Introduction to **grid** Graphics

Paul Murrell

The University of Auckland

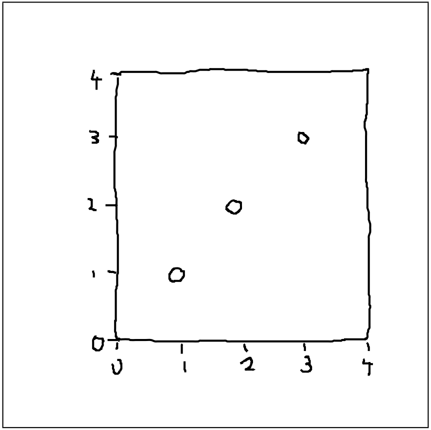
July 2011

Entrée

Entrée

Entrée

- Why do we **not** draw statistical plots by hand with a drawing program like Inkscape or Gimp?



Entrée

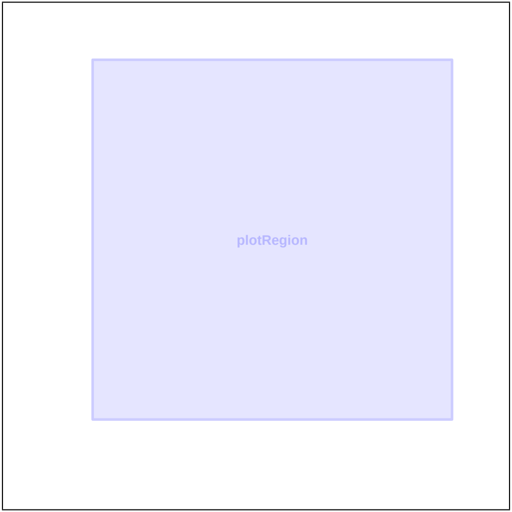
- A plot is just a bunch of shapes, but the **arrangement** of those shapes is critical.
- **grid** provides tools to draw basic shapes **plus** tools that assist in the arrangement of basic shapes.
- Plus it provides a way to produce graphical scenes **programmatically**.

Entrée

- **Viewports** create a context for drawing.

```
> library(grid)
> plotvp <- viewport(x=unit(5, "lines"),
                     y=unit(5, "lines"),
                     width=unit(1, "npc") -
                           unit(8, "lines"),
                     height=unit(1, "npc") -
                            unit(8, "lines"),
                     just=c("left", "bottom"),
                     xscale=c(0, 4),
                     yscale=c(0, 4),
                     name="plotRegion")
> pushViewport(plotvp)
```

Entrée

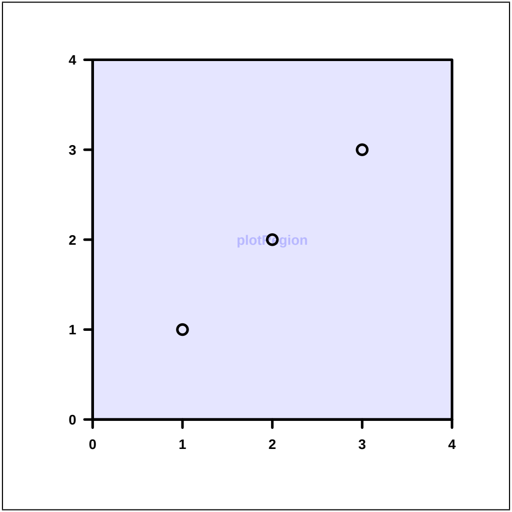


Entrée

- Graphical shapes are drawn within that context.

```
> grid.points(1:3, 1:3, default.units="native")
> grid.rect(x=0.5, y=0.5, width=1, height=1)
> grid.xaxis(at=0:4)
> grid.yaxis(at=0:4)
```

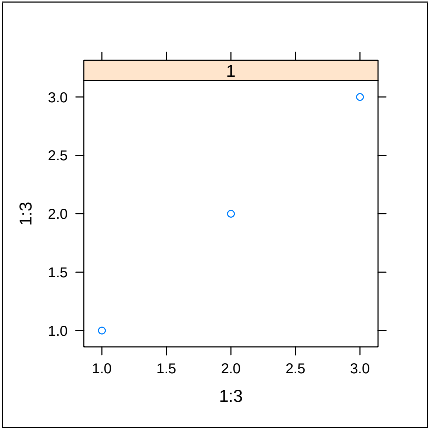
Entrée



Entrée

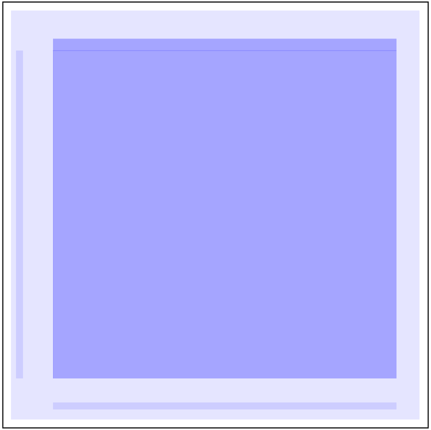
- This is what **lattice** is doing ...

```
> library(lattice)
> xyplot(1:3 ~ 1:3 | 1)
```



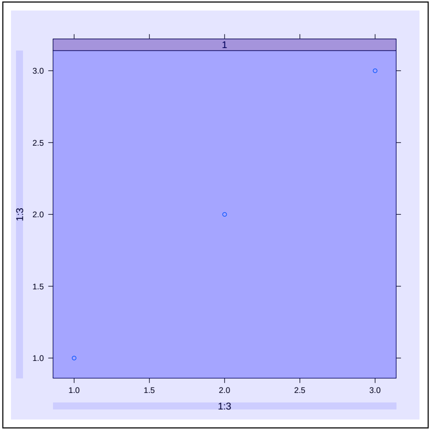
Entrée

- ... creating viewports ...



Entrée

- … and drawing shapes in the viewports.



Review

Review

Review

- Drawing with **grid** involves defining contexts for drawing (viewports) and drawing basic shapes in those contexts.
- We need to know what shapes **grid** can draw and how to position and size those shapes.
- We need to know how to create viewports.

