title.
`cex.lab`	The expansion factor for `xlab` and `ylab`.
`cex.axis`	The expansion used for the text printed at axis tickmarks.
`cex.sub`	The expansion used for plot subtitles (`sub`).

Controlling Colour

Setting the value of `col` controls the default colour of text elements and symbols in graphs.

More specialized `col` values are as follows:

`col.main`	The colour for the main title.
`col.lab`	The colour for `xlab` and `ylab`.
`col.axis`	The colour used for the text printed at axis tickmarks.
`col.sub`	The colour used for plot subtitles (`sub`).

Controlling Fonts

Setting the value of `font` controls the default font of the text elements in graphs.

More specialized `font` values are as follows:

`font.main`	The font for the main title.
`font.lab`	The font for `xlab` and `ylab`.
`font.axis`	The font used for the text printed at axis tickmarks.
`font.sub`	The font used for plot subtitles (`sub`).

The value of `font` is `1` for normal font, `2` for bold, `3` for italic and `4` for bold-italic.

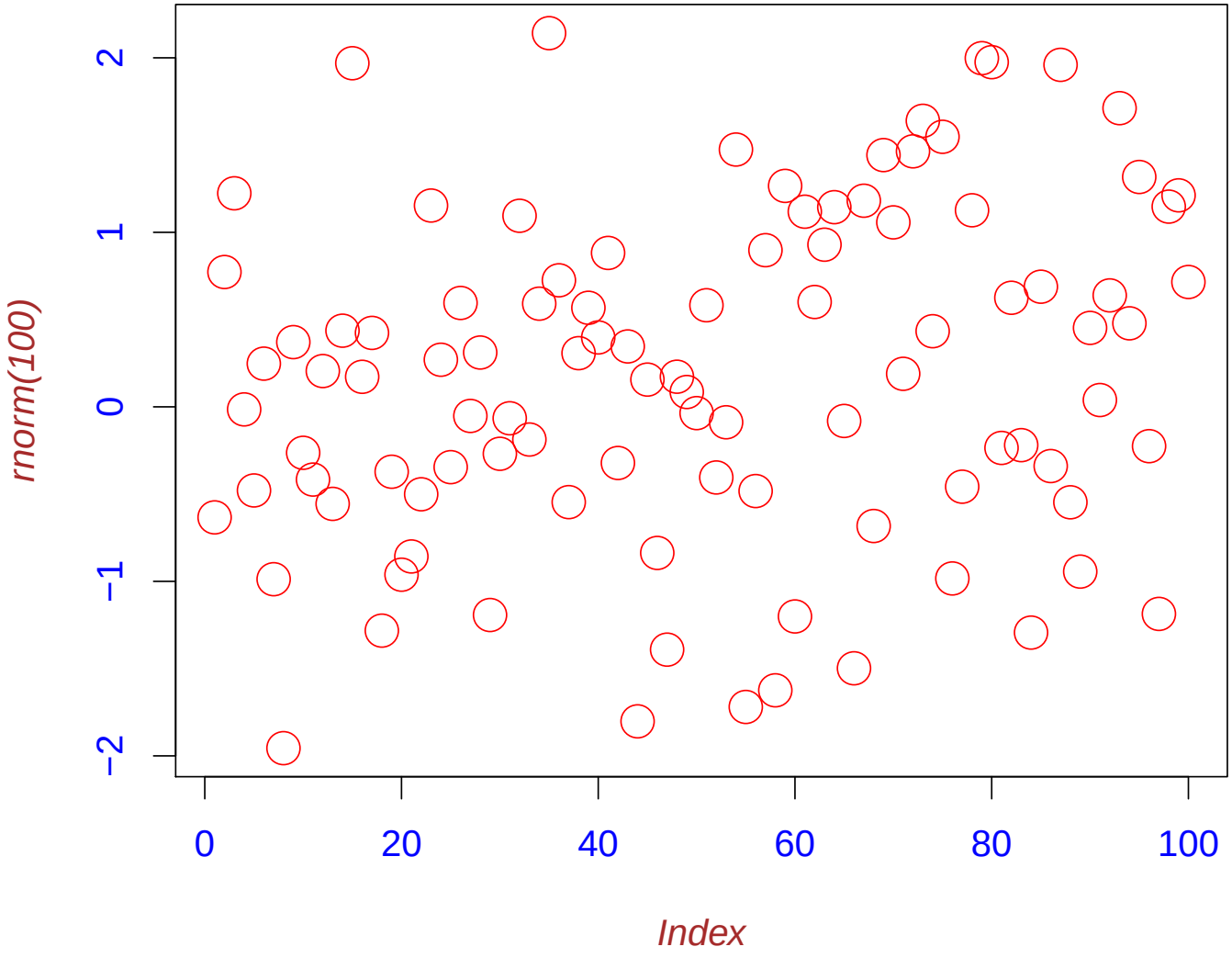
An Example

Many `par` values can be passed as arguments to plotting functions or passed to `par` to set the values for all future plots.

```
> plot(rnorm(100), col = "red", cex = 2,
       font.axis = 1, col.axis = "blue",
       font.main = 4, col.main = "green4",
       font.lab = 3, col.lab = "brown",
       cex.main = 2, main = "Par Overload")
```

```
> par(col = "red", cex = 2,
      font.axis = 1, col.axis = "blue",
      font.main = 4, col.main = "green4",
      font.lab = 3, col.lab = "brown",
      cex.main = 2)
```

Par Overload



All Those Other pars ...

There are many other `par` values which we have not discussed.

You get a description of all the `par` values by looking at the documentation for `par`.

```
> ?par
```

```
> help("par")
```

More Flexible Layouts

The multifigure layouts which can be specified with `par` are very rigid.

All the panels in the plot have exacly the same size.

The `layout` function provides an alternative way of producing multiple figures.

The function still imposes constraints, but they only require that the widths of figures in the same column be equal and that the heights of figures in the same row be equal.

1	2	3
4	5	6
7	8	9

Combining Figures

The arrangements produced by layout are a little more flexible because adjacent figures can be combined.

In the next arrangement, the 4th and 5th figures of the previous layout are combined into a single figure.

