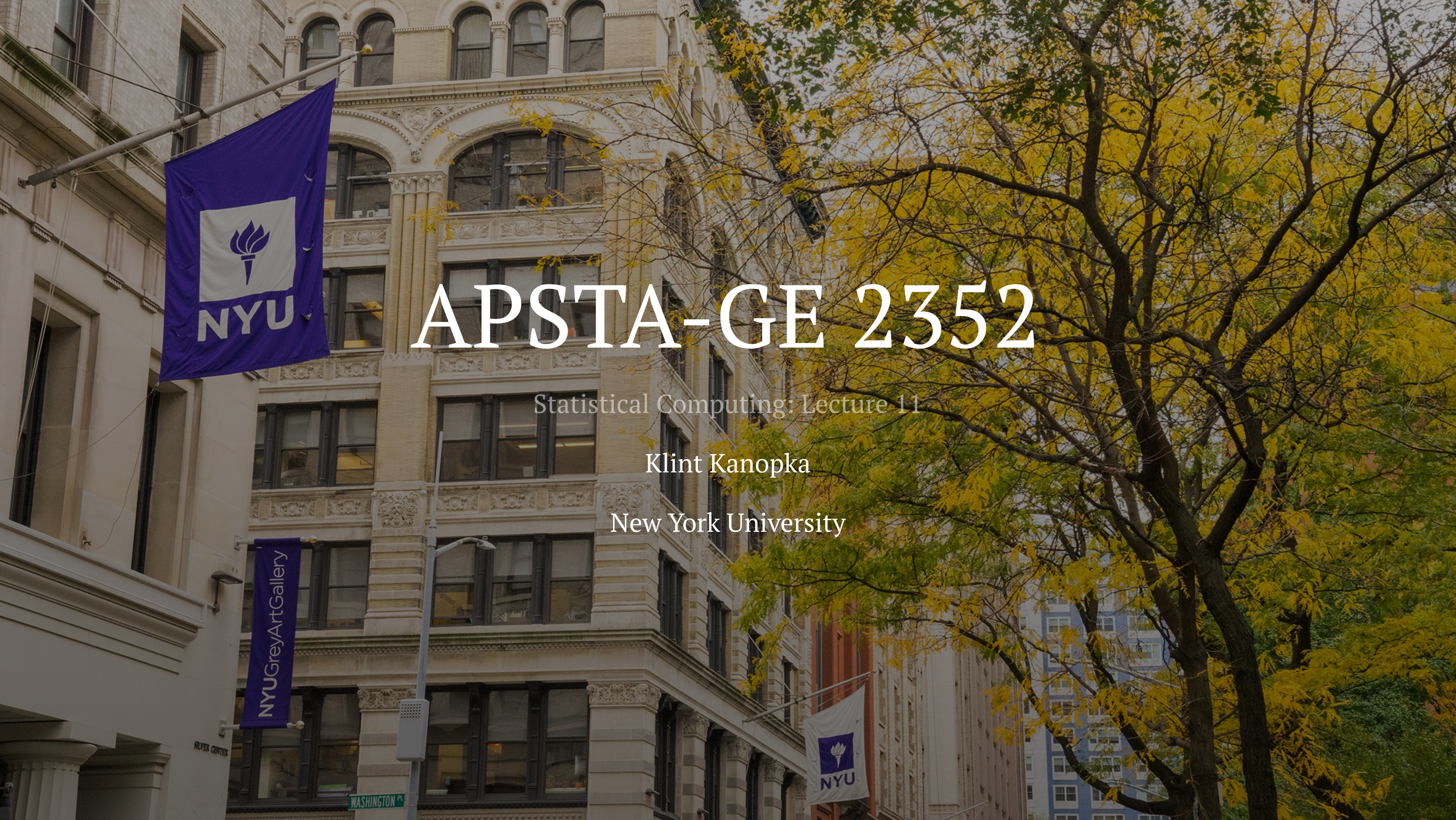


APSTA-GE 2352

Statistical Computing: Lecture 11

Klint Kanopka

New York University

Table of Contents

Check-In

- PollEv.com/klintkanopka

Announcements

- PS6 is posted!
  - It's your 2nd to last pset
  - It's shorter than PS5
- Lab on Wednesday before Thanksgiving is cancelled
- After today:
  - 3 more lectures
  - 3 more labs
  - 2 more psets
- Reminder about next week's PRIISM seminar talk with my friend Charlie Rahal from Oxford!

