Adventures in Simulation in R

EC 425/525, Lab 6

Edward Rubin
11 May 2019

Prologue

Schedule

Last time

Plotting

Today

Simulation

Simulation

Simulation

Motivation

As we've discussed, simulation can be a quick and effective way to better understand how an estimator performs/behaves.

You just need to be careful to **ask a clear, answerable question** and then **run a simulation** that corresponds/answers this question.

In addition, simulations can be computationally intense—they are often the first time you have to really think about efficiency in coding.

Simulation

Generic outline

The general outline for a simulation is fairly consistent.

1. **Define the population** via a data-generating process (DGP).[†]

2. Iterate. In each iteration:

   - **Sample** from your population.
   - Construct **estimates/inferences** that relate to your original question.

3. **Summarize** results.

[†] Some people prefer to actually construct the population in this step and then repeatedly sample from this fixed population. Others stick with a population defined by the DGP.

Simulation

Practical issues

This semi-theoretical framework needs a few practical reminders.

1. Always set a seed *at the beginning* of your simulation (`set.seed()`).

1. Parallelize where/when possible (*e.g.*, the `furrr` package).

1. Writing a function for a single iteration can be helpful (see above).

1. There is a (big) difference between unbiasedness and consistency.

1. You build simulations/DGPs with assumptions.

1. Analytical results can inform and/or replace simulations.

Example simulation

Simulation

The question

Q We've shown that instrumental variables (IV) is consistent, how does it perform (*i.e.*, is it unbiased) in finite (small) samples?

Note This question is definitely answerable analytically.

Nevertheless, let's see how IV performs at several small-ish sample sizes.

While we're at it, let's confirm OLS is indeed biased in this setting.

Simulation

DGP

We want a valid instrument for a setting in which treatment is endogenous.

$$\mathrm{Y}_i = \alpha + \tau \mathrm{D}_i + \varepsilon_i$$

So we want

1. **Endogenous treatment:** $\mathrm{Cov}(\mathrm{D}_i,\ \varepsilon_i) \neq 0$
2. **Predictive:** $\mathrm{Cov}(\mathrm{Z}_i,\ \mathrm{D}_i) \neq 0$
3. **Excludability:** $\mathrm{Cov}(\mathrm{Z}_i,\ \varepsilon_i) = 0$

where (2) and (3) imply Z_i is a valid instrument.

Simulation

DGP

In other words, the variance-covariance matrix of D_i, ε_i, and Z_i is

$$\Sigma = \begin{bmatrix} \sigma^2_{\mathrm{D}} & \sigma_{\mathrm{D},\varepsilon} & \sigma_{\mathrm{D},\mathrm{Z}} \\ \sigma_{\mathrm{D},\varepsilon} & \sigma^2_{\varepsilon} & 0 \\ \sigma_{\mathrm{D},\mathrm{Z}} & 0 & \sigma^2_{\mathrm{Z}} \end{bmatrix}$$

If we assume unit variances and covariances are 0.6, then

$$\Sigma = \begin{bmatrix} 1 & 0.6 & 0.6 \\ 0.6 & 1 & 0 \\ 0.6 & 0 & 1 \end{bmatrix}$$

Simulation

DGP

To simplify our lives, let's assume that $\mathbf{D}_i$, ε_i, and $\mathbf{Z}_i$ come from a multivariate normal distribution.

We defined their covariance matrix. We need to define their means.

$\mu_{\mathrm{D}} = \mathbf{10}$, $\mu_{\varepsilon} = \mathbf{0}$, and $\mu_{\mathrm{Z}} = \mathbf{3}$.

Finally, we need to define the way in which $\mathbf{D}_i$ and ε_i affect $\mathbf{Y}_i$.

$$\mathbf{Y}_i = \mathbf{7} + \mathbf{1} \times \mathbf{D}_i + \varepsilon_i$$

i.e., $\tau = 1$.

Simulation

DGP

Lucky for us, `R`'s `MASS` package has a function `mvrnorm()` that draws `n` random observations from a multivariate normal distribution with means `mu` and variance-covariance matrix `Sigma`.