Main Course

Main Course

Basic Shapes

The following basic shapes can be drawn using **grid**:

circles	`grid.circle(x, y, r)`
lines	`grid.lines(x, y)`
	`grid.segments(x0, y0, x1, y1)`
	`grid.polylines(x, y, id)`
rectangles	`grid.rect(x, y, width, height)`
	`grid.roundrect(x, y, width, height, r)`
text	`grid.text(label, x, y)`

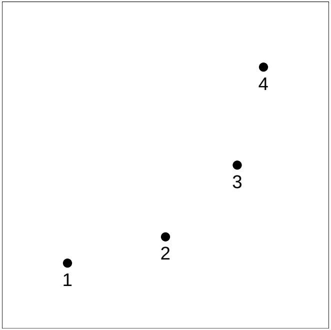
Basic Shapes

The following basic shapes can be drawn using **grid**:

polygons	`grid.polygon(x, y, id)` `grid.path(x, y, id)`
curves	`grid.xspline(x, y, shape)` `grid.curve(x1, y1, x2, y2)`
raster images	`grid.raster(image, x, y, width, height)`
data symbols	`grid.points(x, y, pch)`

Basic Shapes

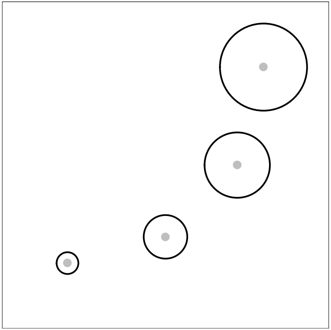
```
> t <- seq(90, 0, -30)
> x <- .2 + cos(t/180*pi)*.6
> y <- .8 - sin(t/180*pi)*.6
```



Basic Shapes

Locations and dimensions are vectors so multiple shapes can be drawn at once.

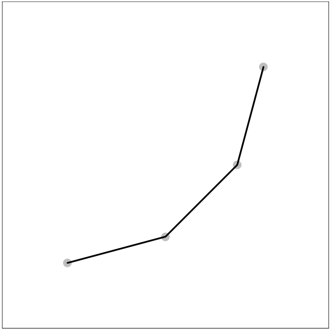
```
> grid.circle(x, y, r=1:4/30)
```



Basic Shapes

Some shapes require multiple locations to describe a single shape.

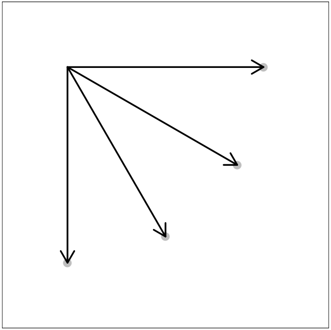
```
> grid.lines(x, y)
```



Basic Shapes

Any line or curve shape can have arrows at either end.

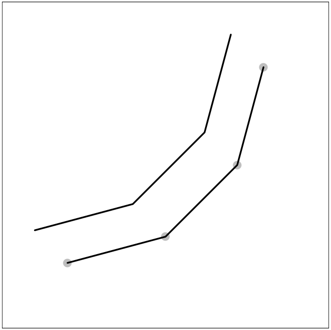
```
> grid.segments(.2, .8, x, y,
                arrow=arrow())
```



Basic Shapes

Some functions have an `id` argument to allow multiple shapes from a single call.

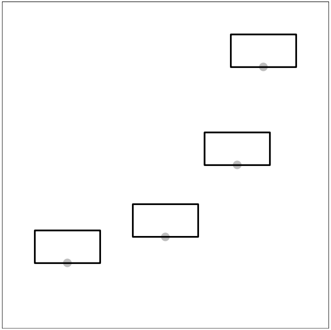
```
> grid.polyline(c(x, x - .1), c(y, y + .1),
                id=rep(1:2, each=4))
```



Basic Shapes

Rectangles are "justified" relative to the `x` and `y` locations.

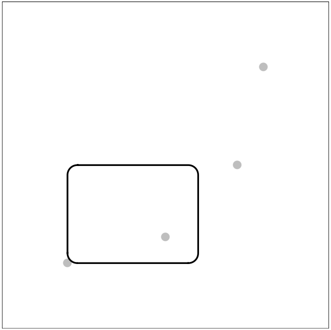
```
> grid.rect(x, y, width=.2, height=.1,
            just="bottom")
```



Basic Shapes

Only one rounded rect can be drawn at a time.

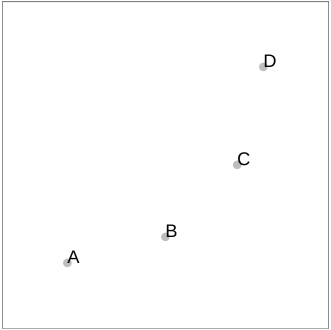
```
> grid.roundrect(x[1], y[1], width=.4, height=.3,
                 just=c("left", "bottom"))
```



Basic Shapes

Text can also be justified relative to `x` and `y`.

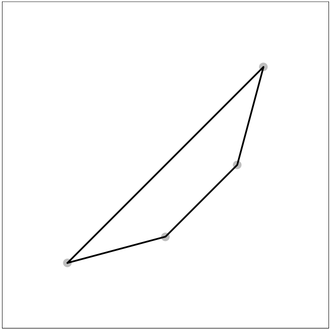
```
> LETTERS[1:4]
[1] "A" "B" "C" "D"
> grid.text(LETTERS[1:4], x, y,
            just=c("left", "bottom"))
```



Basic Shapes

Polygons are automatically "closed".

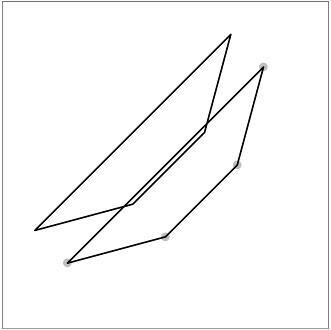
```
> grid.polygon(x, y)
```



Basic Shapes

Paths describe a single shape from multiple disjoint pieces.

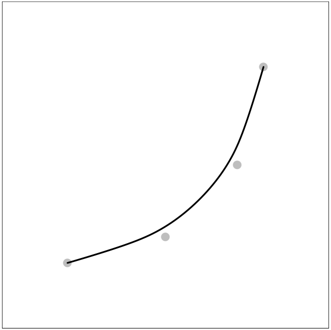
```
> grid.path(c(x, x - .1), c(y, y + .1),
            id=rep(1:2, each=4))
```



Basic Shapes

Xsplines describe a smooth curve relative to control points.