Advanced Debugging

The tools we have so far

- Googling error messages
- `print()` statements
- `traceback()`
- Any others?

What about generative AI?

- I honestly don't know much about debugging code with tools like ChatGPT
- What's your experience been with using ChatGPT to help you debug or write code?
- Are there other models/tools that you use?
- What prompting strategies do you use?
- Do you feel like using generative AI to help with coding tasks increases your understanding?
- Any other notes?

A more powerful debugging tool: `browser()`

- `browser()` lets you step into the code and execute it line by line while you monitor what is loaded in each environment and the values of intermediate objects
- You can further step into functions you encounter along the way
- How does it work?
  - You insert the `browser()` function call into some code you want to debug
  - From here, the right pane shows you the values and variables for the environment you’re currently executing code in
  - You get a special prompt in the console, `Browse[1]>` to let you know you’re doing browser stuff
  - All execution halts and the next line to be run appears above the prompt

Using `browser()`

- First, you can use the `Browse[1]>` prompt like any other sort of console prompt and execute code from there
- Second, and more importantly, you get access to a bunch of new commands:

1. `n` runs the **next** line of code (whatever is currently above the prompt)
2. `s` is like next, but if the next line is a function, you'll **step into** it and run it interactively (line-by-line)
3. `c` stops running code line-by-line and **continues** executing the current function you're currently in
4. `Q` **quits** out of the browser

Why and how to use `browser()`?

- Essentially it’s not too far off from inserting `print()` statements after every single line
- It allows you to see what changes after each line of code is run
- You can also insert code into a function to see if it fixes your problem
- A few other ways to use it:
    - Running `debug(FUN())` inserts `browser()` into the first line of `FUN()`, and so running `FUN()` will *always* open the browser. Stop this with `undebug(FUN())`
    - `debugonce(FUN(args))` runs `FUN(args)` immediately opening a browser, but doesn’t modify `FUN()`
- **Next let’s use `browser()` with the activity from lab yesterday**

The `apply()` Family

Side Effects

- If a function or operation modifies things outside its local environment, it has “side effects”
- These side effects can be hard to observe, but cause serious errors in analysis!
- Side effects most often occur when we are trying to hack together a solution to a problem
  - Functions that take and modify information from the global environment
  - Janky loops

Avoiding Side Effects

- Writing things in functions is a good start!
  - Calls to functions generate new environments
  - New environments protect data in the global environment
- We can “hide loops”
  - The `apply()` family of functions does this really well
  - These nest the entire loop we want to carry out inside of a function call

The `apply()` Family

- `apply()` takes a function and applies it to each element of a data object
- Whole bunch of different `apply()` functions
  - Different functions take different data objects as input
  - Different functions spit out different data objects
- Note that `apply()` functions aren’t really any faster than loops!
  - They are not vectorized
  - They are, however, very easy to *parallelize*

The main `apply()` functions

function	input	output	comment
`apply()`	matrix or array	vector or array or list	
`lapply()`	list or vector	list	
`sapply()`	list or vector	vector or matrix or list	simplify
`vapply()`	list or vector	vector or matrix or list	safer simplify
`tapply()`	data, categories	array or list	ragged
`mapply()`	lists and/or vectors	vector or matrix or list	multiple

