Plotting II

Important Ingredients in the Best Plot Recipes

Paul E. Johnson[1,2]

[1]Department of Political Science
University of Kansas

[2]Center for Research Methods and Data Analysis
University of Kansas

2013

Outline

The Screen as an Output Device

- You make a nice plot and view it on the screen.
- The "window" is a device, as far as R is concerned.
- You can create more than one plotting device. Try:

```
dev.new()
```

or

```
dev.new(height = 3, width = 7)
```

That should create a new "screen" device and the next plot command you run will draw on it.

- Possible to have several devices, and alternate among them.
- What's my current device? Run:

```
dev.cur()
```

- Related functions: "dev.next()" and "dev.set()" and "dev.off()"

A Printer is also a Device

- Send the current device's output to the default printer

```
dev.print(paper="letter", height=4, width=4)
```

- Here's our major problem. If the screen device is a different size & shape, then copying or printing it to another device & shape will usually create a ugly result.
- Users frequently note that a plot looks great if they draw it on a 8 inch by 6 inch plot, but when they try to print that in a smaller section of paper, the result is horrible.
- Furthermore, on-screen devices do not always gracefully resize when we stretch and resize them.

Save pdf, postscript, png, etc

- Things we call "file types" or "formats" are **devices** in R.
- Recall the "window" where you see the graph is a "device" in the eyes of R.
- Problems arise translating "that" device into a file-formatted device.
  - Screen devices are generally "square", say 7" by 7"
  - A screen device output sized for 7" x 7" will look bad if it is written into a 6 inch x 4 inch output device.
  - This is a major problem with Rstudio, because it constrains the graphic device to be a particular rectangle on the screen.

Many Devices Are Available

- Today, I almost always save in `pdf` or `png` devices. `pdf` is scalable, `png` is for web pages.
- Many other excellent devices exist, I'm considering studying up on `svg` or `tikz`
- `xfig` device creates output that can be revised in xfig (Godfather of free line art software). That's useful sometimes.
- `pdf` is now R's default recommendation (was postscript when I started)

Ask your system what it has.

- Run

```
?Devices
```

  should list the base devices and some other possibilities your system *might* have.

- Check your particular system for extra devices.

```
capabilities()
```

  Today I see

```
jpeg       png      tiff     tcltk        X11       aqua  http/ftp
    sockets
TRUE      TRUE      TRUE      TRUE      TRUE      FALSE      TRUE
         TRUE
libxml      fifo    cledit     iconv       NLS   profmem     cairo
TRUE      TRUE      TRUE      TRUE      TRUE      TRUE      TRUE
```

"But I Can Just Point and Click," you say...

Almost all of my students don't pay attention to me on this. They use a "graphical environment" with a pull down menu "File Save As". I say STOP that.
Why?

- **Quality** The result will be bad if the output device shape and size is not similar to the screen device. Better to create a "certain" sized device and let R write on it.
- **Reproducibility**: Pointing and Clicking does not equal reproducible output. You can't "give" that to students & co-authors
- **Speed**: Suppose you need to scan through 200 datasets and make 3 plots each. Do you really want to point-and-click through there? Really?

The Step-by-Step Procedure

Your Code will

1. Create an output device (pdf, png, etc)
2. Run the commands that create the figure (will not show on the screen)
3. Turn off the device in order to finish off the file.

While you are "experimenting" with code, don't execute code in steps 1 and 3, so you can see output on screen. If you see my example code with `if (SAVEME) pdf(file = "somefile.pdf", ...)`, you will know what what I mean.
Check WorkingExample scripts like "glm-logit-unbalanced-1.R" or "glm-logit-compareLinks-1.R"

Save Files from the R Script

- I try to remember to have this "boiler plate" at the top of every R program that I write when I might need to save graphs

```
pdf.options(onefile = FALSE, family = "Times", paper =
    "special", height = 4, width = 6)
ps.options(horizontal = FALSE, onefile = FALSE, family
    = "Times", paper = "special", height = 4, width =
   6 )
options(papersize="special")
```

- These set my taste for the "postscript" and "pdf" output devices.
- *IF* I remember those, then saving a file is not so tedious.

Explain those arguments

horizontal = FALSE : I prefer "portrait" output

onefile = FALSE : Don't try to put all plots into a single file (I want separate images in separate files)

onefile = FALSE : vital for postscript devices. Otherwise, no bounding box is included and the result is not true EPS (Encapsulated Postscript)

paper = "special" : Helps with pdf output. Otherwise, it assumes all pdf images are full page size

family="Times" : I prefer to see figures with fonts that match the text.

Typical Experience

- First, begin with the process of creating a graph. Run it on screen until you get it right, and then "wrap" your commands in this way:

```
pdf(file = "myimage.pdf")
##or postscript(file = "myimage.eps")
##"put your plot commands here")
dev.off()    #vital to save your file!
```

- BIG WARNING 1. If you "turn on" the file device, you will not get to see the output on the screen in the R graphic device.
- BIG WARNING 2. If you "turn on" an output device, and forget to run dev.off(), you will cause lots of trouble for yourself. Try it and see.

If You Forget the Boilerplate

- You omit pdf.options, then you need to write in details each time, e.g.

```
pdf(file = "myfavorite1.pdf", onefile = FALSE, family
    = "Times", paper = "special", height = 4, width =
    6)
#hist(x) or whatever
dev.off()
```

- Instead of

```
pdf(file="myfavorite1.pdf")
#hist(x) or whatever
dev.off()
```

- Even if you did set pdf.options, you can customize

```
pdf(file="myfavorite1.pdf", height=8, width=7)
#hist(x) or whatever
dev.off()
```

Conditionalize Code To Make This Less Tedious

- Do your on-screen work like this

```
SAVEME <- FALSE
if (SAVEME==TRUE) pdf(file="myfile1.pdf")
## your graph code here
if (SAVEME==TRUE) dev.off()
```

- To save figures in files, change

  `SAVEME <- TRUE`

  and then run the whole script over again.

Another Way To Make This Less Tedious

- Let R number the figures for you.
- Put this in the program BEFORE creating plots

```
pdf(file = "myfile1%003d.pdf", onefile =FALSE)
## Add other arguments if pdf.options not set
    previously.
```

- When finished, tell R to stop creating output files.

```
dev.off()
```

- When you run this program, no graphs will "show on the screen."
- Each graph you draw will go in a separate file, `myfile001.pdf`, `myfile002.pdf`, `myfile003.pdf`.
- The downside here is that all figures have to have same height, width

Example Code that Writes a Bunch of Figures

```
for (i in 1:10){
    fn <- paste("fig", i, ".pdf", sep="")
    mm <- 3*i; msd <- i^2;
    mtitle <- paste("Histogram", i, "m =", mm, "sd =", msd)
    pdf(file = fn)
    hist(rnorm(1000, mean = 3*i, sd = i^2),  main = mtitle)
    dev.off()
   }
```

Brief Note on par

- Many settings that affect graphs are sitting "out there" in memory
- To see them, run

```
par()
```

- To see a particular one, run

```
par("oma")
```

- To set one, run

```
par(oma = c(1, 1, 1, 1))
```

Brief Note on par

- Danger: those settings will remain in effect for the duration of the device.
- When you close that device, all of the changes are "forgotten" and par() for that device reverts.
  - Need to re-run if you change devices
  - Many people recommend getting in the routine of saving par settings and replacing them when done.

```
old.par <- par(no.readonly = TRUE)
## make your changes to par
par(xpd = TRUE, mar = c(3, 4, 1, 2))
## make desired figure. When done, restore device.
par(old.par)
```

  - Seems somewhat unnecessary to me because you can close the device and "start fresh"

par: mfcol and mfrow

- Either of these will cause R to subdivide a plotting device into 3 rows

```
par(mfcol = c(3,1))
par(mfrow = c(3,1))
```

- Either of these will subdivide a plotting device into 3 rows, 2 columns

```
par(mfcol = c(3, 2))   # Fill columns first
par(mfrow = c(3, 2))   # Fill rows first
```

mfrow fills rows, mfcol fills columns

- The order of "drawing" is the difference between mfcol and mfrow
- Suppose 6 plots are A, B, C, D, E, F

mfcol fills the columns first

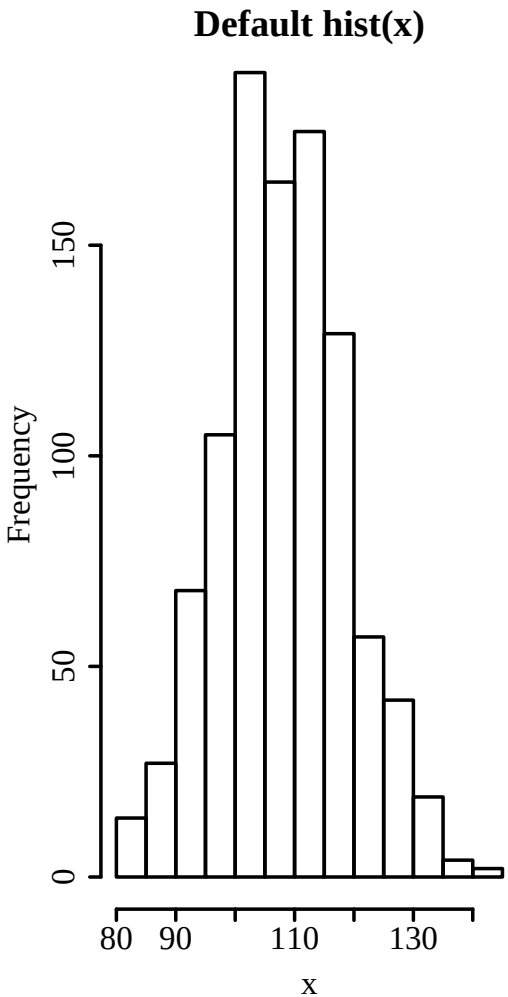
A	D
B	E
C	F

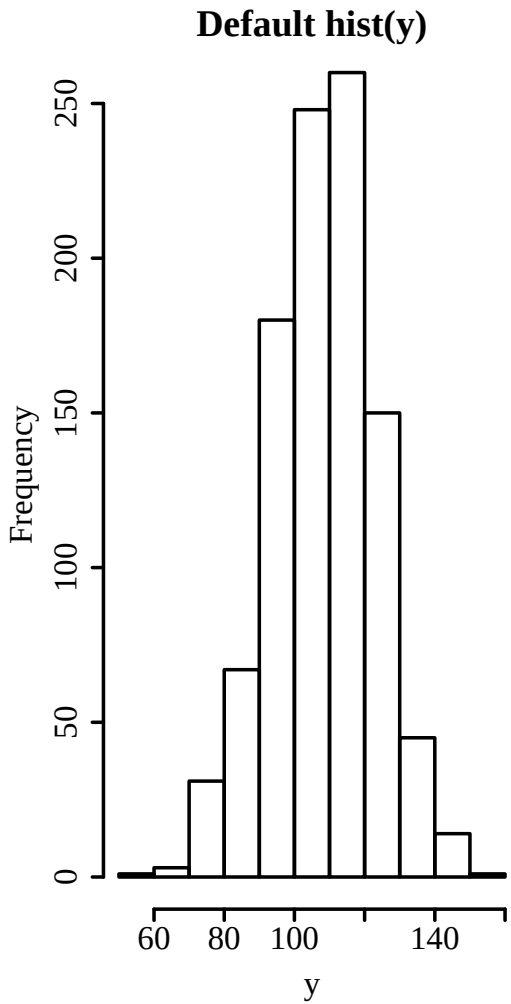
mfrow fills the rows first

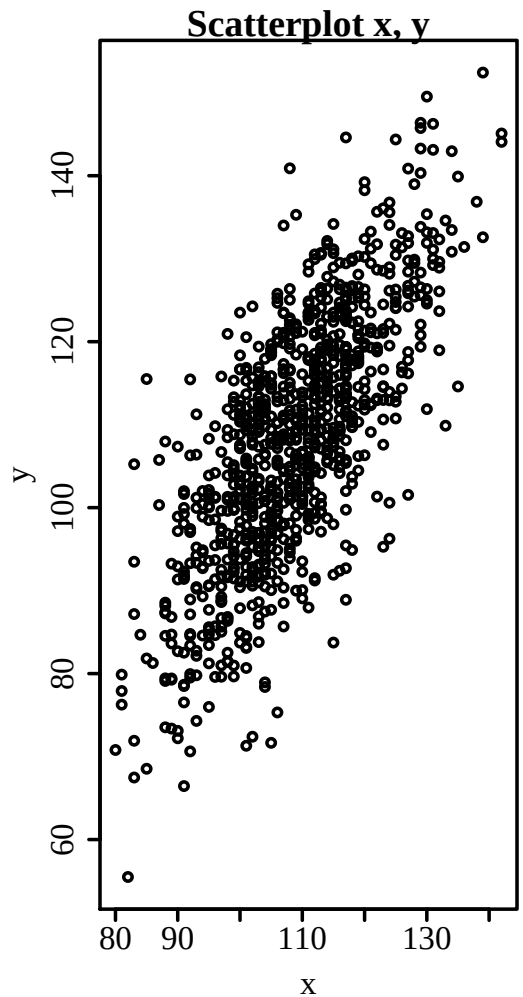
A	B
C	D
E	F

Example: to fit horizontal screen

```
x <- rpois(1000, lambda = 108);
y <- rnorm(1000, m = x, sd = 10);
par(mfcol = c(1,3))
hist(x, main = "Default hist(x)")
hist(y, main = "Default hist(y)")
plot(x, y, main = "Scatterplot x, y")
```


Default hist(x)
Frequency
0
50
100
150
80
90
110
130
x


Default hist(y)
Frequency
0
50
100
150
200
250
60
80
100
140
y


Scatterplot x, y
y
60
80
100
120
140
80
90
110
130
x

When you are finished...