Beware, however, that attempts to combine non-adjacent figures produces very strange results.

1	2	3
4		5
6	7	8

Arguments to Layout

There are three main arguments to `layout`; `mat`, `widths` and `heights`.

The `mat` argument associates a figure number with each element of a rectangular layout.

The first example layout had the following `mat` argument.

```
> layout(matrix(1:9, nc = 3, byrow = TRUE),
                    ... )
```

The second layout combined the 4-th and 5-th figures from this layout as follows

```
> layout(matrix(c(1:4,4:8), nc = 3, byrow = TRUE),
                    ... )
```

Arguments to Layout

The widths of the columns in the arrangement are specied by the `widths` argument.

These can either be relative widths - e.g. `c(1, 2)` specifies that the second column is twice as big as the first, or they can be absolute widths specified in centimetres.

The specification

```
widths = c(lcm(1), 1, 2)
```

says that the first column is exactly 1cm wide and the third column is twice the width of the second.

The row heights are specified by heights in a similar fashion.

Layouts and Margins

It is important to note that `layout` ignores the outer margin parameters set with `par`.

All `par` values which work within figures (such as `mar` and `mai`) still have an effect.

It can be useful to set figure margins to 0 when working with sets of figures which are to be juxtaposed.

The effect of outer margins can be obtained by specifing extra rows and columns in the `mat` argument.

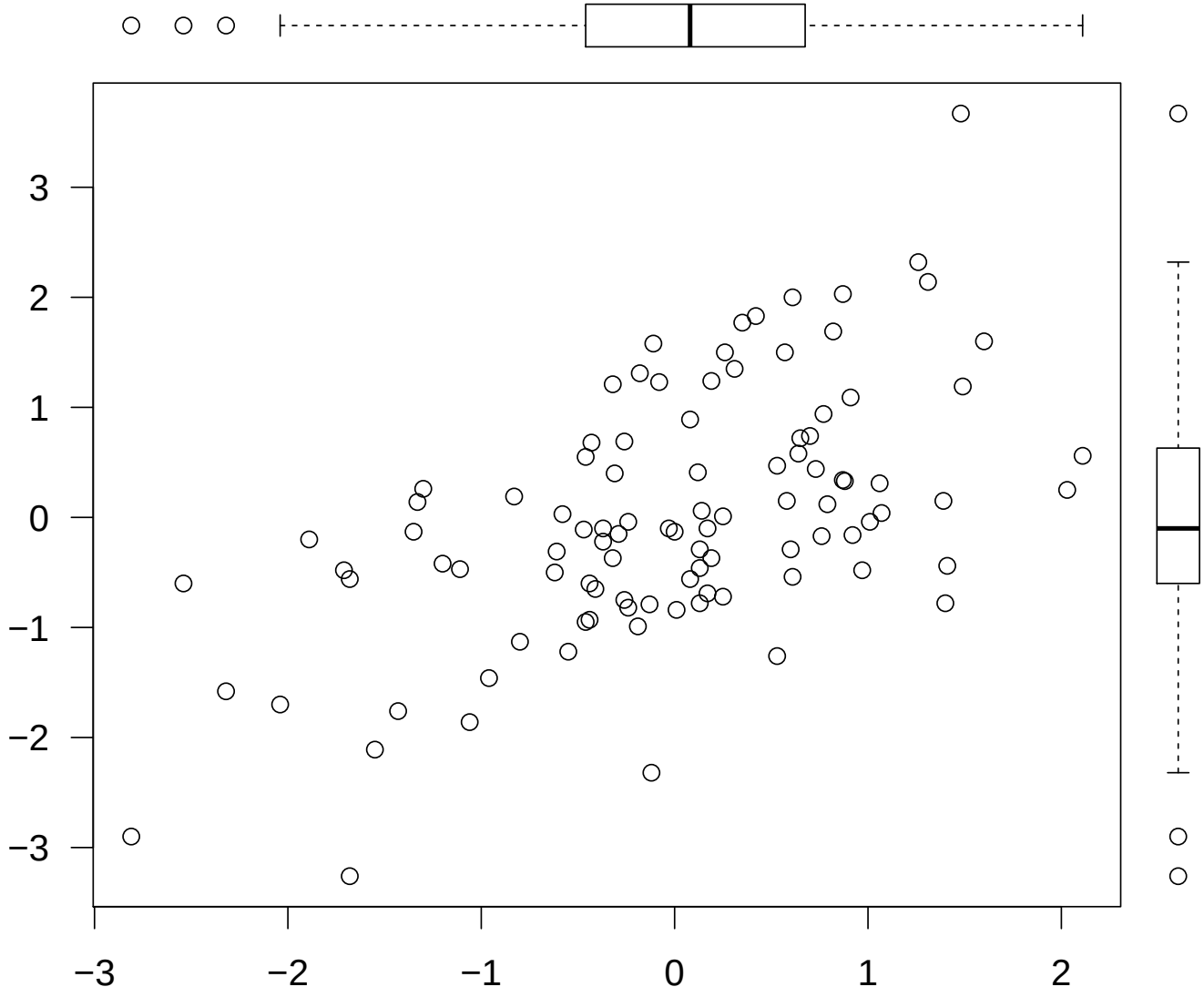
These extra rows and columns can be associated with "figure 0" so that no plotting is done in them.

Example: An Augmented Scatterplot

We'll illustrate what layouts can be used for with a specific example.

The example will produce a standard scatterplot which is enhanced with boxplots of the marginal distributions of the x and y variables.

