Drawing Contexts in Grid Graphics

December 17, 2001

In Grid, all drawing is ultimately performed by a set of predefined primitives (lines, rectangles, text, ...) and the output produced by a primitive depends on two sorts of things:

1. the location and size of the primitive; I call these *structural* parameters.
2. the colour, linewidth, fontsize, ... of the primitive; I call these *graphical* parameters.

Furthermore, all drawing occurs within a viewport, which consists of three sorts of things:

1. the location and size of the viewport (structural parameters).
2. the colour, linewidth, fontsize, ... of the viewport (graphical paramters).
3. a number of coordinate systems, which I call the *drawing context* defined by the viewport.

For both primitives and viewports, the same set of parameters may produce different final results when drawn within different drawing contexts. One consequence of this is that, in order to replicate the result from drawing a primitive or setting a viewport (e.g., when redrawing an image), it is vital that the same drawing context can be reestablished.

1 Structural parameters

Location and size parameters are specified using `"unit"` objects. For example, `unit(1, "inches")` specifies a location or size of 1`''`.

Some units are absolute (such as inches and centimetres), but most are relative to the current drawing context - the coordinate systems defined by the current viewport.

In this sense, the location and size of objects are *declarative*.

2 Graphical parameters

Graphical parameters are specified using a `"gpar"` object. A primitive or viewport does not have to specify any or all possible graphical parameters. Anything that is specified overrides any previous (parent) settings and provides the default for any subsequent (child) settings.

In this sense, graphical parameters are *inherited*.

A primitive's graphical parameter settings take effect just before the primitive is drawn, and their effects are reversed just after the primitive has finished drawing[1].

A viewport's graphical parameter settings take effect just befire the viewport is pushed onto the viewport stack, and their effects are reversed just after the viewport is popped off the stack.

[1]Technically, the graphical parameters are only in effect during the call to the `draw.details` method

3 The Drawing Process

Because primitives and viewports contain a declarative specification of their location and size, and because graphical parameters are inherited, a lot of calculation is required *at drawing time* to determine where the output should end up and what it should look like.

For example, the following command requires establishing the current physical location and size of the parent viewport in order to determine the location that is `1''` in from the bottom-left corner, and the current fontsize must be established in order to determine how large the text should be.

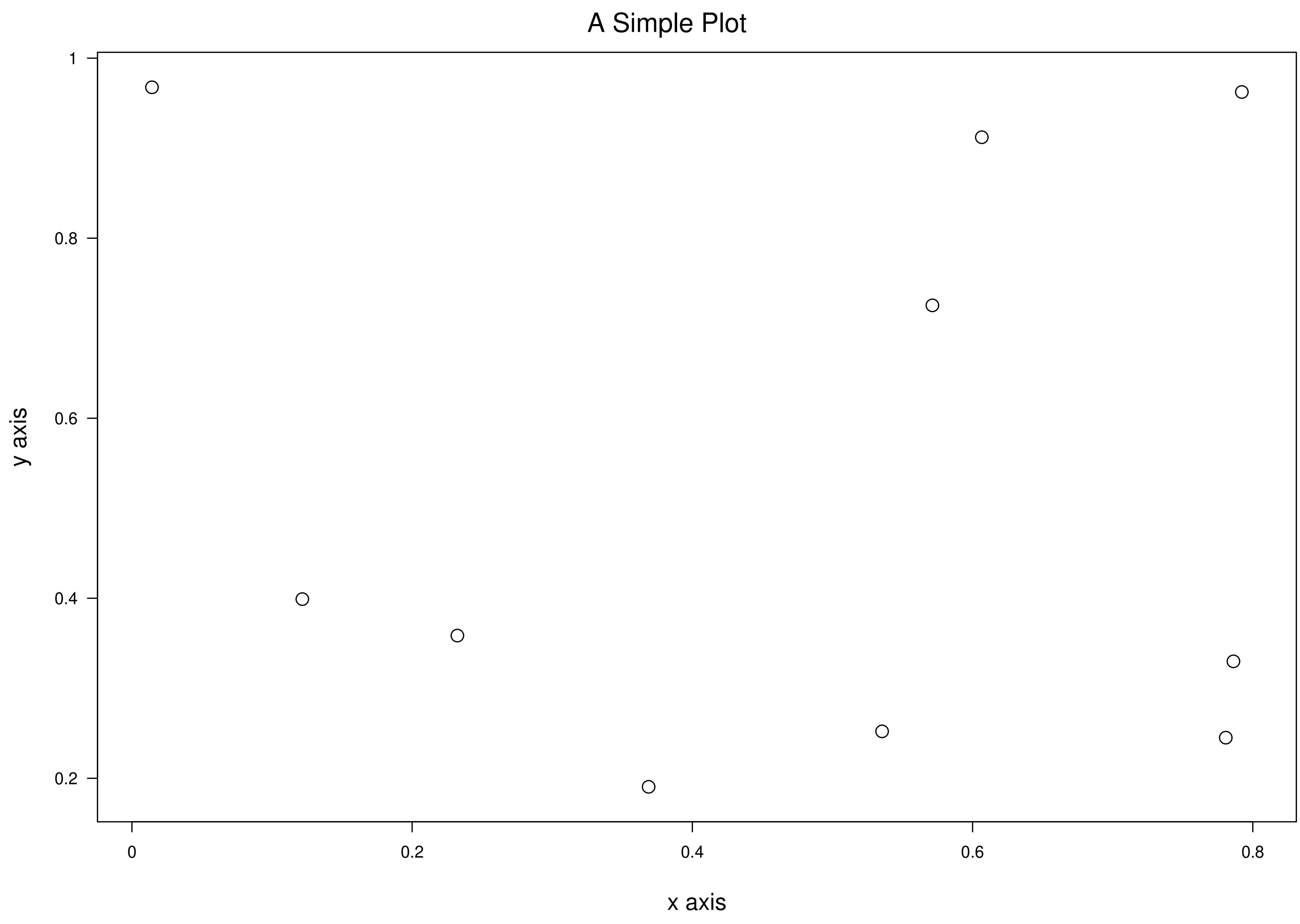
```
grid.text("hi", x=1, y=1, default.unit="inches")
```

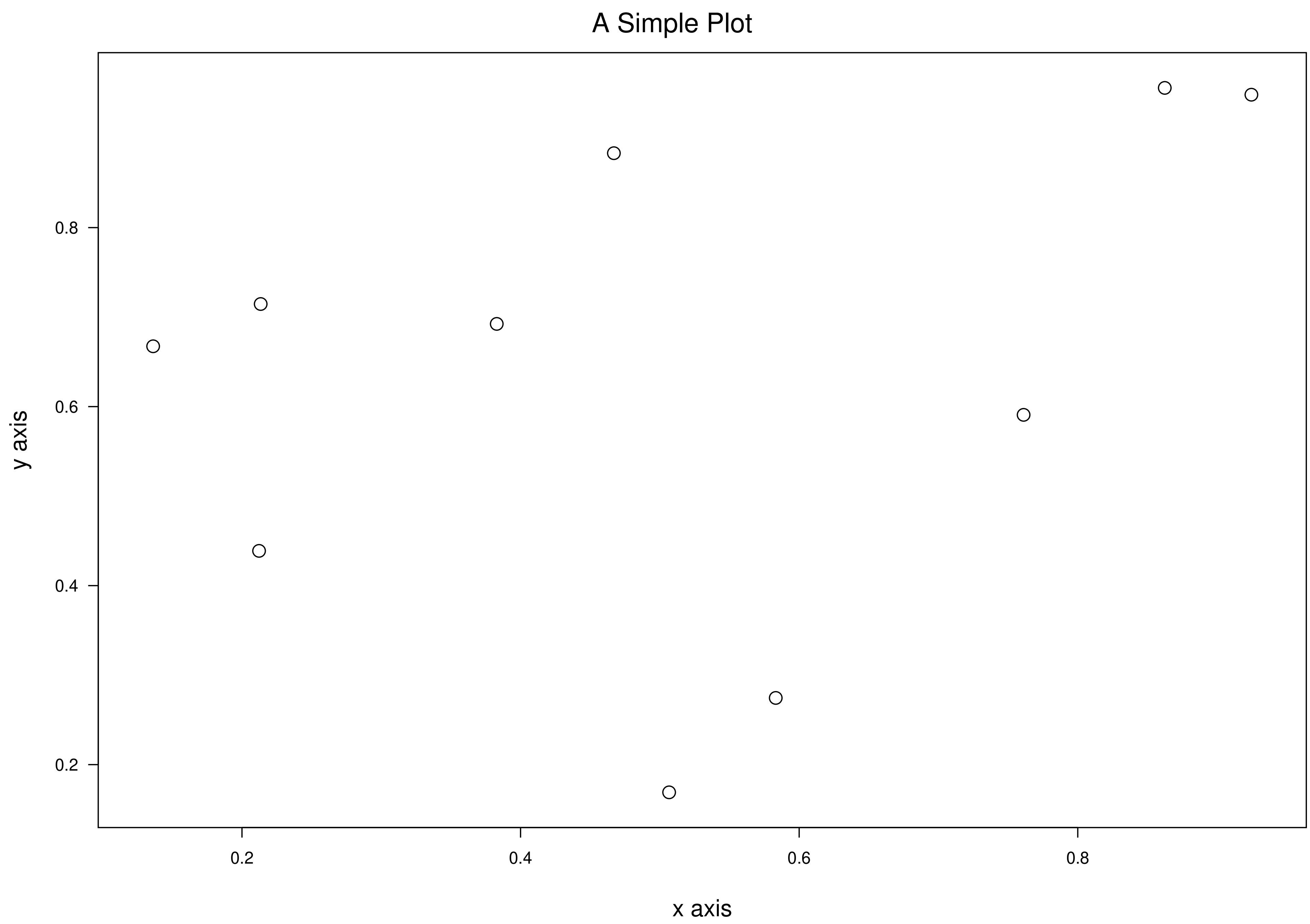
4 Coordinate Systems

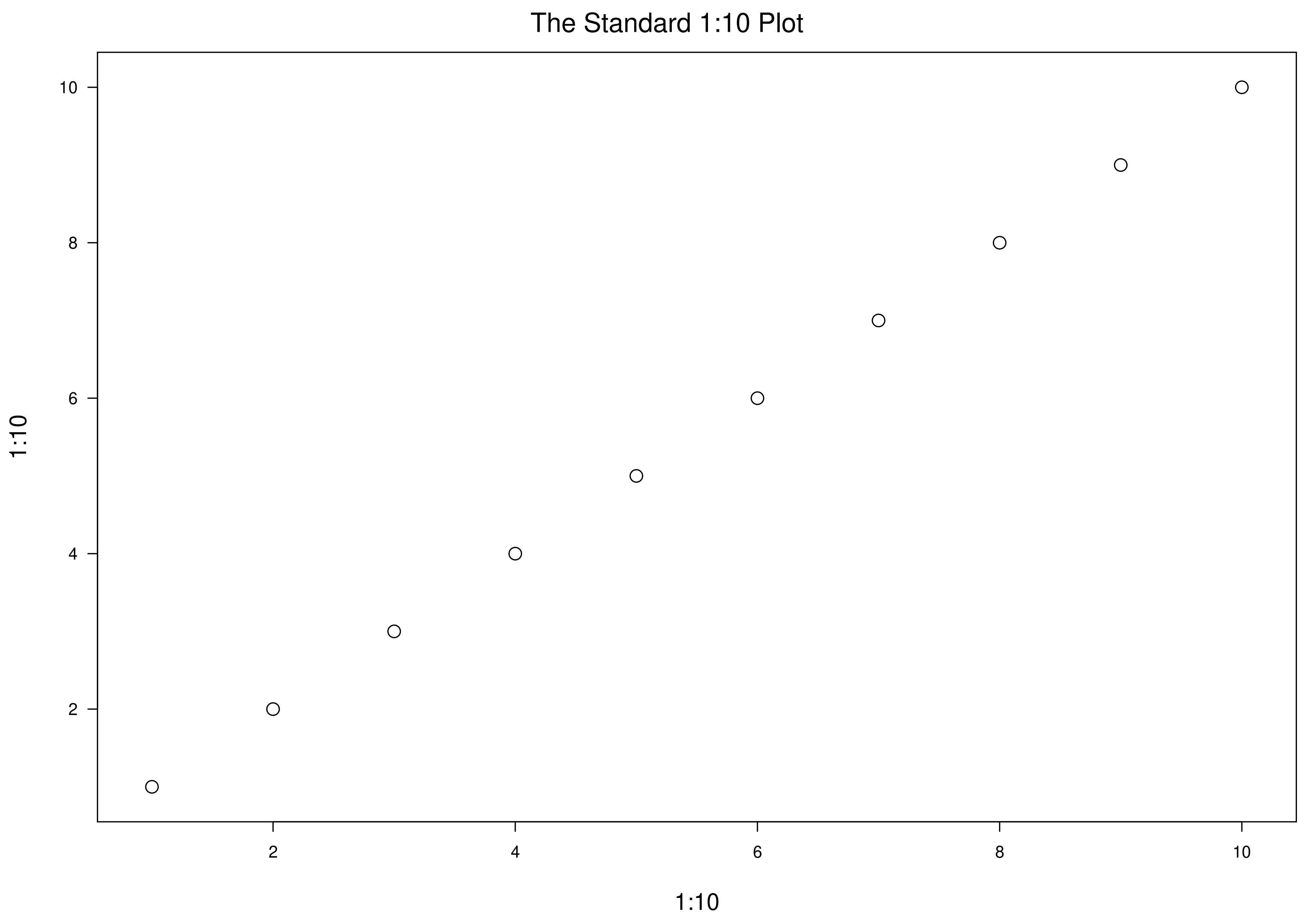
Each coordinate system within a viewport is specified through one or more parameters. For example, the `"native"` units on the x-dimension are specified through a minimum value and a maximum value.

