Statistical Computing 1
Stat 590

Chapter 06
R programming

Christian Gunning and Erik Erhardt

Department of Mathematics and Statistics
MSC01 1115
1 University of New Mexico
Albuquerque, New Mexico, 87131-0001
Office: MSLC 312
erike@stat.unm.edu

Fall 2015

Part 1, Outline

Flow control: Looping

`http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Control-statements`

Functions: How to write your own

`http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Writing-your-own-functions`

Plotting: A brief intro to **lattice**.

`http://lattice.r-forge.r-project.org/`

Debugging: How to identify and fix problems.

`http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR/debug.shtml`

Syntax Reminder

```
# Define an object
# Use parens () for grouping & order of operations
my.vector <- (1:5) / 10

# Use brackets [] to index object
my.vector[-1:-2]
## [1] 0.3 0.4 0.5
# Function calls also use parens
my.sum <- sum(my.vector)

# Normally a new-line separates expressions. We can also use ;
# Try to avoid this.
aa <- 1:5;  bb <- 5:1; sum(aa*bb)
## [1] 35
```

Flow Control

if

```
# Inspect
my.sum
## [1] 1.5
# Each line is a single expression.
# Use braces {} to group multiple expressions together.
if (my.sum < 10) {
  my.vector <- my.vector * 10
  my.sum <- sum(my.vector)
}

# Has anything changed?
my.sum
## [1] 15
```

if/else

```
# Inspect
my.sum
## [1] 15
# Each line is a single expression.
# Use braces {} to group multiple expressions together.
if (my.sum < 10) {
  my.vector <- my.vector * 10
  my.sum <- sum(my.vector)
} else {
  my.sum <- NA
}

# Has anything changed?
my.sum
## [1] NA
```

for loop

When is a **for** loop useful?

```
my.sum <- sum(my.vector)

# bad use of for loop
my.sum.loop <- 0
# add up element-by-element
for (ii in my.vector) {
  my.sum.loop <- my.sum.loop + ii
}

# compare the results
my.sum == my.sum.loop
## [1] TRUE
```

for loops are slow. Use a vectorized R function when possible.

for loop

A **for** loop is required when each iteration depends on the previous iteration:

```
N <- 10

# compute Fibonacci numbers
# good use of for loop
my.fib <- c(0,1)

for (ii in 2:N) {
  # we use previous iteration: ii - 1
  my.fib[ii + 1] <- my.fib[ii] + my.fib[ii - 1]
}

# Examine results
my.fib
##  [1]  0  1  1  2  3  5  8 13 21 34 55
```

Writing Functions

Functions are magic!

- Anything you do in R can be turned into a function.
- Functions will make your life easier.
- Use comments in function code to explain behavior. You'll thank yourself later.

Write a function

```
# let's turn the previous slide into a function:
mk.fib <- function(N, seed = c(0, 1)) {
  # takes a integer of length 1
  # and a seed of 2 fibonacci numbers to start with
  # return a length N+1 vector of fibonacci numbers
  fib <- rep(NA, N + 1) # pre-initialize vector
  fib[1:2] <- seed
  for (ii in 2:N) {
    # we use previous iteration: ii - 1
    fib[ii + 1] <- fib[ii] + fib[ii - 1]
  }
  # Return the results
  return(fib)
}
# Now use it.  Note that seed has a default value
mk.fib(5)
```

```
## [1] 0 1 1 2 3 5
```

```
mk.fib(10)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34 55
```

Function arguments

```
# the seed argument has a default value
# that is used unless another is specified
mk.fib(5)
```

```
## [1] 0 1 1 2 3 5
```

```
# We can specify the seed if desired
mk.fib(5, seed = c(5, 8))
```

```
## [1]  5  8 13 21 34 55
```

```
# If we specify all arguments by name,
# then order doesn't matter
mk.fib(seed = c(5, 8), N = 5)
```

```
## [1]  5  8 13 21 34 55
```

Debugging

Read the error/warning message!

```
my.vector <- (1:3) / 1e6
# A common mistake
for ( ii in 1:my.vector ) { print(ii) }
## Warning in 1:my.vector:  numerical expression has 3 elements:
only the first used
## [1] 1
# What's going on?
length(my.vector)
## [1] 3
1:length(my.vector)
## [1] 1 2 3
```

Understanding error messages takes practice.

How are these different?

```
# We want either
for ( ii in 1:length(my.vector) ) { print(ii) }
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
# or
for ( ii in my.vector ) { print(ii) }
```

```
## [1] 1e-06
## [1] 2e-06
## [1] 3e-06
```

```
# but not this!
1:(1:5)
```

```
## Warning in 1:(1:5): numerical expression has 5 elements: only
the first used
## [1] 1
```

Inspection

```
# Let's use the previous function definition.
# Open a new script file and paste in the following:

mk.fib <- function(N, seed = c(0, 1)) {
  # If the author had commented this code better,
  # maybe he would have spotted the mistake.
  # Can you see what's wrong?
  for (ii in 1:N) {
    seed[ii + 1] <- seed[ii] + seed[ii - 1]
  }
  return(seed)
}
# Now use it.
mk.fib(5)
```

```
## Error in seed[ii + 1] <- seed[ii] + seed[ii - 1]:  replacement
has length zero
```

What does the error say? Does it make sense?