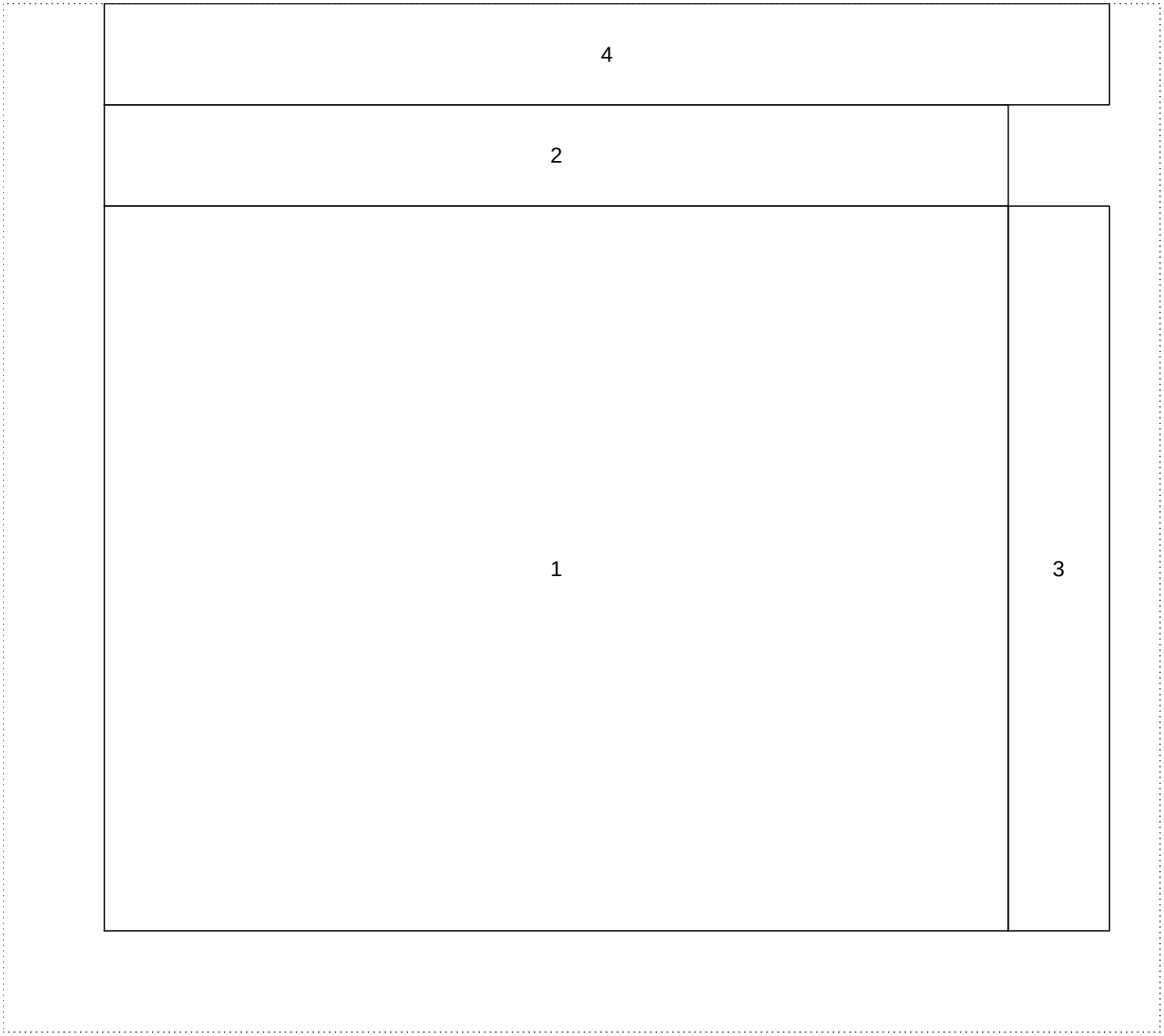
An Enhanced Scatterplot



Choosing a Layout

We need a big figure for the scatterplot, regions along its top and right for the scatterplots and a figure at the very top for a title.

```
> layout(rbind(c(0,4,4,0),
               c(0,2,0,0),
               c(0,1,3,0),
               c(0,0,0,0)),
       height = c(lcm(2), lcm(2), 1, lcm(2)),
       width = c(lcm(2), 1, lcm(2), lcm(1)))
> layout.show(4)
> box("outer", lty = "dotted")
```