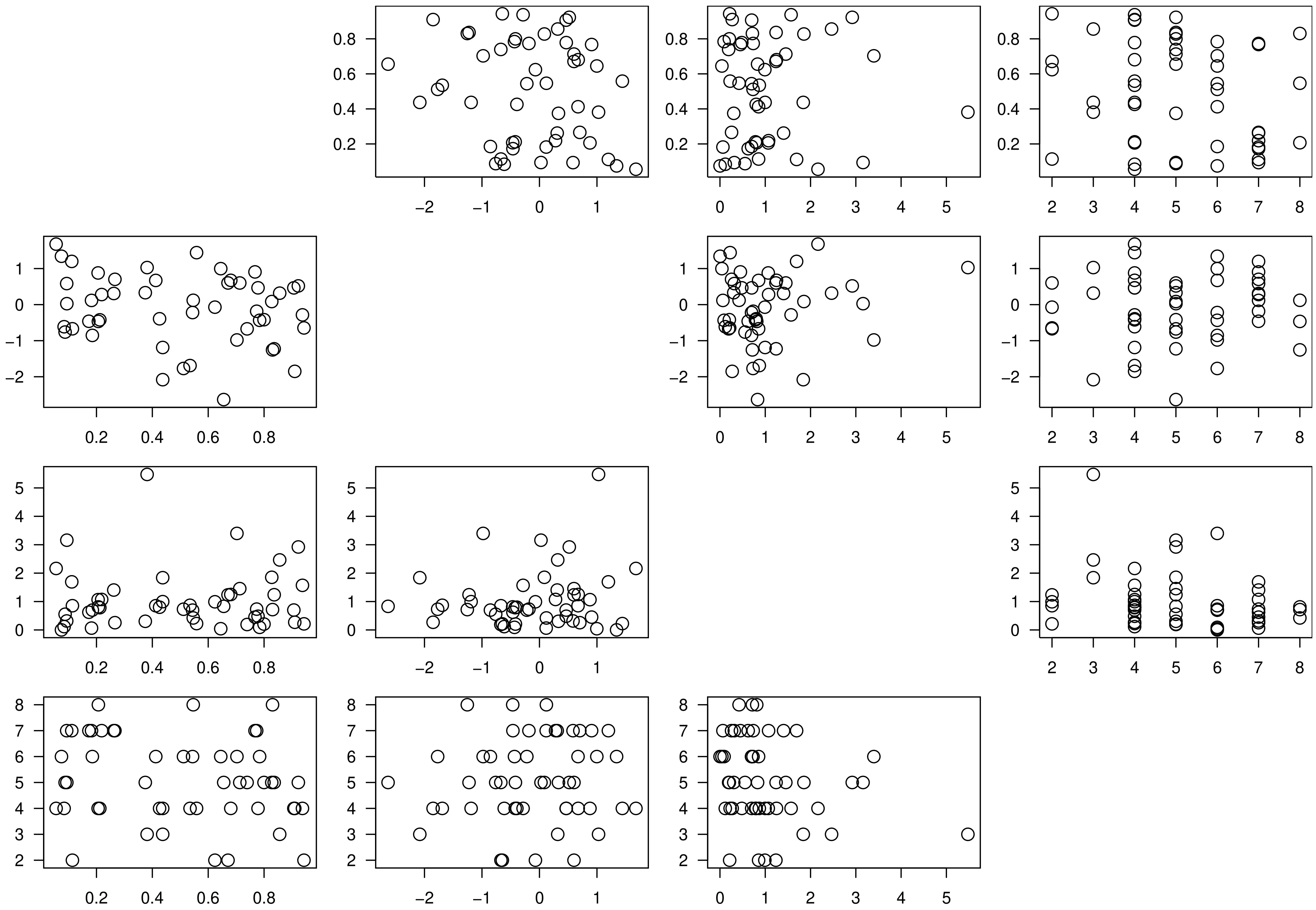
Most of these coordinate system parameters are stored in the viewport, however, two of the coordinate systems have special parameters. The `"lines"` and `"char"` units are based on the current `lineheight` and `fontsize`, which are general graphical parameters. Viewports *can* have a lineheight and/or fontsize specified, but this is not required. For example, a viewport may not specify either of fontsize or lineheight if it just wants to inherit these values from its parent viewport.

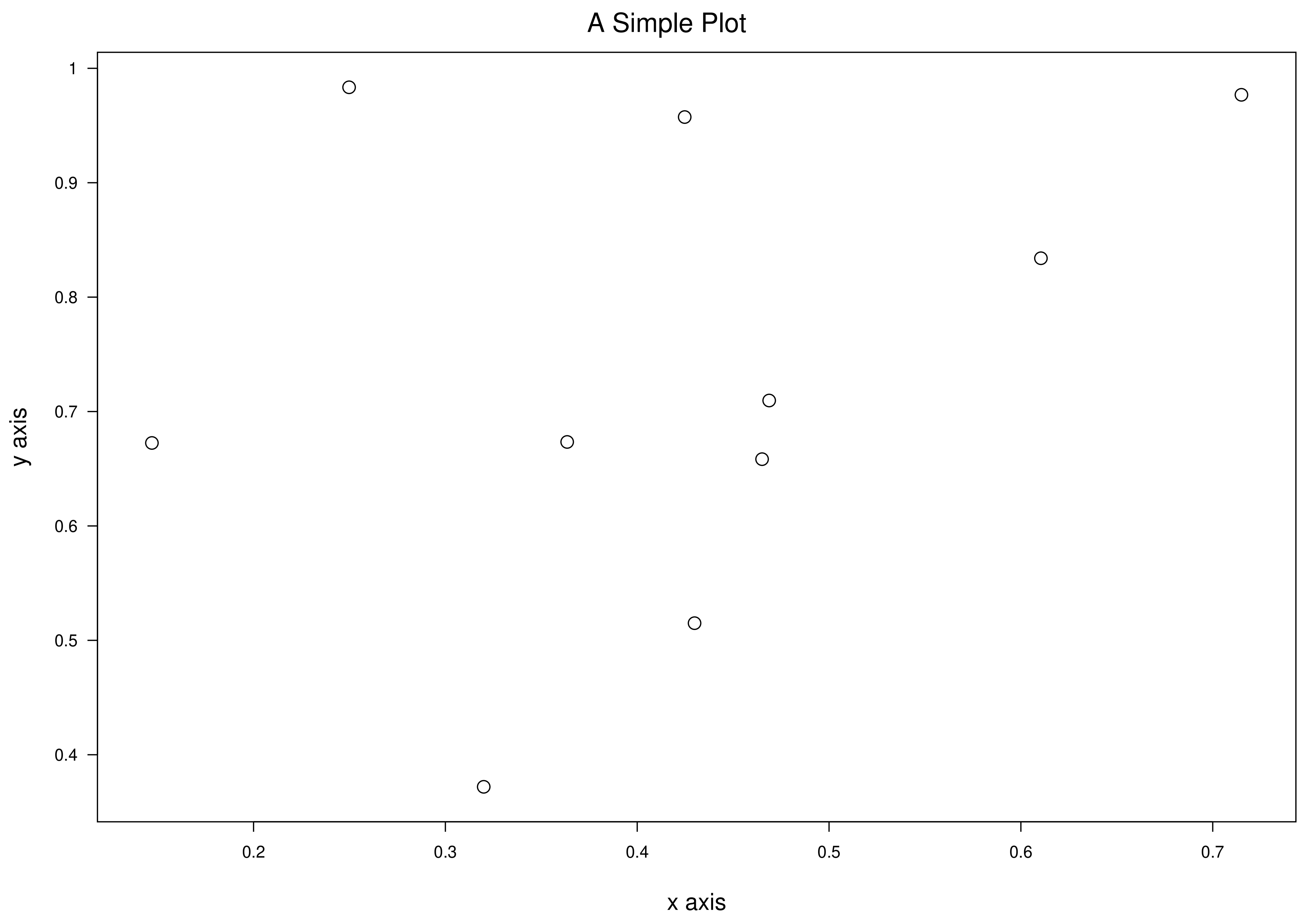
The drawing process requires that we can establish things like the current fontsize and lineheight for each viewport.


A Simple Plot
y axis
1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
x axis