Inspecting a function with **print()**

```
# Edit your script file
mk.fib <- function(N, seed = c(0, 1)) {
  # Can you see what's wrong?
  for (ii in 1:N) {
    print(ii); print(seed)    # ADD THIS LINE
    seed[ii + 1] <- seed[ii] + seed[ii - 1]
  }
  return(seed)
}
# Test the function
mk.fib(5)
```

```
## [1] 1
## [1] 0 1
```

```
## Error in seed[ii + 1] <- seed[ii] + seed[ii - 1]:  replacement
has length zero
```

- Look closely at the output. *When* does the error happen?
- Raise your hand if you understand what's wrong!

Inspecting a function with **browser()**

browser()

is my favorite debugging tool.

```
# Edit your script file again
mk.fib <- function(N, seed = c(0, 1)) {
  # Can you spot the mistake?
  for (ii in 1:N) {
    print(paste("!! Entering browser with ii =", ii)) # ADD THIS LIN
    # browser()   # ADD THIS LINE
    seed[ii + 1] <- seed[ii] + seed[ii - 1]
  }
  return(seed)
}
# Test the function
mk.fib(5)
```

```
## [1] "!! Entering browser with ii = 1"
```

```
## Error in seed[ii + 1] <- seed[ii] + seed[ii - 1]:  replacement
has length zero
```

Inspection – Cont.

At the **Browse[1]>** prompt, try the following:

```
# Show the local environment
ls()

# Check where we are in the loop
ii

# Check each part of the code
seed[ii]
seed[ii + 1]
seed[ii - 1]
```

- Can you correct the error now?
- Look at the help for **browser()** (e.g. **?browser**), especially the **Details** section. When the error is corrected, how does **browser()** work?

Software Development Best Practices

These guidelines will help you write better code in less time:

- When you get stuck, take a break. Avoid working when frustrated or upset.
- Learn your text editor: use keyboard shortcuts, syntax highlighting, and proper code indenting.
- Seek help early and often: ?help, Google, other students.
- Ask good questions – prepare a minimal, commented, fully-reproducible example.

Part 2, Outline

Three powerful R programming techniques.

Partner with someone if you want, and fire up Rstudio!

Scoping: Using with() and within()

`http://cran.r-project.org/doc/manuals/R-intro.html#Scope`

reshape2: Manipulating data.frames

`http://cran.r-project.org/web/packages/reshape2/index.html`

plyr: Split-apply-combine

`http://cran.r-project.org/web/packages/plyr/index.html`

Let's start with packages.

```
# install.package("reshape2")
require(reshape2)
# install.package("plyr")
require(plyr)
# install.package("ggplot2")
require(ggplot2)
```

Scoping
Where the Variables Live

data.frame()

```
# Initialize some variables
# Number of things
N <- 1e4

# Make a new dataframe of quantile functions
# for several distributions

# Why do we use , instead of ;
# And = instead of <-?
quants <- data.frame(
  # Probability, from 0 to 1
  Pr = (1:(N - 1)) / N,
  norm = qnorm(Pr),
  pois = qpois(Pr, 5),
  gamma = qgamma(Pr, 3)
)
```

```
## Error in qnorm(Pr):  object 'Pr' not found
```

Why is `Pr` not found?

data.frame()

```r
# create Pr first, then use in data.frame() function.

# Probability, from 0 to 1
Pr = (1:(N - 1)) / N

# Make a new dataframe of quantile functions
# for several distributions
quants <- data.frame(
  Pr = Pr,
  norm = qnorm(Pr),
  pois = qpois(Pr, 5),
  gamma = qgamma(Pr, 3)
)
# removing the variable we don't need anymore
rm(Pr)
```

Can you guess what **quants** looks like?

Where does pois live?

```
# Inspect
head(quants, 2)
```

```
##      Pr      norm pois      gamma
## 1 1e-04 -3.719016    0 0.08617606
## 2 2e-04 -3.540084    0 0.10919865
```

```
str(quants)
```

```
## 'data.frame': 9999 obs. of  4 variables:
##  $ Pr   : num  1e-04 2e-04 3e-04 4e-04 5e-04 6e-04 7e-04 8e-04 9e
##  $ norm : num  -3.72 -3.54 -3.43 -3.35 -3.29 ...
##  $ pois : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ gamma: num  0.0862 0.1092 0.1255 0.1386 0.1497 ...
```

```
# Can we look at just one column?
head(pois, 2)
```

```
## Error in head(pois, 2):  object 'pois' not found
```

```
# Let's tell R where to find it.
with(quants, head(pois, 2))
```

```
## [1] 0 0
```

```
head(quants$pois, 2)
```

within() – like with() only more so.

```
# Make a new variable by modifying quants
# Why are we using { and ; now?
quants.within <- within( quants, {
  norm.big <- (norm > pois) & (norm > gamma)
  pois.big <- (pois > norm) & (pois > gamma)
  # we can now use the above variables
  gamma.big <- !(norm.big | pois.big)
})

# Inspect
head(quants.within, 2)
##      Pr      norm pois      gamma gamma.big pois.big norm.big
## 1 1e-04 -3.719016    0 0.08617606      TRUE    FALSE    FALSE
## 2 2e-04 -3.540084    0 0.10919865      TRUE    FALSE    FALSE
tail(quants.within, 2)
##          Pr     norm pois    gamma gamma.big pois.big norm.big
## 9998 0.9998 3.540084   15 13.12493     FALSE     TRUE    FALSE
## 9999 0.9999 3.719016   15 13.92817     FALSE     TRUE    FALSE
```