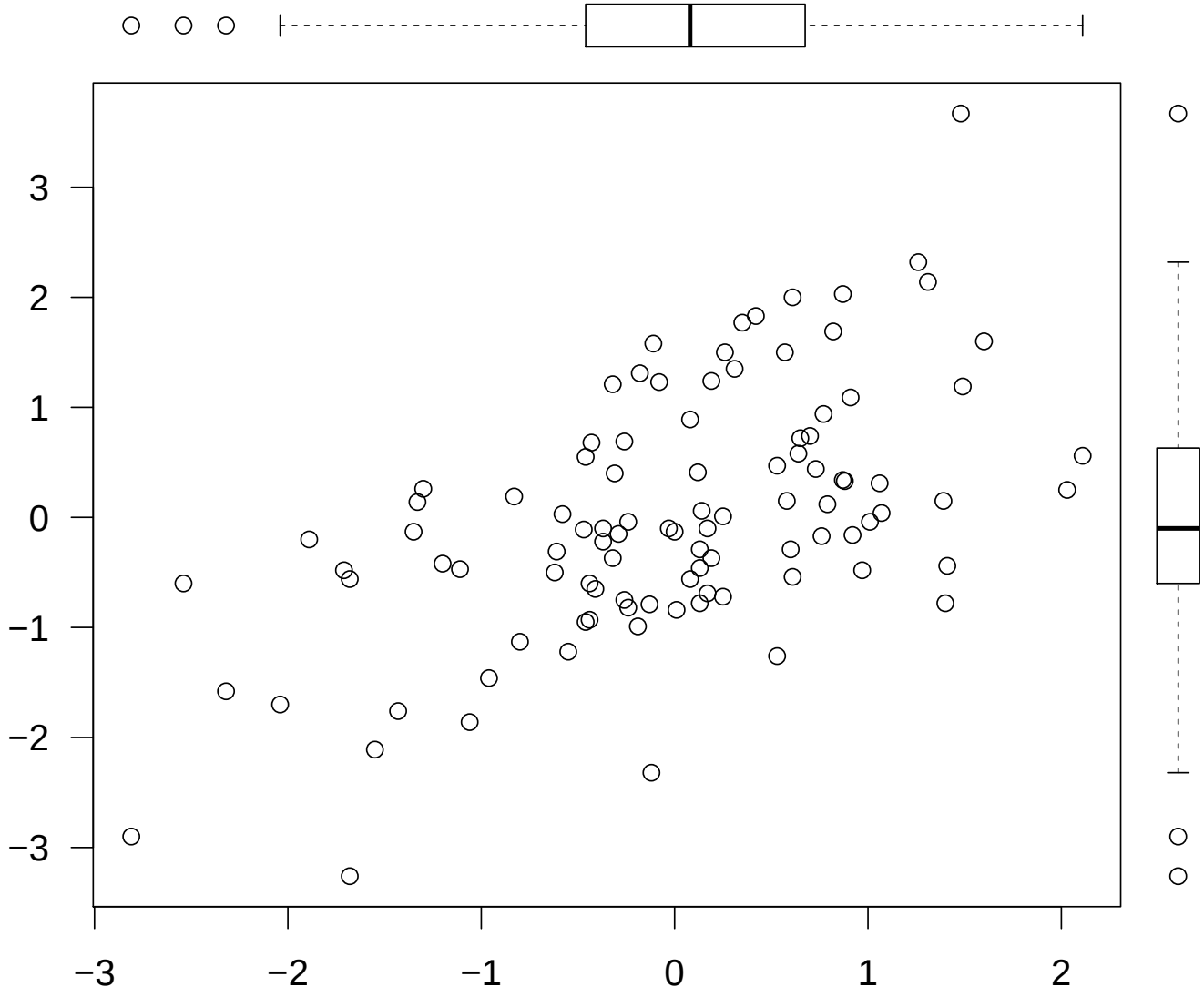

4
2
1
3

Producing the Graph

We constrain the margins of the plots to be zero so that the plots are juxtaposed. Then we produce the plots, in the order specified in `mat`. The last plot is a simple one which contains just the title.

```
> par(mar = rep(0, 4), cex = 1)
> plot(x, y, las = 1)
> boxplot(x, horizontal = TRUE, axes = FALSE)
> boxplot(y, axes = FALSE)
> plot.new()
> plot.window(xlim = c(0, 1), ylim = c(0, 1))
> text(.5, .25, "An Enhanced Scatterplot",
       cex = 1.5, font = 2)
```