Applying `apply()`

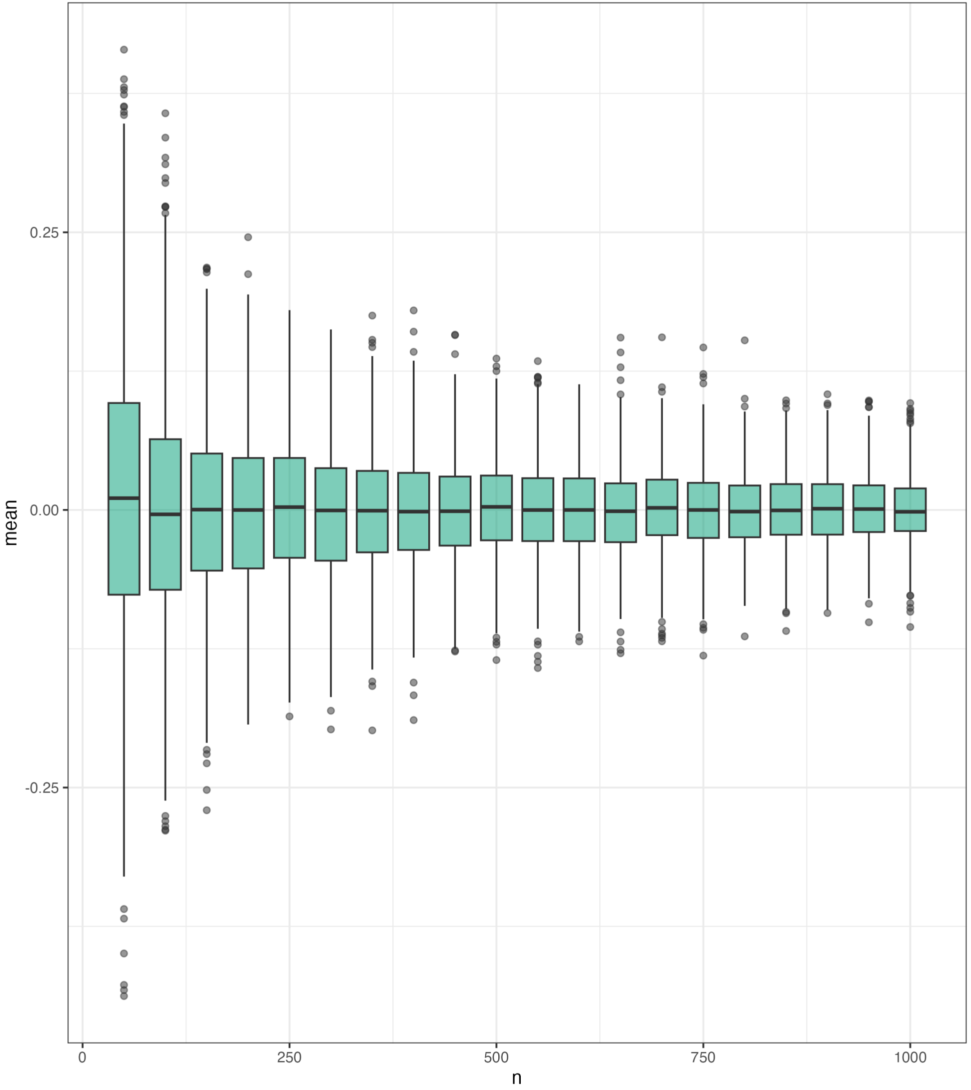
Uncertainty in mean estimates

- First we conduct a simulation study to observe how the uncertainty in estimating the mean of a normal distribution depends on the number of samples we draw from it

```r
Ns <- seq(from = 50, to = 1000, by = 50)
M <- 1000
means <- list(length = length(Ns))

for (i in seq_along(Ns)) {
  means[[i]] <- vector(length = M)
  for (j in 1:M) {
    means[[i]][j] <- mean(rnorm(Ns[i]))
  }
}

d_mean <- data.frame(n = rep(Ns, each = M), mean = do.call(c, means))
```

Visualizing uncertainty in mean estimates

```r
ggplot(d_mean, aes(x = n,
                   y = mean,
                   group = n)) +
  geom_boxplot(alpha = 0.5,
               fill = okabeito_colors(3)) +
  theme_bw()
```