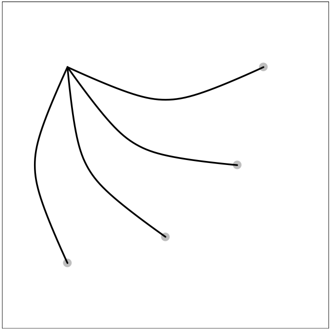
```
> grid.xspline(x, y, shape=1)
```



Basic Shapes

Curves describe a smooth curve between two end points.

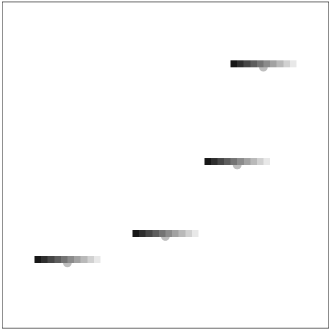
```
> grid.curve(.2, .8, x, y, square=FALSE,
             curvature=.5, shape=1)
```



Basic Shapes

Raster images can be vectors or matrices or (with help from an extension package) external files.

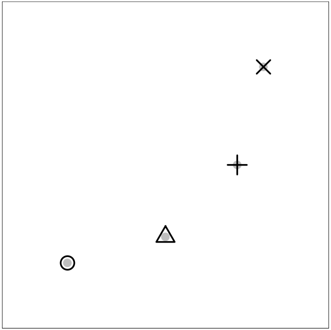
```
> grid.raster(t(1:10/11), x, y, width=.2,
              interpolate=FALSE, just="bottom")
```



Basic Shapes

A predefined set of data symbols is available.

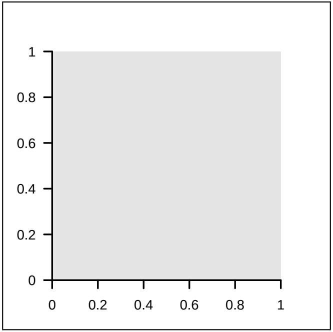
```
> grid.points(x, y, pch=1:4)
```



Axes

grid also provides functions for drawing basic axes.

```
> grid.xaxis()
> grid.yaxis()
```

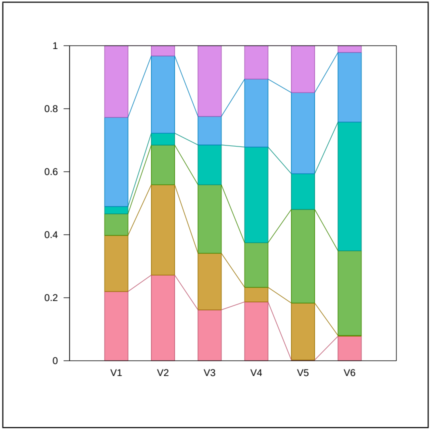


EXERCISES

EXERCISE

EXERCISES

- The ultimate goal of the exercises in the first half of this course is to produce a complete plot with a novel style.

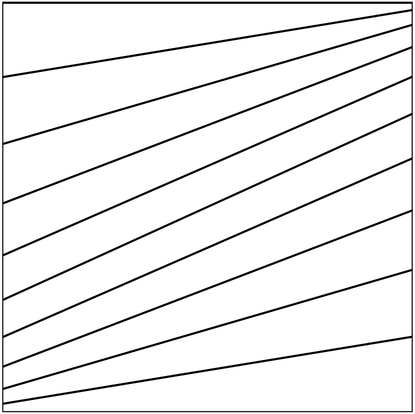


EXERCISE

- We will develop the plot in separate stages that will allow us to experiment with the various **grid** concepts that we encounter.
- At each stage, a code skeleton is provided to perform ancillary tasks such as data preparation, so that you just have to add code to do the drawing.

EXERCISES

- The goal of this exercise is to draw a series of line segments as shown below.



EXERCISES

- The raw data consist of two vectors of values.

```
> y1 <- 1:10
> y2 <- 10:1
```

- A function is provided to generate cumulative proportions from a vector.

```
> cprop <- function(x) {
      prop <- x/sum(x)
      cumsum(prop)
  }
```

- Each vector is converted into a set of cumulative proportions, which provide the start and end y-values for the line segments.

```
> cp1 <- cprop(y1)
> cp2 <- cprop(y2)
```

Main Course

Main Course

Units and Coordinate Systems

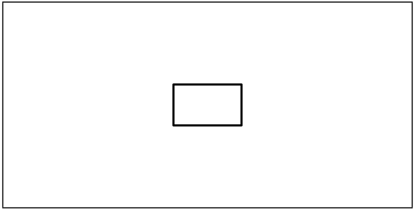
- The locations and dimensions of shapes are **units**, which consist of a **value** plus a **coordinate system**.
- The main coordinate systems are:

`"npc"`	Normalised Parent Coordinates
`"native"`	Relative to the current x-scale/y-scale
`"in"` or `"cm"`	Inches or centimetres
`"lines"`	Lines of text

Units and Coordinate Systems

- The `unit()` function is used to create unit objects.

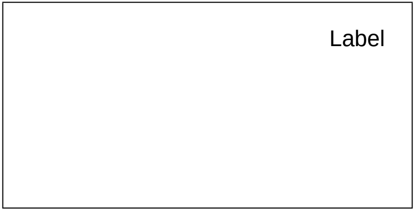
```
> unit(1, "in")
[1] 1in
> unit(.2, "npc")
[1] 0.2npc
> grid.rect(width=unit(1, "in"),
            height=unit(.2, "npc"))
```



Units and Coordinate Systems

- Simple operations on units are possible, including basic arithmetic.

```
> unit(1, "npc") - unit(1, "cm")
[1] 1npc-1cm
> grid.text("Label",
            x=unit(1, "npc") - unit(1, "cm"),
            y=unit(1, "npc") - unit(1, "cm"),
            just=c("right", "top"))
```



Graphical Parameters

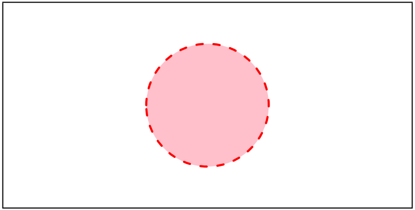
- Every basic shape has a gp argument that allows graphical parameters to be specified.
- The main graphical parameters are:

`col`	colour (for borders)
`fill`	colour (for interiors)
`lwd`	line width
`lty`	line type
`cex`	text size multiplier

Graphical Parameters

- The `gpar()` function creates a list of graphical parameter settings.