subset knows where to look

```
Pr < 0.1 # Just checking, we removed this variable
```

```
## Error in eval(expr, envir, enclos):  object 'Pr' not found
```

```
# Only return rows matching the condition
# Subset looks inside quants for Pr
quants.tails <- subset(quants, Pr < 0.005 | Pr > 0.995)

# Inspect dimensions: how many rows did we start with?
dim(quants.tails)
```

```
## [1] 98  4
```

```
# Use subset to remove a column
quants.sub <- subset(quants, select = -pois)

head(quants.sub, 2)
```

```
##      Pr      norm      gamma
## 1 1e-04 -3.719016 0.08617606
## 2 2e-04 -3.540084 0.10919865
```

reshaping data – wide vs. long

- **Wide** data has measurements in separate columns. Wide data is often required for linear models: $lm(y \sim x1 + x2 + x3, wide.df)$
- **Long** data has a single column of measurements. Other columns identify the type of measurement. Long data is often easier to plot: **facet_wrap(), facet_grid()**.

```
# melt is a function in the reshape2 package
# quants is in wide form.
# Which variable "identifies" each measurement?
quants.melt <- melt(quants, id.vars = "Pr")

# Inspect
head(quants.melt, 3)
##      Pr variable     value
## 1 1e-04     norm -3.719016
## 2 2e-04     norm -3.540084
## 3 3e-04     norm -3.431614
```

reshaping data – cont.

```
# Wide format
summary(quants)
```

```
##        Pr                 norm                pois          gamma
##  Min.   :0.0001   Min.   :-3.7190   Min.   : 0   Min.   : 0.08618
##  1st Qu.:0.2500   1st Qu.:-0.6743   1st Qu.: 3   1st Qu.: 1.72749
##  Median :0.5000   Median : 0.0000   Median : 5   Median : 2.67406
##  Mean   :0.5000   Mean   : 0.0000   Mean   : 5   Mean   : 2.99950
##  3rd Qu.:0.7500   3rd Qu.: 0.6743   3rd Qu.: 6   3rd Qu.: 3.92007
##  Max.   :0.9999   Max.   : 3.7190   Max.   :15   Max.   :13.92817
```

```
# Long format
summary(quants.melt)
```

```
##        Pr           variable        value
##  Min.   :0.0001   norm :9999   Min.   :-3.7190
##  1st Qu.:0.2500   pois :9999   1st Qu.: 0.5873
##  Median :0.5000   gamma:9999   Median : 2.2397
##  Mean   :0.5000                Mean   : 2.6664
##  3rd Qu.:0.7500                3rd Qu.: 4.3555
##  Max.   :0.9999                Max.   :15.0000
```

reshaping data – cont.

```
# Let's clean up column names:
# rename is a plyr function, better than accessing by position
quants.melt <- rename(quants.melt, c(value="quantile"))

# Inspect
head(quants.melt, 2)
##      Pr variable  quantile
## 1 1e-04     norm -3.719016
## 2 2e-04     norm -3.540084
str(quants.melt)
## 'data.frame': 29997 obs. of  3 variables:
##  $ Pr      : num  1e-04 2e-04 3e-04 4e-04 5e-04 6e-04 7e-04 8e-04
##  $ variable: Factor w/ 3 levels "norm","pois",..: 1 1 1 1 1 1 1 1
##  $ quantile: num  -3.72 -3.54 -3.43 -3.35 -3.29 ...
```