An Enhanced Scatterplot



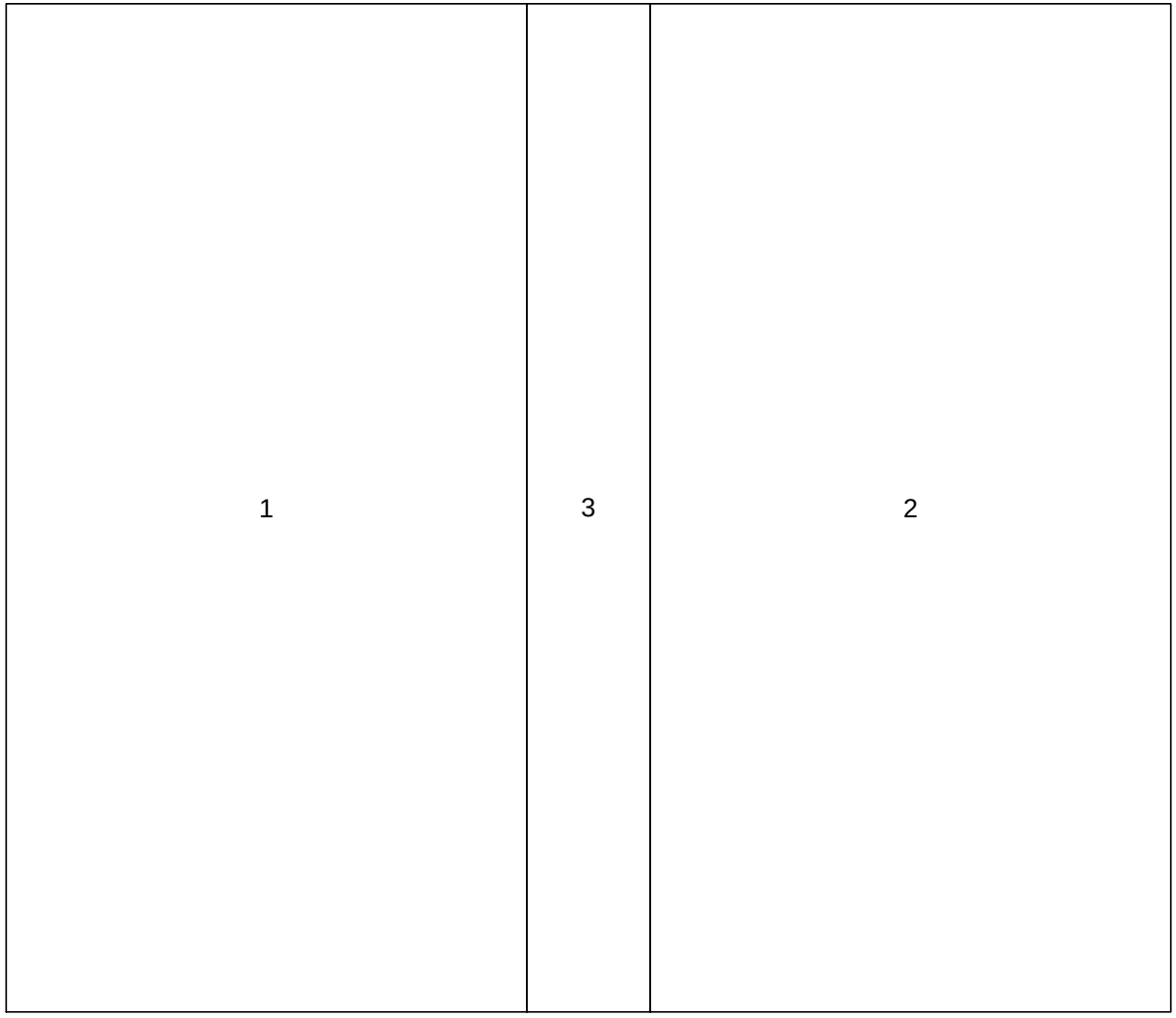
A Two-Panel Plot with Labelling

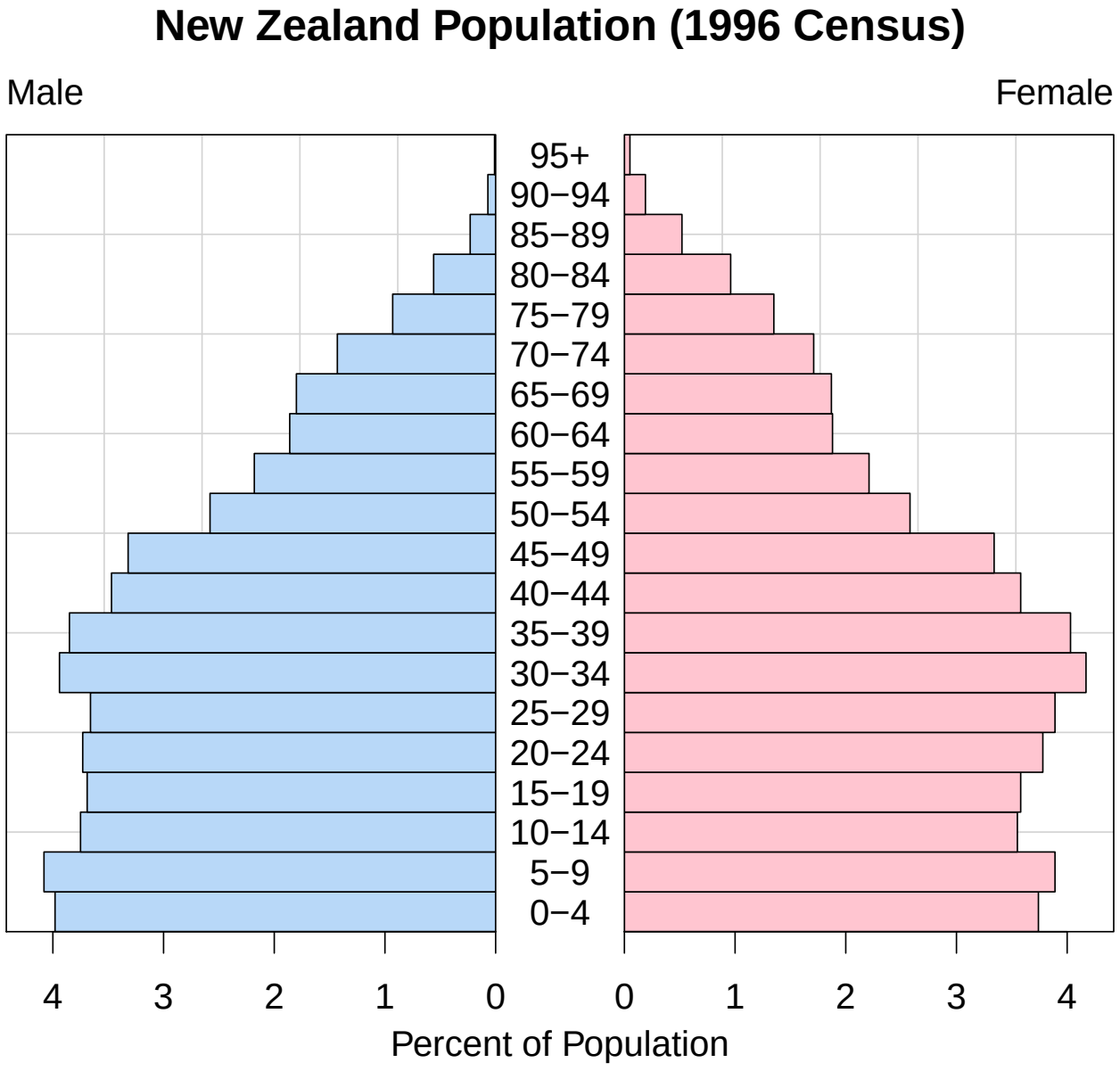
Suppose that (for some reason) we want a plot consisting of two panels with a strip for labelling between them.

How do we arrange this?

```
> layout(matrix(c(0,0,0,0,0,
                  0,1,3,2,0,
                  0,0,0,0,0), nc = 5, byrow = TRUE),
         widths = c(lcm(2), 1, lcm(2), 1, lcm(2)),
         heights = c(lcm(2), 1, lcm(2)))
> layout.show(3)
> box("outer", lty = "dotted")
```

Note that this just a start. You need extra panels for the labelling at the top and bottom, and some space for the axes to appear in.


1
3
2


New Zealand Population (1996 Census)
Male
Female
95+
90−94
85−89
80−84
75−79
70−74
65−69
60−64
55−59
50−54
45−49
40−44
35−39
30−34
25−29
20−24
15−19
10−14
5−9
0−4
4
3
2
1
0
0
1
2
3
4
Percent of Population

Layouts and Plots

We've seen that layouts provide a way to partition the page into a number of regions which can be used to draw multiple figures or a single plot with multiple components.

One simple approach is to use a high-level R plotting function such as `plot`, `hist`, `barplot` or `boxplot`, and to let the high level function take care of all the details.

An alternative, much more flexible, approach is to divide the page into subregions which each forms a single component of a graph.

It is useful to create a software component to match each of the types of region which will appear in the graph.

Handling Margins

Margins are added to plots to provide space for axes and annotations such as axis labels and titles.

Under the component model, such elements are represented by separate plotting areas, so the need for margins is gone.