Simulation

Sampling from our DPG

We're ready to write a function that performs one iteration.

Our function will accept a single argument `n`, the sample size.

```r
sim_iter <- function(n) {
  # Define our variance-covariance matrix (D, ε, Z)
  Σ <- matrix(data = c(1, 0.6, 0.6, 0.6, 1, 0, 0.6, 0, 1), ncol = 3)
  # Our vector of means (D, ε, Z)
  μ = c(10, 0, 3)
  # Draw n observations; convert to tibble
  sample_df <- MASS::mvrnorm(n = n, mu = μ, Sigma = Σ) %>% tibble()
  # Name variables
  names(sample_df) <- c("D", "ε", "Z")
  # Calculate Y
  sample_df %<>% mutate(Y = 7 + 1 * D + ε)
}
```

Simulation

Estimation

Now we just need to estimate β_{IV} and β_{OLS}. We'll use `estimatr`.

Previous OLS estimates of the effect of `x` on `y`

```
lm_robust(y ~ x)
```

New IV estimates of the effect of `x` on `y` with instrument `z`

```
iv_robust(y ~ x | z)
```

```r
sim_iter ← function(n) {
  # Define our variance-covariance matrix (D, ε, Z)
  Σ ← matrix(data = c(1, 0.6, 0.6, 0.6, 1, 0, 0.6, 0, 1), ncol = 3)
  # Our vector of means (D, ε, Z)
  μ = c(10, 0, 3)
  # Draw n observations; convert to tibble
  smpl_df ← MASS::mvrnorm(n = n, mu = μ, Sigma = Σ) %>% data.frame()
  # Name variables
  names(smpl_df) ← c("D", "ε", "Z")
  # Calculate Y
  smpl_df %<>% mutate(Y = 7 + 1 * D + ε)
  # Estimates
  est_df ← bind_rows(
    # The OLS estimates
    lm_robust(Y ~ D, data = smpl_df) %>% tidy() %>% mutate(est = "OLS"),
    # The IV estimates
    iv_robust(Y ~ D | Z, data = smpl_df) %>% tidy() %>% mutate(est = "IV")
  )
  return(est_df)
}
```

Simulation

Repeat

Now we want run `sim_iter()` *many* times.

And we're going to do it in parallel—using the `furrr` package.

The output of `sim_iter()` is a data frame, so we can actually use a function from `furrr` that expects outputted data frames, namely, `future_map_dfr`.

The suffix `_dfr` means the function will row-bind the data frames returned by individual iterations.