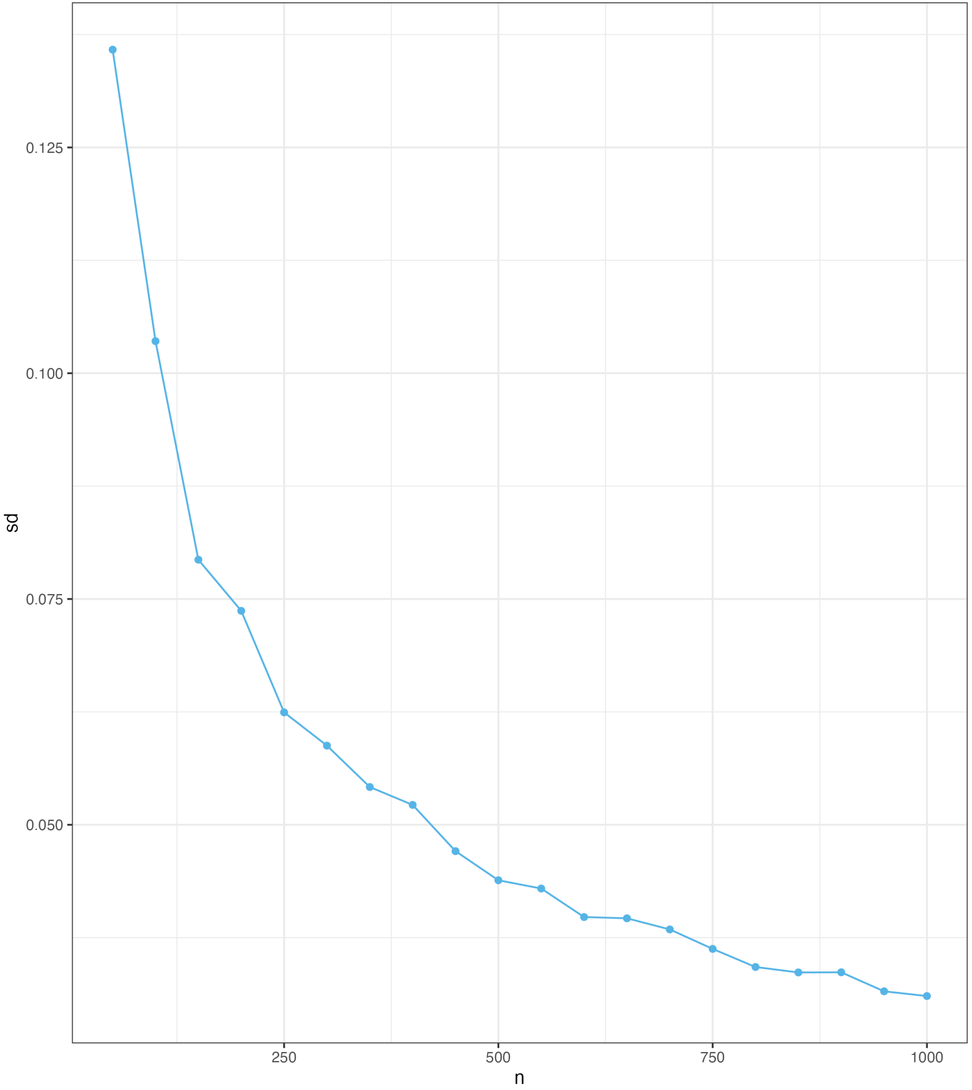
Estimating uncertainty directly

- Next we visualize our estimates of the standard errors of these sampling distributions at each sample size

```r
sds <- vector(length = length(Ns))

for (i in seq_along(Ns)) {
  sds[i] <- sd(means[[i]])
}

d_sd <- data.frame(n = Ns, sd = sds)
```

Visualizing uncertainty estimates

```r
ggplot(d_sd, aes(x = n, y = sd)) +
  geom_point(color = okabeito_colors(2)) +
  geom_line(color = okabeito_colors(2)) +
  theme_bw()
```