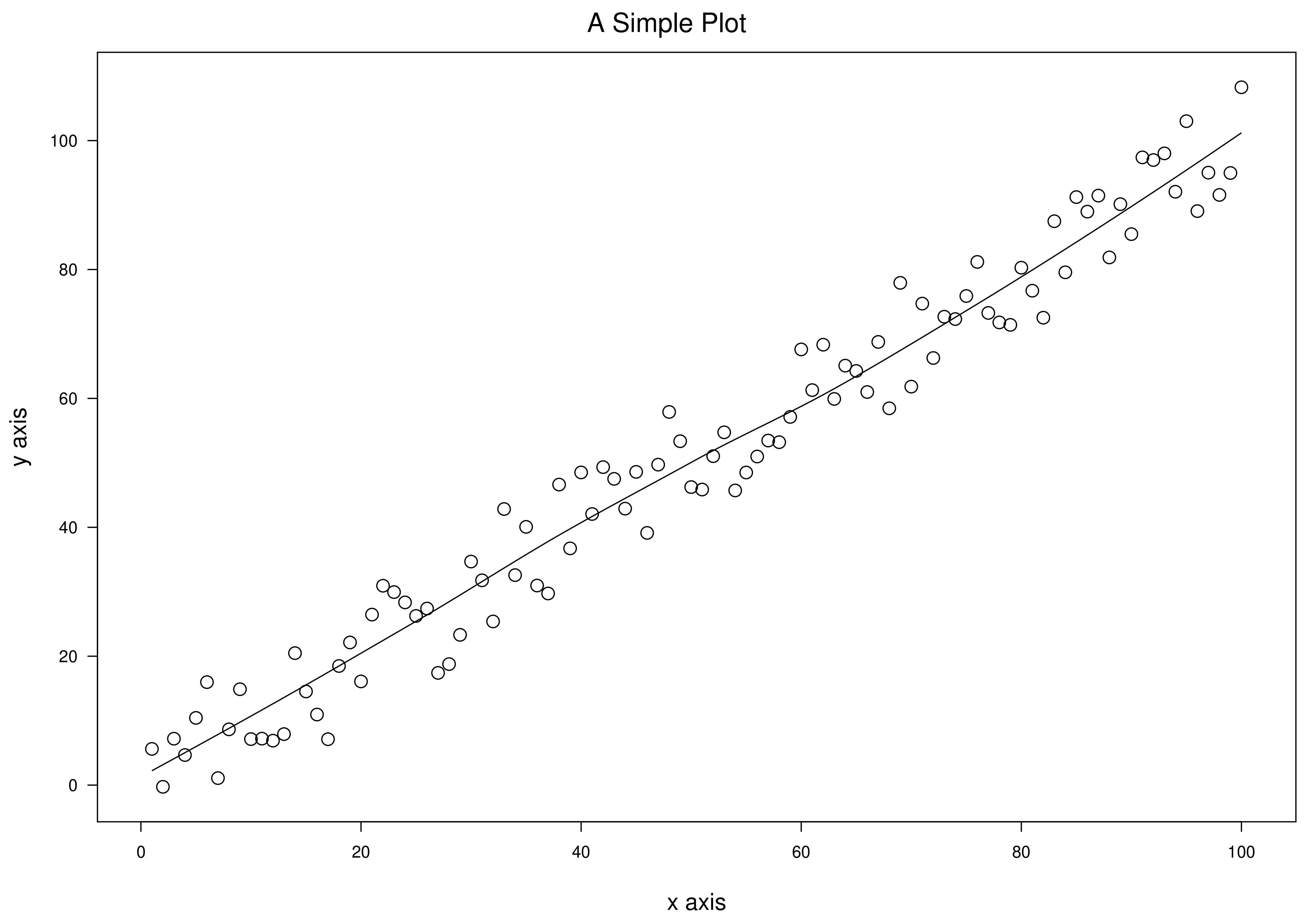

A Simple Plot
y axis
0.8
0.6
0.4
0.2
x axis
0.2
0.4
0.6
0.8

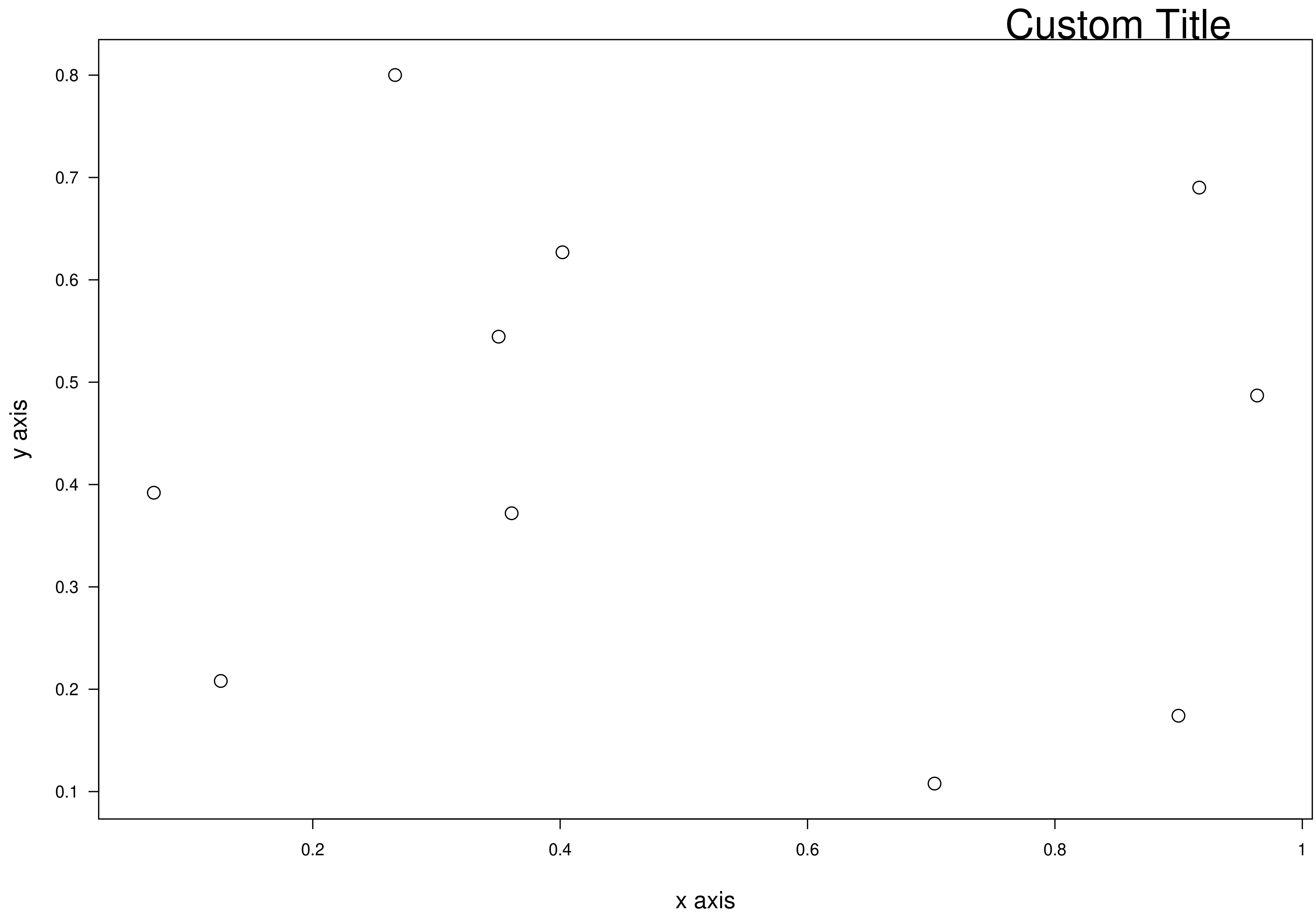

The Standard 1:10 Plot
1:10
2
4
6
8
10
1:10
2
4
6
8
10

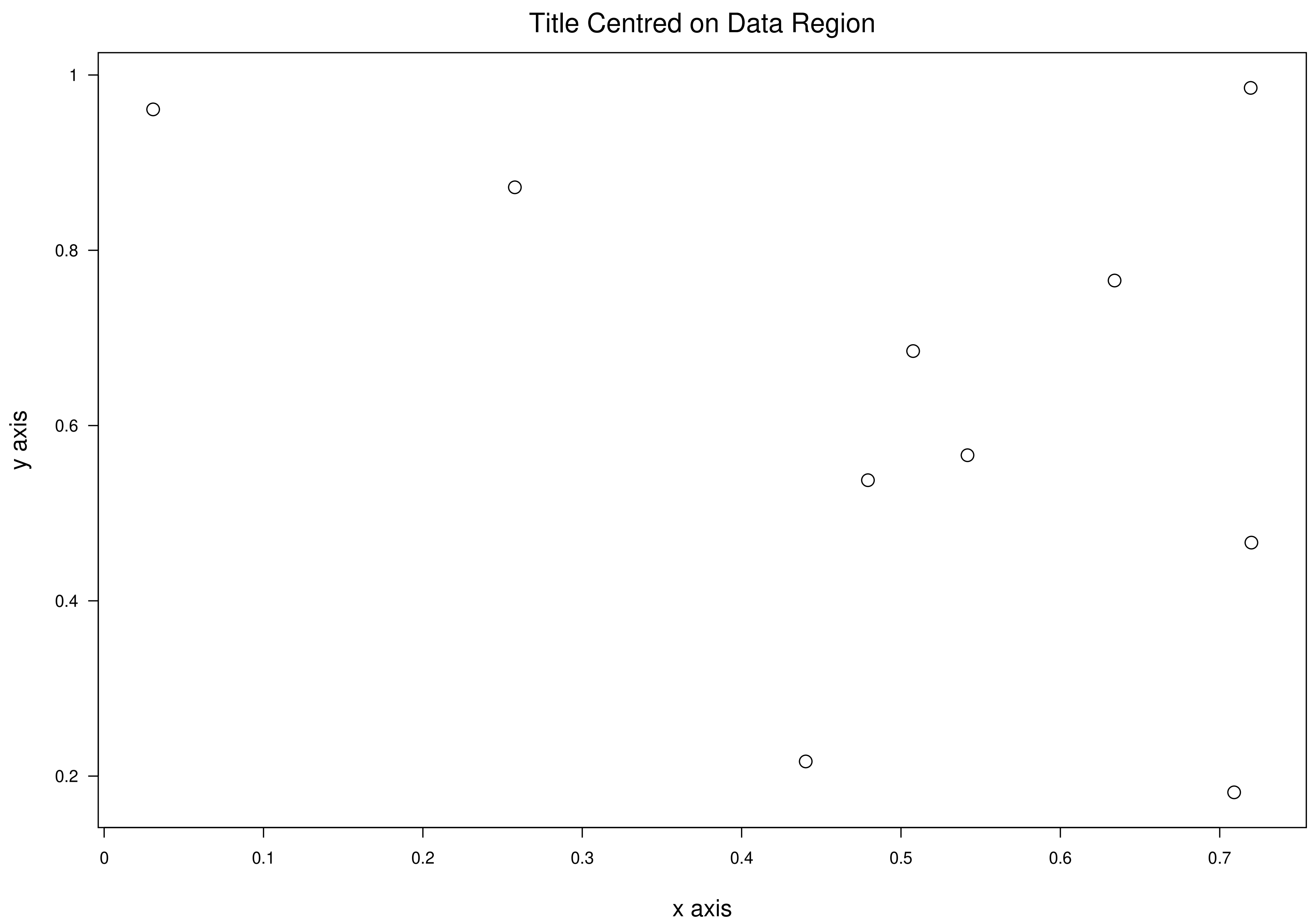


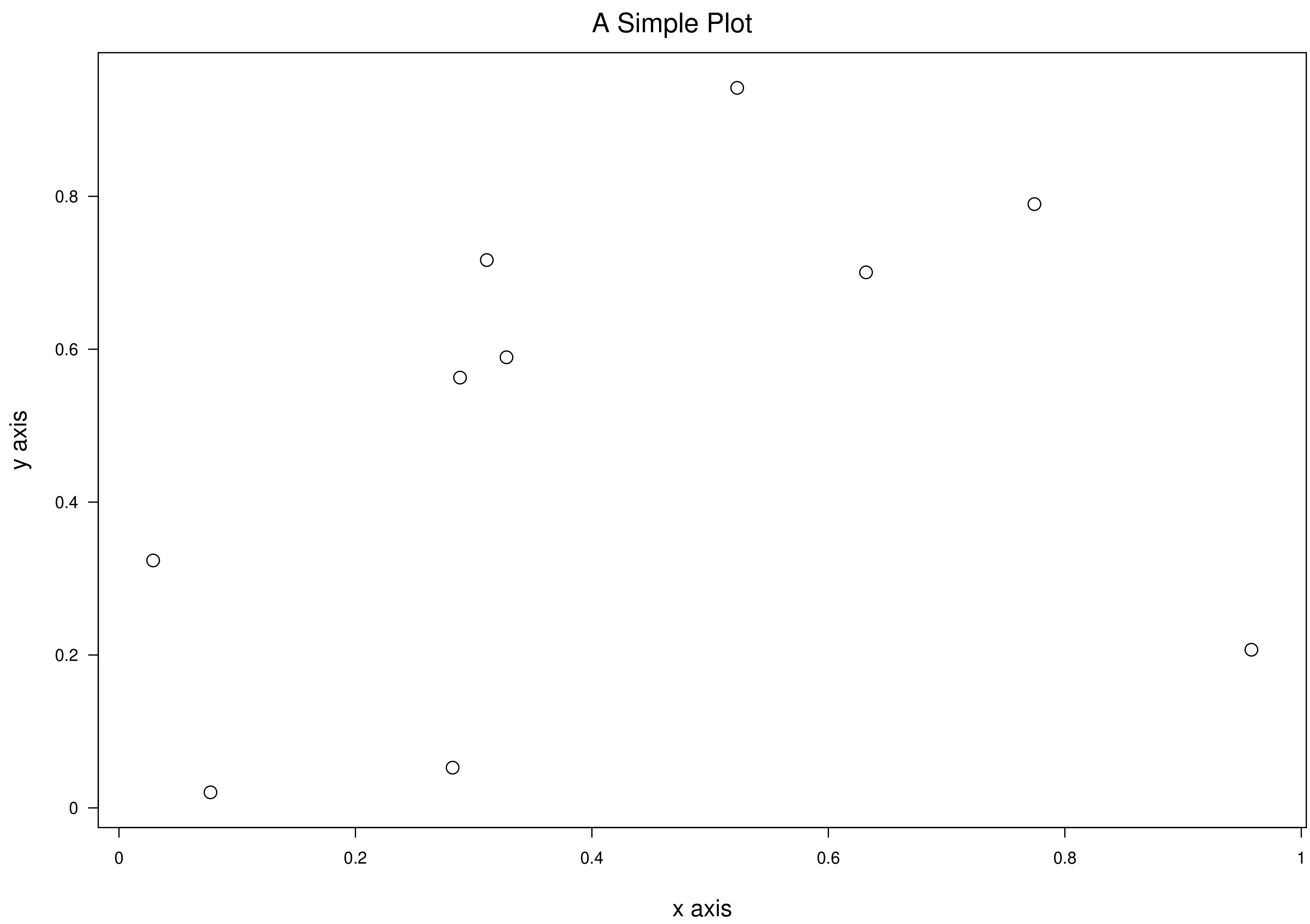


A Simple Plot
y axis
1
0.9
0.8
0.7
0.6
0.5
0.4
0.2
0.3
0.4
0.5
0.6
0.7
x axis


A Simple Plot
y axis
x axis
0
20
40
60
80
100

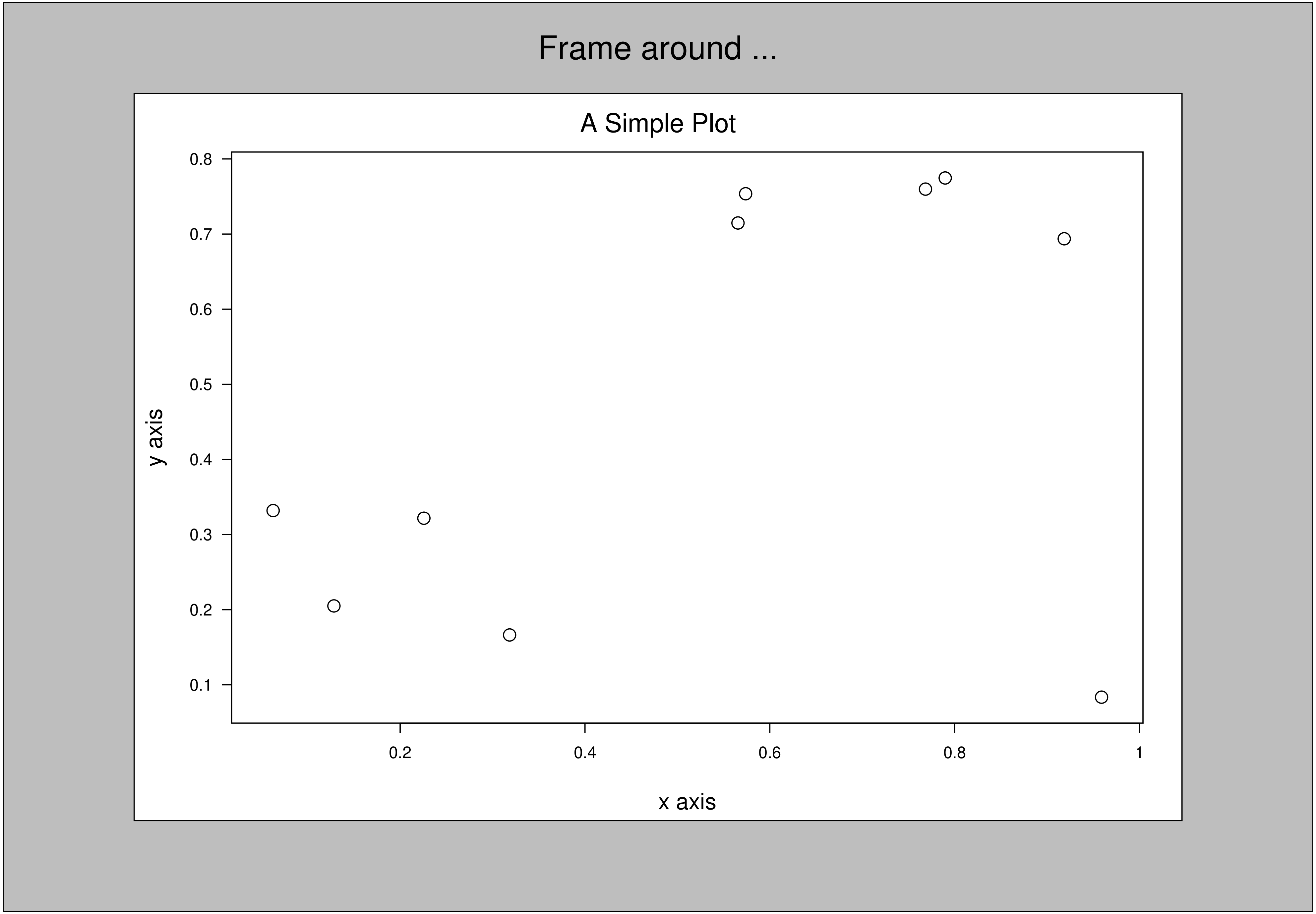

Custom Title
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
y axis
0.2
0.4
0.6
0.8
1
x axis