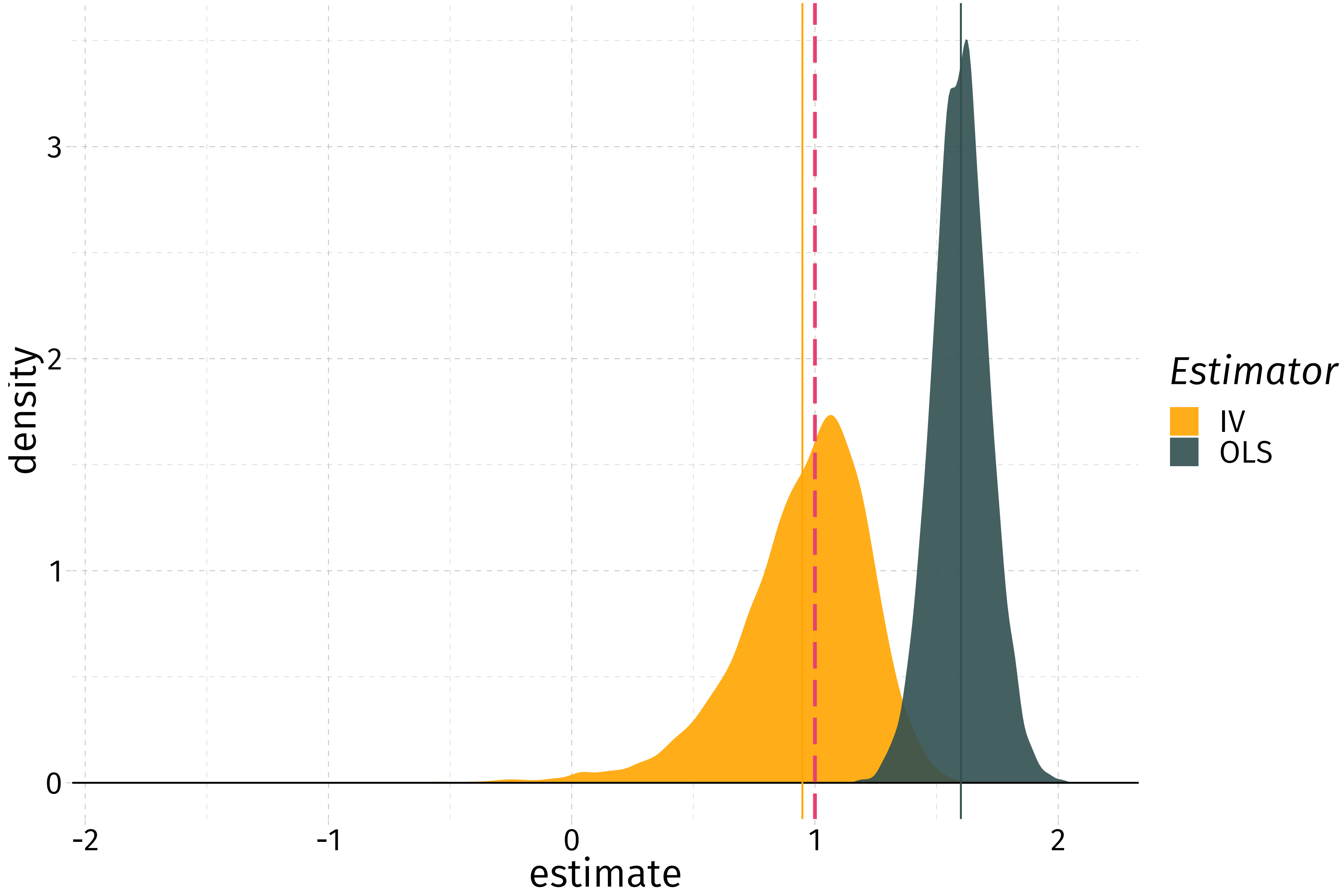
We'll also use the `rep()` function which repeats things, *e.g.*, `rep("a", 3)` repeats `"a"` three times.

Simulation

Assuming we've already entered `sim_iter()` into memory, we can run our simulation 5,000 times, each with sample size 50—in parallel!