Either:

- Reset the option manually
  par(mfcol=c(1,1))
- Or close the device
  dev.off()

Some R Functions Use `mfcol` Behind the Scenes

- Suppose you have a factor variable like “race” or “sex”
- coplot generates separate plot for each value of the factor

```
example(coplot)
```

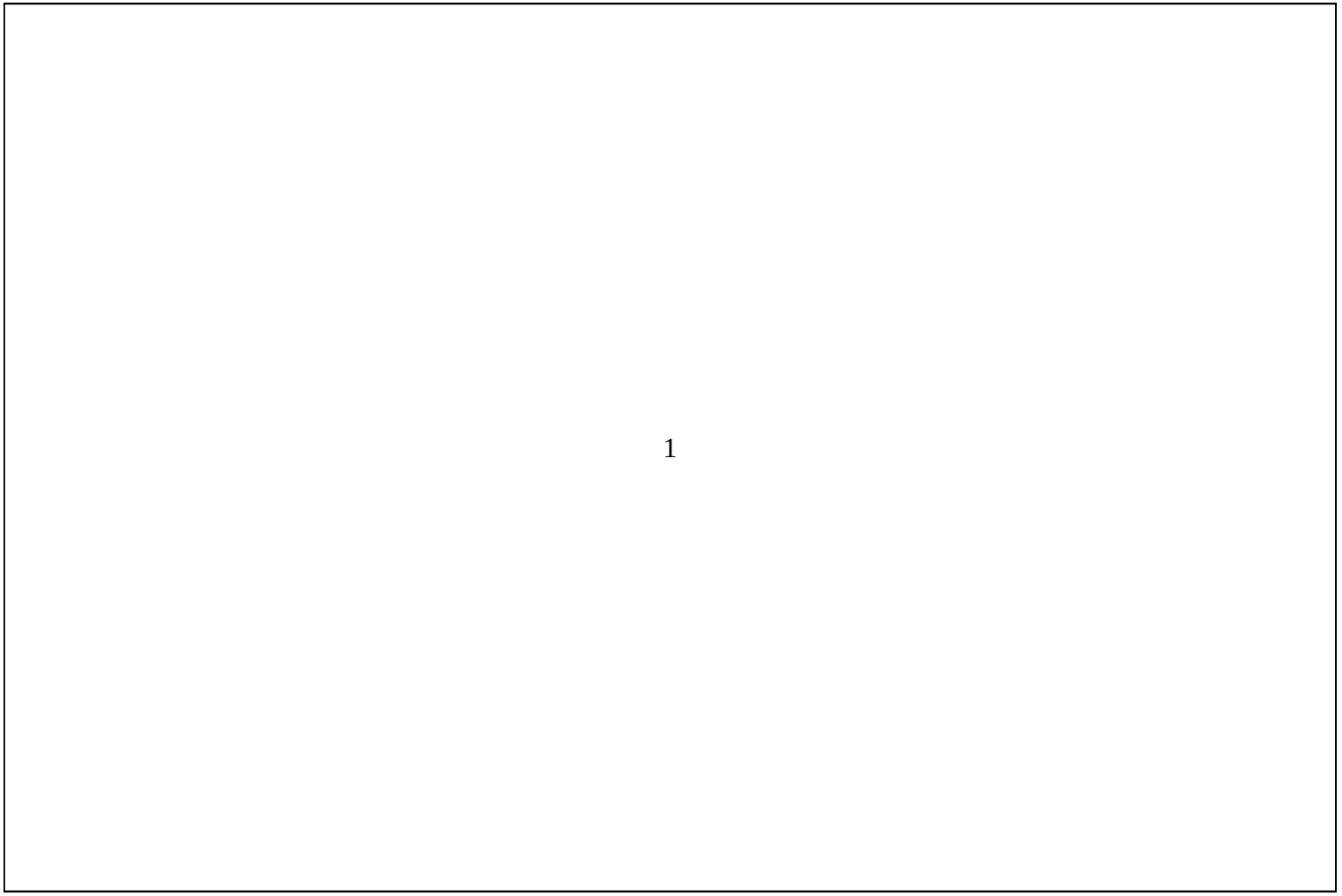
R's Layout() Function Offers More Control

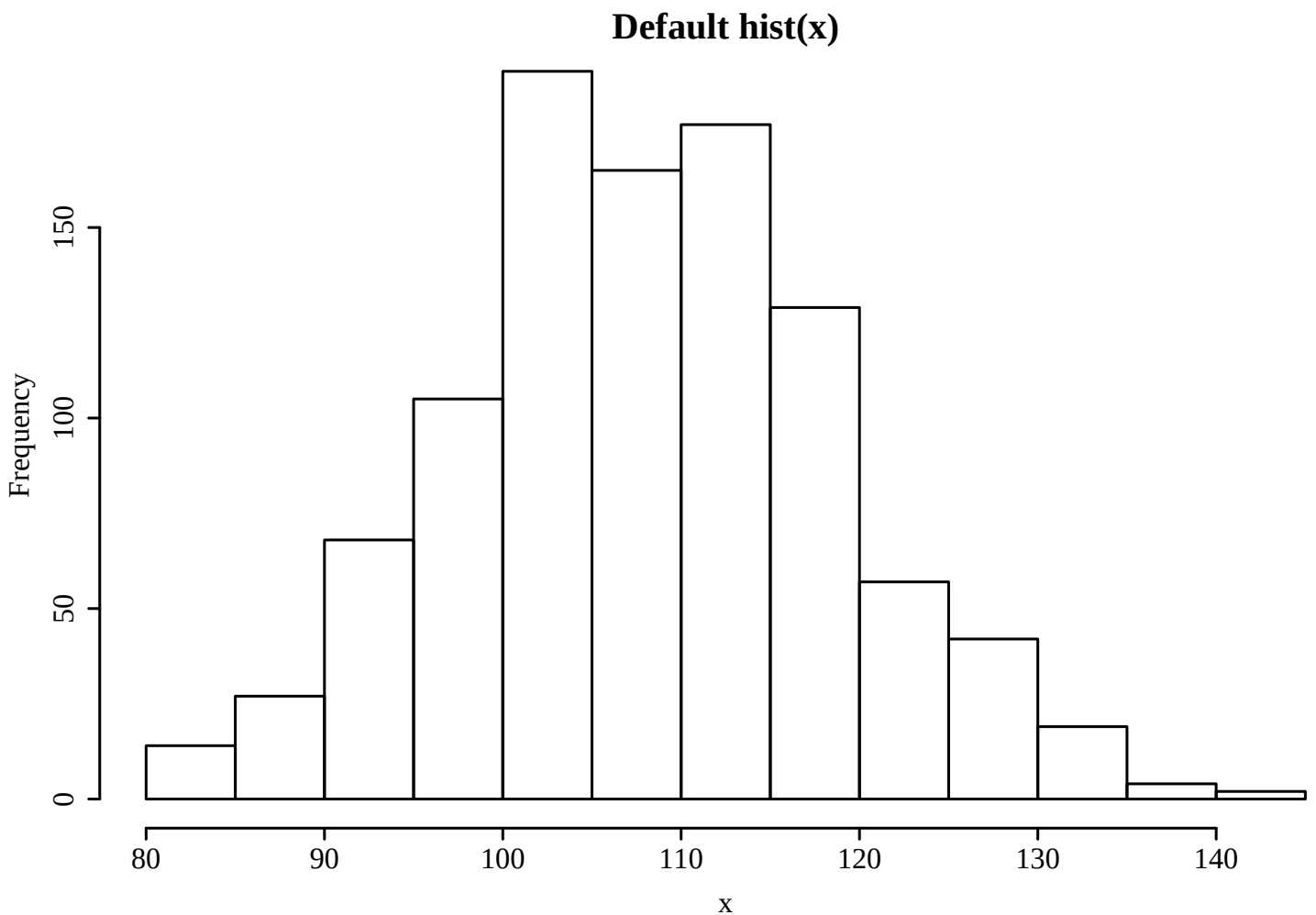
- Check the current layout:
  `"layout.show(1)"`
  It should put a numeral "1" on a screen device.
- layout wants you to specify an R matrix to
  - divide the space by rows and columns, and
  - let R know which figure will be drawn in which area
- Stupid example

$$mymat = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{1}$$

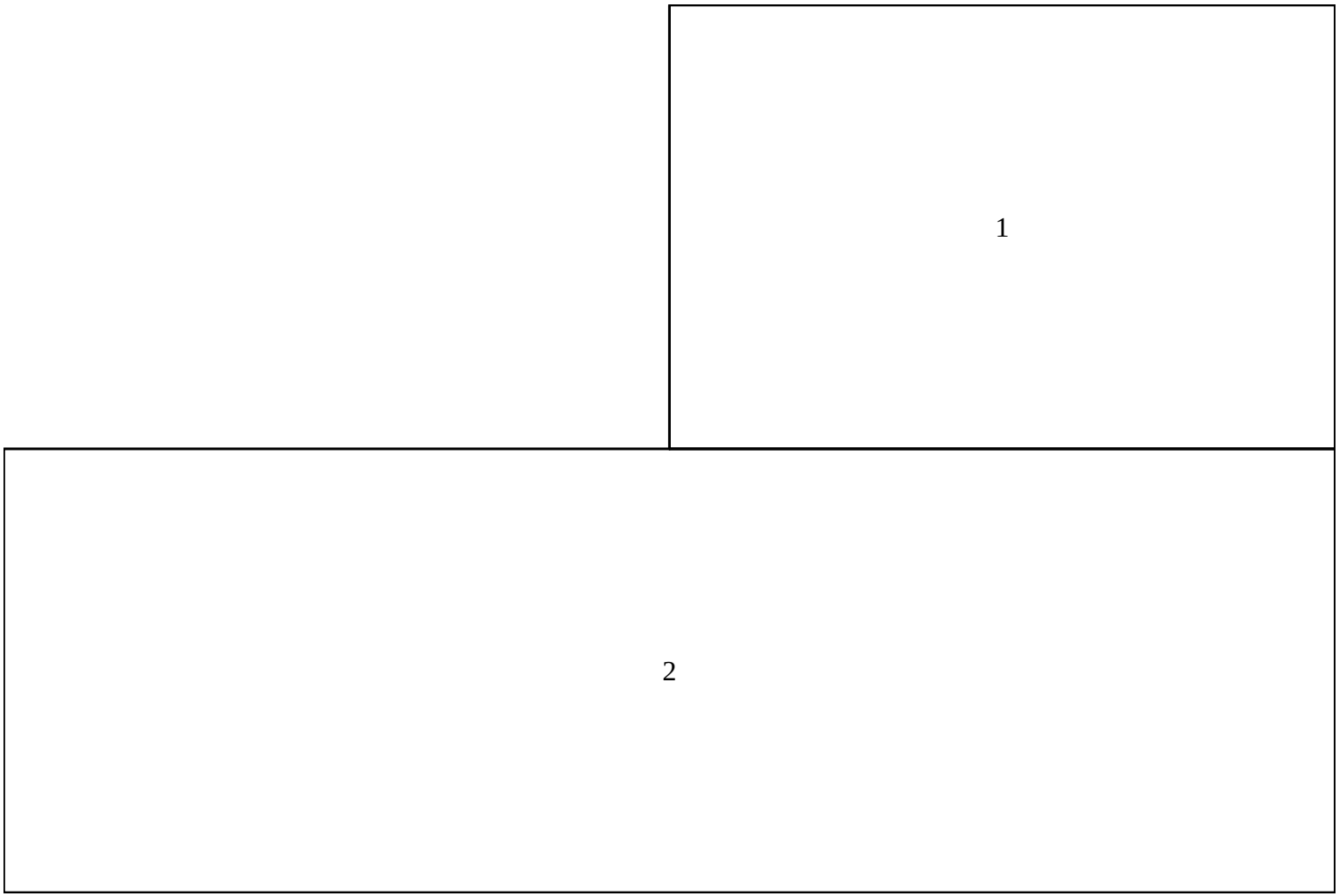
Meaning: divide my plotting area into 4 parts, but let figure "1" use all 4 parts.

```
mymat <- matrix( c(1, 1, 1, 1), ncol = 2)
layout(mat = mymat)
layout.show(1)
```

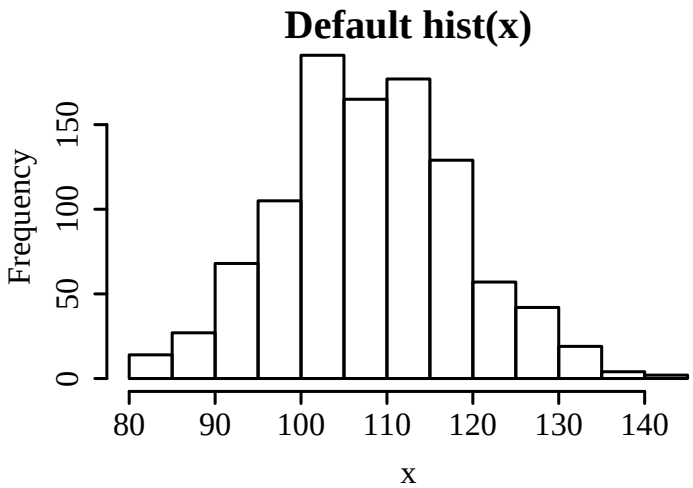


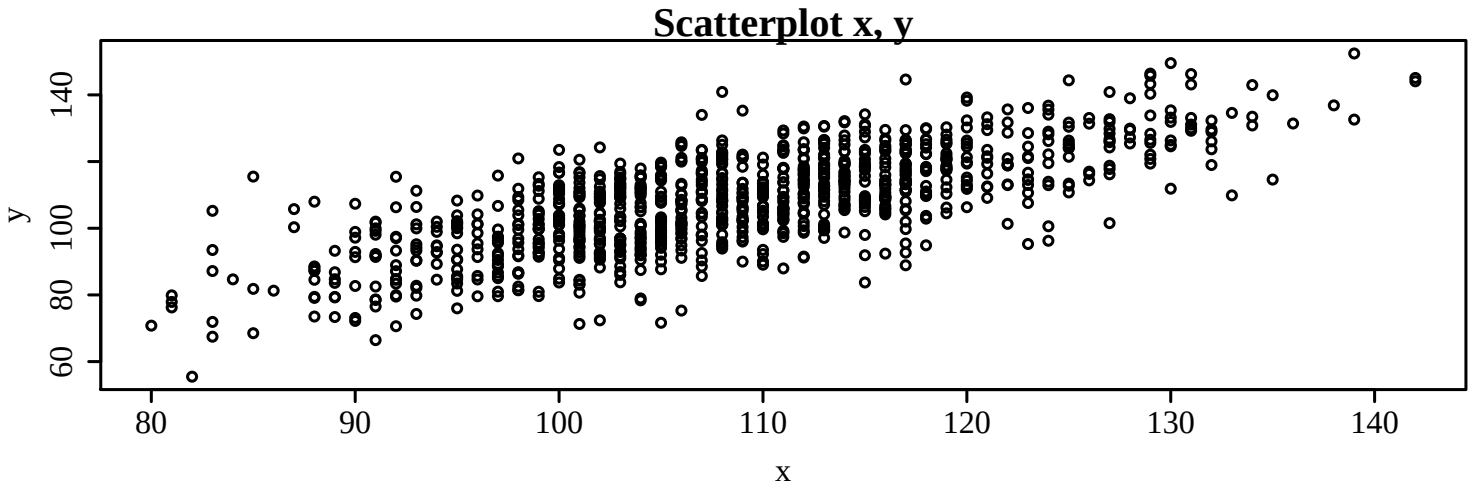

Default hist(x)
Frequency
0
50
100
150
80
90
100
110
120
130
140
x

```
mymat <- matrix(c(0,2,1,2), ncol = 2)
layout(mat = mymat)
layout.show(2)
```



R Squishes the Figures to Fit!