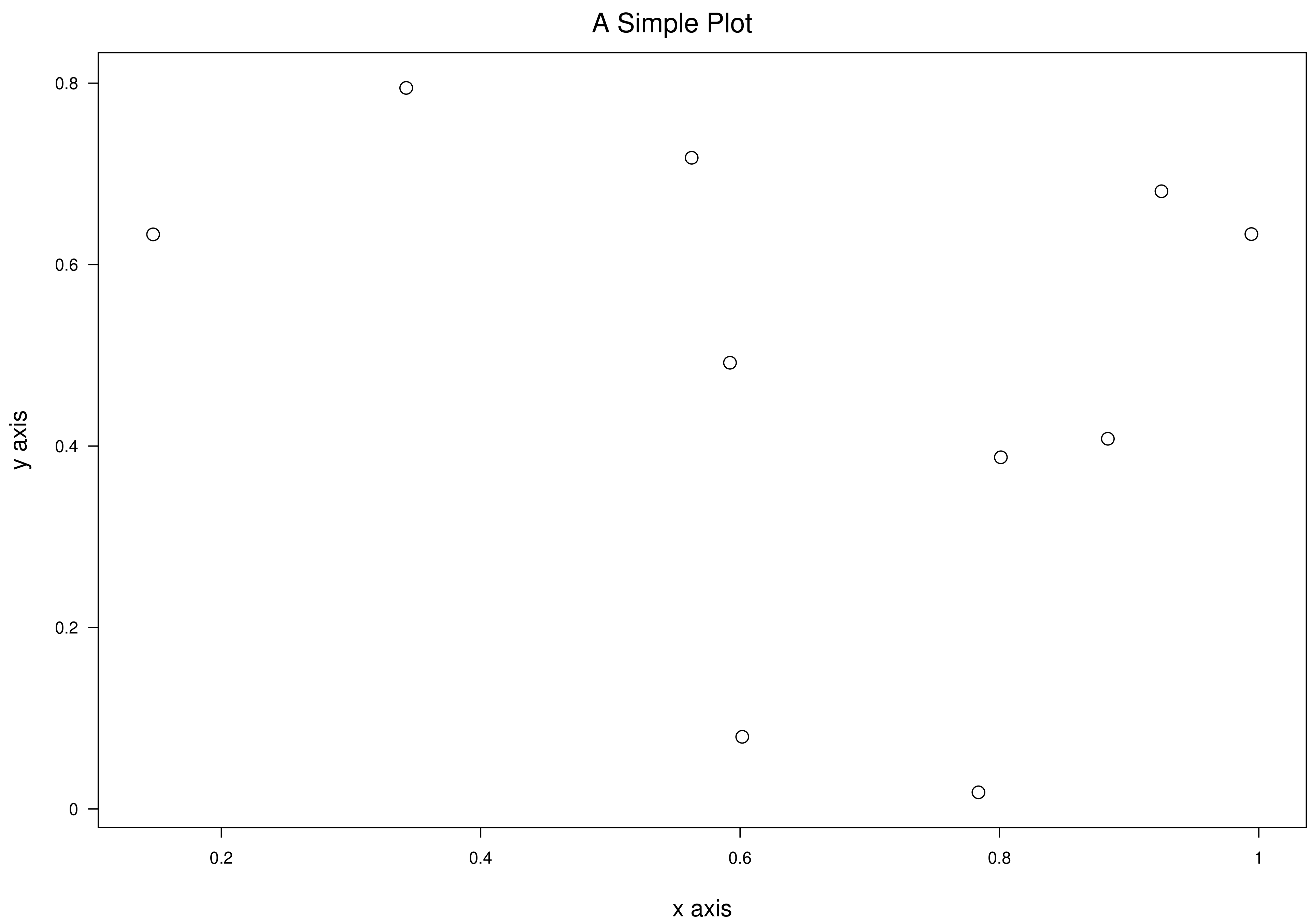

Title Centred on Data Region
y axis
1
0.8
0.6
0.4
0.2
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
x axis

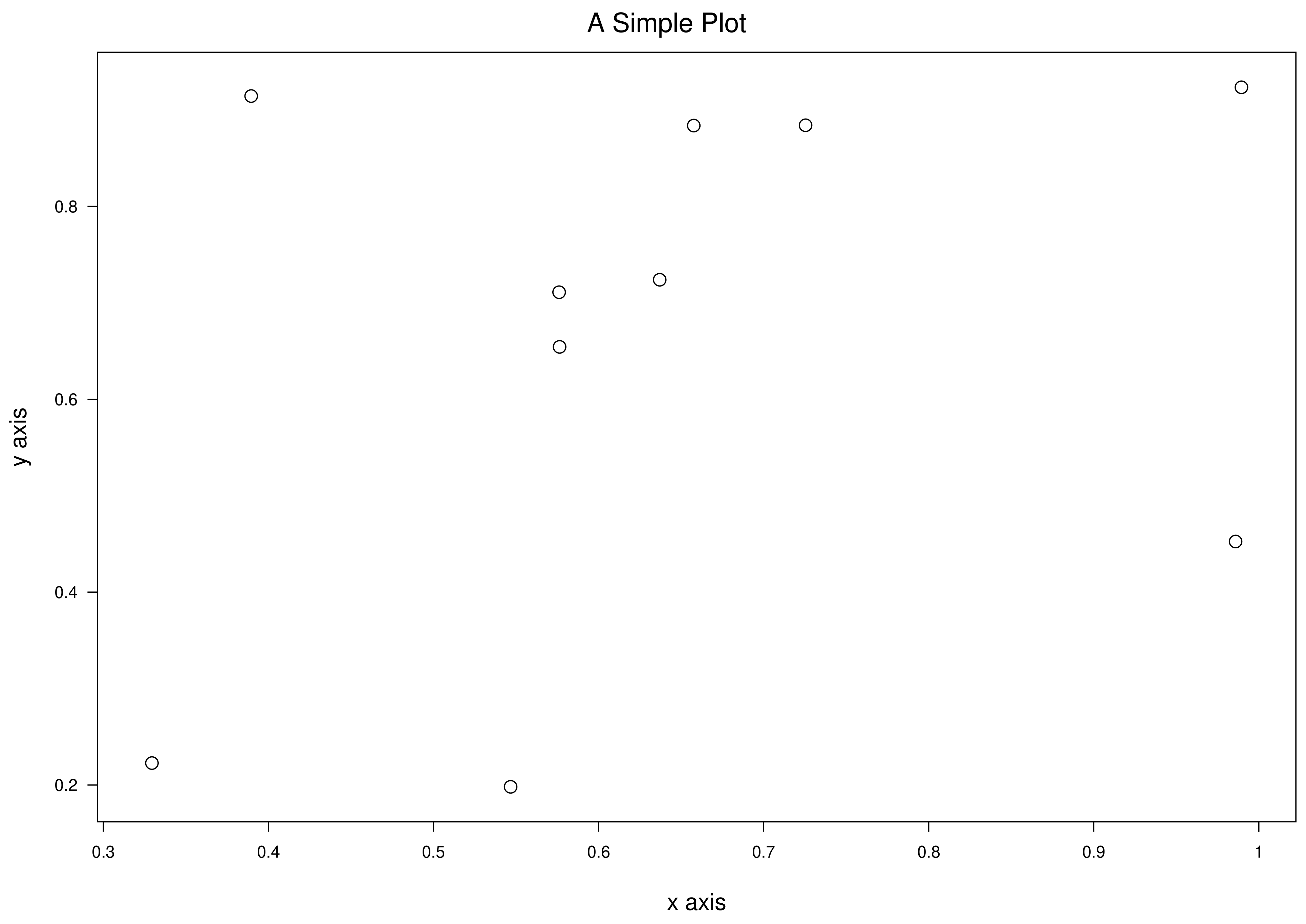

A Simple Plot
y axis
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
x axis

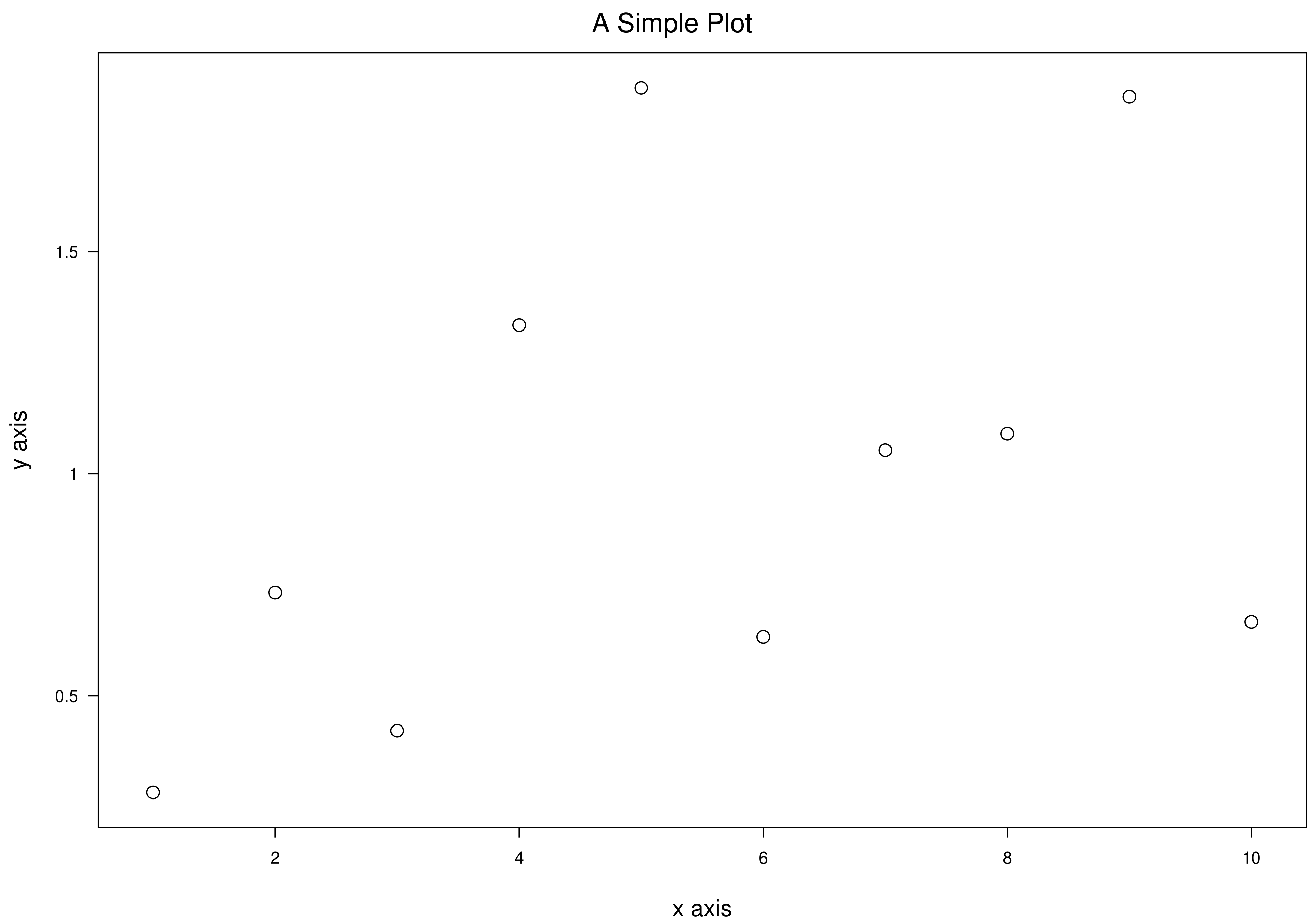
Frame around ...

An Unexciting Default


Frame around ...
A Simple Plot
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
y axis
0.2
0.4
0.6
0.8
1
x axis

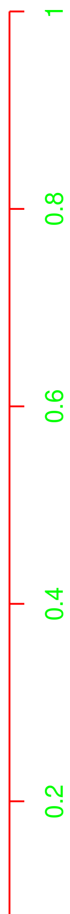

A Simple Plot
0.8
0.6
0.4
0.2
0
y axis
0.2
0.4
0.6
0.8
1
x axis


A Simple Plot
y axis
0.8
0.6
0.4
0.2
x axis
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