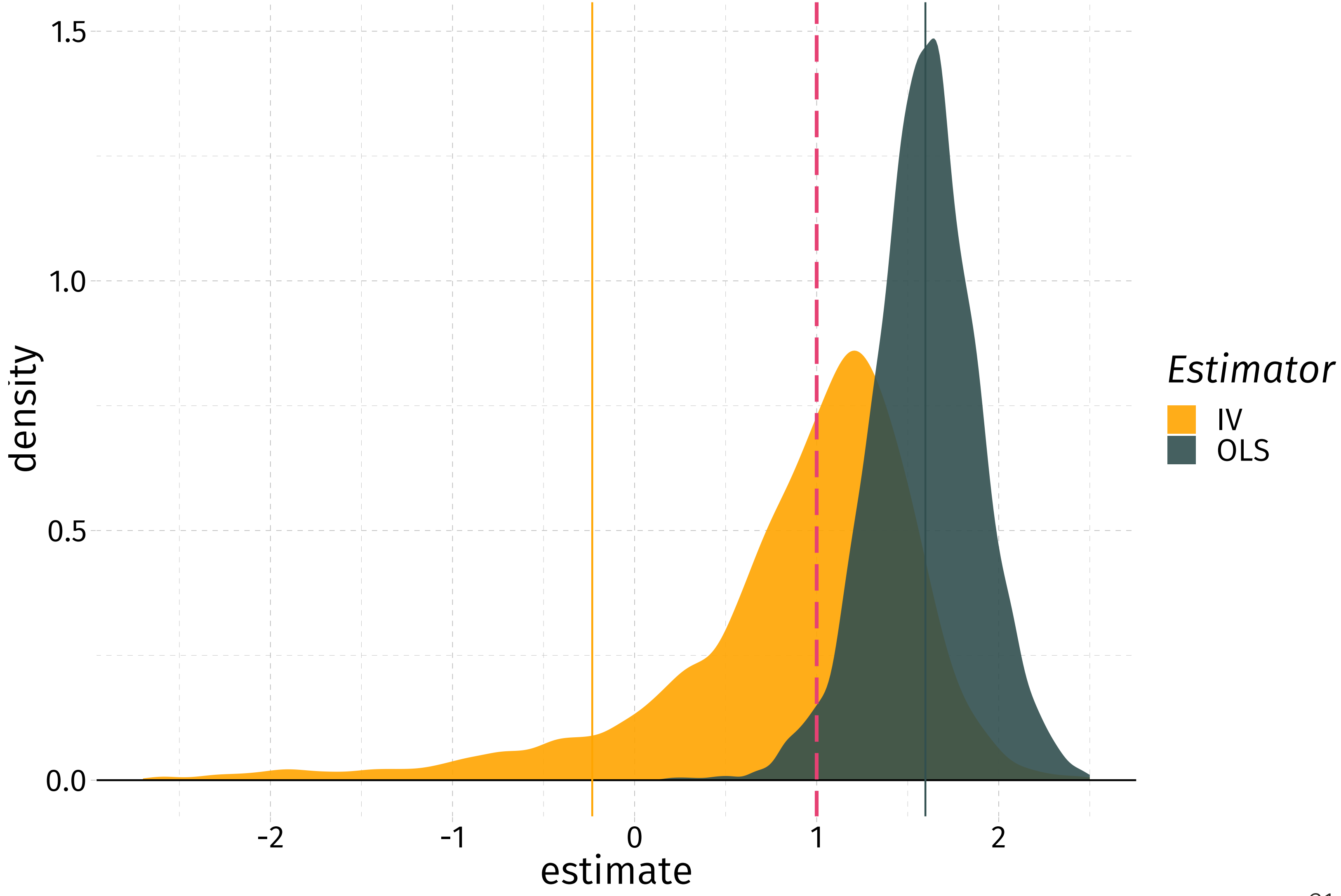
```r
# Load furrr
p_load(furrr)
# Tell R to parallelize with 4 cores
plan(multiprocess, workers = 4)
# Set a seed
set.seed(12345)
# Run simulation with sample size 50
sim50 <- future_map_dfr(
  # Repeat sample size 50 for 5000 times
  rep(50, 5000),
  # Our function
  sim_iter,
  # Let furrr know we want to set a seed
  .options = future_options(seed = T)
)
```

Sample size 50 (5,000 iterations)