Layout 5 images in a "rainbow" shape

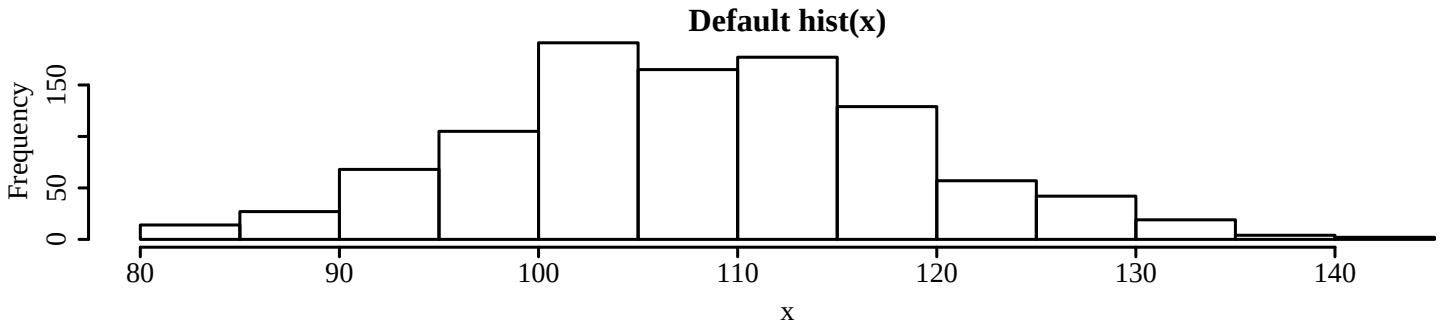
$$mymat = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 3 \\ 4 & 0 & 5 \end{bmatrix} \tag{2}$$

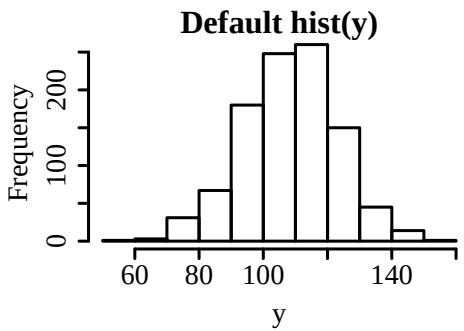
```
mymat <- matrix(c(1,2,4,1,0,0,1,3,5), nrow = 3)
layout(mat = mymat)
layout.show(5)
```

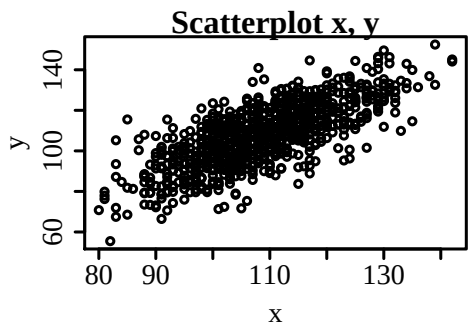
You Need 5 Plots for 5 Indicated Spaces

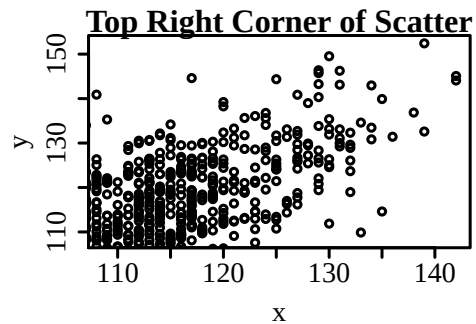
```
layout(mat = mymat)
hist(x, main = "Default hist(x)")
hist(y, main = "Default hist(y)")
plot(x, y, main = "Scatterplot x, y")
plot(x, y, main = "Top Right Corner of Scatter", xlim = c(
    mean(x), max(x)), ylim = c(mean(y),max(y)))
plot(x, y, main = "Bottom Left Corner of Scatter", xlim = c
    (min(x), mean(x)), ylim = c(min(y), mean(y)))
```

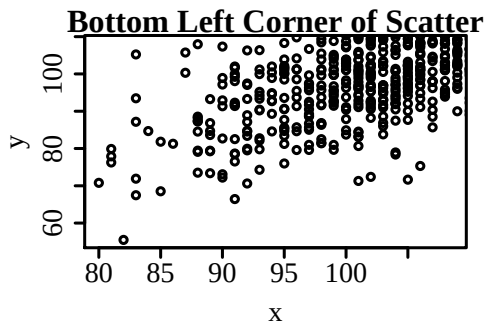
Voila


Default hist(x)
Frequency
0
50
150
80
90
100
110
120
130
140
x


Default hist(y)
Frequency
0
100
200
60
80
100
140
y


Scatterplot x, y
y
60
100
140
80
90
110
130
x


Top Right Corner of Scatter
y
110
130
150
110
120
130
140
x


Bottom Left Corner of Scatter
y
60
80
100
80
85
90
95
100
x

Cautionary Comment about Layout

- Layout affords the ability to fill a page with an arbitrarily complicated array of figures.
- I worry that some of this effort is wasted because some/many journals do not want a full page layout of graphs. If they will give you the space at all, they probably want the individual figures in separate graphs that can be arranged according to their style.
- Exception would be layout used to link together inseparable drawings, e.g. “rug” plots below scatterplots.

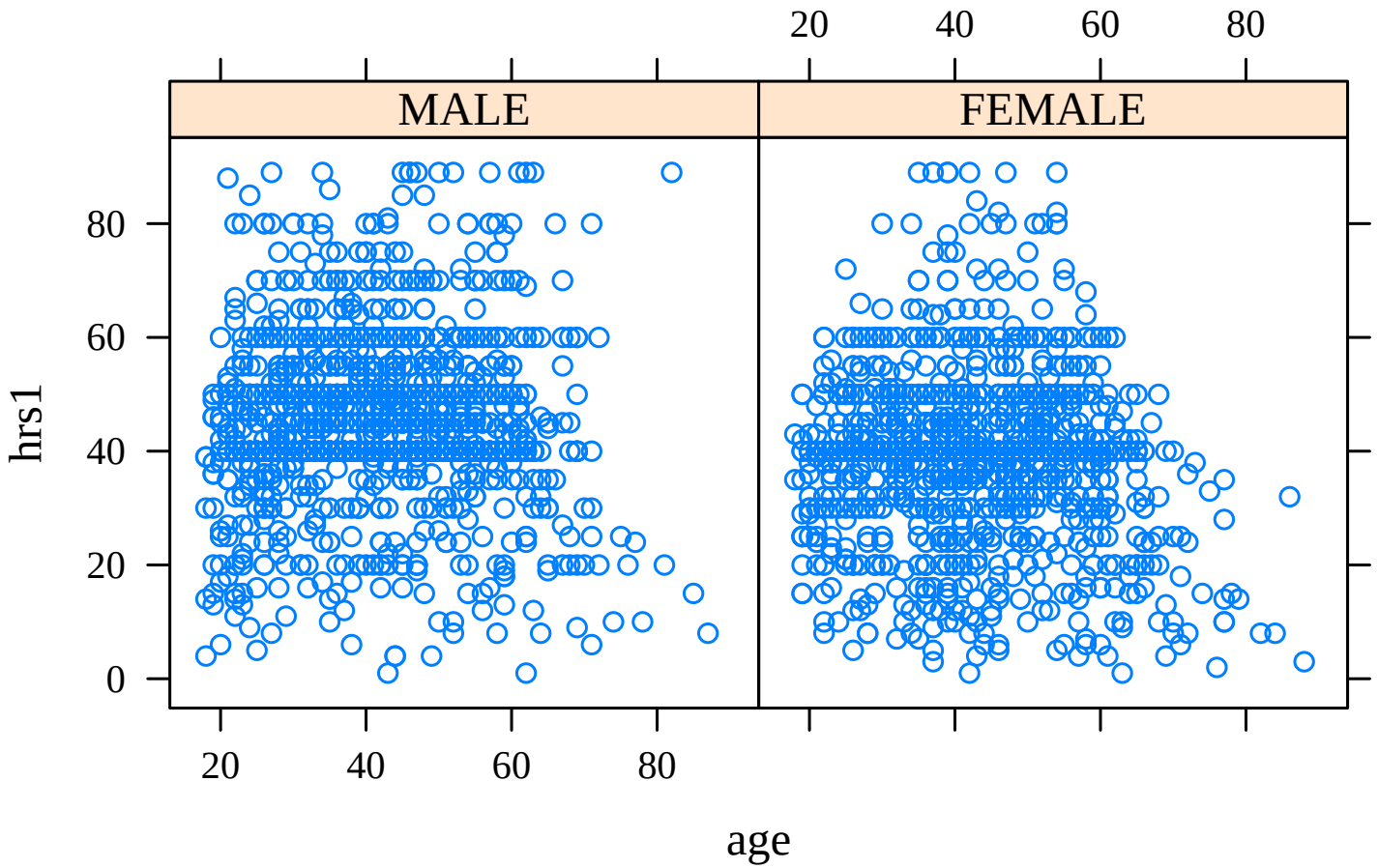
Trellis Graphics with the lattice Package

- Trellis graphics in S/S+ are highly prized (natural scientists especially)
- Main idea: make it easy/simple to create many plots that subdivide a sample and illustrate for subgroups (coplot, but a fully worked out framework)
- Deepayan Sarkar contributed the lattice package for R
- Some see lattice graphs as a complete replacement for existing 2D plotting framework (e.g., Doug Bates).
- I am more cautious, possibly because
  - lattice graphics came along after I had mastered the others
  - seems tougher to get individual pieces "just right"
  - see previous concern about difficulty of layout in publication.

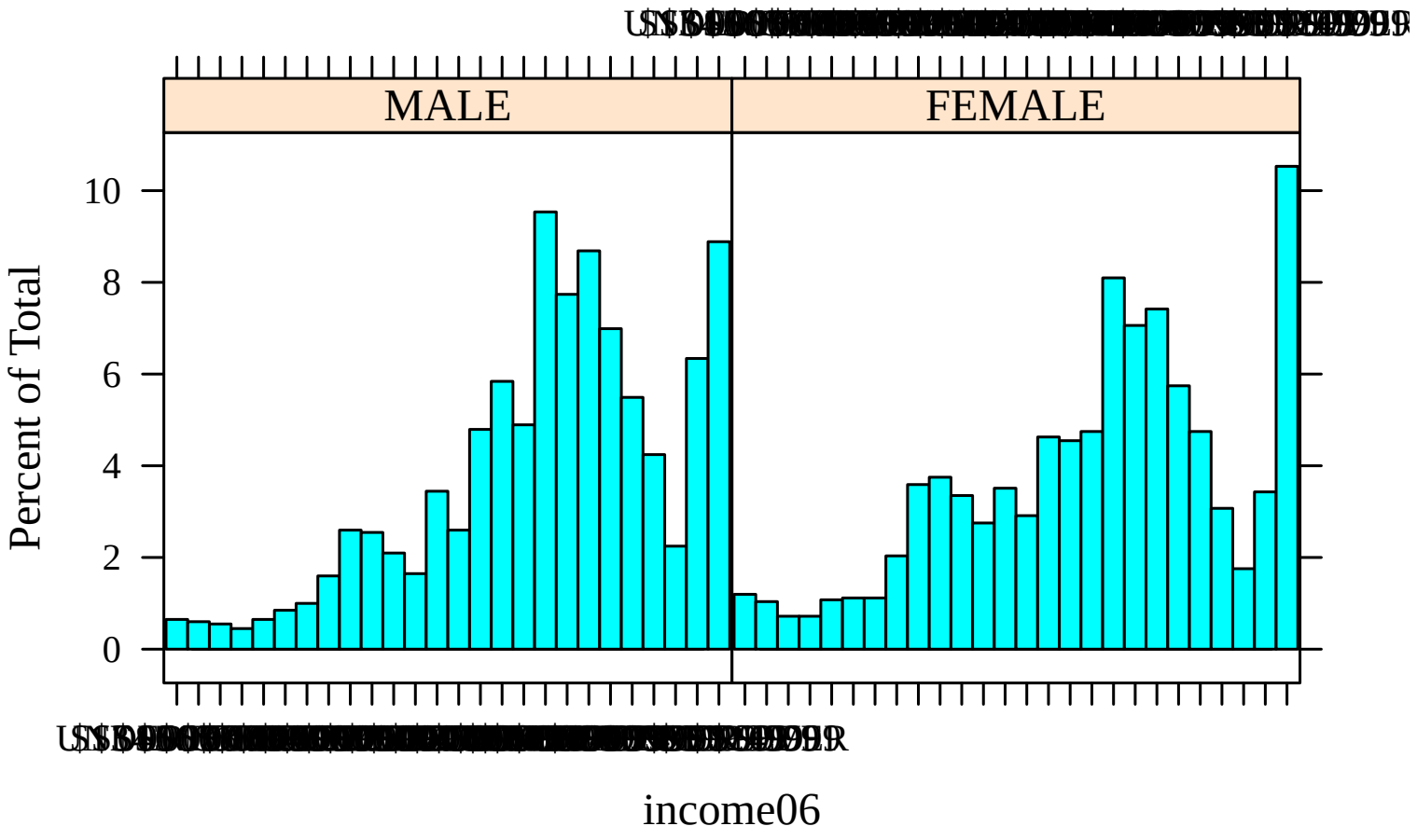
xyplot is the workhorse generic function

- xyplot will accept a formula and figure out what to do with it.
- After a vertical bar, | , factors indicate desired subgroups.

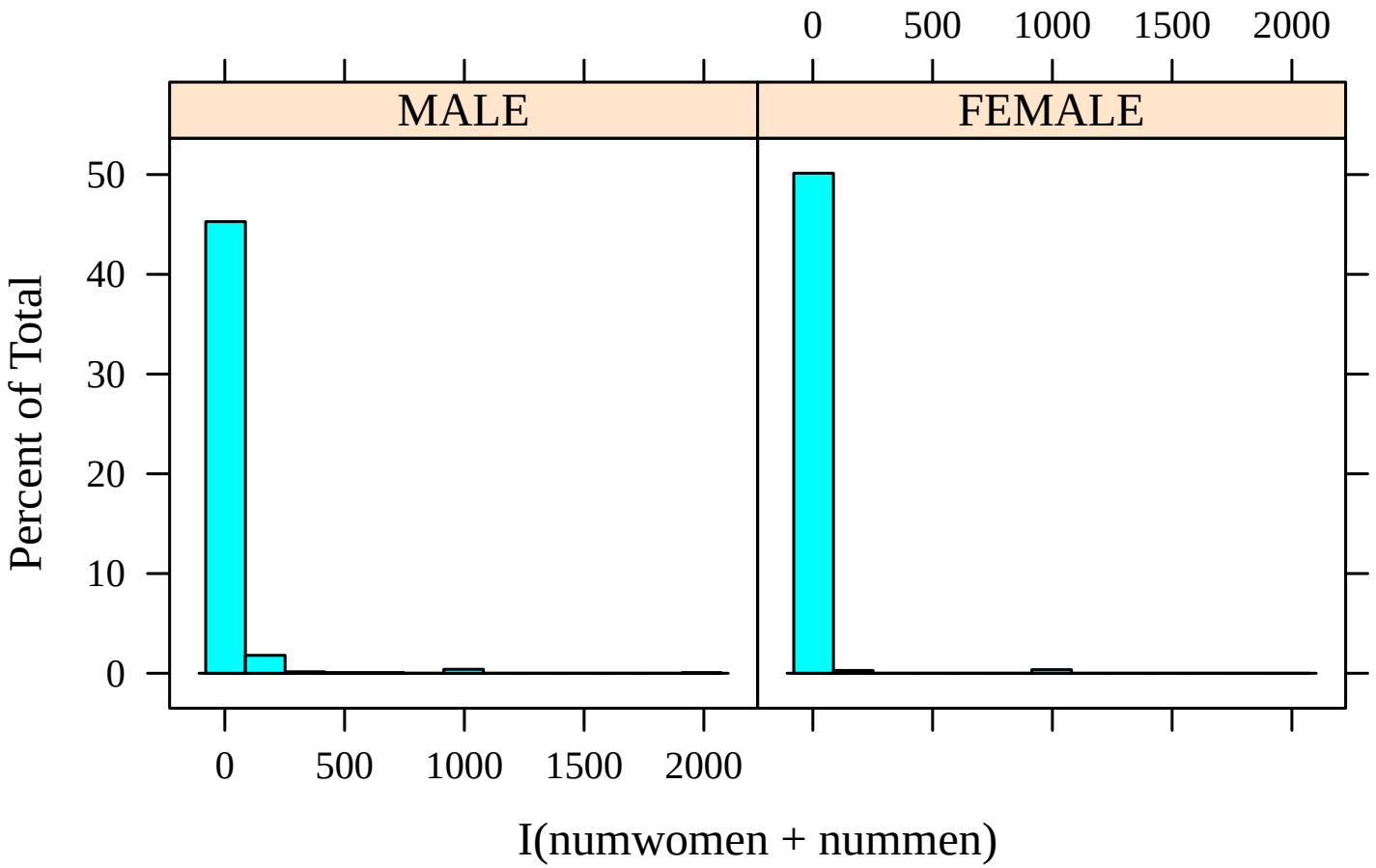
```
xyplot( hrs1 ~ age | sex , data=dat)
```