Plotting

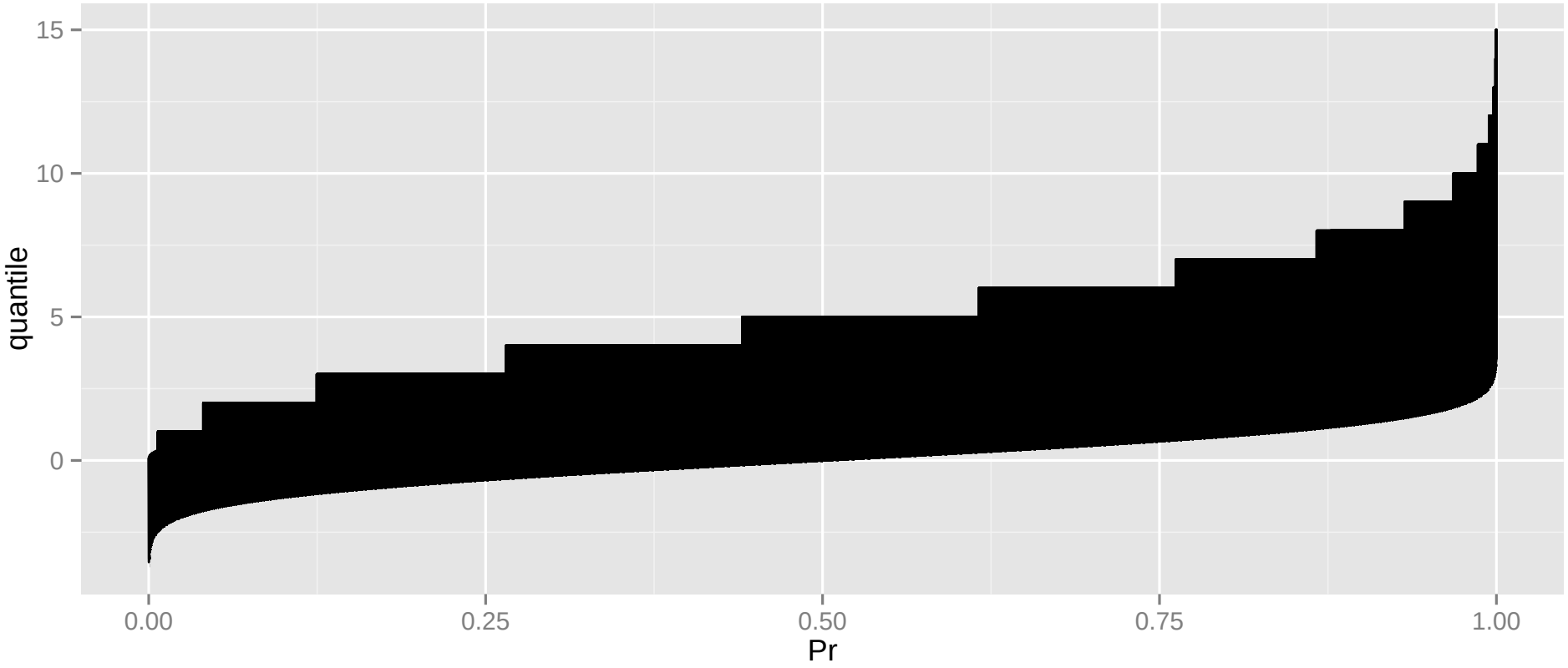
A plotting function

We're going to define a plotting function and reuse it.
The `...` is special. It represents any number of arguments that are passed to another function (including nothing).

```
plot.quant <- function(x, ...) {
  # object to return
  ret <- ggplot(x, aes(x=Pr, y=quantile)) +
    geom_line(...)
}
```

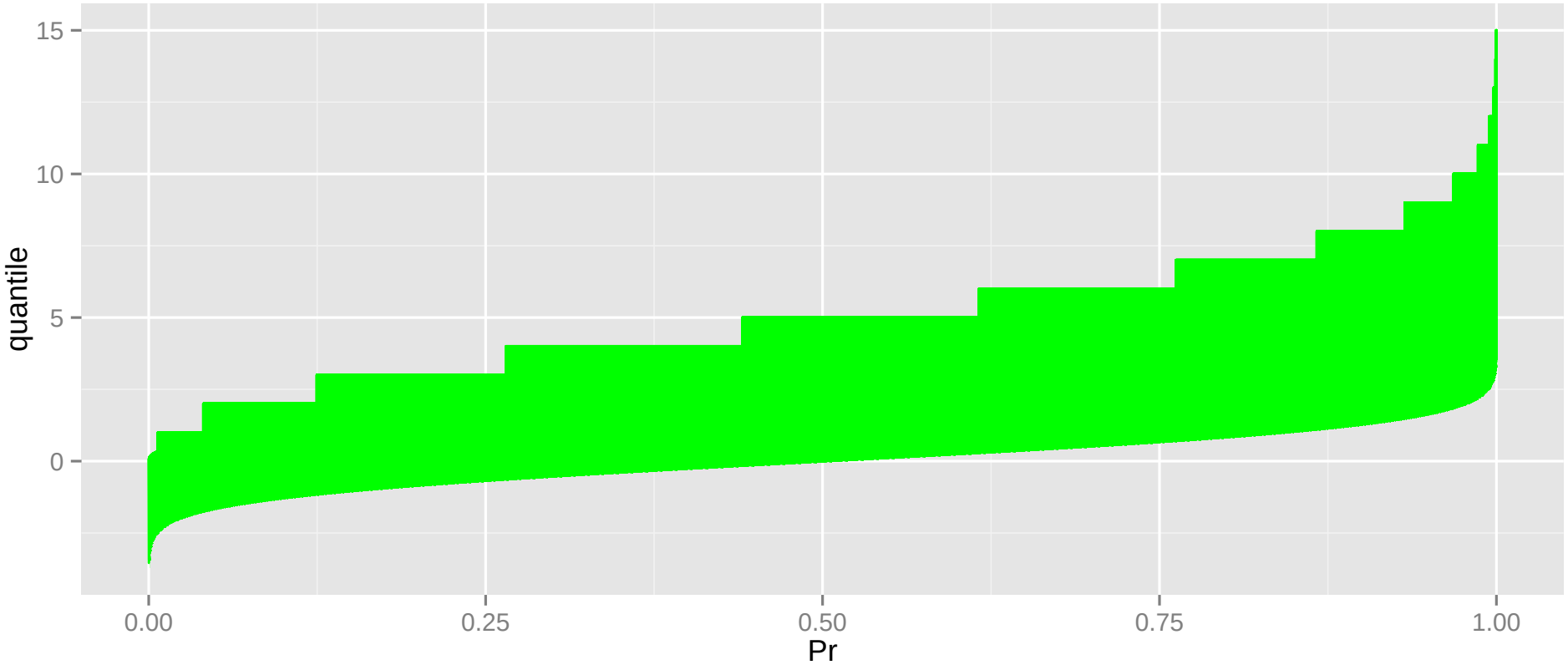
How does R know where to find **Pr** and **quantile**?!

```r
# first try
plot1 <- plot.quant(quants.melt)
plot(plot1)
```