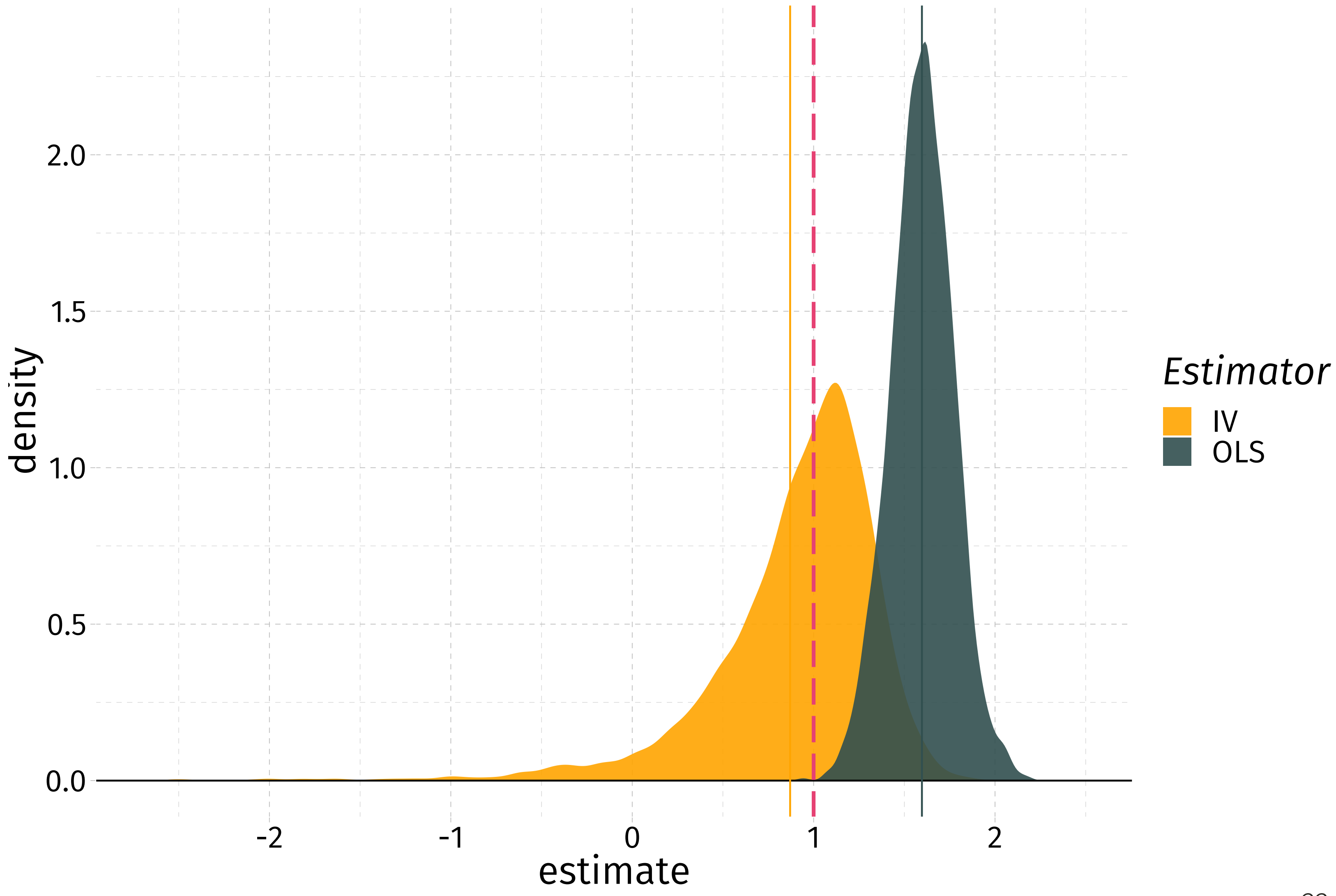


Let's vary the sample size and see what happens.