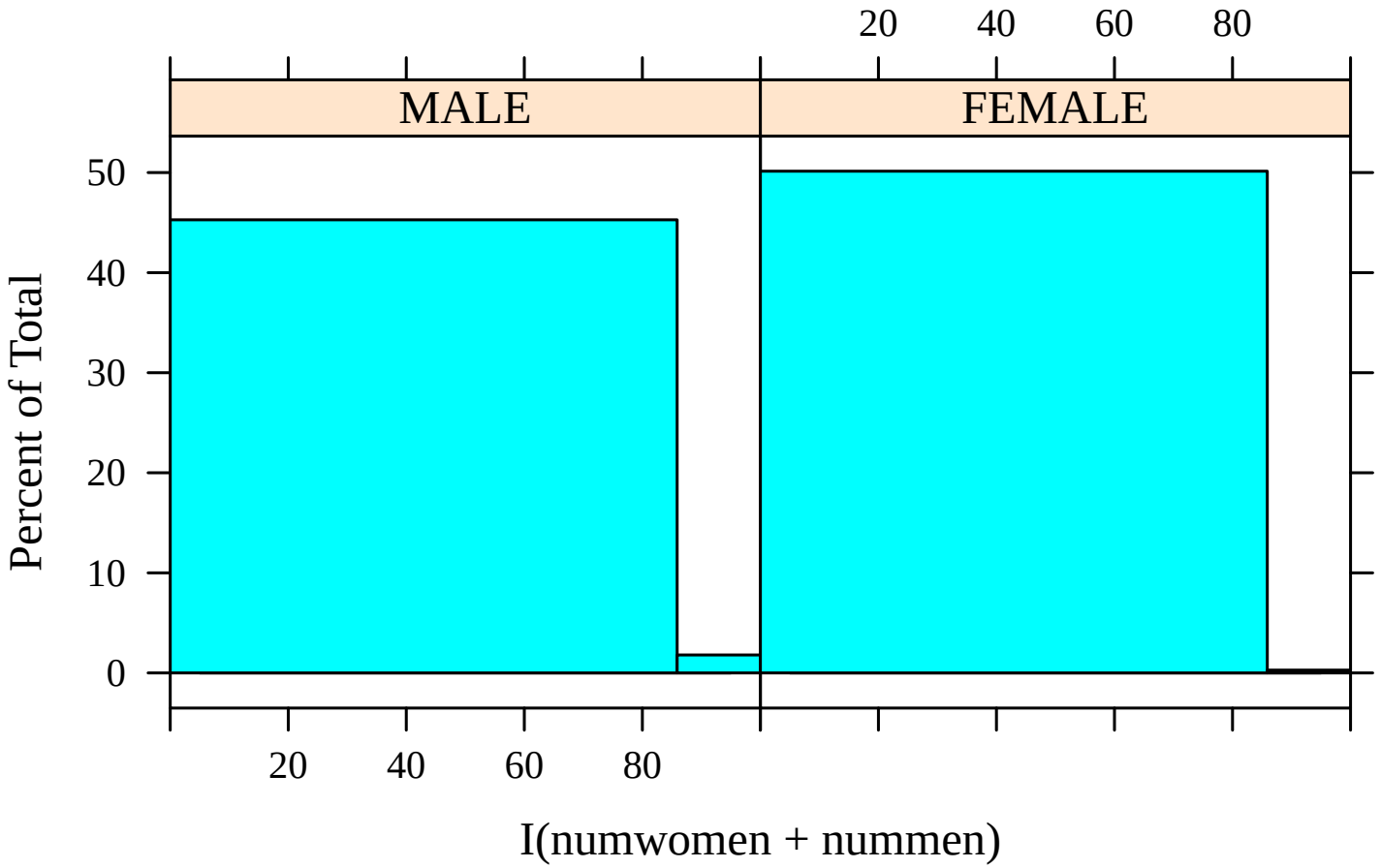
```
histogram(~income06 | sex, data=dat)
```



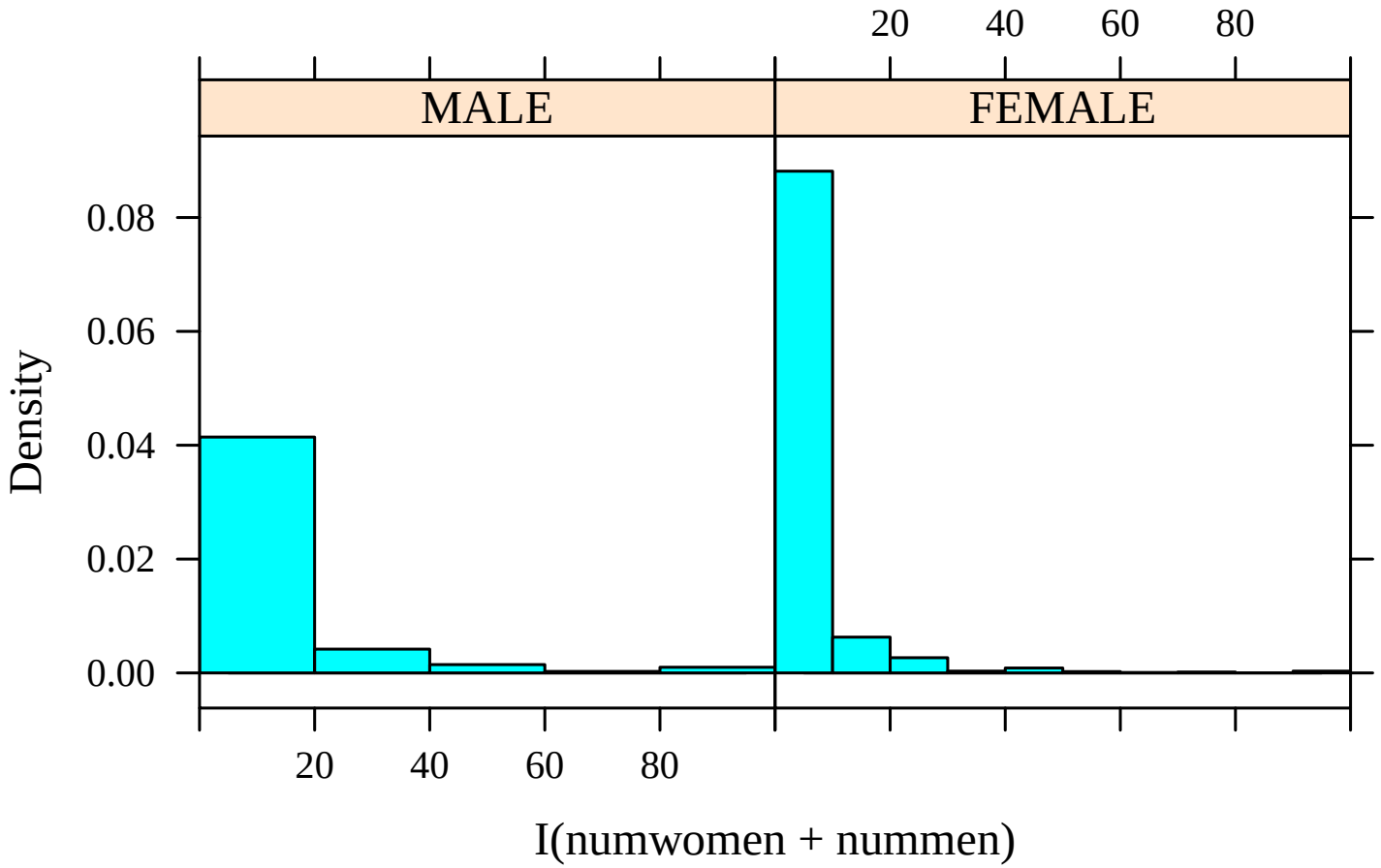
```
histogram(~I(numwomen + nummen) | sex, data = dat)
```



```
histogram(~I(numwomen+nummen) | sex, data=dat, xlim=c
    (0,100))
```



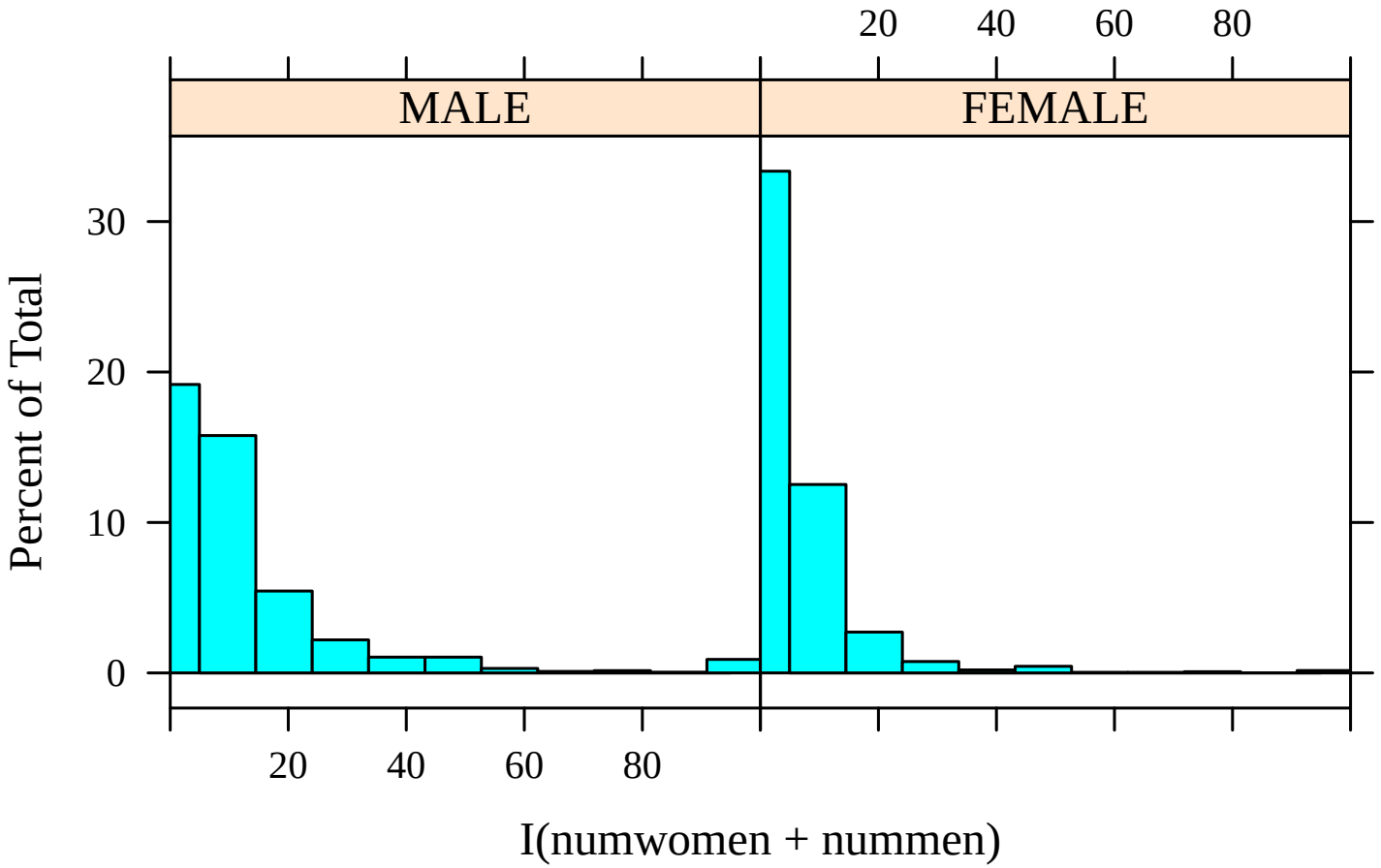
```
histogram(~I(numwomen+nummen) | sex, data=dat, xlim=c
    (0,100), breaks=100)
```



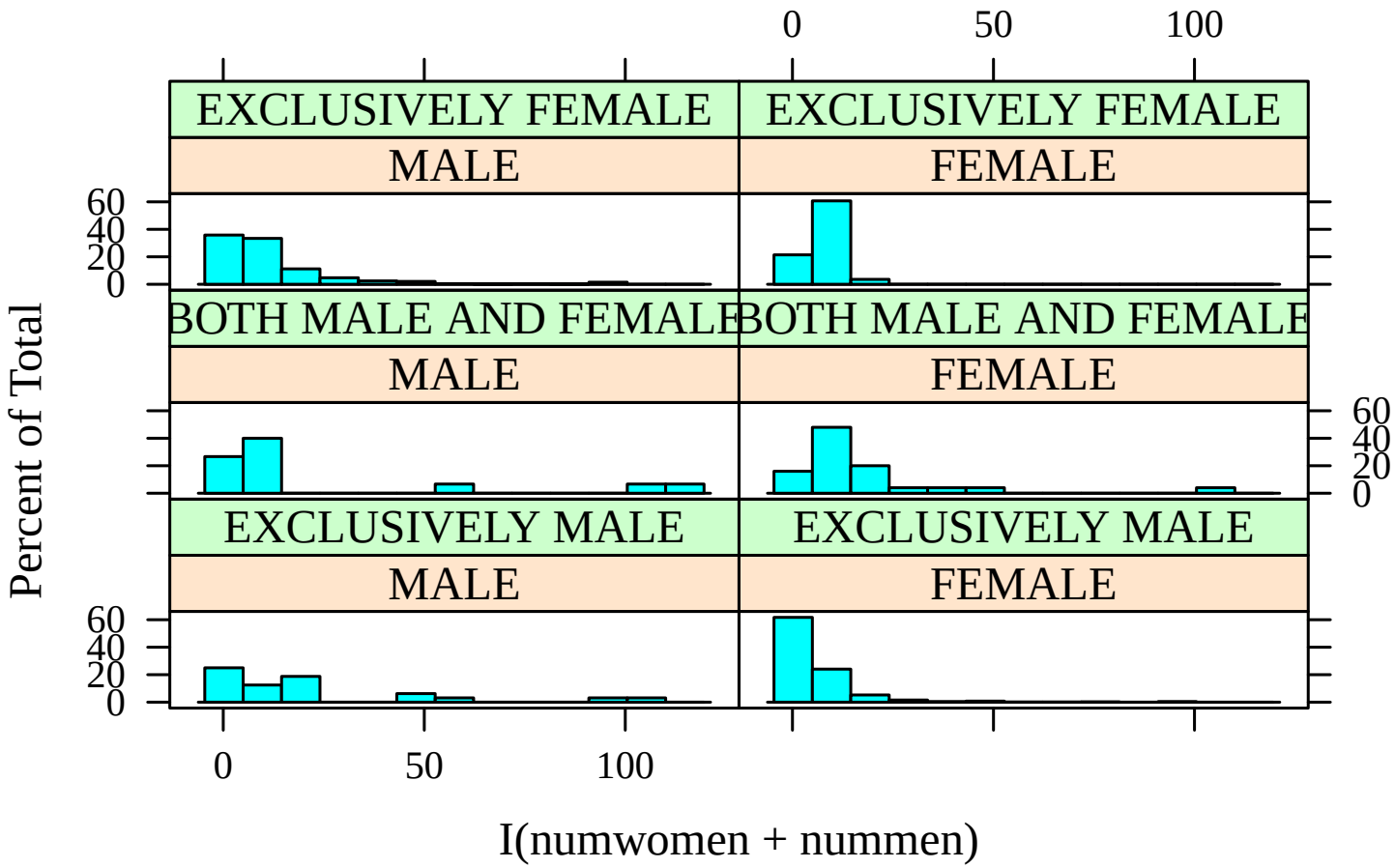
Appears we just need to cut some extreme values.

```
dat$numwomen[dat$numwomen > 100] <- NA
dat$nummen[dat$nummen > 100] <- NA
```