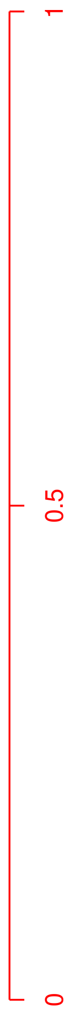

A Simple Plot
y axis
1.5
1
0.5
2
4
6
8
10
x axis


0
0.2
0.4
0.6
0.8
1

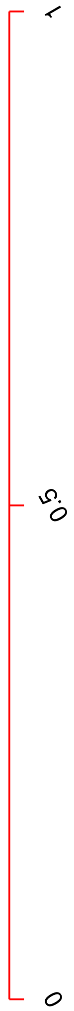

0
0.2
0.4
0.6
0.8
1

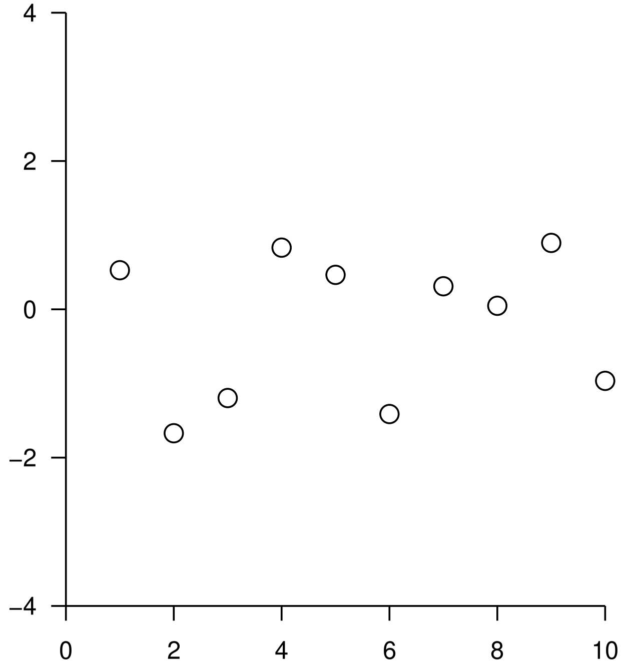


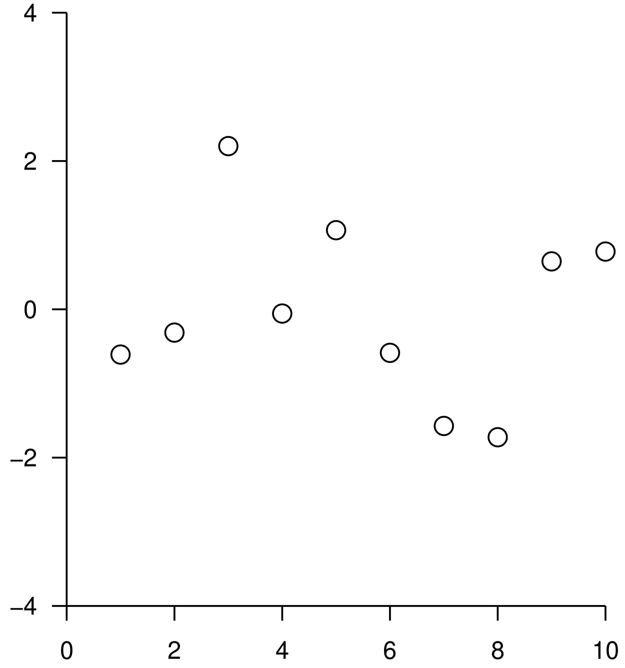
0
0.2
0.4
0.6
0.8
1

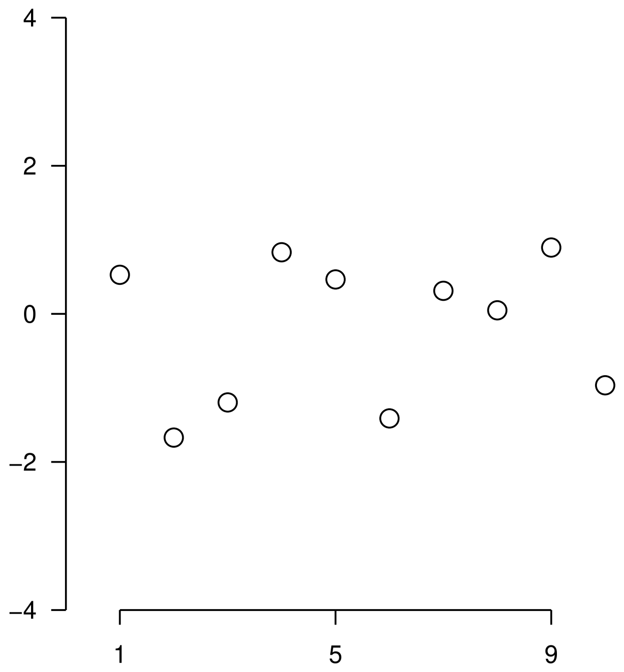


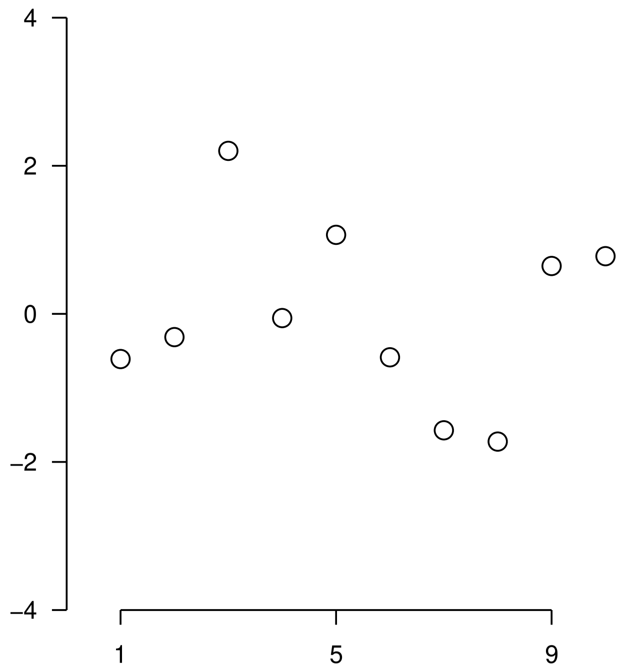
0
0.5
1

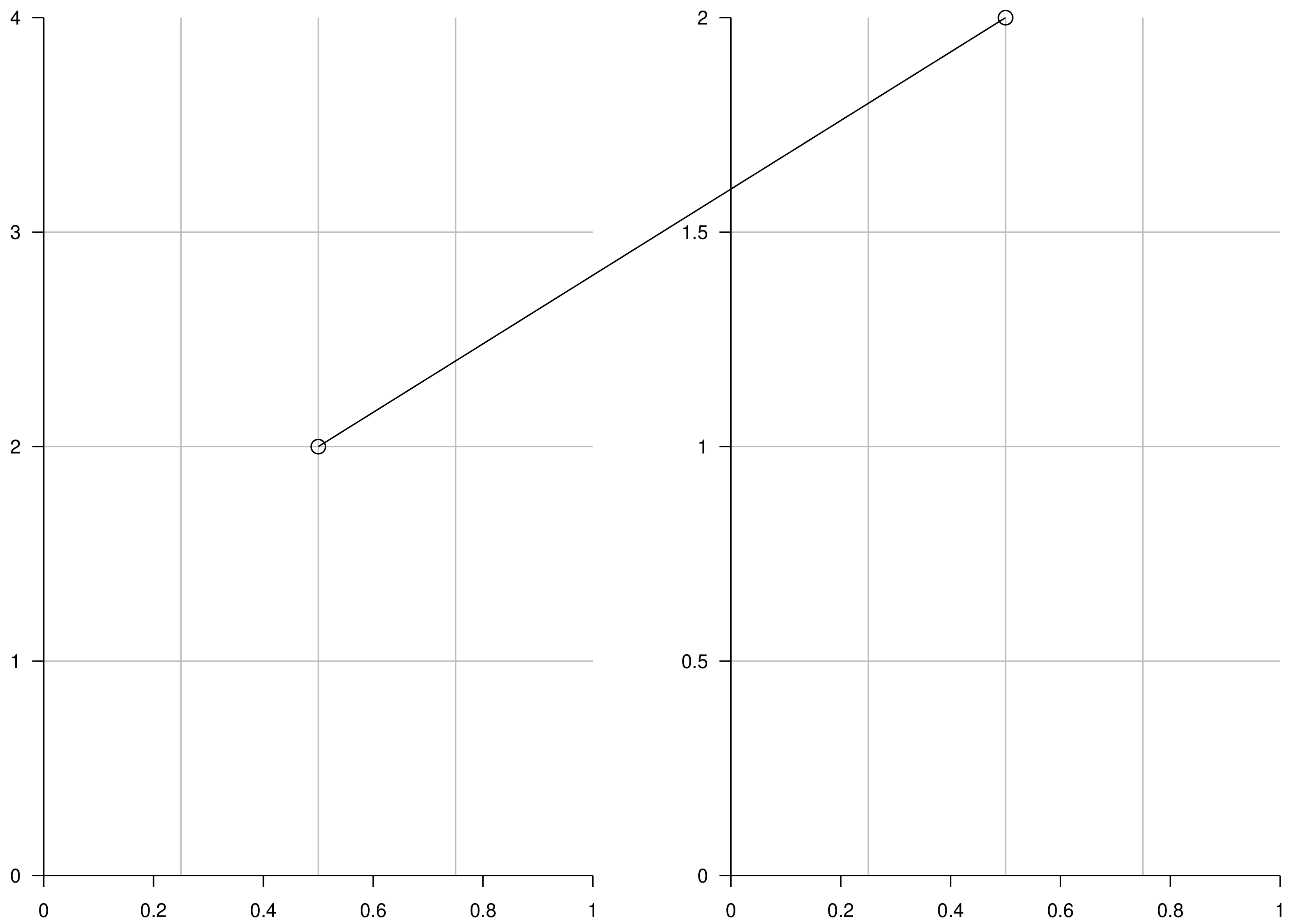

1
0.5
0

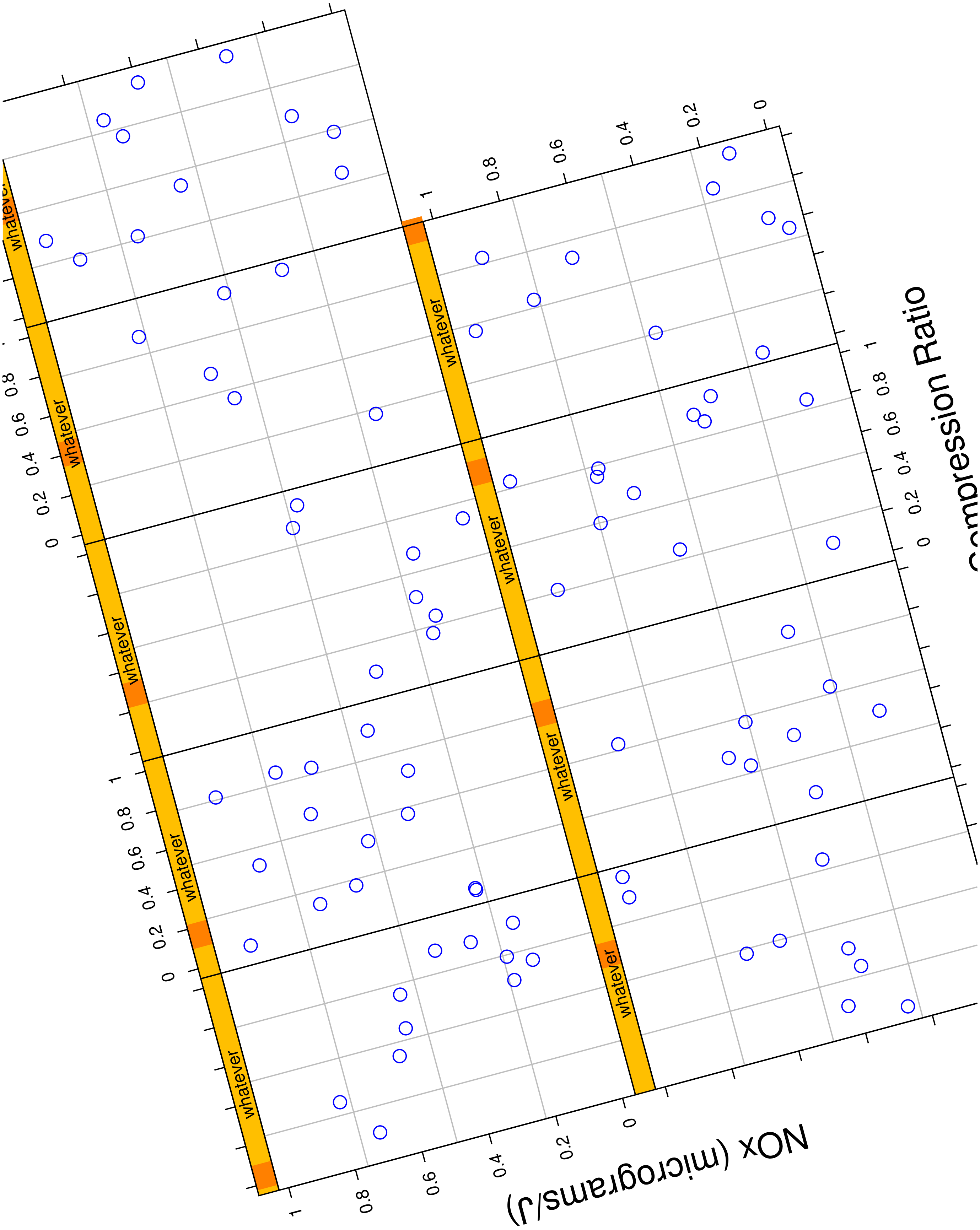

4
2
0
−2
−4
0
2
4
6
8
10


4
2
0
−2
−4
0
2
4
6
8
10

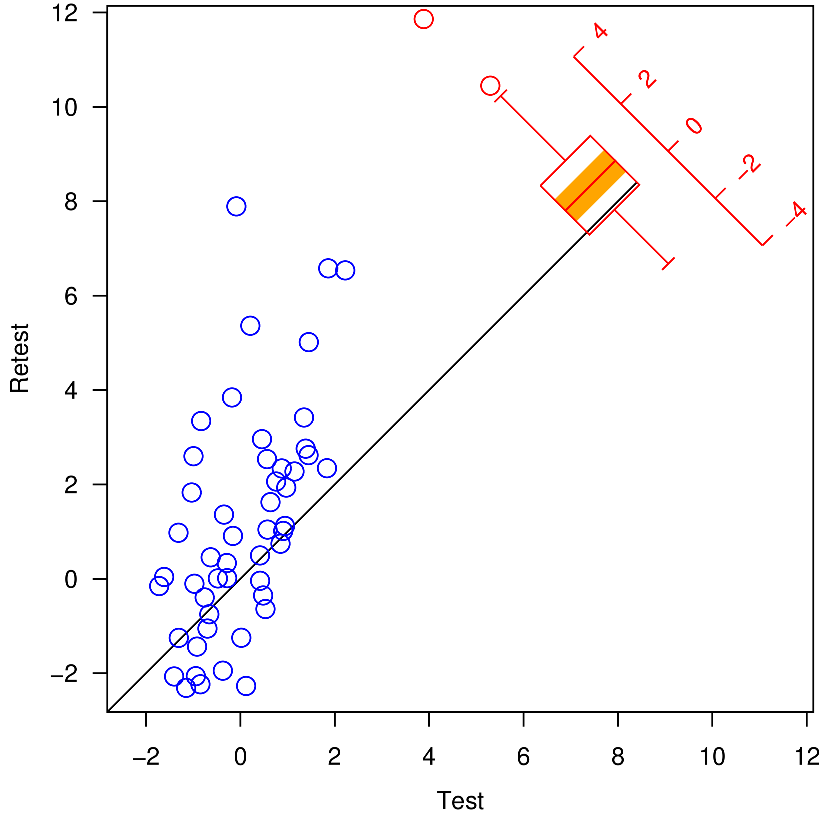

4
2
0
−2
−4
1
5
9


4
2
0
−2
−4
1
5
9

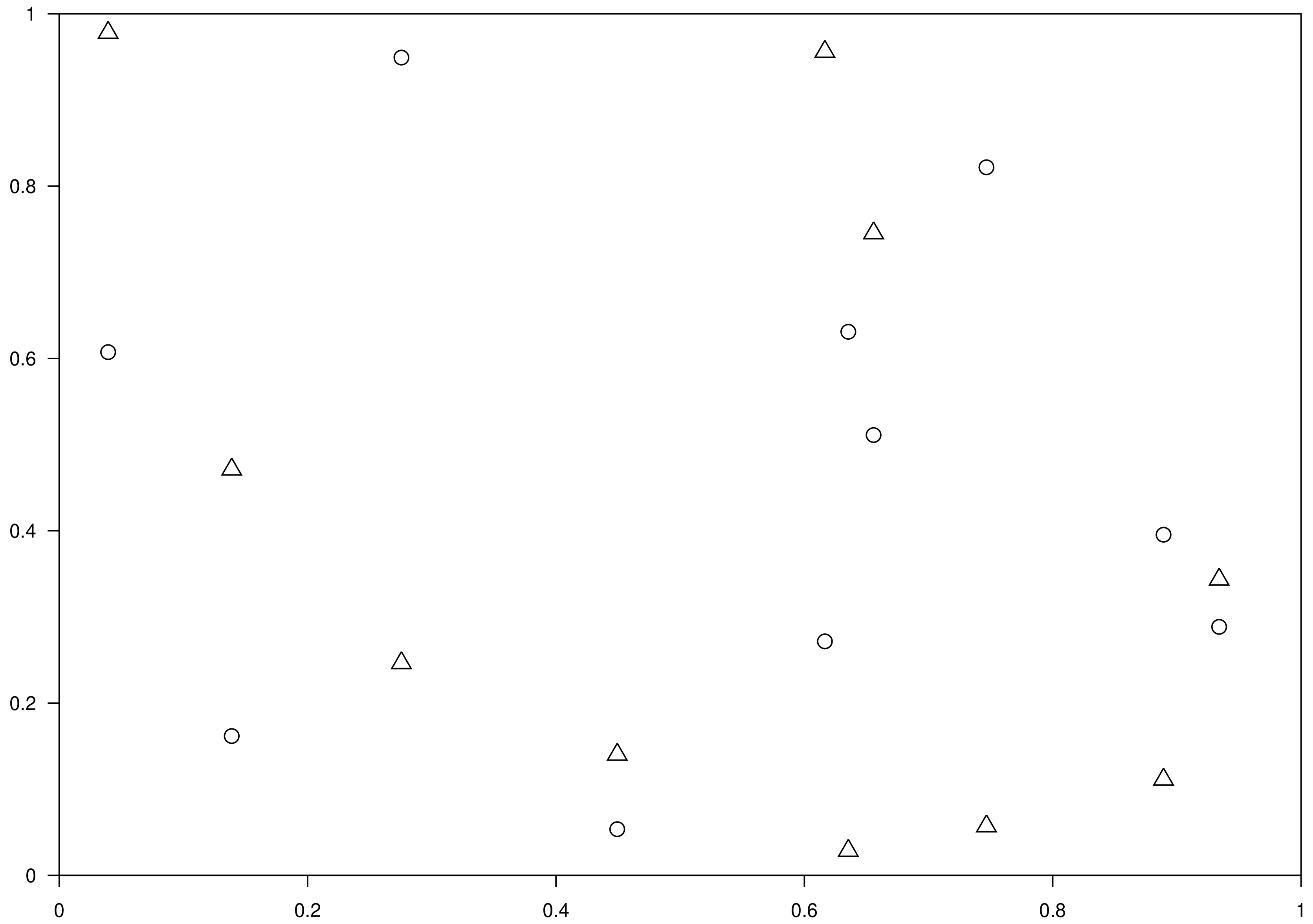

4
3
2
1
0
0
0.2
0.4
0.6
0.8
1
2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1



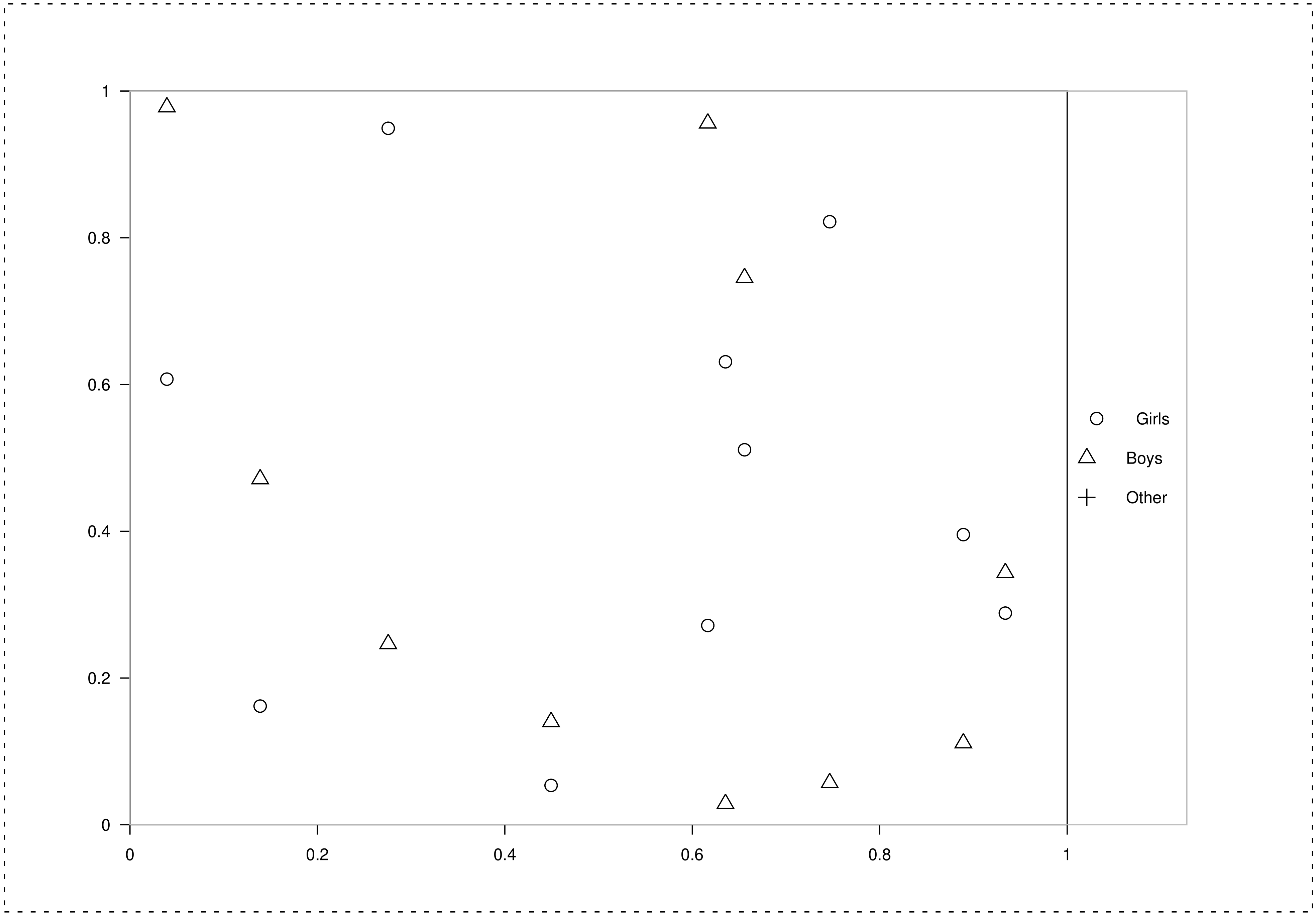
NOx (micrograms/J)
Compression Ratio
whatever
0
0.2
0.4
0.6
0.8
1


Retest
Test
12
10
8
6
4
2
0
−2
−2
0
2
4
6
8
10
12
4
2
0
-2
-4

○	one line
△	two lines
+	three lines of text



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



Girls
Boys
Other
0
0.2
0.4
0.6
0.8
1

some text

An Example of Sharing Graphical Objects in Grid

August 28, 2001

⟨*⟩≡

Grid graphical objects are implemented as "external pointers". This is easiest to explain by example:

⟨*sharing*⟩≡
```
l <- grid.lines()
l2 <- l
grid.edit(l2, gp=gpar(col="green"))
lget(l)
```

The final command produces the following output, which shows that `l` is also now "green".

⟨*sharingOutput*⟩≡
```
$x
[1] 0npc 1npc

$y
[1] 0npc 1npc

$col
[1] "green"

$lty
[1] "solid"

$vp
NULL

attr(,"class")
[1] "lines" "glist"
```