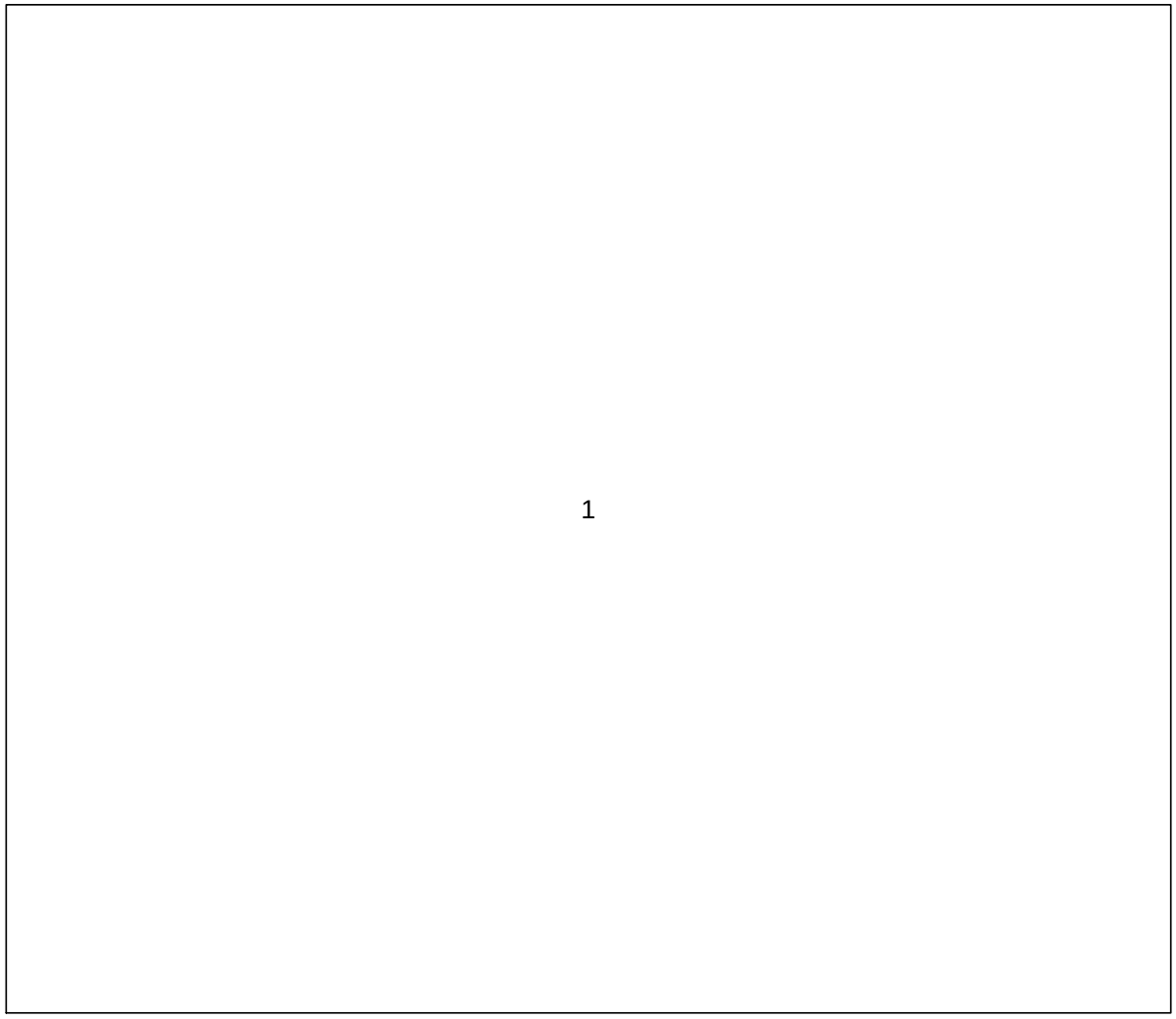
When working with layouts in this way it can be very useful to remove the margins completely with a specification of the form

```
> opar = par(mar = rep(0, 4))
```

Example: A Simple Scatterplot

We can build a layout for a simple scatterplot as follows:

```
> layout(matrix(c(0, 0, 0,
                  0, 1, 0,
                  0, 0, 0), nc = 3, byrow = TRUE),
         widths = c(lcm(2), 1, lcm(2)),
         heights = c(lcm(2), 1, lcm(2)))
> layout.show(1)
```


1

Choice of Margin Size

While it is possible to specify the margin size in centimetres, it is much more natural to specify it in lines of text (just like the `mar` and `oma` parameters).

By defining the function

```
> llines =
      function(x)
      lcm(x * par("csi") * 2.54)
```

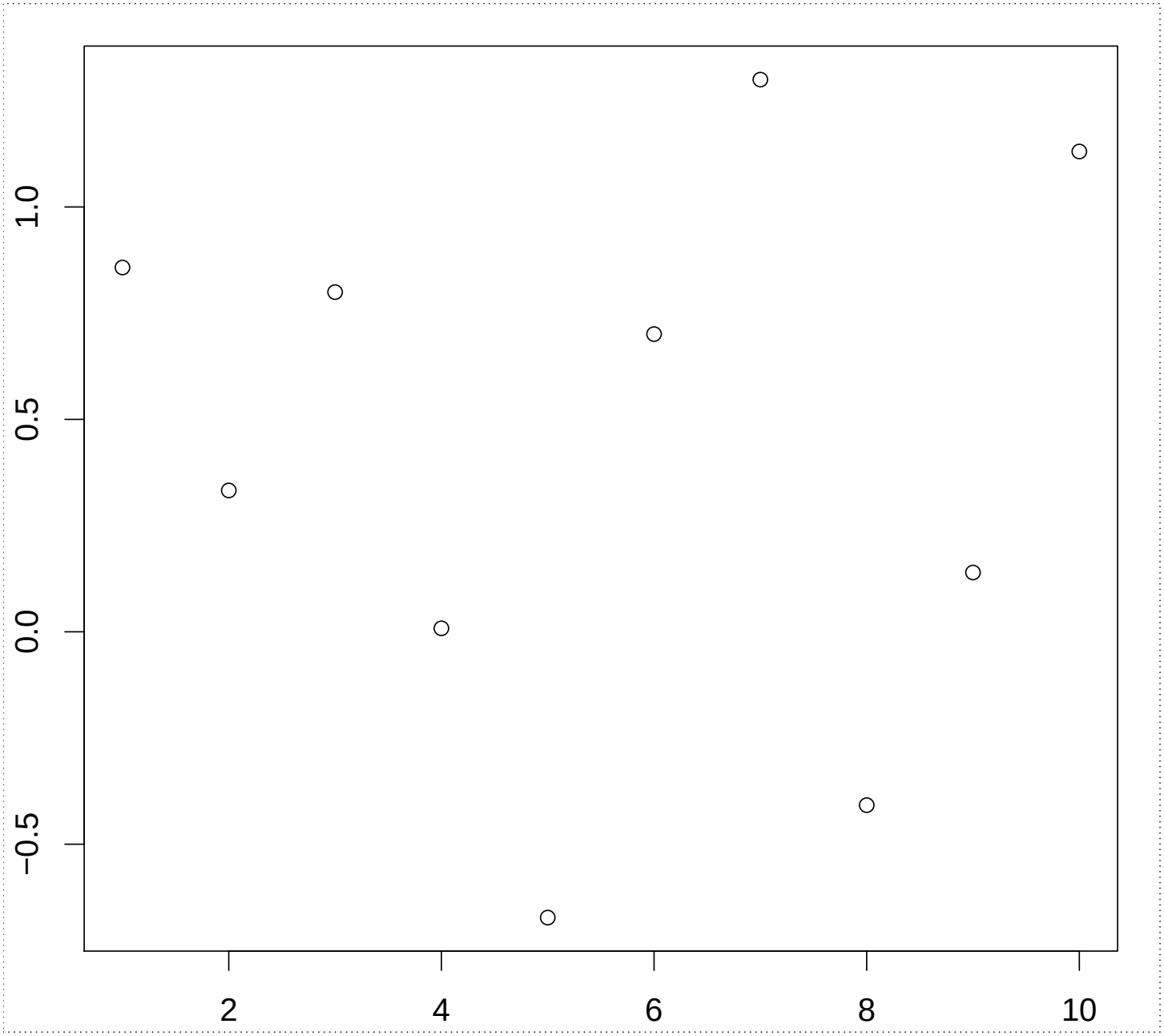
it is possible to specify margin size in lines of text.