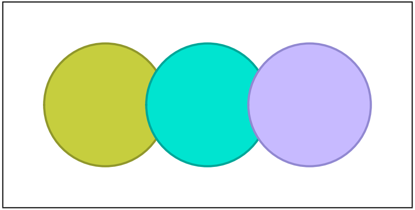
```
> grid.circle(r=.3,
              gp=gpar(col="red", fill="pink",
                      lwd=3, lty="dashed"))
```



Graphical Parameters

- When drawing multiple shapes with a single function call, graphical parameter settings can be vectors so that different shapes can have different appearances.

```
> hcl(1:3/2*180, 60, 60)
[1] "#90972B" "#00A698" "#9188D1"
> grid.circle(x=1:3/4, r=.3,
              gp=gpar(lwd=3,
                      col=hcl(1:3/2*180, 60, 60),
                      fill=hcl(1:3/2*180, 80, 80)))
```

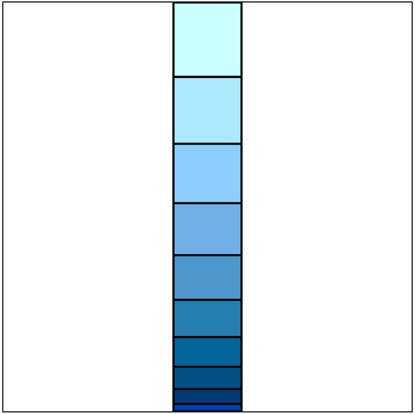


EXERCISES

$EXERCISE$

EXERCISES

- The goal of this exercise is to draw a vertical stack of rectangles as shown below.
- The rectangles are exactly one inch wide and each rectangle has a specific colour.



EXERCISES

- The raw data come from the first vector from the previous exercise (y1).
- A function is provided to generate proportions from a vector.

```
> prop <- function(x) {
      x/sum(x)
  }
```

- The proportions, p1, provide the heights of the rectangles in the stack and the cumulative proportions, cp1, provide the locations of the tops of the rectangles.

```
> p1 <- prop(y1)
```

- The colours for the rectangle fills are also provided.

```
> fills <- hcl(240, 60, seq(10, 100, 10))
```

Main Course

Main Course

Viewports

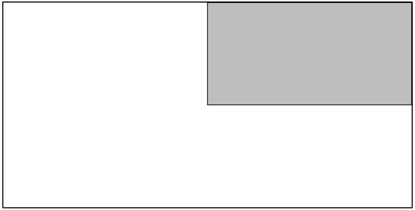
- A viewport is a description of a rectangular region on the page.
- The `viewport()` function creates viewports.
- Viewports have a location and size, both of which can be specified in any coordinate system.
- The viewport can be justified relative to its location.

```
> vp <- viewport(x=.5, y=.5,
                 width=.5, height=.5,
                 just=c("left", "bottom"))
```

Viewports

- The `pushViewport()` function creates a rectangular region on the page.
- All drawing occurs within the current viewport.

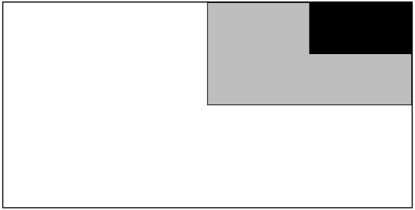
```
> pushViewport(vp)
> grid.rect(gp=gpar(fill="grey"))
```



Viewports

- Pushing of viewports also occurs within the current viewport.

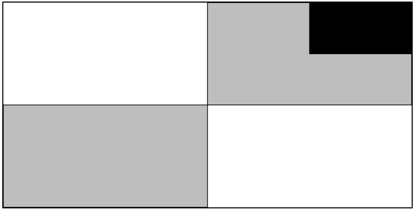
```
> pushViewport(vp)
> grid.rect(gp=gpar(fill="black"))
```



Viewports

- The popViewport() function removes the rectangular region from the page.

```
> popViewport(2)
> pushViewport(viewport(width=.5, height=.5,
                        just=c("right", "top")))
> grid.rect(gp=gpar(fill="grey"))
```

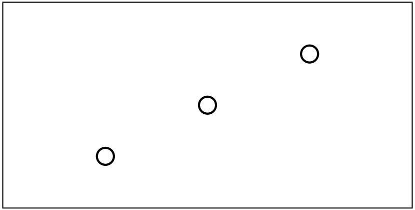


Viewports

- A viewport has an x-scale and a y-scale and these provide context for the `"native"` coordinate system.

- The `grid.newpage()` function starts a fresh page.

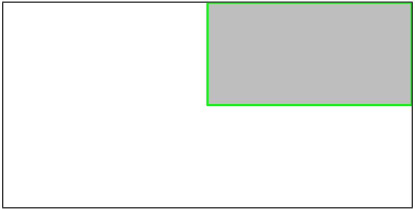
```
> grid.newpage()
> pushViewport(viewport(xscale=c(0, 4),
                        yscale=c(0, 4)))
> grid.points(unit(1:3, "native"),
              unit(1:3, "native"))
```



Viewports

- A viewport has a gp argument for setting graphical parameters.
- These settings provide default values for all drawing within the viewport.

```
> pushViewport(viewport(x=.5, y=.5,
                        width=.5, height=.5,
                        just=c("left", "bottom"),
                        gp=gpar(lwd=3, col="green")))
> grid.rect(gp=gpar(fill="grey"))
```

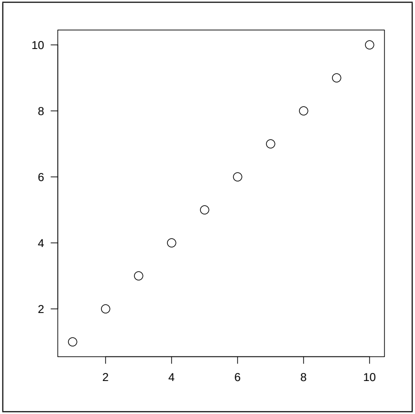


Viewports

- There are two convenience functions that create viewports for a simple plot.
    - The `plotViewport()` function creates a viewport with margins around the outside.
    - The `dataViewport()` function creates a viewport with the x-scale and y-scale based on data values.

```
> x <- 1:10
> y <- 1:10
> grid.newpage()
> pushViewport(plotViewport(c(4, 4, 2, 2)),
               dataViewport(x, y))
> grid.points(x, y)
> grid.xaxis()
> grid.yaxis()
> grid.rect()
```