In the above, we create a graphical object and assign it to `l`. We then assign `l` to `l2`. In most cases in R, this would copy the *value* of `l` to `l2`, but in this case it actually copies a *reference* to the value of `l` to `l2`. In other words, `l` and `l2` "point" to the same value. Thus when we modify `l2`, the change is reflected in `l`.

Shared Axes

Suppose that I am drawing two graphs on a page, which are the results from two subjects in an experiment. I want the graphs to have the same axes to aid in comparison of the subjects.

First of all, I will split the page up into two bits for the two graphs.

⟨*sharedAxes*⟩≡
```
push.viewport(viewport(layout=grid.layout(1, 2, respect=TRUE)))
```

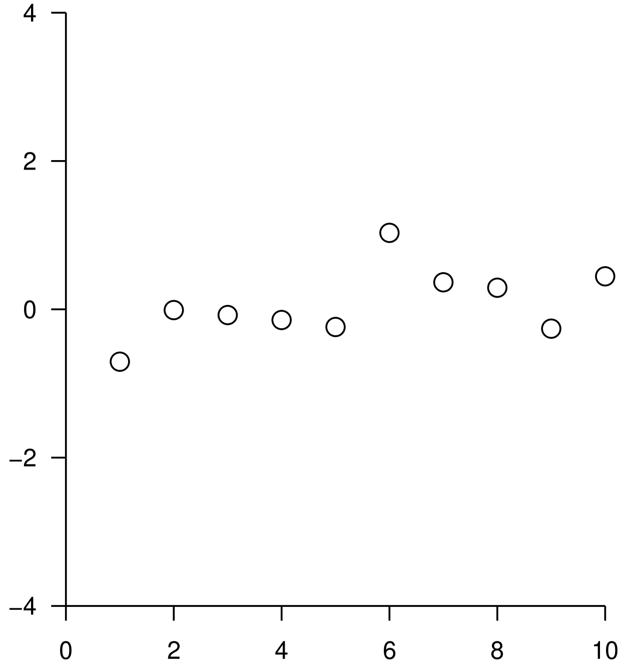
Now I generate some data and draw the first plot.

⟨*sharedAxes*⟩+≡
```
x <- 1:10
y1 <- rnorm(10)
vp1a <- viewport(layout.pos.col=1)
vp1b <- viewport(width=0.6, height=0.6,
                 xscale=c(0, 11), yscale=c(-4, 4))
push.viewport(vp1a, vp1b)
xa <- grid.xaxis()
ya <- grid.yaxis()
grid.points(x, y1)
pop.viewport(2)
```

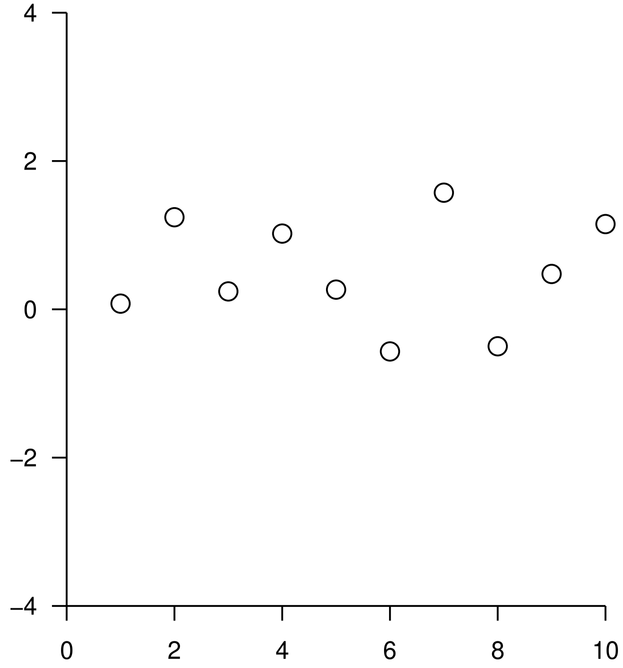
Notice that I have saved the graphical objects for the axes. Now when I draw the second plot I will just reuse these graphical objects rather than creating new ones.

⟨*sharedAxes*⟩+≡
```
y2 <- rnorm(10)
vp2a <- viewport(layout.pos.col=2)
vp2b <- viewport(width=0.6, height=0.6,
                 xscale=c(0, 11), yscale=c(-4, 4))
push.viewport(vp2a, vp2b)
grid.draw(xa)
grid.draw(ya)
grid.points(x, y2)
pop.viewport(2)
```

The output looks like the figure below.