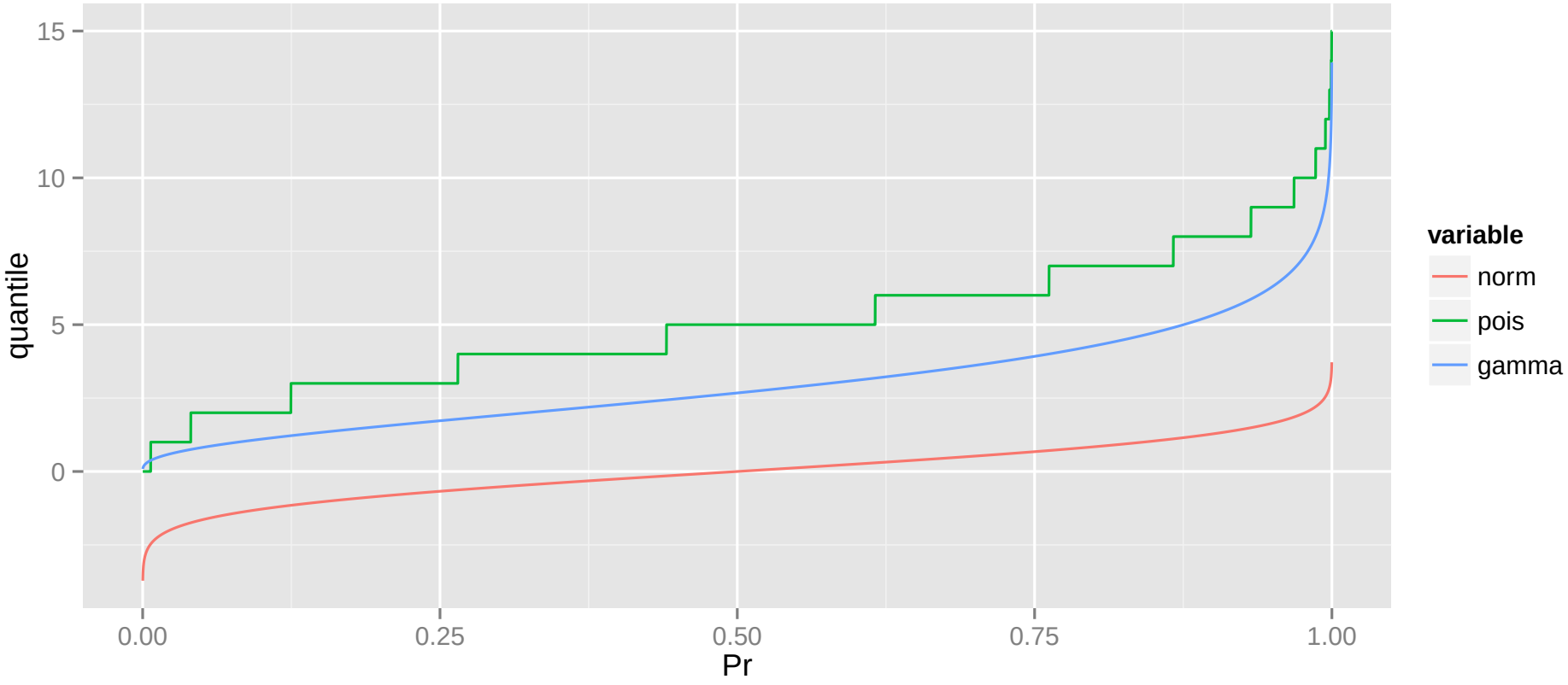


Can you figure out what's happening?