Viewports

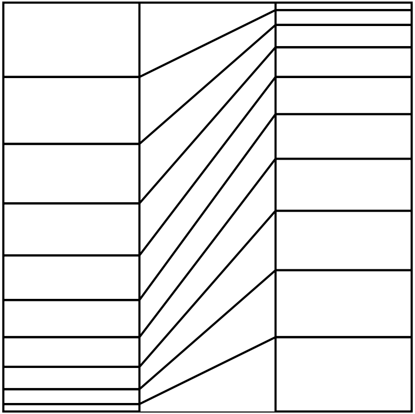


EXERCISES

EXERCISE

EXERCISES

- The goal of this exercise is to draw two vertical stacks of rectangles, with a set of line segments in between, as below.



EXERCISES

- The raw data are the two vectors from the previous exercises.
- Two functions are provided: `spine()` to generate a stack of rectangles and `connector()` to generate a set of line segments.

```
> spine <- function(x) {
      px <- prop(x)
      cpx <- cprop(x)
      grid.rect(y=cpx, height=px, just="top")
  }
> connector <- function(x1, x2) {
      cp1 <- cprop(x1)
      cp2 <- cprop(x2)
      grid.segments(0, cp1, 1, cp2)
  }
```

EXERCISES

- You need to create three viewports: one occupying the left third of the page, one occupying the central third, and one occupying the right third.
- Draw a stack of rectangles based on the data in `y1` in the left viewport, a stack of rectangles based on `y2` in the right viewport, and a set of line segments in the central viewport.

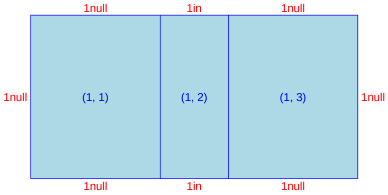
Main Course

Main Course

Layouts

- A layout divides a viewport into rows and columns.
- The height of each row in a layout can be specified in any coordinate system, **plus** the special "null" coordinate system, which is just for layouts. Column widths are similar.

```
> lyt <- grid.layout(1, 3,
                     widths=unit(c(1, 1, 1),
                       c("null", "in", "null")))
```

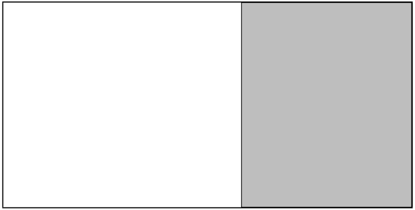


Layouts

- Viewports can be located and sized using a layout (rather than via an explicit location and size).

- A parent viewport can have a layout and then any viewports pushed within that parent can occupy particular rows/columns of the layout.

```
> pushViewport(viewport(layout=lyt))
> pushViewport(viewport(layout.pos.col=3))
> grid.rect(gp=gpar(fill="grey"))
```

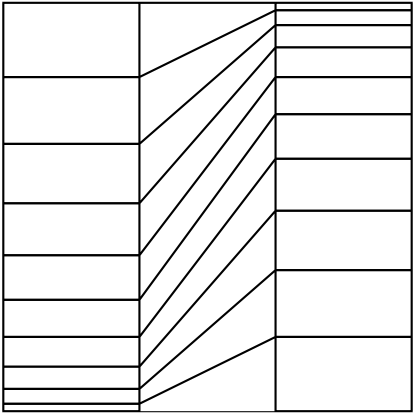


EXERCISES

EXERCISE

EXERCISES

- The goal of this exercise is to produce the same result as the previous exercise, except using a **layout** to position the components of the picture.



EXERCISES

- The raw data are the same two vectors from the previous exercise.
- The `spine()` and `connector()` functions to draw the stack of rectangles and the line segments are the same as in the previous exercise.

Review

Review

Review

grid provides the following tools to facilitate drawing statistical plots (among other things):

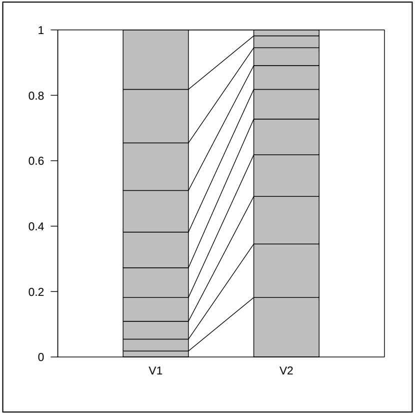
- basic shapes
- units (coordinate systems) for locating and sizing shapes
- graphical parameters for controlling the appearance of shapes
- viewports and layouts for creating local drawing contexts

EXERCISES

EXERCISE

EXERCISES

- The goal of this exercise is to produce a plot composed of stacks of rectangles and sets of line segments.



Main Course

Main Course

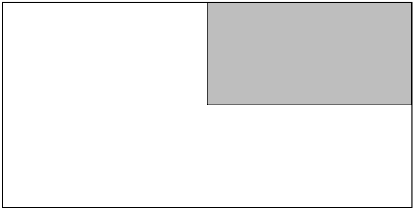
Reusing Viewports

- Viewports can have **names** and a record is kept of all viewports on the page.

- The `upViewport()` function reverts to the parent viewport context, but leaves the current viewport on the page.
- The `current.viewport()` function shows the current viewport.
- The `current.vpTree()` function shows all viewports on a page.
- The `downViewport()` function can be used to return to an existing viewport on the page.