4
2
0
−2
−4
0
2
4
6
8
10



4
2
0
−2
−4
0
2
4
6
8
10

Drawing Lines Between Viewports: `grid.move.to` and `grid.line.to`

August 28, 2001

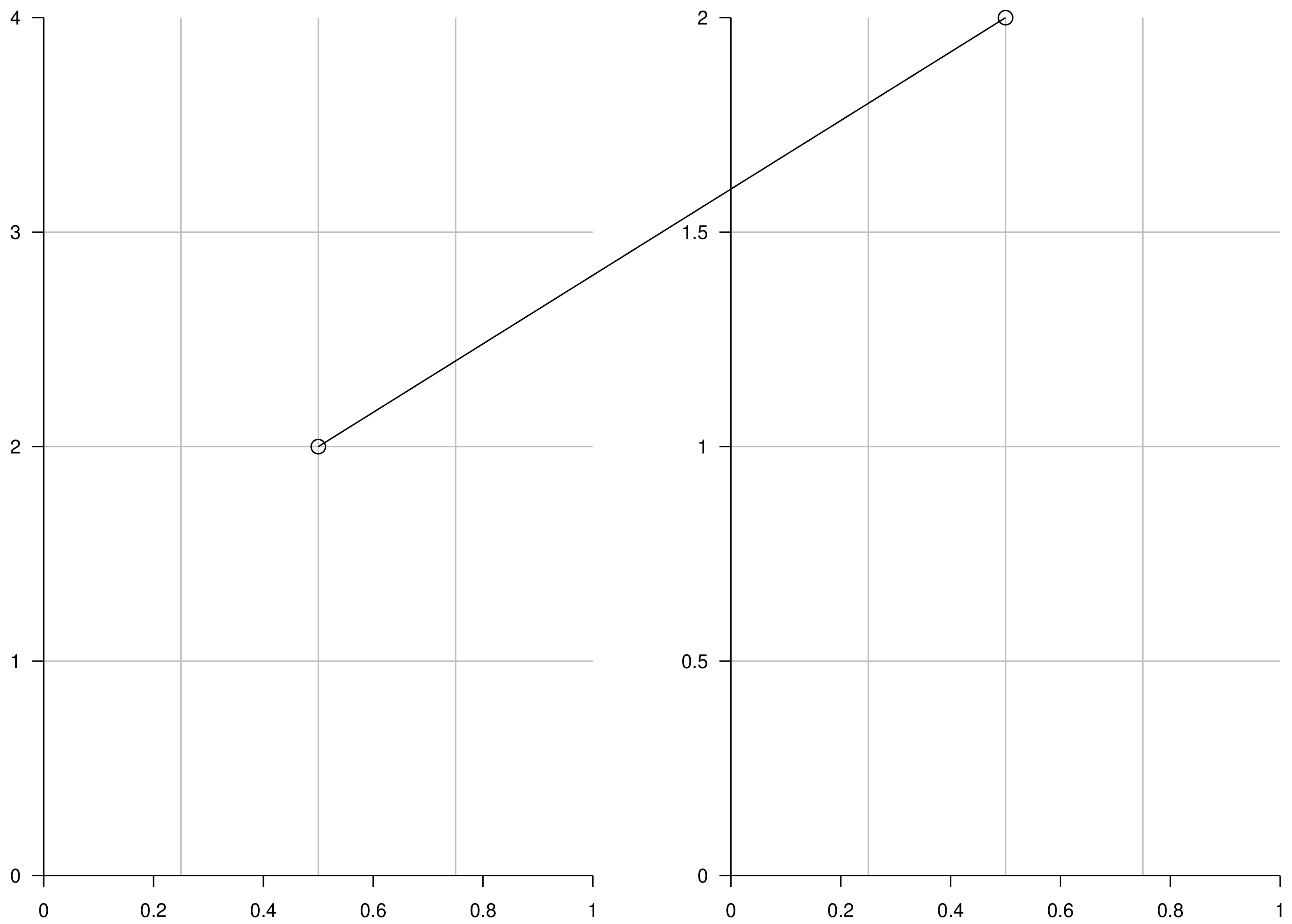
⟨*⟩≡

Many graphics systems have the notion of a current drawing location. This has been added to Grid, with the additional benefit that the drawing location can be specified relative to an arbitrary viewport. This allows drawing across coordinate systems[1].

The following code demonstrates a simple example (the output is given after the code).

⟨*movetolineto*⟩≡

```
push.viewport(
    viewport(w=.8, h=.8,
             layout=grid.layout(1, 3,
                                widths=unit(rep(1, 3),
                                            c("null", "inches", "null")))))
push.viewport(viewport(layout.pos.col=1, yscale=c(0,4)))
grid.grill(); grid.yaxis(); grid.xaxis()
grid.points(.5, unit(2, "native"))
grid.move.to(.5, unit(2,"native"))
pop.viewport()
push.viewport(viewport(layout.pos.col=3, yscale=c(0,2)))
grid.grill(); grid.yaxis(); grid.xaxis()
grid.points(.5, unit(2, "native"))
grid.line.to(.5, unit(2,"native"))
```



4
3
2
1
0
0
0.2
0.4
0.6
0.8
1
2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1

A GUI-Builder Approach to Grid Graphics

December 18, 2001

⟨*⟩≡

Grid contains a lot of support for locating, sizing, and arranging graphical components on a device and with respect to each other. However, most of this support relies on *either* the parent object dictating both location and size (layouts) *or* the child dictating both location and size.

Some sorts of arrangements are more conveniently handled by having the parent dictate the location, but letting the child dictate the size. This is the situation for GUI-builders (software which forms arrangements of GUI components or widgets). The approach taken by (many ?) GUI-builders is to allow the user to create a parent *frame* and then *pack* widgets into this frame. The frame locates and arranges the children with the help of hints such as "place this widget at the bottom of the frame", and the children dictate what size they would like to be.

This document describes a first attempt at such an interface for arranging Grid graphical objects.

The `"frame"` grob

You can create a `"frame"` graphical object using the function `grid.frame()`. This function never produces any output, but it is still important that you specify `draw=TRUE` if you want to "pack" the frame interactively. Conversely, if you want to pack the frame first and then view it, it is important that you specify `draw=FALSE` (the default).

It is *very* important that you assign the result to something. The `grid.pack()` function requires a reference to the `"frame"` object that is being packed.

⟨*frame*⟩≡
```
gf <- grid.frame()
```

The `grid.pack` function

Having created a frame, you can then pack other graphical objects into it using the `grid.pack()` function. This function has a complex interface which allows for a variety of methods of packing graphical objects. The required arguments are:

`frame` is a `"frame"` object created by `grid.frame`.

`grob` is the grob to pack into the frame.

The remaining arguments specify where the grob is located in the frame and possibly how much space the grob should occupy. The frame is effectively just a layout; you can add grobs to existing rows and columns or you can append the grob in a new row and/or column. If the grob is added to an existing row (column) then the height (width) of that row (column) becomes the maximum of the new height (width) and the previous height (width). If the row (column) is new then it just gets the specified height (width).

`col` is the column to put the grob in. This can be 1 greater than the existing number of columns (in which case a new column is added).

`row` is like `col` but for rows.

`col.after` specifies that the grob should be put in a new column inserted between `col.after` and `col.after + 1`.

`col.before` specifies that the grob should be put in a new column inserted between `col.before` and `col.before + 1`.

`row.after` **and** `row.before` do what you would expect.

`side` specifies which side to append the new grob to. The valid values are `"left"`, `"right"`, `"bottom"`, and `"top"`.

`width` is the width of the row that the grob is being packed into. If this is not given then the grob supplies the width.

`height` is like `width` but for rows.

It is possible to modify this default behaviour. For example, it is possible to add a grob to a row and force that row to have the specified height by setting `force.height=TRUE` (and similarly for column widths). It is also possible to pack a graphical object into several rows or columns at once (although you cannot simultaneously affect the heights or widths of those rows and columns).

`"grobwidth"` and `"grobheight"` units

A `"frame"` object allows a grob to specify its size by making use of `"grobwidth"` and `"grobheight"` units. These units may, of course, be used outside of frames too so their use is described here.

Consider a simple example where I want to draw a rectangle around a piece of text. I can get the size of the piece of text from the `"text"` grob as follows:

⟨*simpleGrobWidthHeight*⟩≡
```
my.text <- grid.text("some text")
grid.rect(width=unit(1, "grobwidth", my.text),
          height=unit(1, "grobheight", my.text))
```

$\boxed{\text{some text}}$

"null" Units in Grid Graphics

April 25, 2001

⟨*⟩≡

"null" units are used in layouts in Grid to indicate "relative" widths and heights of columns and rows. For example, `llayout(ncol=2, widths=unit(1:2, "null"))` says that column two will be twice the width of column one.

This is pretty simple to understand, but two situations have arisen which have forced a decision about how to handle "null" units in situations other than the simple one described above:

1. What do "null" units mean outside of layouts ?
2. What do "null" units mean in combination with other unit types ?

"null" units outside of layouts

I have decided that "null" units should basically have an "identity" value outside of layouts. That means:

1. A "null" unit on its own has a value of zero.
2. A "null" unit added to or subtracted from any other unit *or* multiplied by a scalar also has a value of zero.
3. A "null" unit being compared with other unit values to find the maximum has a value of `min-val`.
4. A "null" unit being compared with other unit values to find the minimum has a value of `max-val`.

`"null"` units in combination with other units in layouts

There is a reasonable interpretation for such expressions as `max(unit(c(1, 1), c("inches", "null")))` when used as, for example, the width of a layout column. It could be taken to mean that the column should be 1 inch wide *or* whatever width it would take based on the `"null"` value, if it wasn't 1 inch.

Unfortunately, this sort of interpretation is very difficult to calculate (at least in the way I currently resolve layout widths and heights) especially in the general case when there might be arbitrarily complex arithmetic expressions involving `"null"` and non-`"null"` units (and this can very easily happen in `"frame"` grobs).

On the other hand, arbitrarily complex arithmetic expressions *only involving* `"null"` *units* are straightforward to calculate.

So for now I have the following rule: if the layout width/height only involves `"null"` units then evaluate the total value and treat the result as the specification of a relative width/height, *but* if there are any non-`"null"` units in the width/height expression then evaluate the `"null"` units as if they are outside of the layout.

Some examples:

- `unit(1, "null")` inside a layout, specifies a relative width/height; outside a layout is zero.
- `unit(1, "null") + unit(2, "null")` inside a layout, specifies a relative width/height; outside a layout is zero.
- `unit(1, "inch") + unit(1, "null")` both inside and outside a layout, has a value of `unit(1, "inch")`.

The Relationship Between Parents and Children in Grid Graphics

December 17, 2001

In Grid there are many ways to specify the location of an object. The most simple is to use one of the "basic" units. For example,

⟨*basic*⟩≡
```
grid.rect(x=unit(1, "inches"), y=unit(.1, "npc"),
          width=unit(.5, "native"), height=unit(1, "lines"),
          just=c("left", "bottom"))
```

The idea here is that the child knows exactly where it wants to go within its parent (viewport).

The other "basic" alternative is to use layouts. For example,

⟨*basic*⟩+≡
```
grid.newpage()
push.viewport(viewport(layout=grid.layout(2, 2)))
push.viewport(viewport(layout.pos.col=2, layout.pos.row=2))
grid.rect()
pop.viewport(2)
```

The idea here is that the parent knows exactly where it wants the child to go.

More complex situations are possible. These are accessed via `"grobwidth"` and `"grobheight"` units, which are the basis of the `grid.frame` and `grid.pack` functions. For example,

⟨*complex*⟩≡
```
grid.newpage()
gt <- grid.text("howdy")
grid.rect(width=unit(1, "grobwidth", gt), height=unit(1, "lines"))

grid.newpage()
gf <- grid.frame()
grid.pack(gf, grid.rect(x=unit(1, "npc"),
                         w=unit(1, "char"), h=unit(1, "char"),
                         just=c("right", "centre"), draw=F))
grid.pack(gf, grid.text("A label", draw=F), side="right")
```

The idea here is that the parent asks the child how big it should be and allocates the required space, but takes care of the locations of its children.

When we are dealing with this sort of complex situation, we have to start thinking about what sort of unit is being used. In particular, we make the distinction between *absolute* units and *relative* units. The clearest examples of absolute units are the physical units such as inches and centimetres. In terms of widths and heights, these units are actually completely idependent of the parent context; it does not matter how big the parent is or what scales it has, a physical size gives the same end result. These physical units are sensible to use when the parent is asking the child for its size.

The clear examples of relative units are native units and normalised parent coordinates. These depend on the size and scales of the parent to give a final size for the child. This sort of unit is **not** sensible to use when a parent is asking the child for its size because an infinite loop can be created; the size of the parent depends on the child, which depends on the parent, ...

There are two slightly odd cases. The first case involves `"lines"` and `"char"` units. These do depend on the parent in as much as the parent may have settings for the graphical parameters `lineheight` and `fontsize`. However, these are graphical parameter settings whereas the other things such as scales that define coordinate systems and the size of the parent form what I call the drawing context and structural description of the parent. What is important is to avoid asking a child for its size when the child depends on the drawing context (coordinate systems) or physical size of the parent. It is actually ok to have the child depend on the graphical parameters of the parent - this does not generate any sort of infinite loop.

The second odd case is `"null"` units. In this case, when the parent asks the child how big it should be, the child is effectively saying, “I don’t know, give me whatever is left over after the others have had their share.”

Pratical Implications

All of this stuff about absolute and relative units means that I would really only like `width.details` methods to return absolute units. For one thing, I don’t want the method to return relative units because that could lead to weird results and it might lead to infinite loops. On the other hand, I *do* want the method to return absolute units because that can actually be quite handy (e.g., see the first example in this document).

Because I believe in giving the users plenty of rope to hang themselves, all I have done about this is provide a function which takes a unit and converts it to only absolute units; absolute units are untouched, but relative units get converted to `unit(1, "null")`. Consider the following example,

⟨*absolute*⟩≡

```
grid.newpage()
gf <- grid.frame()
grid.pack(gf, grid.rect(width=unit(1, "inches"),
                        height=unit(.5, "npc"),
                        draw=F))
```

This means that the rectangle is packed into an area which is 1 inch wide, but as high as the parent decides it should be (based on what are is left after any other children have been accommodated), *and* that the rectangle actually only occupies half of the height that it is allocated.