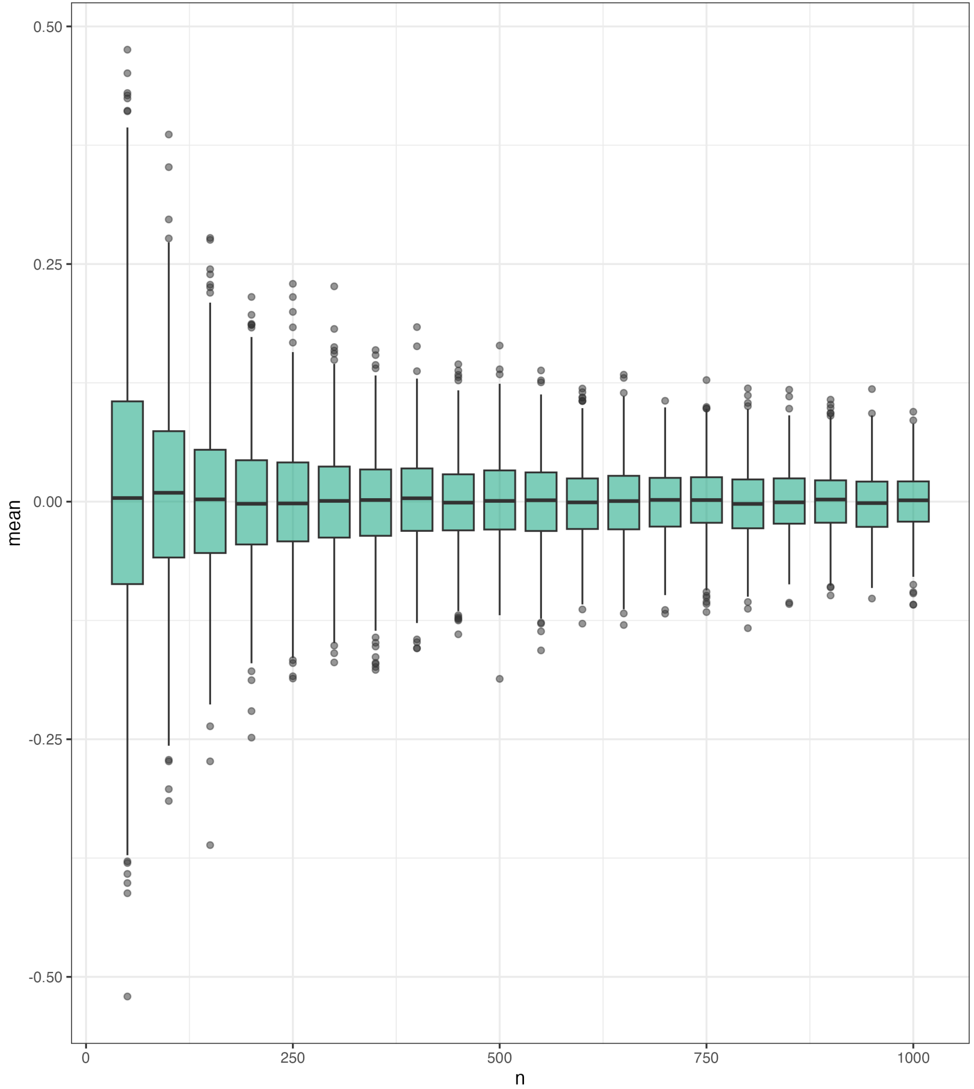
Uncertainty in mean estimates with `apply()`

- Now we redo the first simulation with something from the `apply()` family

```r
# TODO: Write a simulation function

SimFun1 <- function(n_draws) {
  draws <- rnorm(n_draws)
  m <- mean(draws)
  return(m)
}

# TODO: Construct the object to apply() over

n <- rep(Ns, each = M)

# TODO: Construct the output

d_mean_apply <- data.frame(n = n, mean = sapply(X = n, FUN = SimFun1))
```

Visualizing uncertainty in mean estimates with `apply()`

```r
ggplot(d_mean_apply, aes(x = n,
                         y = mean,
                         group = n)) +
  geom_boxplot(alpha = 0.5,
               fill = okabeito_colors(3)) +
  theme_bw()
```