A margin of 2.1 lines is just enough to fit the standard axis (with a small amount of extra space).

Example: A Simple Scatterplot

We can build a layout for a simple scatterplot, using just the minimum of space needed for the axes as follows:

```
> par(mar = rep(0, 4))
> layout(matrix(c(0, 0, 0,
                  0, 1, 0,
                  0, 0, 0), nc = 3, byrow = TRUE),
         widths = c(llines(2.1), 1, llines(1.1)),
         heights = c(llines(1.1), 1, llines(2.1)))
> par(cex = 1)
> plot(rnorm(10), ann = FALSE)
```


1.0
0.5
0.0
−0.5
2
4
6
8
10

Axis Labels

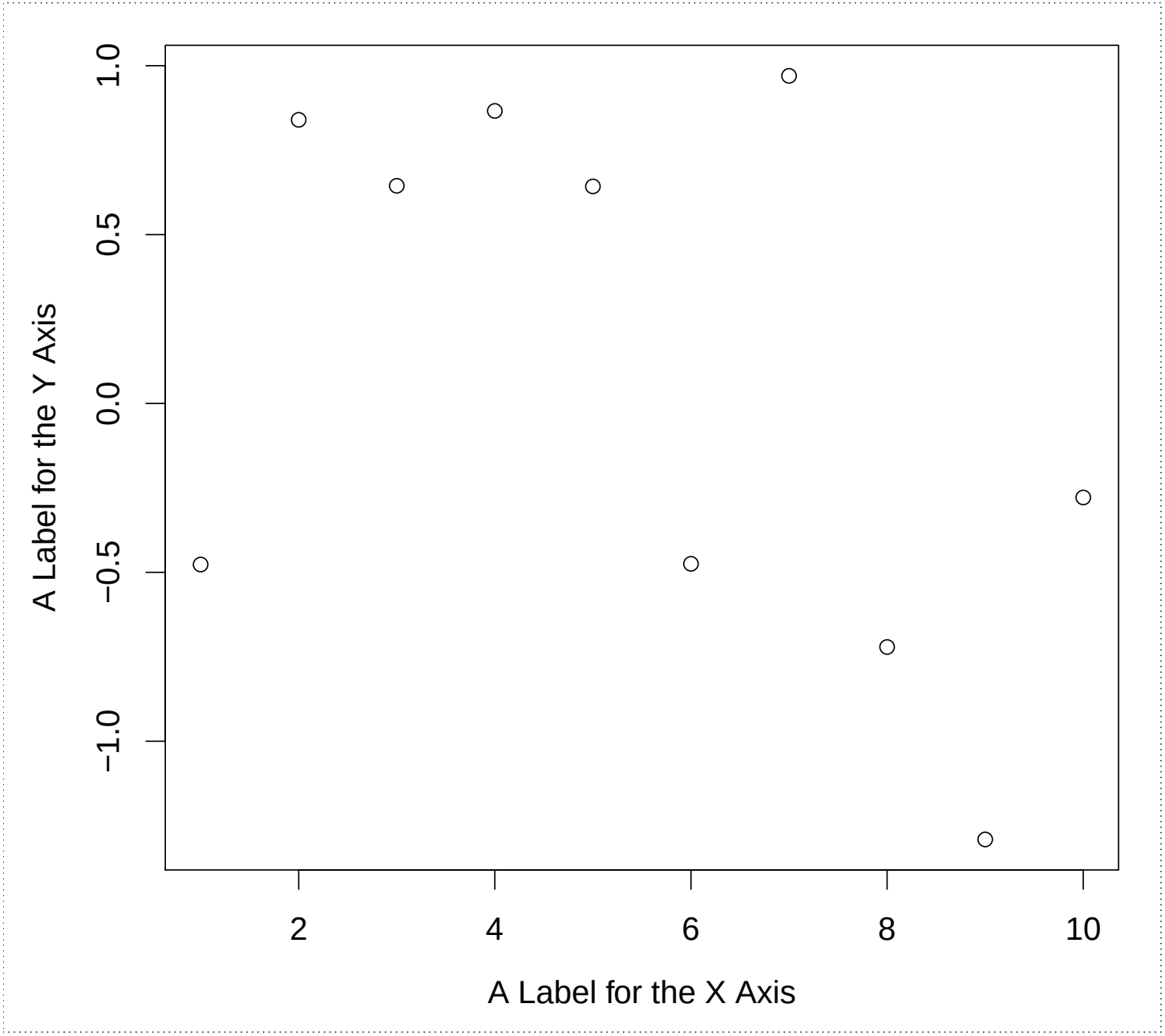
Now suppose that we want to add axis labels to the scatterplot. This means adding another line at the top and left of the layout.

```
> par(mar = rep(0, 4))
> layout(matrix(c(0, 0, 0, 0,
                  3, 0, 1, 0,
                  0, 0, 0, 0,
                  0, 0, 2, 0),
                nc = 4, byrow = TRUE),
         widths = c(llines(2.1), llines(2.1),
                    1, llines(1.1)),
         heights = c(llines(1.1), 1,
                     llines(2.1), llines(2.1)))
> par(cex = 1)
```