Writing Grid Graphics Code

December 17, 2001

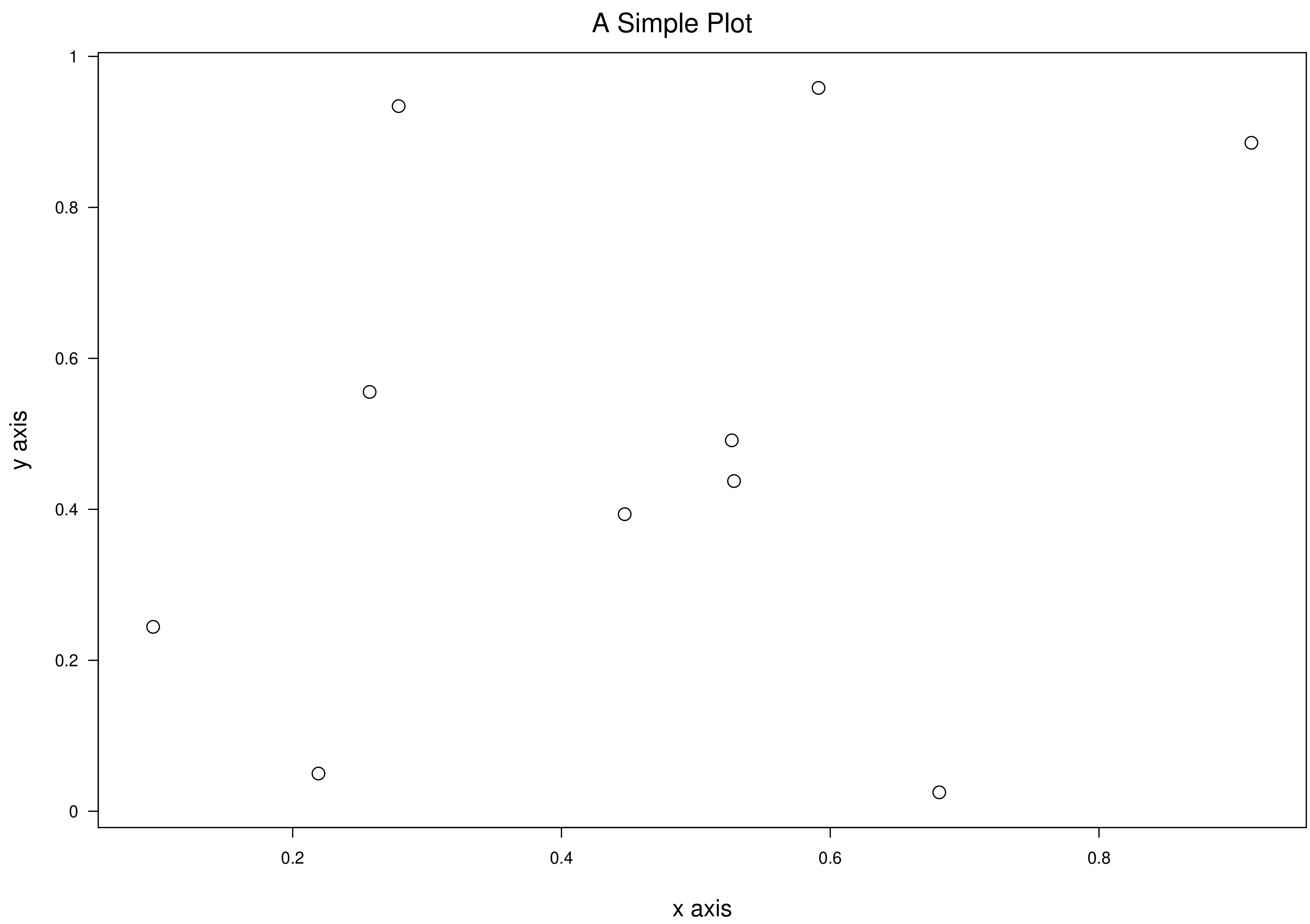
⟨*⟩≡

The Grid Graphics system contains a degree of complexity in order to allow things like interactive graphics. This means that many of the predefined Grid graphics functions are relatively complicated[1].

One design aim of Grid Graphics is to allow users to create simple graphics simply and not to force them to use complicated concepts or write complicated code unless they actually need to. Along similar lines, it is intended that people should be able to prototype even complex graphics very simply and then refine the implementation into a more sophisticated form if necessary.

With the predefined graphics functions being fully-developed and complicated implementations, there is a lack of examples of simple, prototype code. Furthermore, given that the aim is to allow a range of ways to produce the same graphical output, there is a need for examples which demonstrate the various stages, from simple to complex, that a piece of Grid Graphics code can go through.

This document describes the construction of a scatterplot object, like that shown below, going from the simplest, prototype implementation to the most complex and sophisticated. It demonstrates that if you only want simple graphics output then you can do it pretty simply and quickly. It also demonstrates how to write functions that allow your graphics to be used by other people. Finally, it demonstrates how to make your graphics fully interactive (or at least as interactive as Grid will let you make it).


A Simple Plot
y axis
x axis
1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8

Rotated Viewports

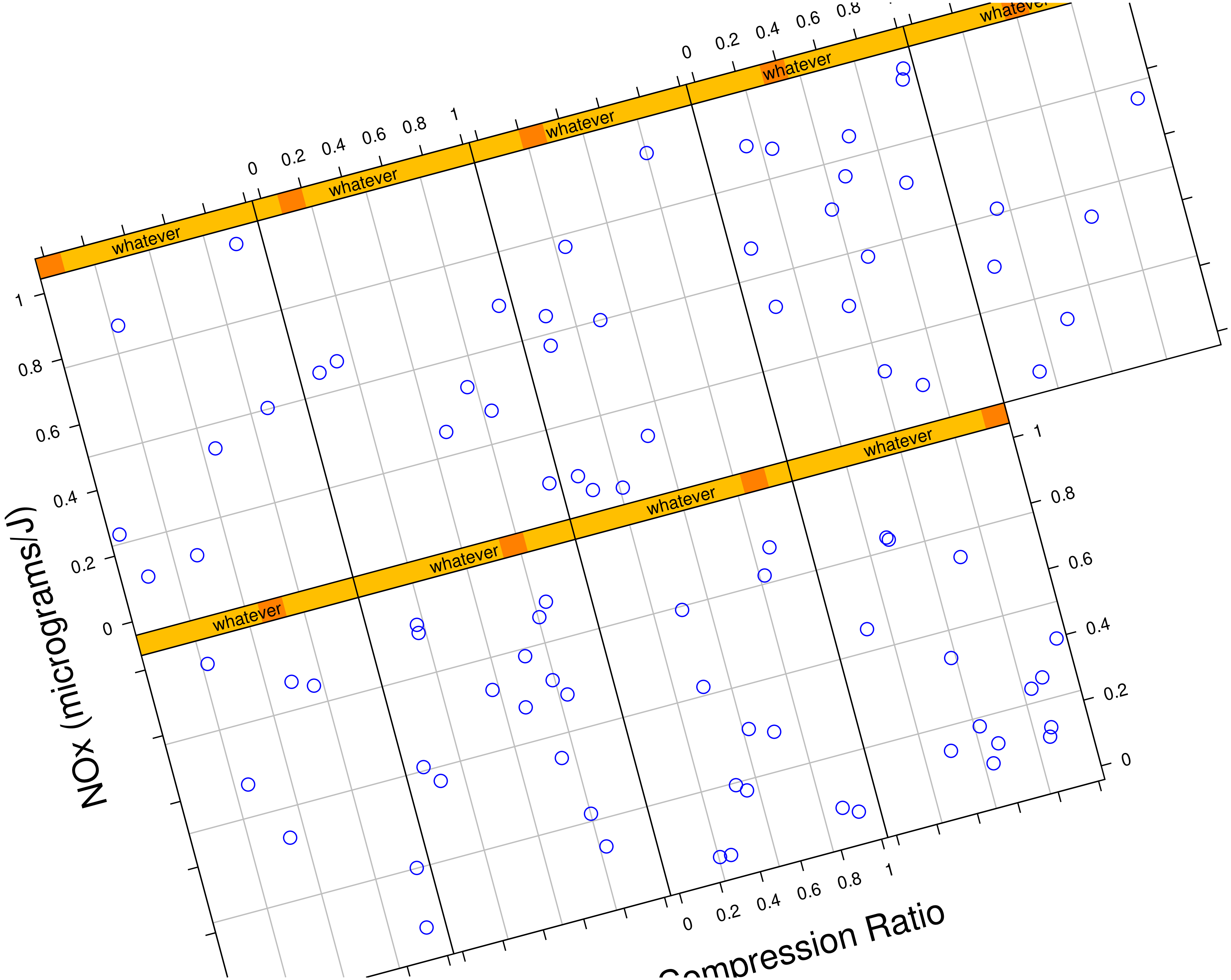
August 29, 2001

⟨*⟩≡

It is possible to specify an angle of rotation for a Grid viewport. For example, the following code draws the example multipanel plot at an angle of 15∘.

⟨*rotmulti*⟩≡

```
push.viewport(viewport(h=.8, w=.8, angle=15))
grid.multipanel(newpage=F)
pop.viewport()
```



NOx (micrograms/J)
ompression Ratio
whatever
0
0.2
0.4
0.6
0.8
1

The Grid Graphics Package

Version 0.5

Table of Contents

1 Introduction

Grid graphics is an alternative graphics interface to the base R graphics (`plot()`, `par()`, `lines()`, ...). It is designed to allow greater low-level power and customisation for graphics. Grid itself mostly consists of the *basis* for producing flash high-level plots; there are a couple of examples of higher-level plotting functions (e.g., `grid.xaxis` and `grid.yaxis`, and more spectacularly, `grid.show.layout` and `grid.show.viewport`). The `lattice` package provides plenty of examples of what can be achieved.

This user guide focuses on the low-level functionality; the purpose of this package is to provide greater power and flexibility for producing plots when the standard high-level plots are not sufficient.

This guide provides an explanation of the system through examples. It is best used while sitting at a console with R running.

A more detailed and complete description of individual functions is available in the form of Rd files (see `help(Grid)` and `library(help=grid)`).

2 Getting Started

Load the Grid library with the command `library(grid)`.

3 Tutorial

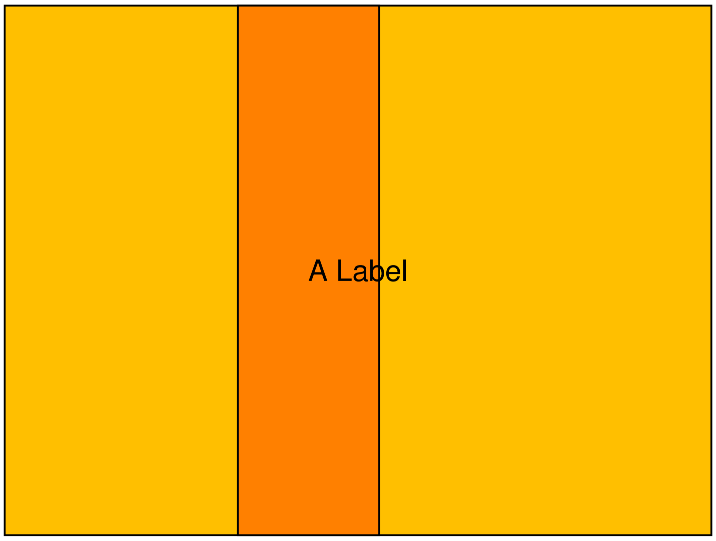
We will now look at some very simple Grid graphics drawing functions. This section assumes that you have loaded Grid graphics.

3.1 Baby Steps

First of all, try typing the following commands:

```
grid.rect(x=0.5, y=0.5, h=1, w=1,
          just="centre", gp=gpar(border=NULL, fill="#FFBF00"))
grid.rect(x=0.33, y=0, h=1, w=0.2,
          just=c("left", "bottom"), gp=gpar(border=NULL, fill="#FF8000"))
grid.text(x=0.5, y=0.5, "A Label")
```

The window is now a light orange colour with a darker orange vertical strip and the text ‘`A Label`’ in the centre.


A Label

Notes on Units in Grid

Paul Murrell

June 12, 2001

This document describes the reasons behind the implementation of units in Grid (at the R level).

Initially, units were specified as an additional argument. For example, `grid.lines(x=c(1, 1), y=c(2, 2), units="inches")`. This has the disadvantage of units not being strongly linked syntactically or visually with their corresponding values. Also, if we ever intend to implement "unit arithmetic", it would be a nightmare trying to specify units on operands in an arithmetic expression.

Two other options were considered:

1. something like `grid.lines(x=inch(1, 1), y=inch(2, 2))`. This is good for unit arithmetic (e.g., `cm(1) + inch(2) + 3`), but has name-space problems because the units names are too likely to be used by others (in fact `cm()` has already been taken.

2. something like `grid.lines(x=unit(c(1, 1), "inch"), y=unit(c(2, 2), "inch"))`. This is ok for unit arithmetic (e.g., unit(1, "cm") + unit(2, "inch") + 3) and avoids name-space problems.

In both of these two approaches, the idea would be for the function to produce an object object of class "unit". This would allow us to write a `Math.unit` function for doing simple arithmetic.

The other thing we thought about was being able to define a vector of units at once. In the first option above, it would have been nice to allow something like `c(1, cm(2))` (i.e., allow the user to specify implicit units; defaults to "npc" for example). We could write a `c.unit` method which would work for `c(cm(1), 2)`, but it wouldn't work for `c(1, cm(2))` because the method dispatch occurs only on the first argument so the result would be just a numeric vector (with no class or attributes). NOTE that this is not a problem for `Math.unit` because method dispatch in that case checks all (both) arguments. This means that we have to go with the second option and do something like `unit(c(1, 2), c("npc", "inch"))` (i.e., NOT allow implicit units). It should, however, be possible to specify a whole vector of values with implicit units (e.g., `llines(c(1, 2))`) and have the function take the numeric vector and create a unit object with default units.