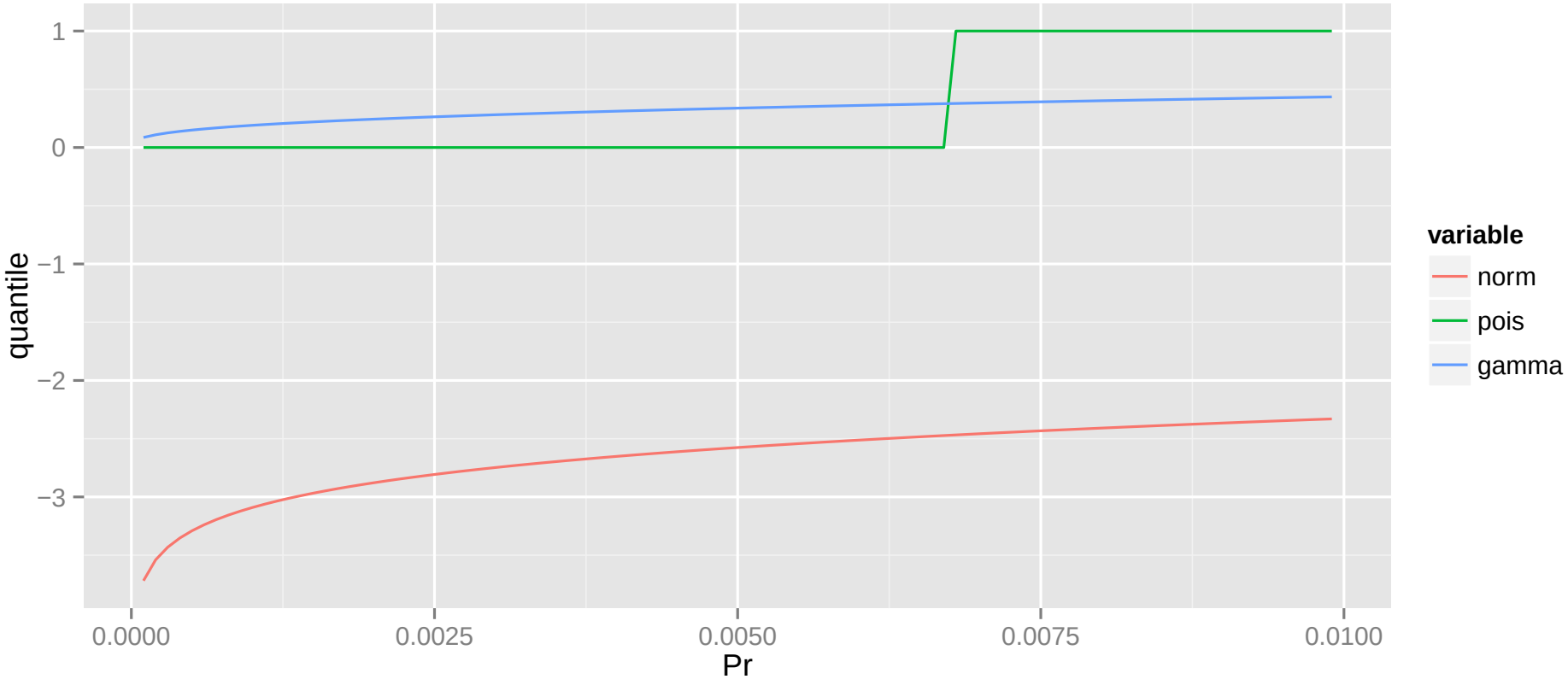
```r
# If at first...
plot2 <- plot.quant(quants.melt, color="green")
plot(plot2)
```



```r
# Where is variable located?
plot3 <- plot.quant(quants.melt, aes(color=variable))
plot(plot3)
```



```
# Let's examine the lower tail
plot4 <- plot.quant( subset(quants.melt, Pr<0.01), aes(color=variable
plot(plot4)
```



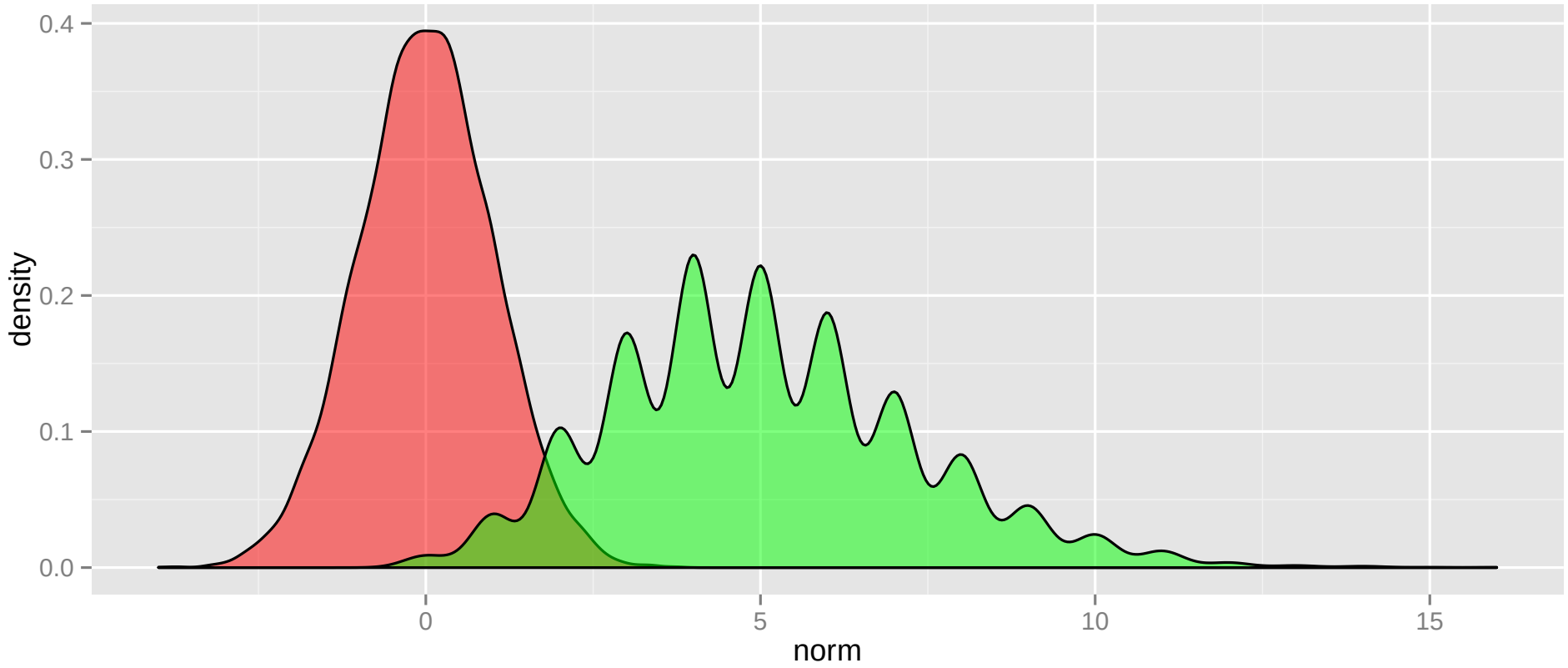
One more example, with random data

```
# like the beginning, only with random deviates
rands <- data.frame(
  # indicator variable
  index = 1:N,
  norm = rnorm(N),
  pois = rpois(N, 5),
  gamma = rgamma(N, 3)
)

summary(rands)
```

```
##       index             norm                pois
##  Min.   :    1   Min.   :-3.993715   Min.   : 0.000
##  1st Qu.: 2501   1st Qu.:-0.657451   1st Qu.: 3.000
##  Median : 5000   Median : 0.008357   Median : 5.000
##  Mean   : 5000   Mean   : 0.004017   Mean   : 5.031
##  3rd Qu.: 7500   3rd Qu.: 0.667325   3rd Qu.: 6.000
##  Max.   :10000   Max.   : 3.729744   Max.   :16.000
##       gamma
##  Min.   : 0.05376
##  1st Qu.: 1.72083
```

```r
# Let's try a density plot
# Why is there no aes() around alpha?
plot5 <- ggplot(rands) +
  # each geom gets its own aes()
  geom_density(aes(x=norm), fill="red", alpha=0.5) +
  geom_density(aes(x=pois), fill="green", alpha=0.5)
plot(plot5)
```