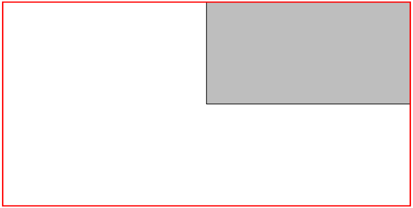
Reusing Viewports

```
> vp <- viewport(x=.5, y=.5,
                 width=.5, height=.5,
                 just=c("left", "bottom"),
                 name="top-right-vp")
> pushViewport(vp)
> grid.rect(gp=gpar(fill="grey"))
```



Reusing Viewports

```
> upViewport()
> grid.rect(gp=gpar(col="red", lwd=3))
```



```
> current.viewport()
```

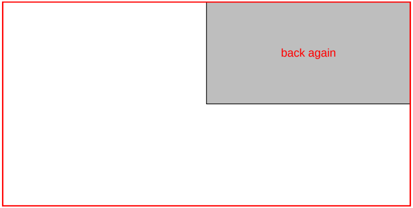
```
viewport[ROOT]
```

```
> current.vpTree()
```

```
viewport[ROOT]->(viewport[top-right-vp])
```

Reusing Viewports

```
> downViewport("top-right-vp")
> grid.text("back again", gp=gpar(col="red"))
```

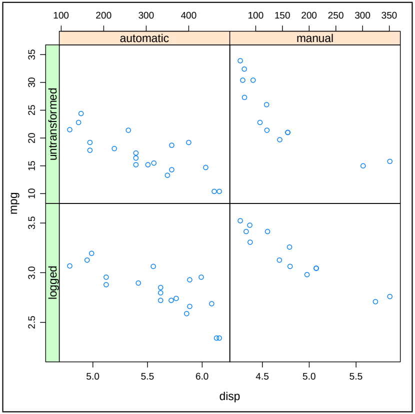


EXERCISES

EXERCISE

EXERCISES

- The goal of this exercise is to modify a **lattice** plot by reusing viewports.
- The modification involves adding the x-axes on the top strips.



EXERCISES

- The data are based on the `mtcars` data frame.

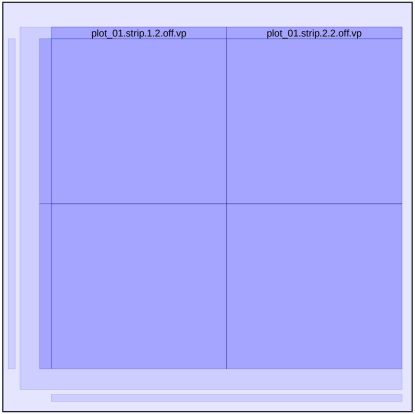
```
> mtcarsExp <- rbind(apply(mtcars[c("mpg", "disp")], 2, log),
                     mtcars[c("mpg", "disp")])
> mtcarsExp$am <- rep(ifelse(mtcars$am, "manual", "automatic"), 2)
> mtcarsExp$logged <- rep(c("logged", "untransformed"),
                          each=nrow(mtcars))
```

- The original plot is produced by the following code.

```
> library(lattice)
> plot <- xyplot(mpg ~ disp | am*logged, mtcarsExp,
                 scales=list(relation="free",
                             x=list(at=list(TRUE, TRUE, NULL, NULL)),
                             y=list(limits=list(c(2.2, 3.6), c(2.2, 3.6),
                                                c(10, 35), c(10, 35)),
                                    at=list(TRUE, NULL, TRUE, NULL))),
                 par.settings=list(layout.heights=list(axis.panel=c(1, 0),
                                                       top.padding=3),
                                   layout.widths=list(axis.panel=c(1, 0))))
> library(latticeExtra)
> print(useOuterStrips(plot))
```

EXERCISES

- The viewports that **lattice** created to draw the top two strips on this plot are called `"plot_01.strip.2.2.off.vp"` and `"plot_01.strip.1.2.off.vp"`.



EXERCISES

- You need to `downViewport()` to the appropriate viewport and call `grid.xaxis()` to add the x-axis (the strip viewports have an appropriate x-scale).
- The `grid.xaxis()` function has an argument `main`; set that to `FALSE` to draw the axis at the top of the viewport rather than the bottom.
- Use `upViewport()` to navigate back to the `ROOT` viewport; `downViewport()` returns the number of viewports that it went down.

Main Course

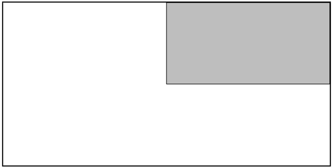
Main Course

Grobs

- Drawing a basic shape with **grid** is a **two-step** process.
  - First, a graphical object, or **grob**, is created, which contains a description of the shape.
  - Second, the shape is drawn on the page.
- Grobs can have **names** and a record is kept of all grobs on the page.
- The `grid.ls()` function lists the grobs that have been drawn on the current page.
- The `grid.edit()` function can be used to access a grob, by name, and modify it.

Grobs

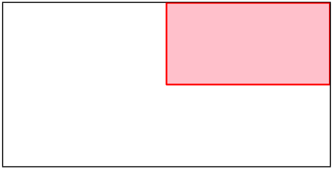
```
> vp <- viewport(x=.5, y=.5,
                 width=.5, height=.5,
                 just=c("left", "bottom"),
                 name="top-right-vp")
> pushViewport(vp)
> grid.rect(gp=gpar(fill="grey"), name="top-right-rect")
```



```
> grid.ls()
top-right-rect
```

Grobs

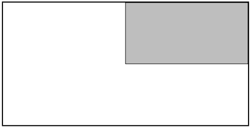
```
> upViewport()
> grid.edit("top-right-rect",
            gp=gpar(col="red", lwd=3, fill="pink"))
```



Viewports

- The `grid.ls()` function can also lists viewports.

```
> vp <- viewport(x=.5, y=.5,
                 width=.5, height=.5,
                 just=c("left", "bottom"),
                 name="top-right-vp")
> pushViewport(vp)
> grid.rect(gp=gpar(fill="grey"), name="top-right-rect")
```



```
> grid.ls(viewports=TRUE, fullNames=TRUE)
```

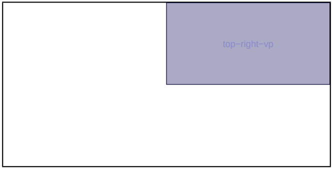
```
viewport[ROOT]
  viewport[top-right-vp]
    rect[top-right-rect]
```

Grobs and Viewports

In addition to `grid.ls()` ...

- The `showViewport()` function draws semitransparent rectangles and labels to represent the locations of viewports on the page.

```
> showViewport()
```



Grobs and Viewports

In addition to `grid.ls()` ...

- The `showGrob()` function draws semitransparent rectangles and labels to represent the locations of grobs on the page.