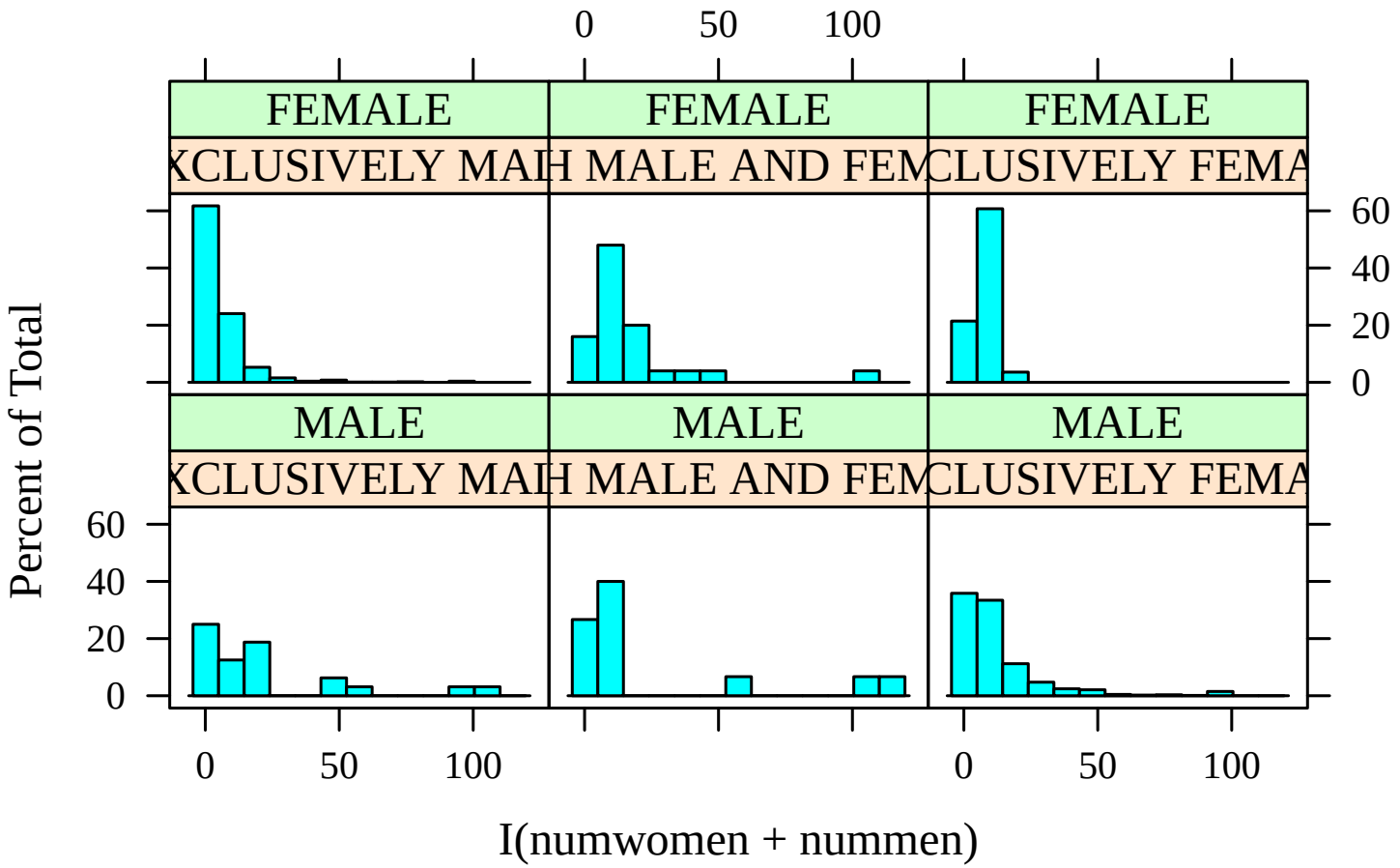
```
histogram(~I(numwomen+nummen) | sex, data=dat, xlim=c
    (0,100))
```



```
histogram(~I(numwomen+nummen) | sex * sexsex5, data=dat)
```



```
histogram(~I(numwomen+nummen) | sexsex5 * sex , data=dat)
```



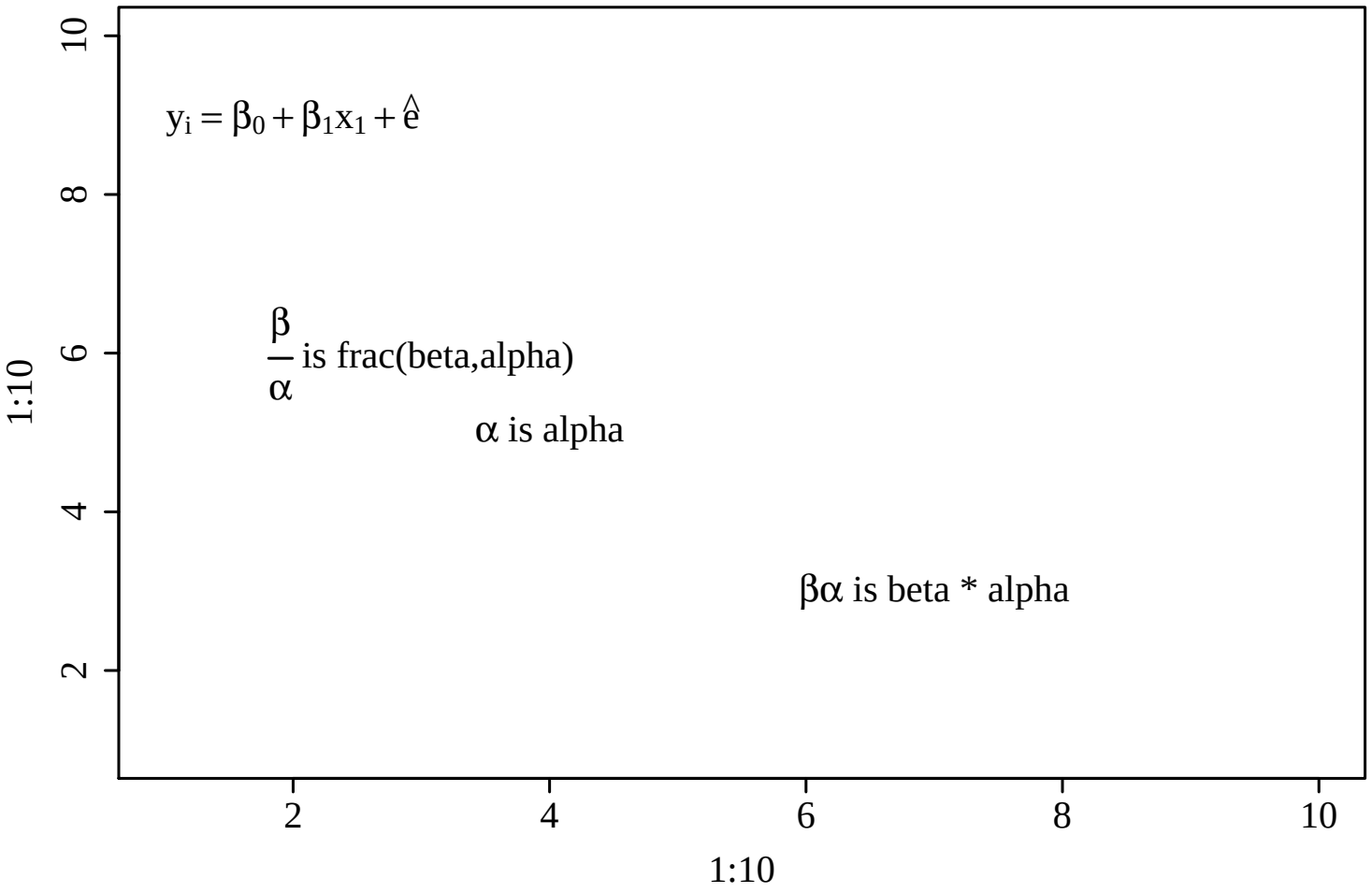
The plotmath Framework

- R has a "LaTeXlike" math markup called "plotmath"
- Create an R "expression", then when plotted, math will appear!
- read `?plotmath`
- run `example(plotmath)`

Simple example

```
plot(1:10, 1:10, type="n")
text(4, 5, expression(paste( alpha  ," is alpha")))
text(7, 3, expression(paste( beta * alpha, " is beta *
    alpha")))
text(3, 6, expression(paste(frac(beta, alpha), " is frac(
    beta,alpha)")))
text(2,9, expression(paste(y[i]==beta[0]+beta[1]*x[1]+hat(
    e))))
```

Simple example



A Few plotmath Tips

- Two Equal Signs (== outputs =)
- Want * to show? Type %*%
- Want $\cdot$? Type cdot

The Only Really Big Gotcha Is...

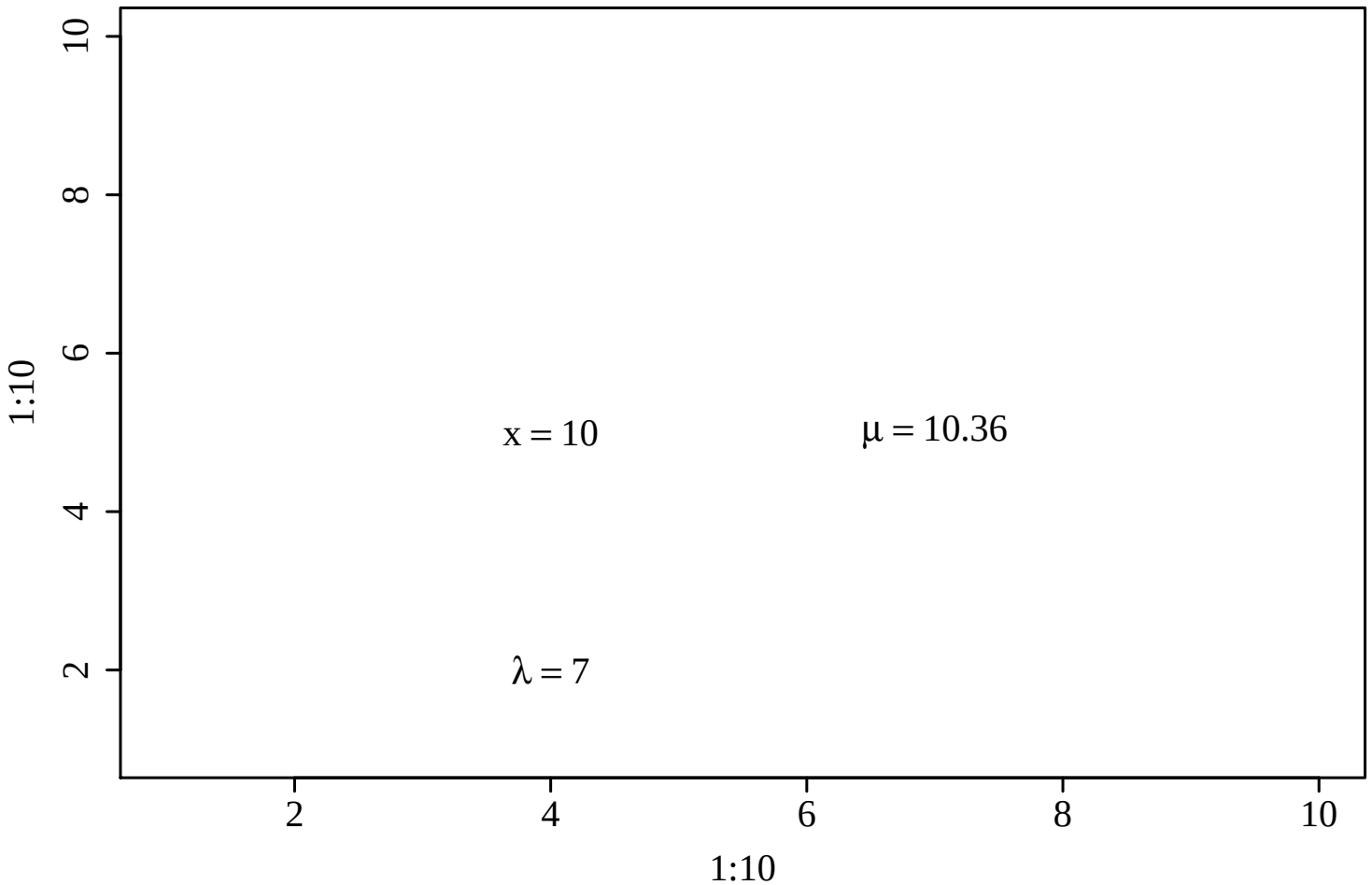
- Blending calculations and math.
- This does not work:

```
x <- 10
text(4, 5, expression(paste("x is ", x)))
```

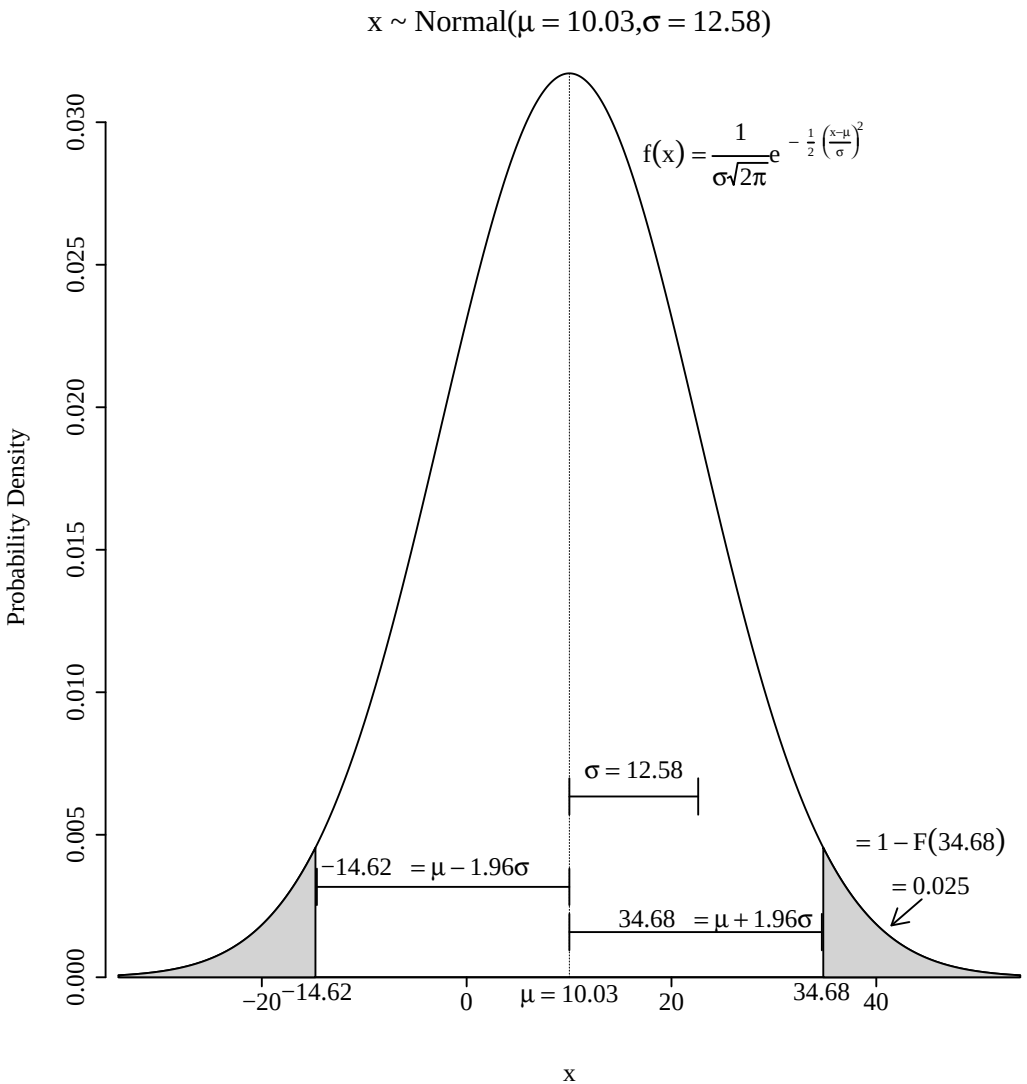
- Instead, a more elaborate procedure is required.
- One method is bquote

bquote example usage

```
x <- 10
plot( 1:10, 1:10, type="n")
anexpression <-   bquote(x==.(x))
text( 4, 5, anexpression)
mnorm <- round(mean(rnorm(1000, m=10, sd=20)), 2 )
text( 7,5, bquote(mu==.(mnorm)))
text(4, 2, bquote(lambda==.(round(mean(rpois(100,lambda=7)
    )))))
```