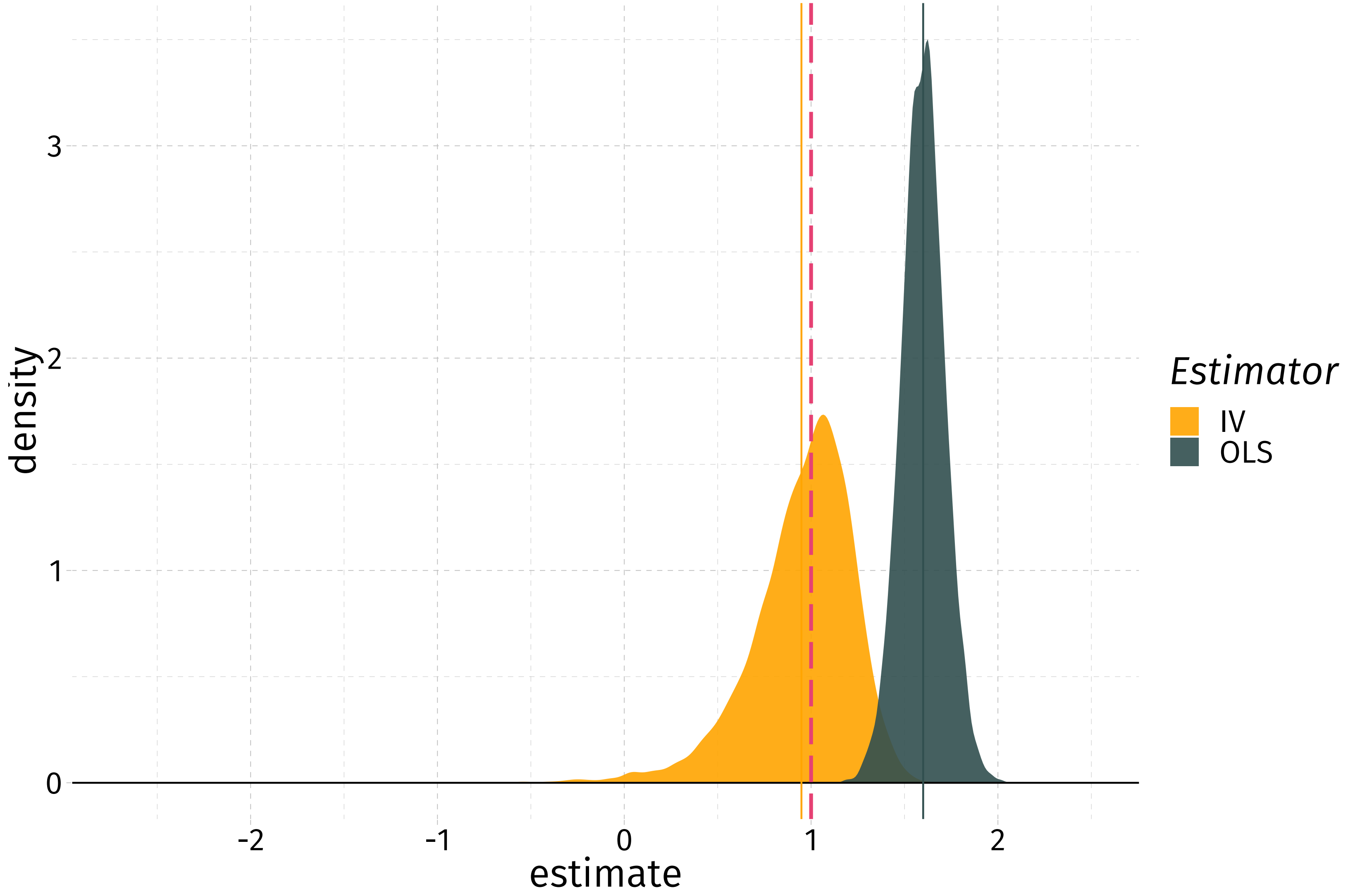
Sample size 10 (5,000 iterations)