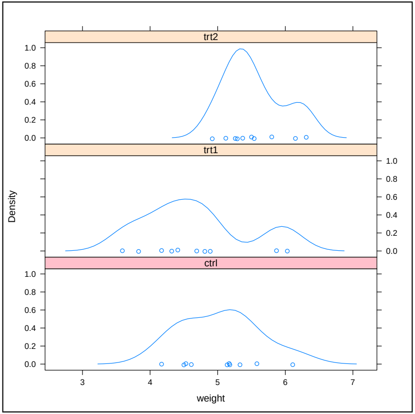
```
> showGrob()
```

EXERCISES

$EXERCISE$

EXERCISES

- The goal of this exercise is to modify a **lattice** plot by editing grobs.
- The modification involves changing the background colour of a single strip.



EXERCISES

- The original plot is produced by the following code.

```
> print(
  densityplot( ~ weight | group, PlantGrowth,
               layout=c(1, 3))
  )
```

- Use `grid.ls()` and/or `showGrob()` to inspect the grobs that **lattice** has created to find the one that corresponds to the bottom strip region.

- You need to `grid.edit()` the appropriate grob and set its `fill` to be `"pink"`.

Review

Review

Review

- A record is kept of the viewports and grobs that are drawn on a page.
- Viewports and grobs can have names.
- If other people name their viewports and grobs, it is easier for you to make modifications.
- If you name your viewports and grobs, it is easier for others to make modifications.

Dessert

Dessert

Modular Graphics

- Do NOT assume that you have the whole page to draw into.
- Name any viewports that you create.
- Use `upViewport()` so that the viewports remain available for others.
- Always end up in the viewport where you started.

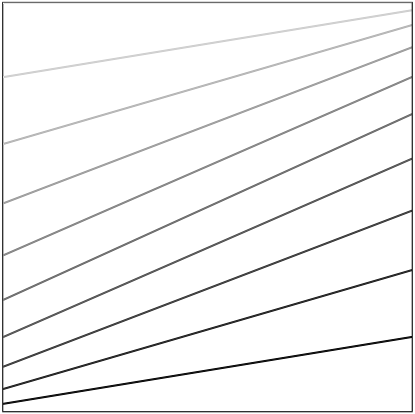
Modular Graphics

- A `connector()` function that draws line segments.

```
> connector <- function(x1, x2,
                        gp=gpar(),
                        name=NULL) {
      cp1 <- cprop(x1)
      cp2 <- cprop(x2)
      grid.segments(0, cp1, 1, cp2,
                    gp=gp, name=name)
  }
```

Modular Graphics

```
> connector(1:10, 10:1,
            gp=gpar(col=grey(1:10/11), lwd=3),
            name="connectorDemo")
```



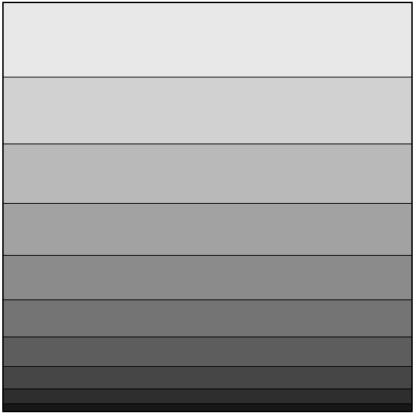
Modular Graphics

- A `spine()` function that draws rectangles.

```
> spine <- function(x,
                    gp=gpar(),
                    name=NULL) {
      px <- prop(x)
      cpx <- cprop(x)
      grid.rect(y=cpx, height=px, just="top",
                gp=gp, name=name)
  }
```

Modular Graphics

```
> spine(1:10,
        gp=gpar(fill=grey(1:10/11)),
        name="spineDemo")
```



Modular Graphics

- A `cplot()` function that draws a series of line segments and rectangles based on the columns of a data frame.
- The `width` argument controls the widths of the spines.

```
> cplot <- function(df, gp=gpar(), name="cplot") {
      for (i in 1:length(df)) {
          spineName <- paste(name, "spine", i, sep="-")
          pushViewport(viewport(x=unit(i, "native"),
                                width=unit(0.5, "native"),
                                name=spineName))
          spine(df[[i]], gp=gp, name=spineName)
          upViewport()
          if (i > 1) {
              conName <- paste(name, "con", i, sep="-")
              pushViewport(viewport(x=unit(i - 0.5, "native"),
                                    width=unit(0.5, "native"),
                                    name=conName))
              connector(df[[i - 1]], df[[i]], gp=gp, name=conName)
              upViewport()
          }
      }
  }
```

Modular Graphics

Some data preparation ...

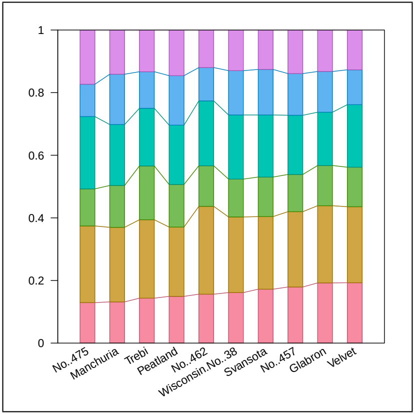
```
> barley1931 <- subset(barley, year == 1931)
> barley1931$variety <- reorder(barley1931$variety,
                                barley1931$yield,
                                FUN=function(x) {
                                    prop(x)[1]
                                })
> barley1931 <- barley1931[order(barley1931$variety), ]

> col <- hcl(seq(0, 300, 60), 70, 50)
> fill <- hcl(seq(0, 300, 60), 70, 70)
```

Modular Graphics

```
> grid.newpage()
> pushViewport(plotViewport(c(5, 4, 2, 2),
                            xscale=c(0, 11),
                            yscale=0:1),
               viewport(clip=TRUE,
                        xscale=c(0, 11),
                        yscale=0:1))
> df <- as.data.frame(split(barley1931$yield,
                            barley1931$variety))
> cplot(df, gp=gpar(col=col, fill=fill))
> popViewport()
> grid.text(colnames(df),
            x=unit(1:10, "native"),
            y=unit(-0.5 ,"lines"),
            rot=30, just="right")
> grid.yaxis()
> grid.rect()
> popViewport()
```