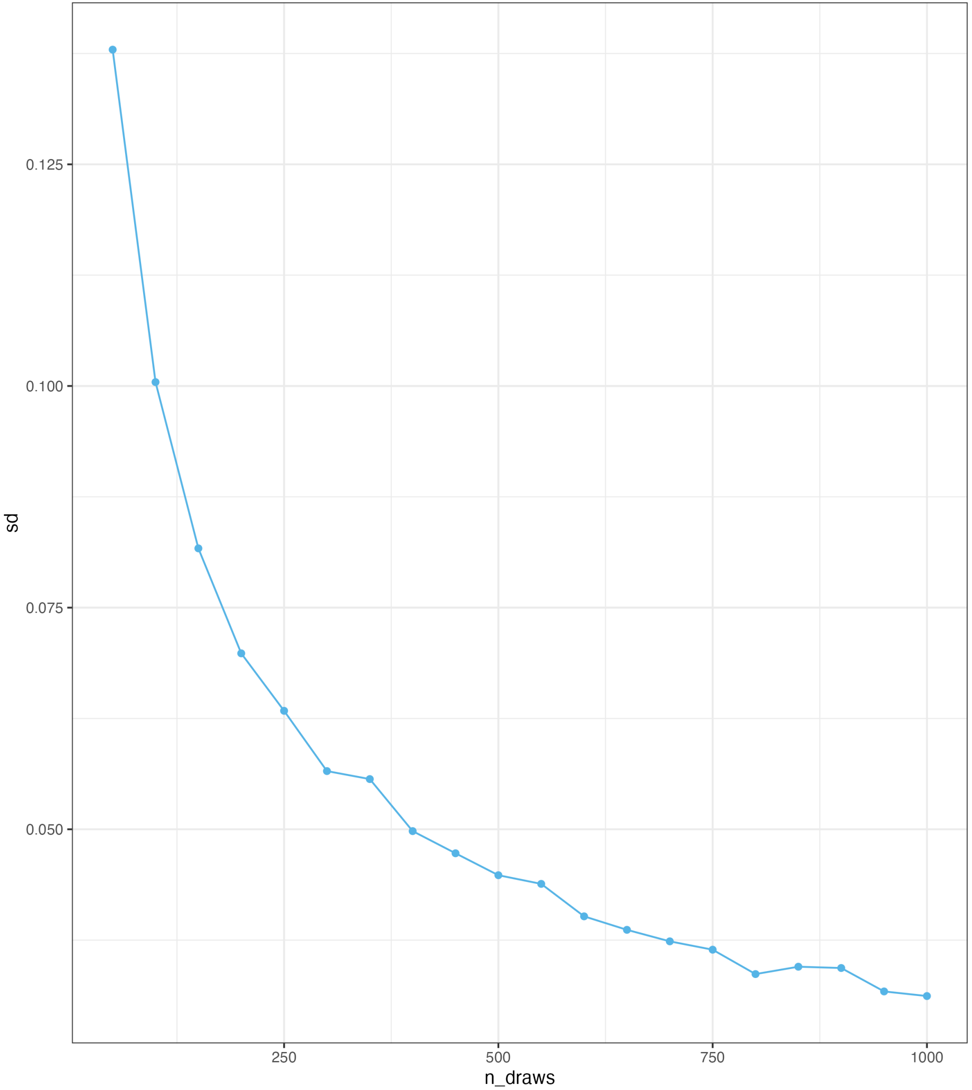


Estimating uncertainty directly with `apply()`

```r
# TODO: Write a simulation function

SimFun2 <- function(n_draws, n_reps) {
  draws <- matrix(rnorm(n = n_draws * n_reps),
                  nrow = n_draws,
                  ncol = n_reps)
  means <- apply(X = draws, MARGIN = 2, FUN = mean)
  sdev <- sd(means)
  return(sdev)
}

# TODO: Construct the object to apply() over

sim_control_2 <- data.frame(n_draws = Ns, n_reps = M)

# TODO: Construct the output

sim_control_2$sd <- mapply(FUN = SimFun2,
                           sim_control_2$n_draws,
                           sim_control_2$n_reps)
```

Visualizing uncertainty estimates with `apply()`

```r
ggplot(sim_control_2, aes(x = n_draws,
                          y = sd)) +
  geom_point(color = okabeito_colors(2)) +
  geom_line(color = okabeito_colors(2)) +
  theme_bw()
```