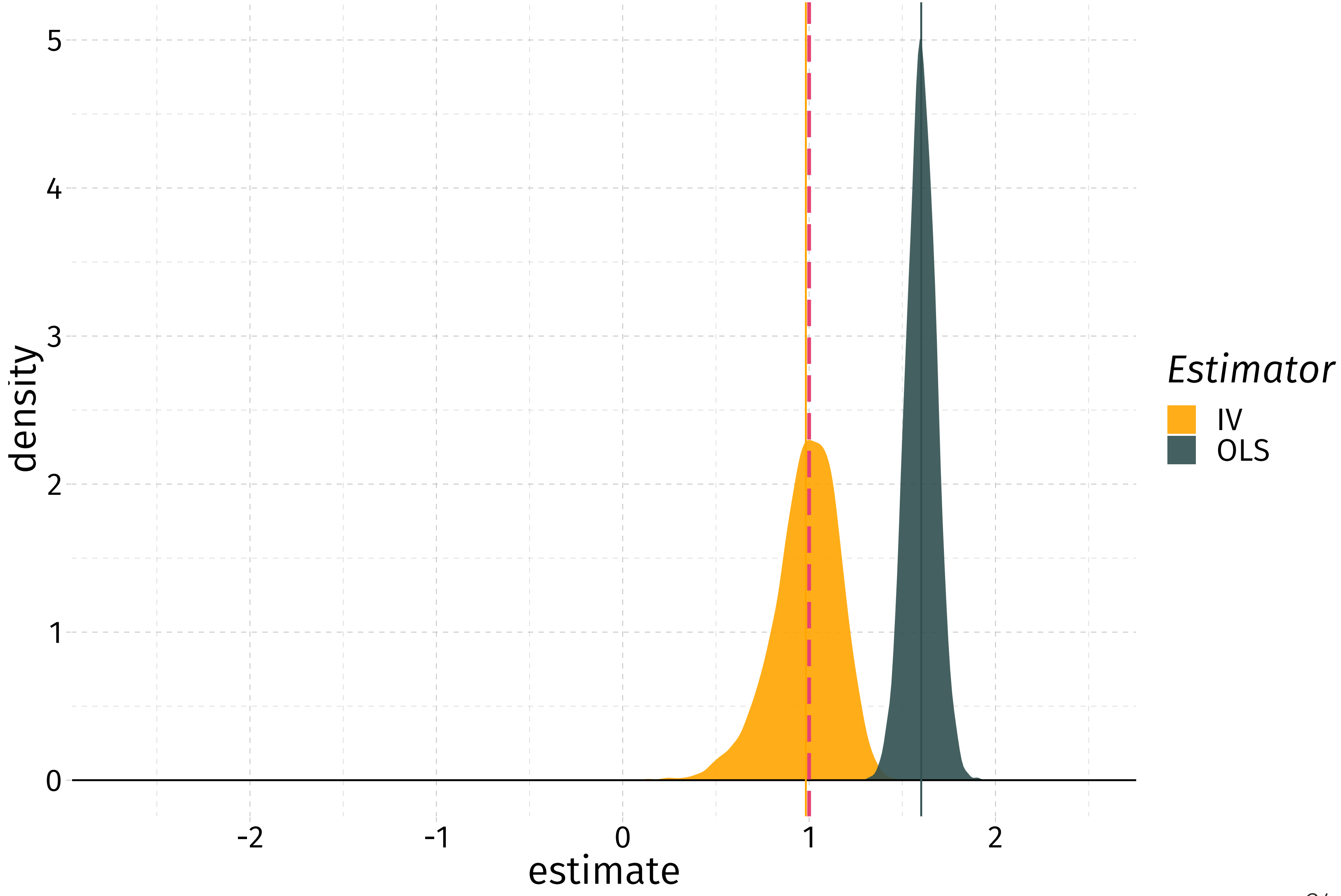
Sample size 25 (5,000 iterations)



Sample size 50 (5,000 iterations)



Sample size 100 (5,000 iterations)



Simulation

Assumptions

Keep in mind that we made several assumptions about

- the distribution (joint normality is very restrictive)
- variance (all equal, independent, and homoskedastic)
- covariances (again, all equal)
- strong instrument

Simulation

Looping

There are **many** ways to iterate/loop in `R`:

- `for()`, `while()`, *etc.*
- `lapply()`, `mapply()`, *etc.*
- `parallel`: `mclapply()`, `mcmapply()`, *etc.*
- `foreach`
- `future`, `furrr`, and `future.apply`: `future_lapply()`, `future_map()`, *etc.*

They are not all equal/identical.

- Few can access values from previous iterations (`for()` and `foreach`).
- A subset is parallelizable (`parallel`, `foreach`, the `future` family).
- Behavior can be OS specific (especially `parallel`).

Simulation

`for()`

You'll often hear that you should never use `for()` loops in R.

This opinion is a bit extreme, but there are a few reasons to avoid them.

1. `for()` is not parallelized (though `foreach` can be).

2. `for()` doesn't clean up after itself—leaving objects in memory between iterations and after the loop finishes.[†]

3. `for()` loops generally "grow" data which can be slow/inefficient.

See Grant McDermott's lectures for further justification.

[†] This feature *can* be desirable—*e.g.*, linking iterations.

Table of contents

Simulation

1. Motivation
2. Generic outline
3. Example
   - The question
   - DGP
   - Sampling from the DGP
   - Iterating
   - Assumptions
4. Loops