Modular Graphics

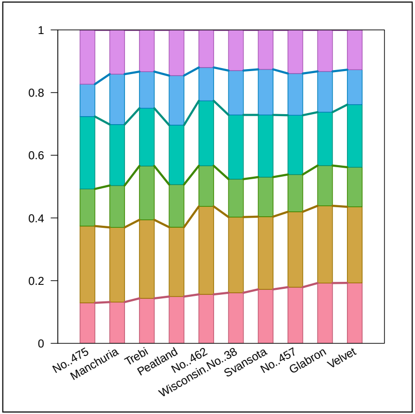


Modular Graphics

```
> grid.ls(viewports=TRUE, fullNames=TRUE)
viewport[ROOT]
  viewport[GRID.VP.114]
    viewport[GRID.VP.115]
      viewport[cplot-spine-1]
        rect[cplot-spine-1]
        upViewport[1]
      viewport[cplot-spine-2]
        rect[cplot-spine-2]
        upViewport[1]
      viewport[cplot-con-2]
        segments[cplot-con-2]
        upViewport[1]
      viewport[cplot-spine-3]
        rect[cplot-spine-3]
        upViewport[1]
      viewport[cplot-con-3]
        segments[cplot-con-3]
        upViewport[1]
      viewport[cplot-spine-4]
        rect[cplot-spine-4]
        upViewport[1]
      viewport[cplot-con-4]
        segments[cplot-con-4]
        upViewport[1]
      viewport[cplot-spine-5]
        rect[cplot-spine-5]
        upViewport[1]
      viewport[cplot-con-5]
        segments[cplot-con-5]
        upViewport[1]
      viewport[cplot-spine-6]
```

Modular Graphics

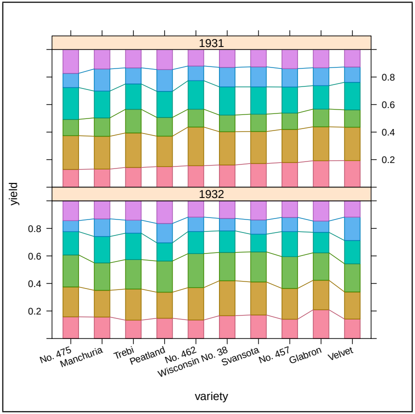
```
> grid.edit("con", grep=TRUE, global=TRUE,
            gp=gpar(lwd=3))
```



Modular Graphics

```
> barley$variety <- factor(barley$variety,
                           levels=levels(barley1931$variety))
> panel.cplot <- function(x, y, groups, subscripts, ...) {
      cplot(as.data.frame(split(y, x)),
            gp=gpar(col=col, fill=fill))
  }
> print(
  xyplot(yield ~ variety | year, barley,
         groups=site, layout=c(1, 2),
         scales=list(x=list(rot=20), y=list(limits=0:1)),
         panel=panel.cplot)
  )
```

Modular Graphics



Coffee & Cigars

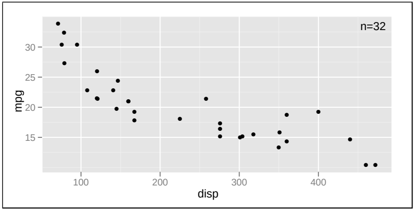
Coffee & Cigars

Editing **ggplot2**

- **ggplot2** creates viewports and grobs when it draws a plot, BUT ...
- ... the viewport for the plot region has a 0-to-1 scale AND ...
- ... the grobs that it creates a more complex, hierarchical objects SO ...
- ... some **grid** changes are not as easy to make compared to editing **lattice**.

Editing **ggplot2**

```
> library(ggplot2)
> qplot(disp, mpg, data=mtcars)
> downViewport("panel-3-3")
> grid.text("n=32",
            x=unit(1, "npc") - unit(2, "mm"),
            y=unit(1, "npc") - unit(2, "mm"),
            just=c("right", "top"))
```

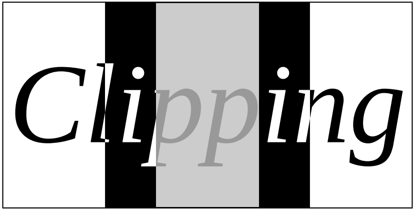


Clipping

- It is possible to set a rectangular clipping region so that drawing can only occur inside that region.
- Viewports have a `clip` argument to indicate whether drawing should be clipped to the viewport.
- The `grid.clip()` function sets the clipping region within a viewport.

Clipping

```
> grid.text("Clipping")
> pushViewport(viewport(width=0.5, clip=TRUE))
> grid.rect(gp=gpar(fill="black"))
> grid.text("Clipping", gp=gpar(col="white"))
> grid.clip(width=0.5)
> grid.rect(gp=gpar(fill="grey80"))
> grid.text("Clipping", gp=gpar(col="grey60"))
```



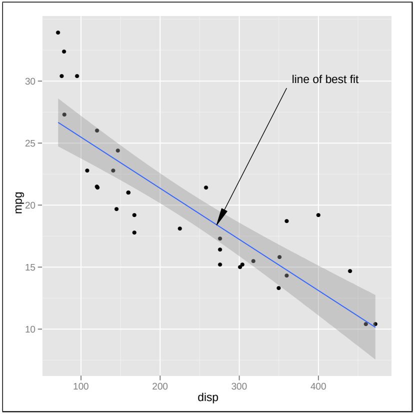
Querying Grobs

- It is possible to ask a grob about its location and size.
- The `grobWidth()` function returns the width of a grob. There is also `grobHeight()`.
- The `grobX()` function returns an x-location on the boundary of a grob. There is also `grobY()`.

Querying Grobs

```
> ggplot(aes(x=disp, y=mpg), data=mtcars) +
      geom_point() +
      geom_smooth(method="lm")
> downViewport("panel-3-3")
> sline <- grid.get(gPath("smooths", "polyline"),
                    grep=TRUE)
> grid.segments(.7, .8,
                grobX(sline, 45), grobY(sline, 45),
                arrow=arrow(angle=10, type="closed"),
                gp=gpar(fill="black"))
> grid.text("line of best fit",
            x=unit(.7, "npc") + unit(2, "mm"),
            y=unit(.8, "npc") + unit(2, "mm"),
            just=c("left", "bottom"))
```

Querying Grobs



Finis!

Finis!