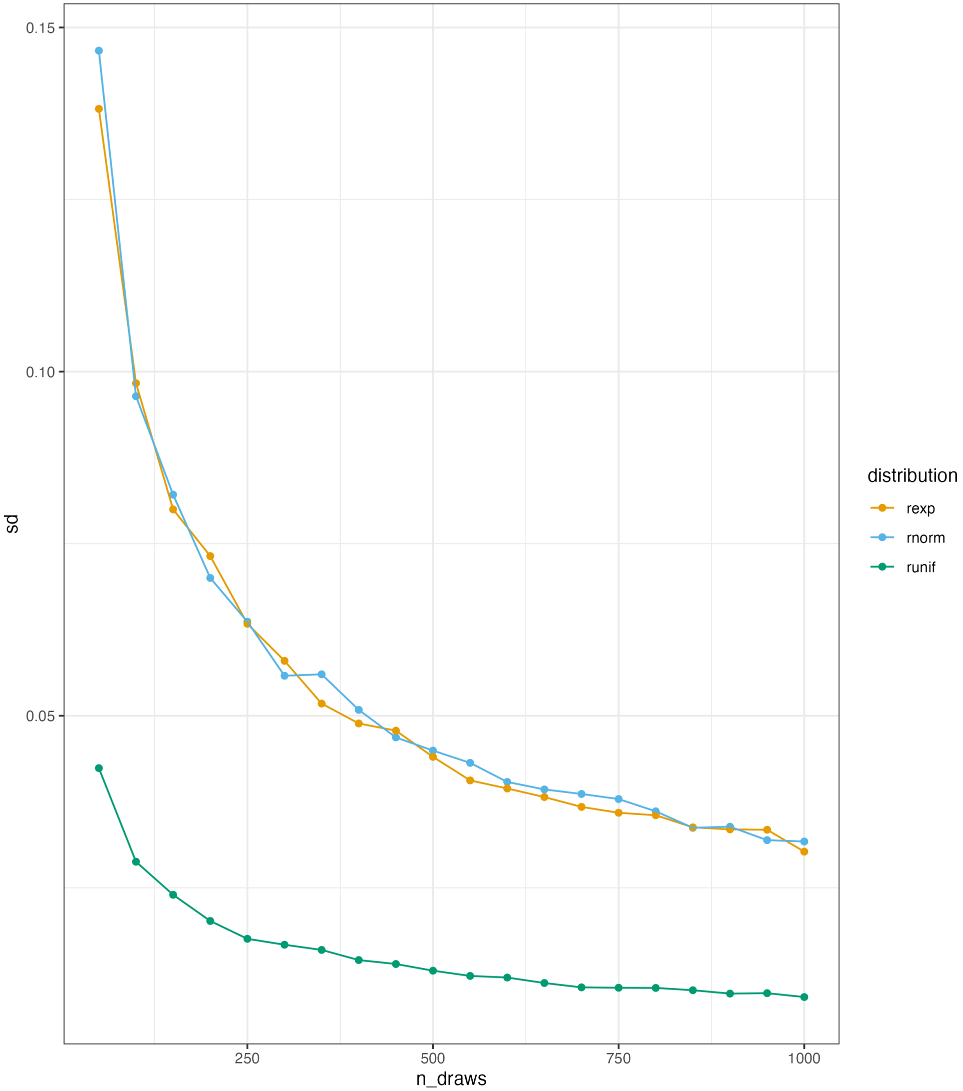


Extending to more distributions

```r
# TODO: Write a simulation function

SimFun3 <- function(n_draws, n_reps, dist) {
  draws <- do.call(what = dist, args = list(n = n_draws * n_reps))
  draws <- matrix(draws, nrow = n_draws, ncol = n_reps)
  means <- apply(X = draws, MARGIN = 2, FUN = mean)
  sdev <- sd(means)
  return(sdev)
}

# TODO: Construct the object to apply() over

distributions <- c('rnorm', 'runif', 'rexp')
sim_control_3 <- data.frame(n_draws = rep(Ns, times = length(distributions)),
                            n_reps = M,
                            distribution = rep(distributions, each = length(Ns)))

# TODO: Construct the output

sim_control_3$sd <- mapply(FUN = SimFun3,
                           sim_control_3$n_draws,
                           sim_control_3$n_reps,
                           sim_control_3$distribution)
```