1:10
2
4
6
8
10
x = 10
μ = 10.36
λ = 7
2
4
6
8
10
1:10

I've Been Proposing This as a plotmath Example

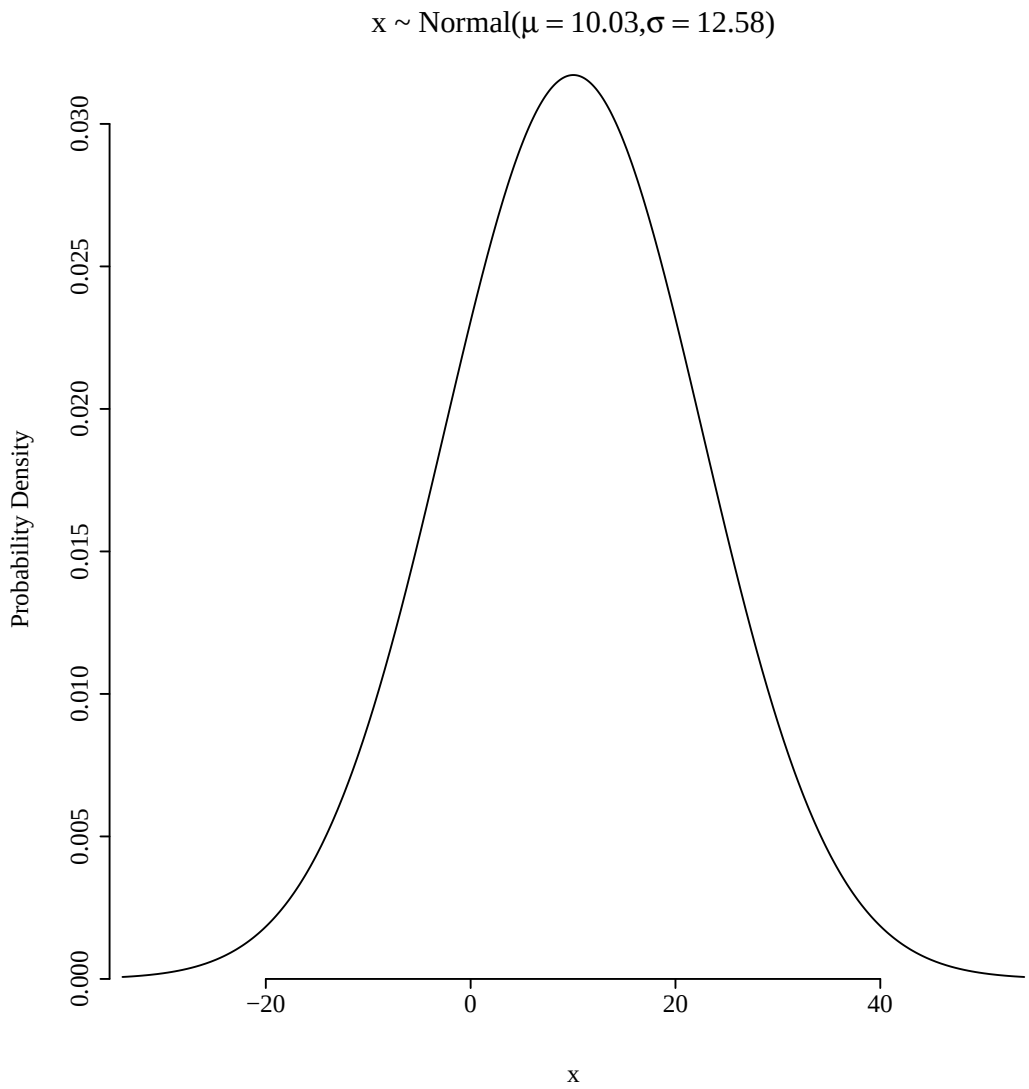


Set Parameters, Create x, y, Make Title

```
###Set mu and sigma at your pleasure:
mu <- 10.03
sigma <- 12.5786786
myx <- seq( mu - 3.5*sigma ,  mu+ 3.5*sigma , length.out=500)
myDensity <- dnorm(myx,mean=mu,sd=sigma)
# It is challenging to combine plotmath with values of mu
    and sigma in one expression.
# Either bquote or substitute can be used.  First use
    bquote:

myTitle1 <- bquote (paste("x ~ Normal(", mu==.(round(mu,2))
    , ',', sigma==.(round(sigma,2)),")") )

### Using substitute:
### myTitle1 <-  substitute( "x ~ Normal" ~~ group( "(",
    list(mu==mu1, sigma^2==sigma2)#, ")") ,  list(mu1=round(
    mu,2), sigma2=round(sigma^2,2)))

### Or combine the two:
### t1 <- substitute ( mu == a ,  list (a = mu))
### t2 <- substitute ( sigma == a, list(a = round(sigma,2)))
### myTitle1 <- bquote(paste("x ~ Normal(", .(t1),",", .(t2
```

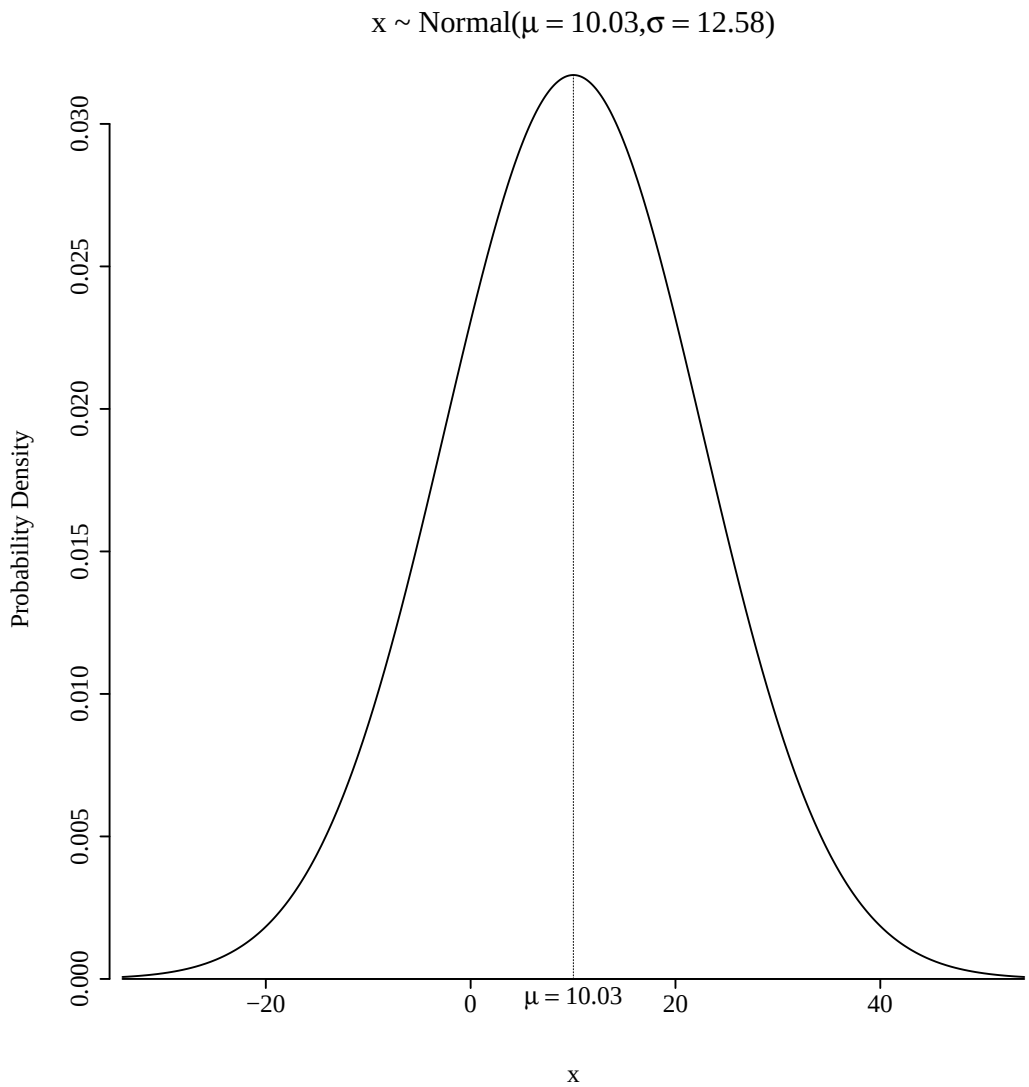
Create Plot and axes

```
### xpd needed to allow writing outside strict box of graph
par(xpd=TRUE, ps=10)
plot(myx, myDensity, type="l", xlab="x", ylab="Probability
    Density ", main=myTitle1, axes=FALSE)
axis(2, pos= mu - 3.6*sigma)
axis(1, pos=0)
```


x ~ Normal(μ = 10.03,σ = 12.58)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
−20
0
20
40
x

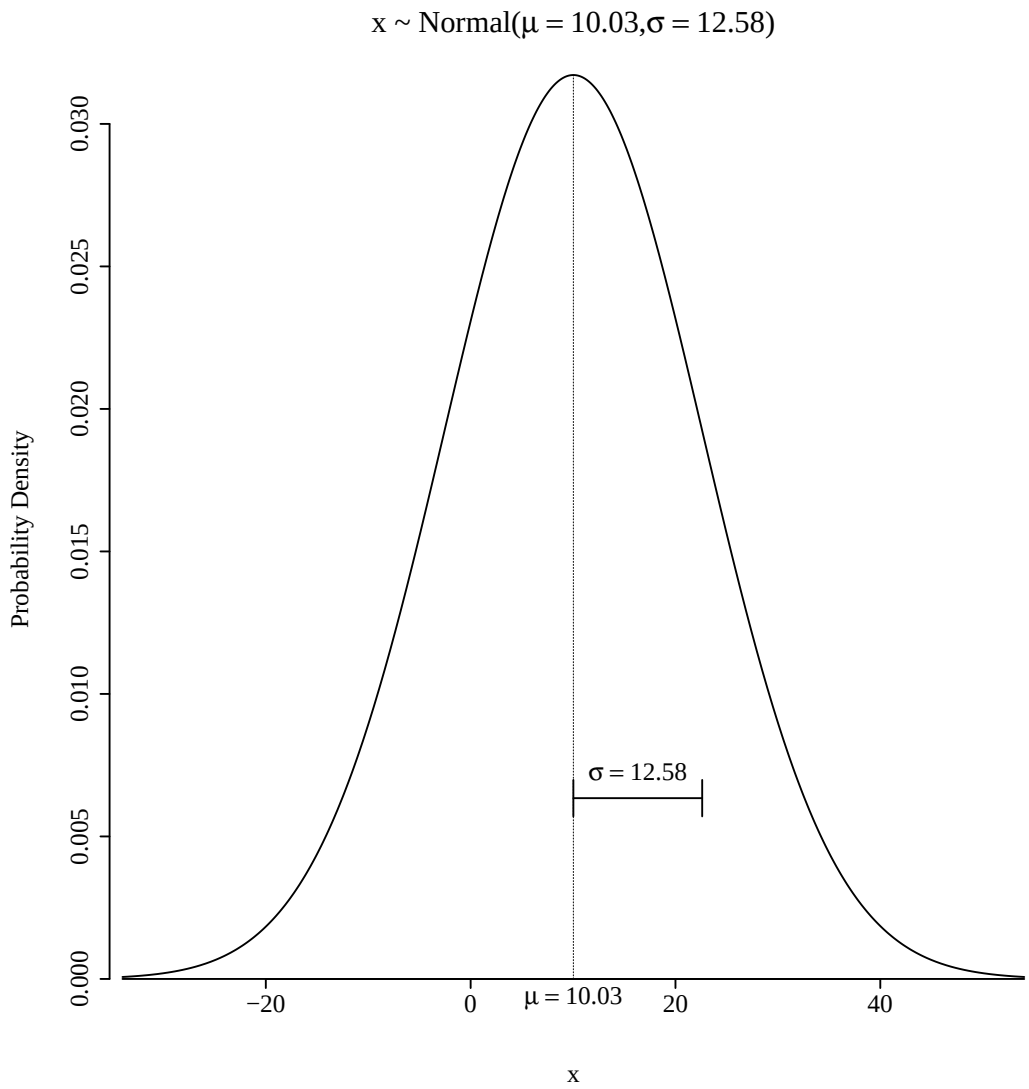
Label Lower Axis, Draw Line Across Range

```
lines(c(myx[1],myx[length(myx)]),c(0,0)) ### closes off
    axes
# bquote creates an expression that text plotters can use
t1 <-  bquote( mu==.(mu))
# Find a value of myx that is "very close to" mu
centerX <- max(which (myx <= mu))
# plot light vertical line under peak of density
lines( c(mu, mu), c(0, myDensity[centerX]), lty= 14, lwd=.2
    )
# label the mean in the bottom margin
mtext(bquote( mu==.(mu)), 1, at=mu, line=-1)
```


x ~ Normal(μ = 10.03,σ = 12.58)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
−20
0
μ = 10.03
20
40
x

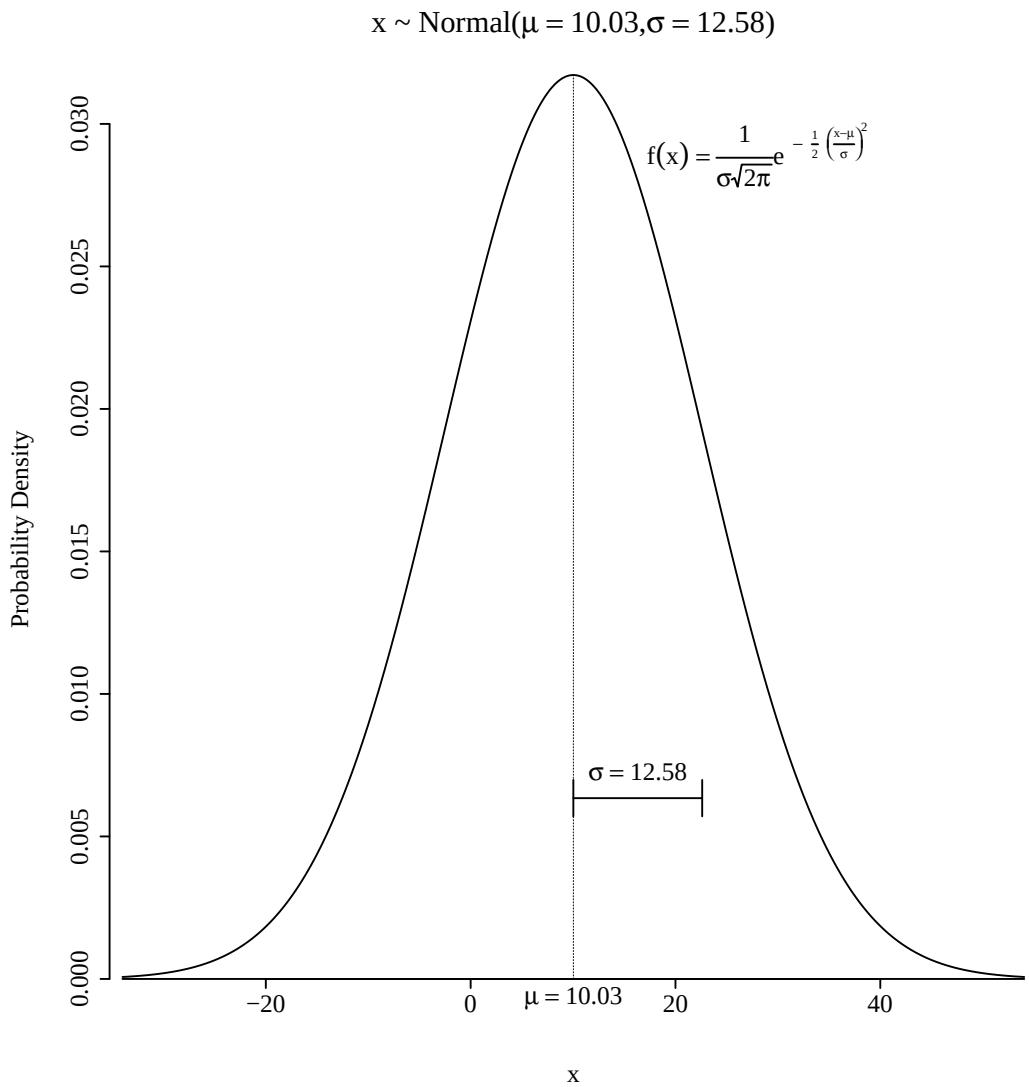
Mark Interval from mean to σ

```
### find position 20% "up" vertically, to use for arrow
    coordinate
ss = 0.2 * max(myDensity)
# Insert interval to represent width of one sigma
arrows( x0=mu, y0= ss, x1=mu+sigma, y1=ss, code=3, angle
    =90, length=0.1)
### Write the value of sigma above that interval
t2 <-  bquote( sigma==.(round(sigma,2)))
text( mu+0.5*sigma, 1.15*ss, t2)
```


x ~ Normal(μ = 10.03,σ = 12.58)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
σ = 12.58
−20
0
μ = 10.03
20
40
x