Producing the Labels

The plot labels are produced by moving to the region for the label (with `plot.new`) and drawing the text at the centre of the region.

```
> plot.new()
> text(0.5, 0.5, "An X Label")
> plot.new()
> text(0.5, 0.5, "A Y Label", srt = 90)
```

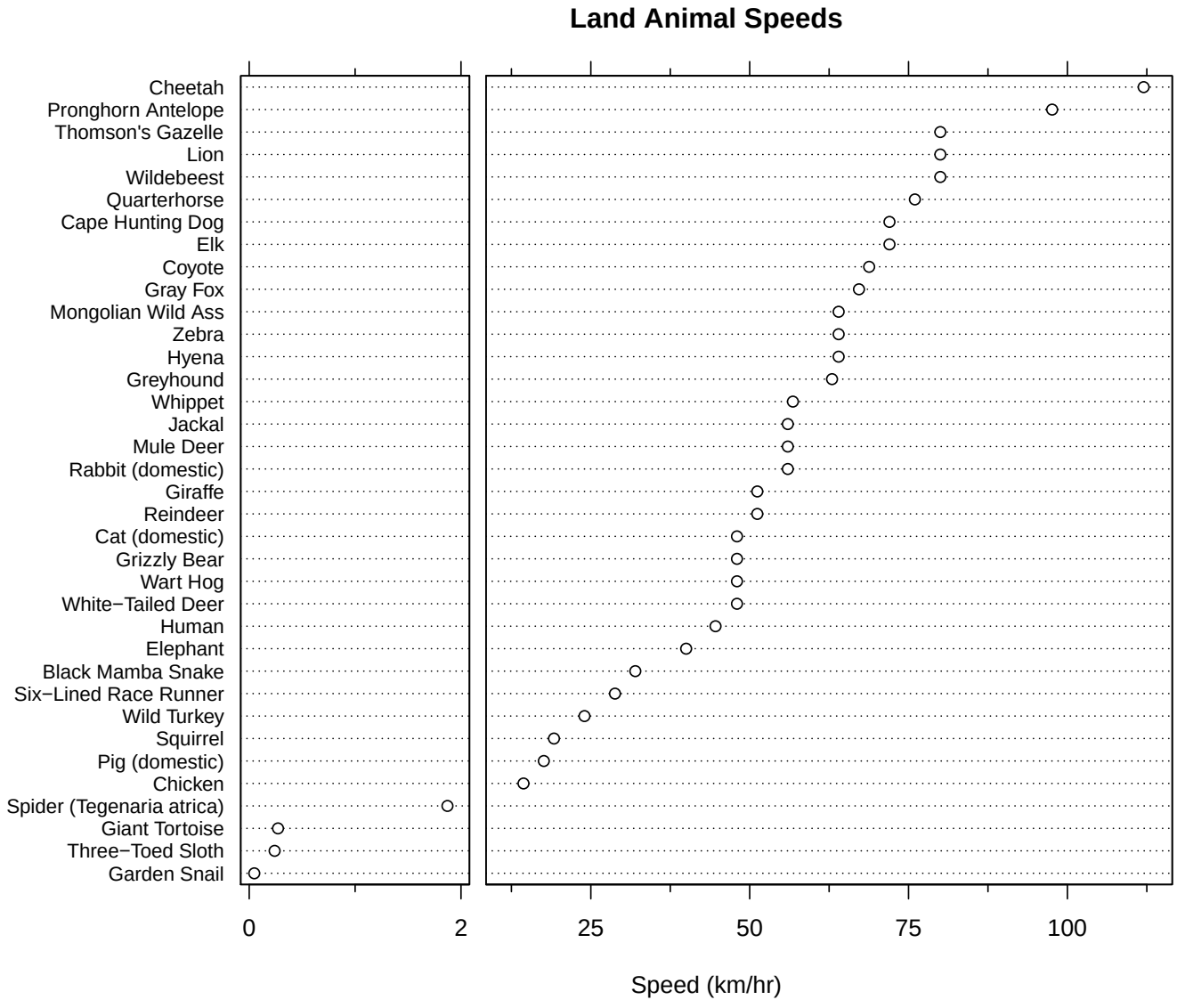

A Label for the Y Axis
1.0
0.5
0.0
−0.5
−1.0
2
4
6
8
10
A Label for the X Axis

Other Features

It is possible to add other features to the plot such as additional axes at the top and right or an overall title by adding further plotting regions.

With a little willingness to experiment, there is virtually no limit to the kinds of plot which you can produce in this way.

The following plot shows a dotchart comparison of land animal speeds produced in exactly this way.


Land Animal Speeds
Cheetah
Pronghorn Antelope
Thomson's Gazelle
Lion
Wildebeest
Quarterhorse
Cape Hunting Dog
Elk
Coyote
Gray Fox
Mongolian Wild Ass
Zebra
Hyena
Greyhound
Whippet
Jackal
Mule Deer
Rabbit (domestic)
Giraffe
Reindeer
Cat (domestic)
Grizzly Bear
Wart Hog
White−Tailed Deer
Human
Elephant
Black Mamba Snake
Six−Lined Race Runner
Wild Turkey
Squirrel
Pig (domestic)
Chicken
Spider (Tegenaria atrica)
Giant Tortoise
Three−Toed Sloth
Garden Snail
0
2
25
50
75
100
Speed (km/hr)