That was a pain. Can you think of a better way?

```r
# How about melt?
# We can specify the measured variables, instead
rands.melt <- melt( rands,
  measure.vars=c("norm", "pois", "gamma")
)
head(rands.melt)
```

```
##   index variable      value
## 1     1     norm -0.9481572
## 2     2     norm -0.5272498
## 3     3     norm  0.1235671
## 4     4     norm  0.1028999
## 5     5     norm -0.3368674
## 6     6     norm  1.1322124
```

```r
# An easier density plot
plot6 <- ggplot(rands.melt, aes(x=value, fill=variable)) +
  geom_density(alpha=0.5)
plot(plot6)
```

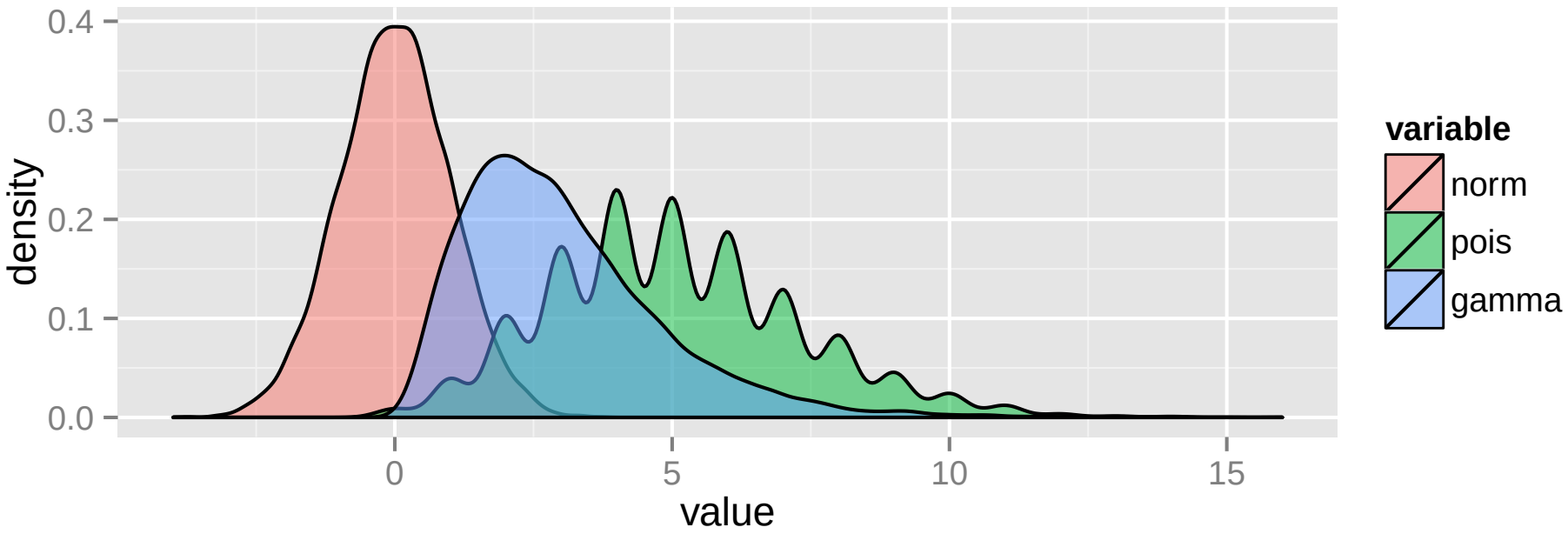


Figure: Much better. Now, why do we have a stegosaurus?