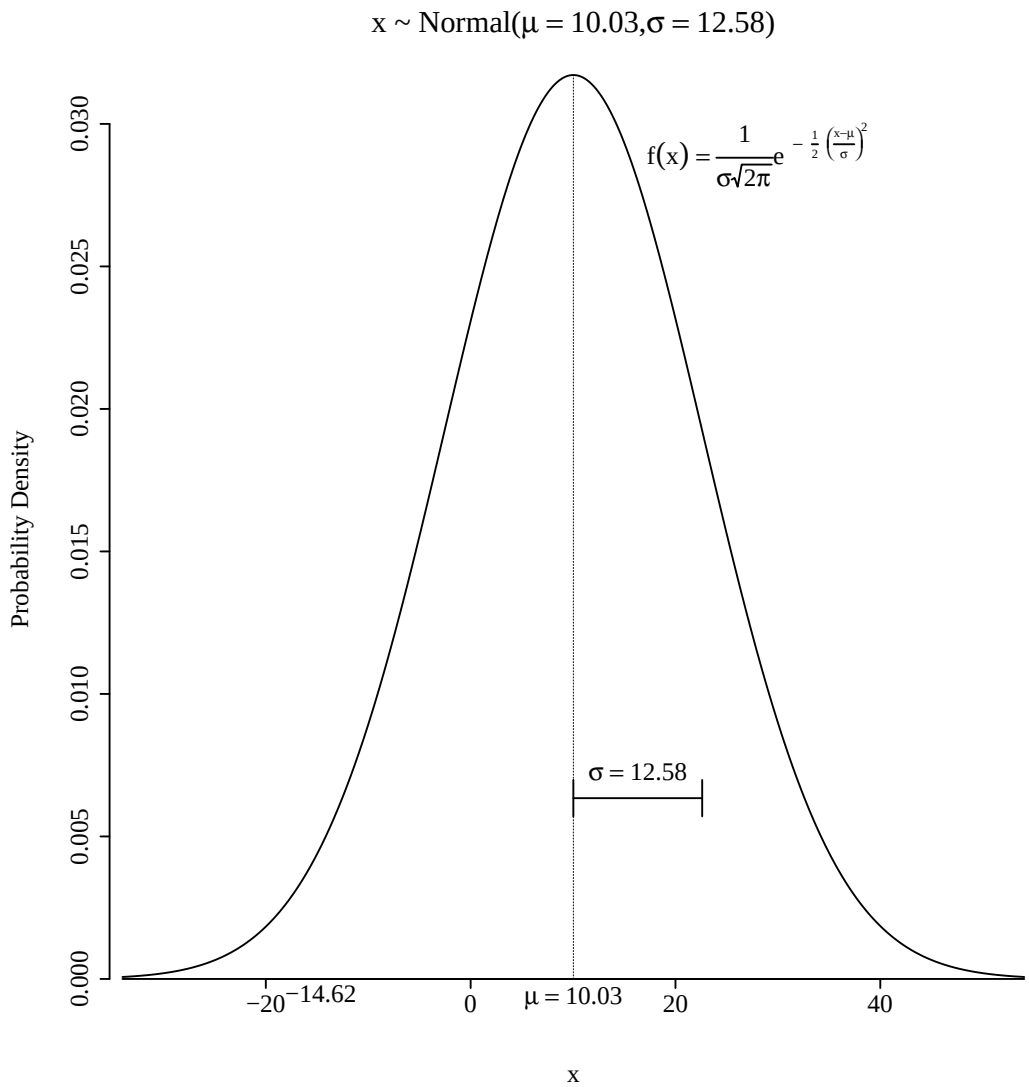
Create formula for Normal Distribution

```
### Create a formula for the Normal
normalFormula <- expression (f(x)==frac (1, sigma* sqrt(2*
    pi)) * e^{~~ - ~~ frac(1,2)~~ bgroup("(", frac(x-mu,
    sigma),")")^2})
# Draw the Normal formula
text ( mu + 0.5*sigma, max(myDensity)- 0.10 * max(myDensity
    ),  normalFormula, pos=4)
```


x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
σ = 12.58
−20
0
μ = 10.03
20
40
x

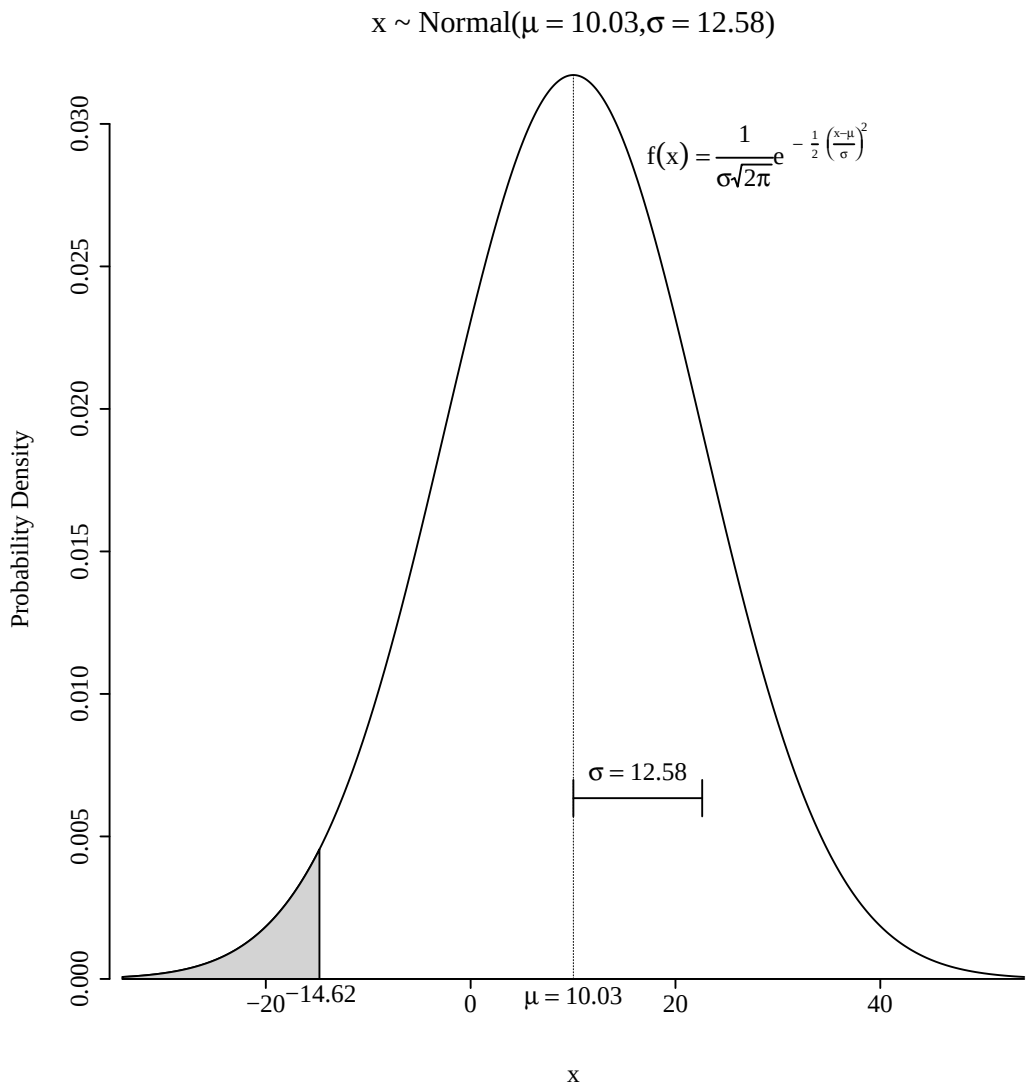
Write Left Side Critical Value Below Axis

```
### Theory says we should have 2.5% of the area to the left
      of: -1.96 * sigma.
### Find the X coordinate of that "critical value"
criticalValue <- mu -1.96 * sigma
criticalValue <- round(criticalValue, 2)
### Then grab all myx values that are "to the left" of that
      critical value.
specialX <- myx[myx <= criticalValue]
### mark the critical value in the graph
text ( criticalValue, 0 ,
       label=criticalValue, pos=1)
```


x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
σ = 12.58
−20
−14.62
0
μ = 10.03
20
40
x

Use polygon to Shade the Area Under Curve on Left

```
### Take sequence parallel to values of myx inside critical
      region
specialY <- myDensity[myx < criticalValue]
#  Polygon makes a nice shaded illustration
polygon(c(specialX[1], specialX, specialX[length(specialX )
    ]), c(0, specialY, 0), density=c(-110),col="lightgray"
    )
```

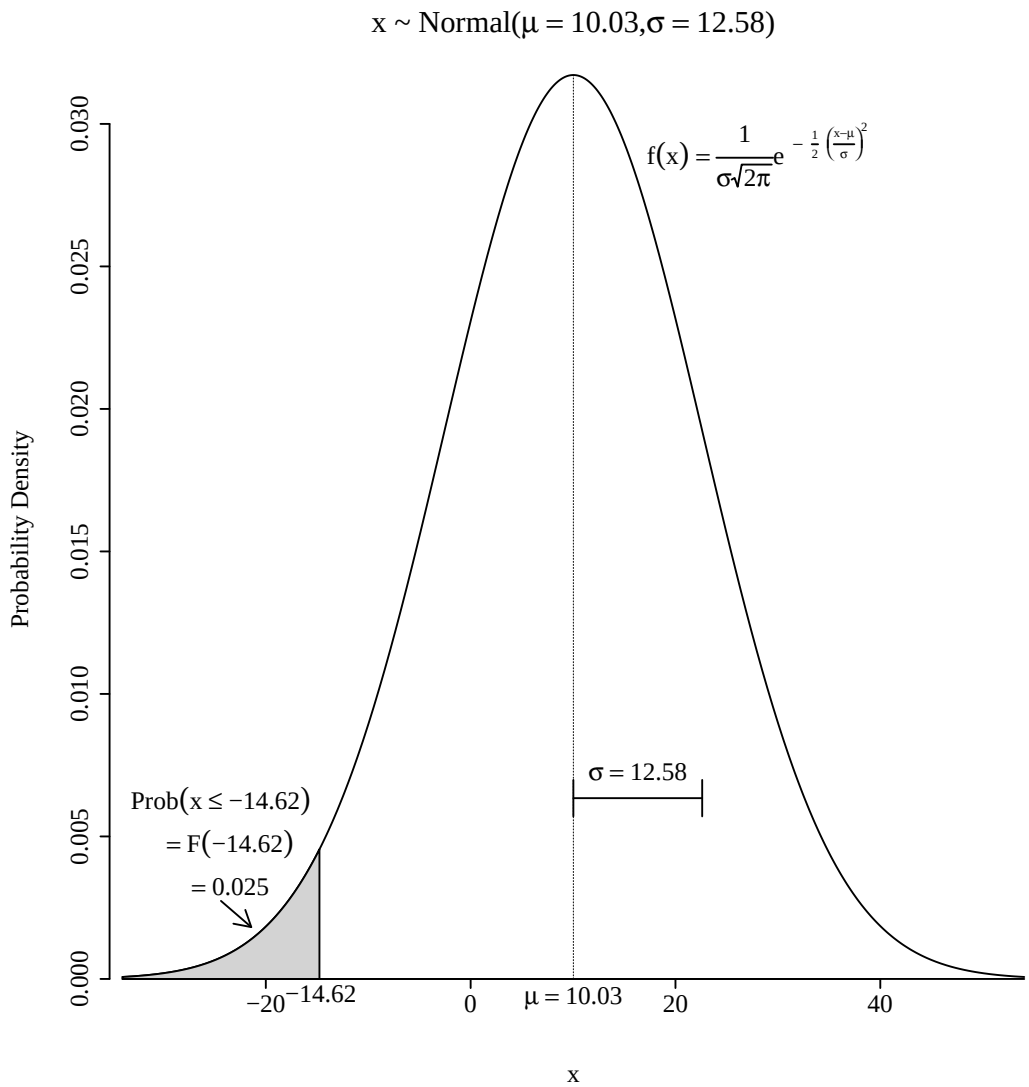

x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
σ = 12.58
−20
−14.62
0
μ = 10.03
20
40
x

Make Calculations about Area on Left Side

```
### I want to insert annotation about area on left side.
shadedArea <- round(pnorm(mu - 1.96 * sigma, mean=mu, sd=
    sigma),4)
al1 <- bquote(Prob(x <= .(criticalValue)))
al2 <- bquote(phantom(0)==F( .(criticalValue) ))
al3 <- bquote(phantom(0)==.(round(shadedArea,3)))
### Hard to position this text "just right"
### Have tried many ideas, this may be least bad.
### Get center position in shaded area
medX <- median(specialX)
### density at that center point:
denAtMedX <- myDensity[indexMed <- max(which(specialX <
    medX))]
```

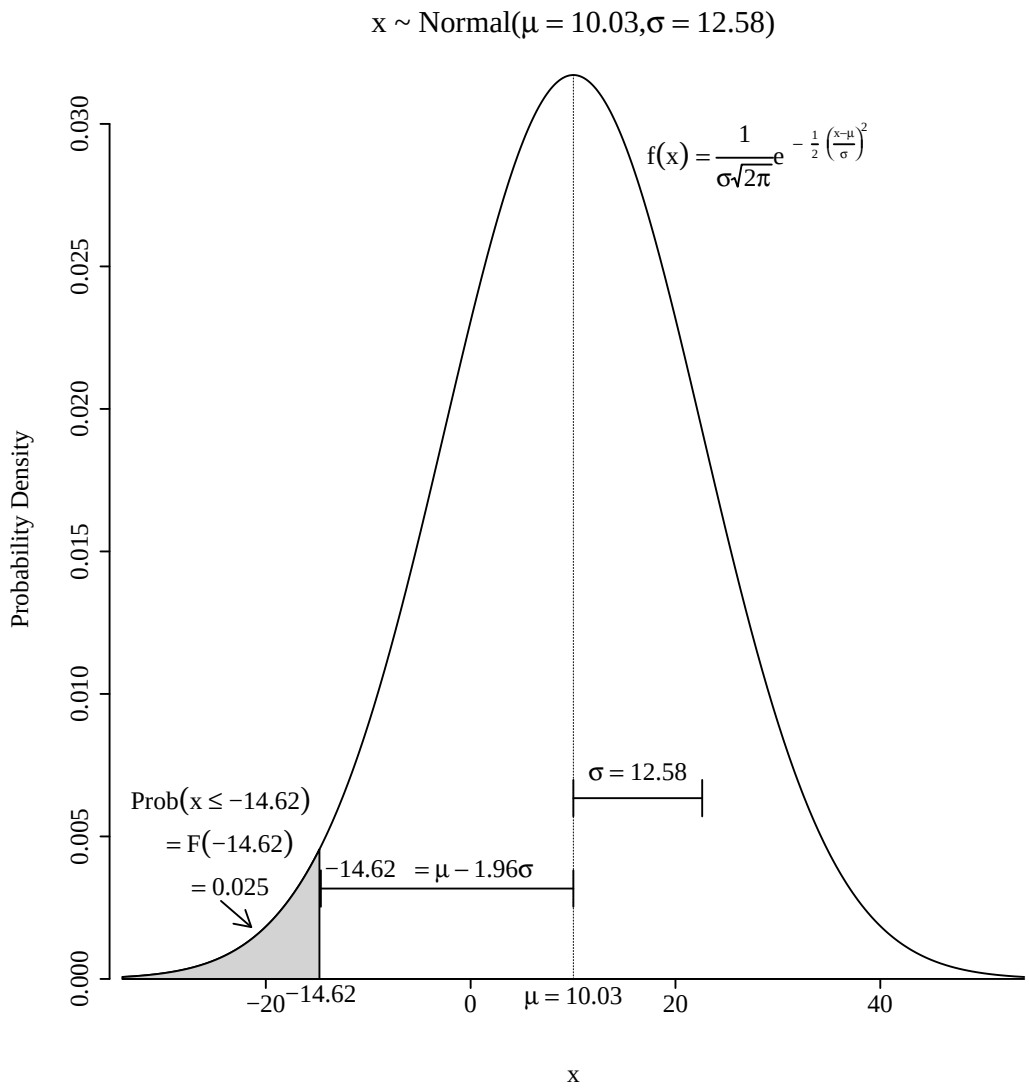
Write Annotation Above Shaded Area on Left

```
text(medX, denAtMedX+0.0055, labels = al1)
text(medX, denAtMedX+0.004, labels = al2)
text(medX, denAtMedX+0.0025, labels = al3)
### point from text toward shaded area
arrows( x0 = medX, y0 = myDensity[indexMed]+0.002,
        x1 = mu-2.5 *sigma, y1 = 0.40*myDensity[length(
            specialX)], length = 0.1)
```


x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
σ = 12.58
Prob(x ≤ −14.62)
= F(−14.62)
= 0.025
−20
−14.62
0
μ = 10.03
20
40
x