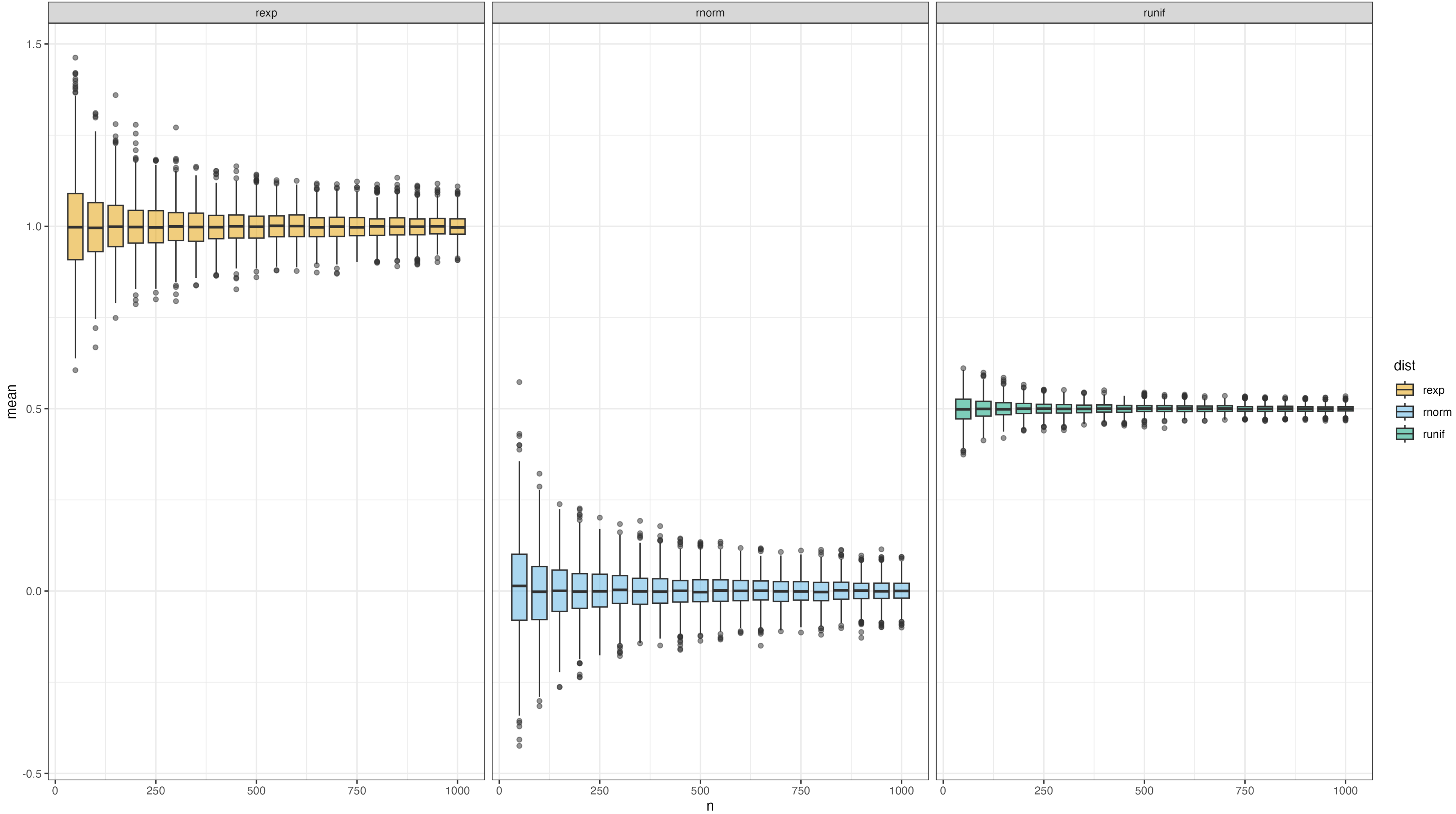
Visualizing uncertainty across distributions

```r
ggplot(sim_control_3,
       aes(x = n_draws,
           y = sd,
           color = distribution)) +
  geom_point() +
  geom_line() +
  scale_color_okabeito() +
  theme_bw()
```



Constructing more complex data frame output

```r
# TODO: Write a simulation function

SimFun4 <- function(n_draws, n_reps, dist) {
  draws <- do.call(dist, args = list(n = n_draws * n_reps))
  draws <- matrix(draws, nrow = n_draws, ncol = n_reps)
  out <- data.frame(n = n_draws, dist = dist, mean = apply(X = draws, MARGIN = 2, FUN = mean))
  return(out)
}

# TODO: Construct the output

out <- mapply(FUN = SimFun4,
              sim_control_3$n_draws,
              sim_control_3$n_reps,
              sim_control_3$distribution,
              SIMPLIFY = FALSE)

d_sim_4 <- do.call('rbind', out)

ggplot(d_sim_4, aes(x = n, y = mean, group = n, fill = dist)) +
  geom_boxplot(alpha = 0.5) +
  facet_grid(. ~ dist) +
  scale_fill_okabeito() +
  theme_bw()
```


rexp
rnorm
runif
mean
1.5
1.0
0.5
0.0
-0.5
n
0
250
500
750
1000
dist
rexp
rnorm
runif

Wrap Up

Recap

- Debugging is challenging, but we have four (five?) main tools:
  - Googling error messages
  - `print()` statements
  - `traceback()`
  - `browser()`
  - Maybe ChatGPT if you like it?
- The `apply()` family of functions allows you to hide loops
  - Not more efficient, but *safer* than loops
  - Encourages functionalization of code
  - Makes code easier to read

Final Thoughts

- PollEv.com/klintkanopka