I used the **knitr** chunk option **fig.cap="Much better..."** to make this caption (which doesn't work quite right in slides). It would be a **great** idea to use this on your homework.

```r
# Use a narrower smoothing bandwidth for density estimation
# geom_density passes adjust=0.2 to density()
plot7 <- ggplot(rands.melt, aes(x=value, fill=variable)) +
  geom_density(alpha=0.5, adjust=0.5)
plot(plot7)
```

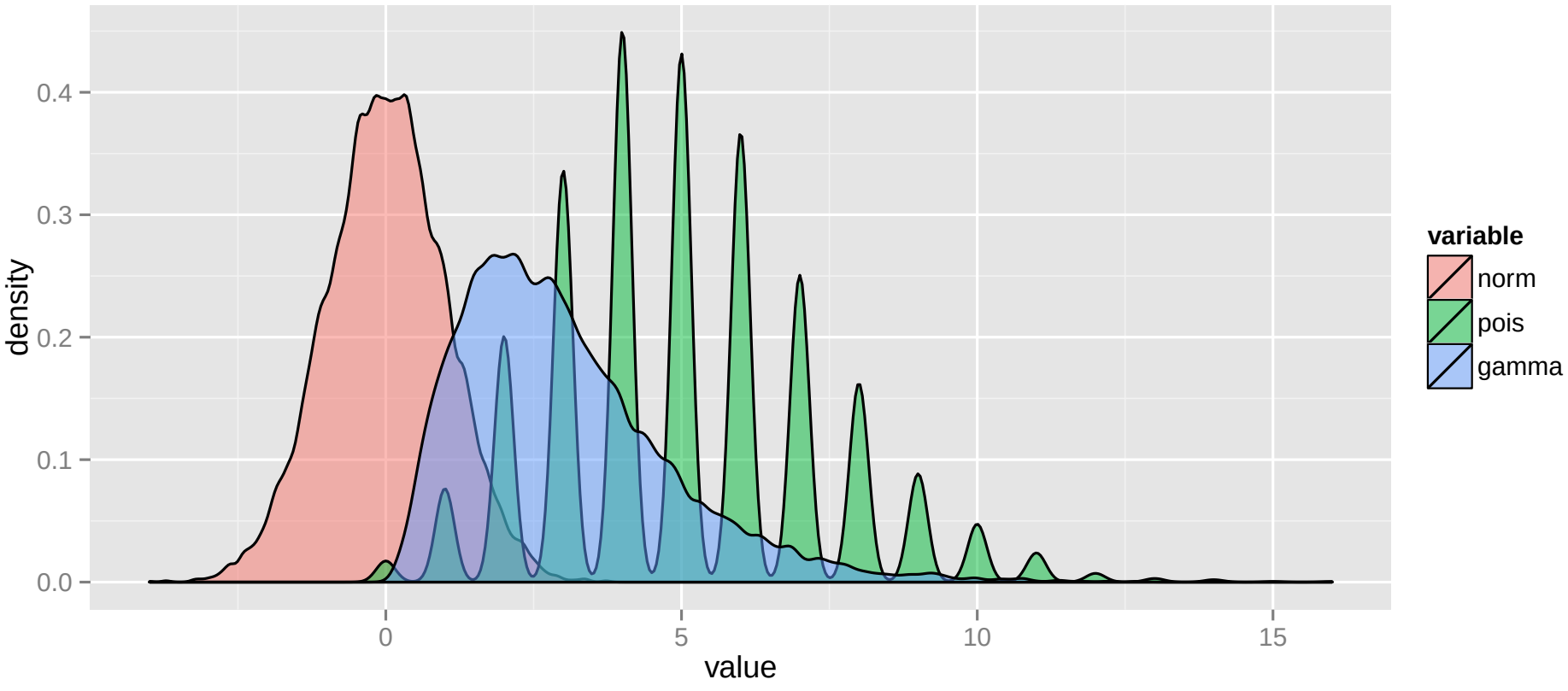


Figure: Does it make sense to mix continuous and discrete distributions in this figure? It doesn't matter how pretty a figure is if it doesn't make sense!

plyr – split, apply, combine

```
# min, max, and quantile summaries for a single variable
my.probs <- c(0, 0.25, 0.5, 0.75, 1)
quants.norm <- quantile( rands$norm, probs=my.probs)
quants.norm
```

```
##          0%         25%         50%         75%        100%
## -3.99371492 -0.65745088  0.00835688  0.66732521  3.72974398
```

```
# 5-number summary for each variable
quants.all <- ddply( rands.melt, "variable", function(x) {
  # what variable/colname do we want to compute on?
  # returning a data.frame gives most control over, e.g., colnames
  data.frame(prob=my.probs
             , quantile=quantile(x$value, probs=my.probs ))
  })
```

plyr – split, apply, combine

```
# Inspect
quants.all
```

```
##    variable prob    quantile
## 1      norm 0.00 -3.99371492
## 2      norm 0.25 -0.65745088
## 3      norm 0.50  0.00835688
## 4      norm 0.75  0.66732521
## 5      norm 1.00  3.72974398
## 6      pois 0.00  0.00000000
## 7      pois 0.25  3.00000000
## 8      pois 0.50  5.00000000
## 9      pois 0.75  6.00000000
## 10     pois 1.00 16.00000000
## 11    gamma 0.00  0.05376136
## 12    gamma 0.25  1.72083459
## 13    gamma 0.50  2.68221263
## 14    gamma 0.75  3.91267941
## 15    gamma 1.00 12.75033078
```

knitr chunk options

Chunk options go in the $<< label, ... >>=$ part.

- fig.cap="My caption for this figure"
- fig.width=7 is default. Using a larger number will shrink your figures (confusing).
- fig.height=7 is default. Use smaller numbers to make shorter figures.