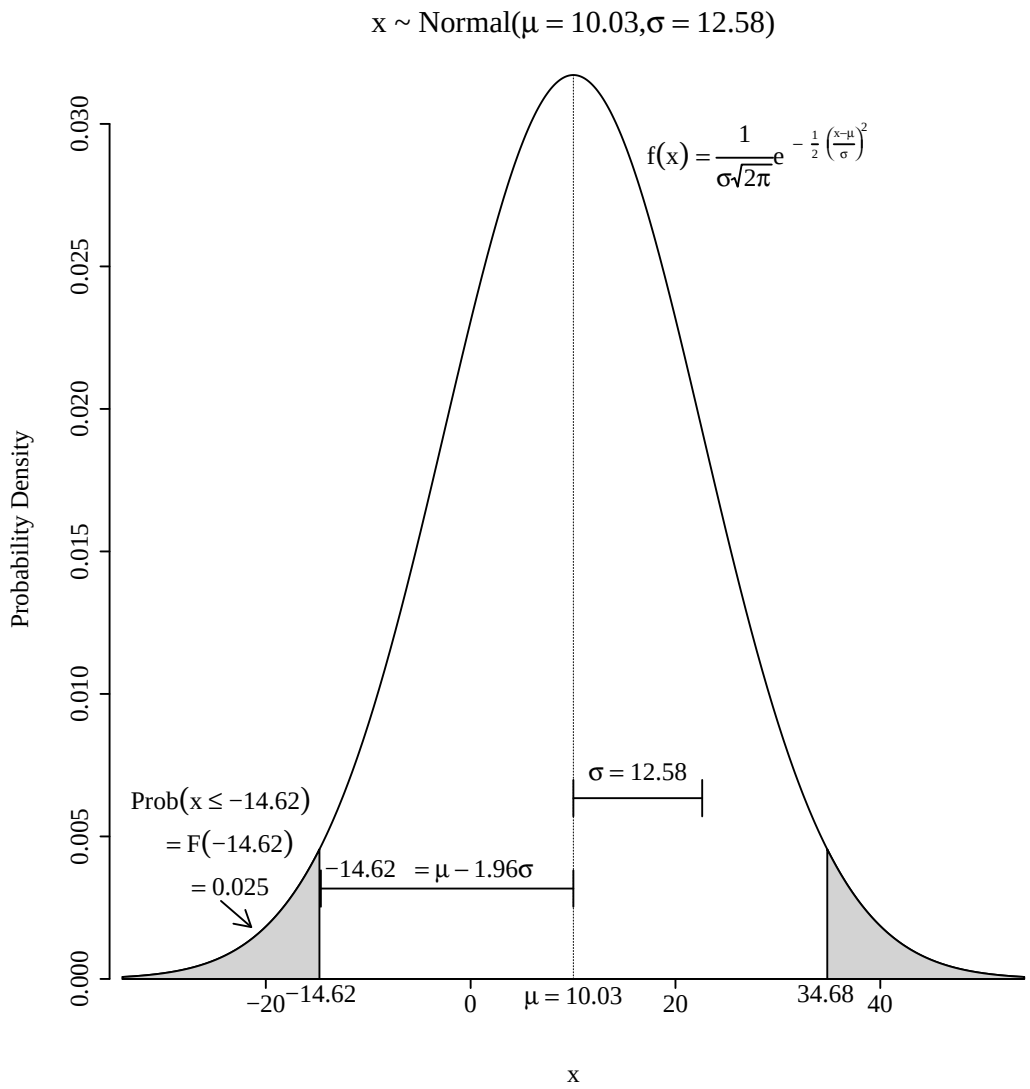
Insert arrow and text from mean to -1.96σ

```
ss <- 0.1 * max(myDensity)
### Mark interval from mu to mu-1.96*sigma
arrows( x0 = mu, y0 = ss, x1 = mu-1.96*sigma, y1 = ss, code
     = 3, angle = 90, length = 0.1)
### Put text above interval
text( mu - 2.0*sigma,1.15*ss, bquote(paste(.(criticalValue)
     , phantom(1) == mu-1.96 * sigma, sep="")), pos=4)
```


x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Prob(x ≤ −14.62)
= F(−14.62)
= 0.025
−14.62 = μ − 1.96σ
σ = 12.58
−20
−14.62
0
μ = 10.03
20
40
x

Shade right side critical region

```
criticalValue <- mu +1.96 * sigma
criticalValue <- round(criticalValue, 2)
### Then grab all myx values that are "to the left" of that
    critical value.
specialX <- myx[myx ≥ criticalValue]
### mark the critical value in the graph
text ( criticalValue, 0 , label= paste(criticalValue), pos
    =1)
### Take sequence parallel to values of myx inside critical
    region
specialY <- myDensity[myx > criticalValue]
#  Polygon makes a nice shaded illustration
polygon(c(specialX[1], specialX, specialX[length(specialX )
    ]), c(0, specialY, 0), density=c(-110), col = "
    lightgray" )
```

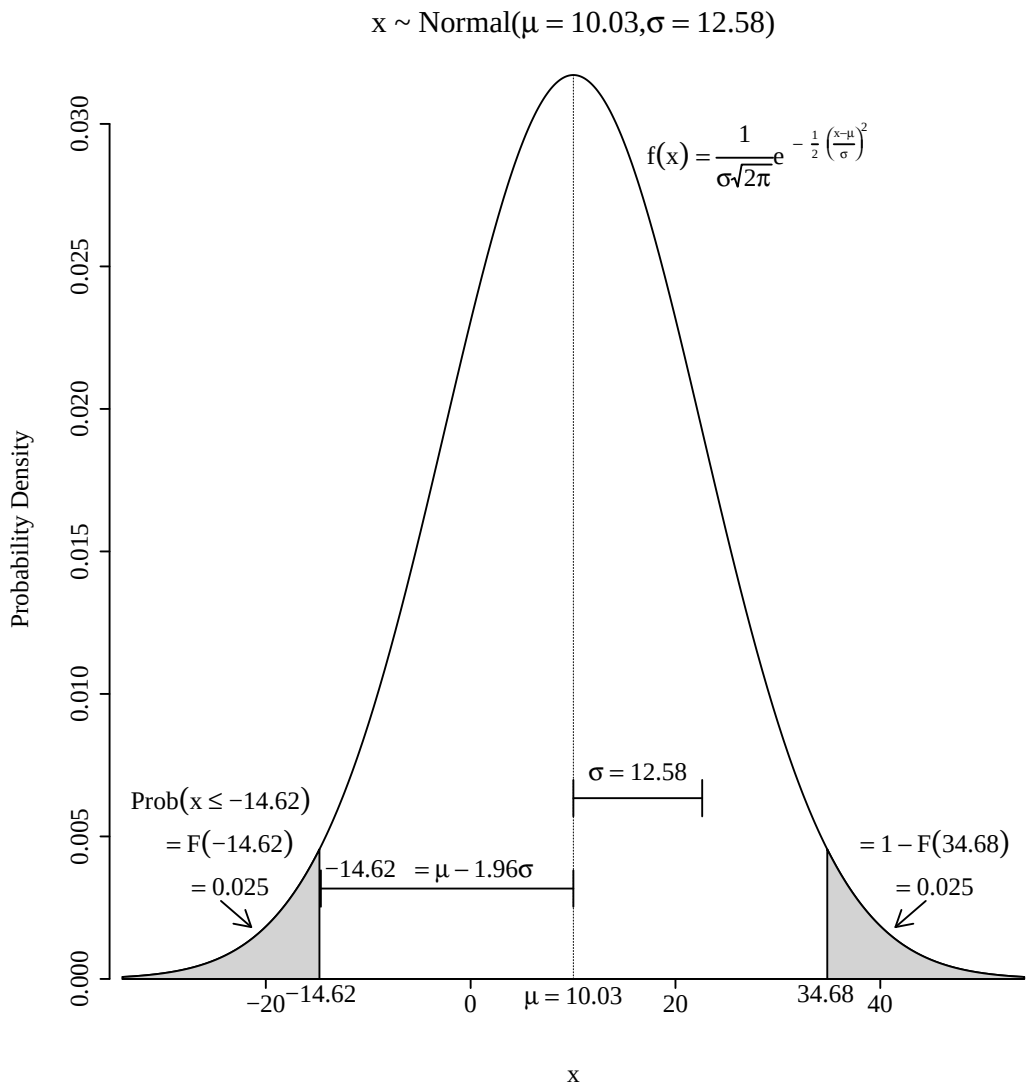

x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Prob(x ≤ −14.62)
= F(−14.62)
= 0.025
−14.62 = μ − 1.96σ
σ = 12.58
−20
−14.62
0
μ = 10.03
20
34.68
40
x

Gather data on right side shaded area

```
shadedArea <- round(pnorm(mu + 1.96 * sigma, mean = mu, sd
    = sigma, lower.tail = F),4)
### Insert simpler comment on right side.

al2 <- bquote(phantom(0)==1 - F( .(criticalValue) ))
al3 <- bquote(phantom(0)==.(round(shadedArea,3)))
```

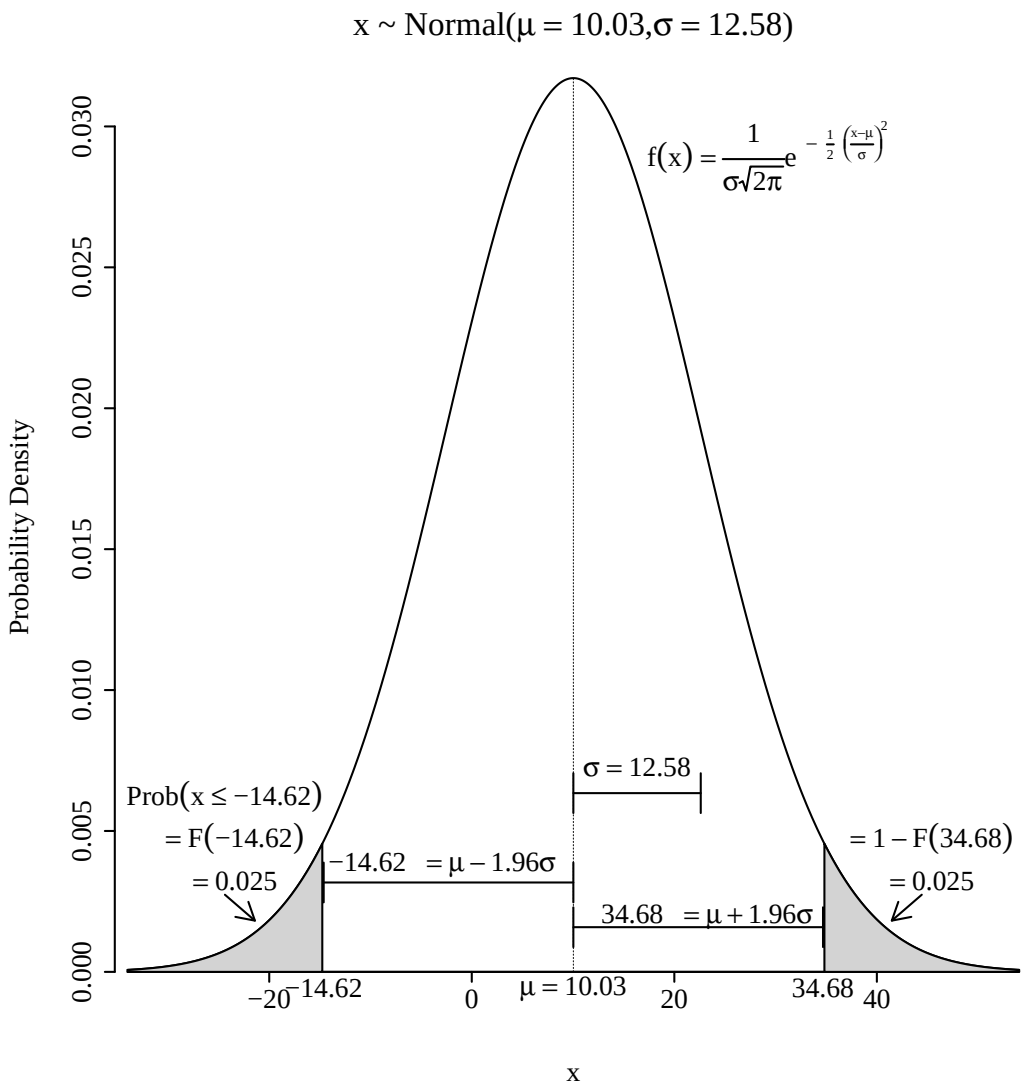
Annotate right shaded area

```
### Hard to position this text "just right"
### Have tried many ideas, this may be least bad.
### Get center position in shaded area
medX <- median(specialX)
### density at that center point:
denAtMedX <- myDensity[indexMed <- max(which(specialX <
    medX))]
text(medX, denAtMedX+0.004, labels = al2)
text(medX, denAtMedX+0.0025, labels = al3)
### point from text toward shaded area
arrows(x0 = medX, y0 = myDensity[indexMed]+0.002, x1 = mu+2
    .5 *sigma, y1 = 0.40*myDensity[length(specialX)] ,
    length = 0.1)
```


x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Prob(x ≤ −14.62)
= F(−14.62)
= 0.025
−14.62 = μ − 1.96σ
σ = 12.58
= 1 − F(34.68)
= 0.025
−20
−14.62
0
μ = 10.03
20
34.68
40
x

Insert arrow from mean to 1.96σ

```
ss <- 0.05 * max(myDensity)
### Mark interval from mu to mu+1.96*sigma
arrows(x0 = mu, y0 = ss, x1 = mu+1.96*sigma, y1 = ss, code
    = 3, angle = 90, length = 0.1)
### Put text above interval
text(mu + 1.96*sigma,1.15*ss, bquote(paste(.(criticalValue)
    , phantom(1)<=mu+1.96 * sigma,sep="")), pos=2)
```


x ~ Normal(μ = 10.03,σ = 12.58)
f(x) = 1/(σ√2π) e^(−½((x−μ)/σ)²)
Probability Density
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Prob(x ≤ −14.62)
= F(−14.62)
= 0.025
σ = 12.58
−14.62 = μ − 1.96σ
34.68 = μ + 1.96σ
= 1 − F(34.68)
= 0.025
−20
−14.62
0
μ = 10.03
20
34.68
40
x