W203 Supplementary Exercise 3

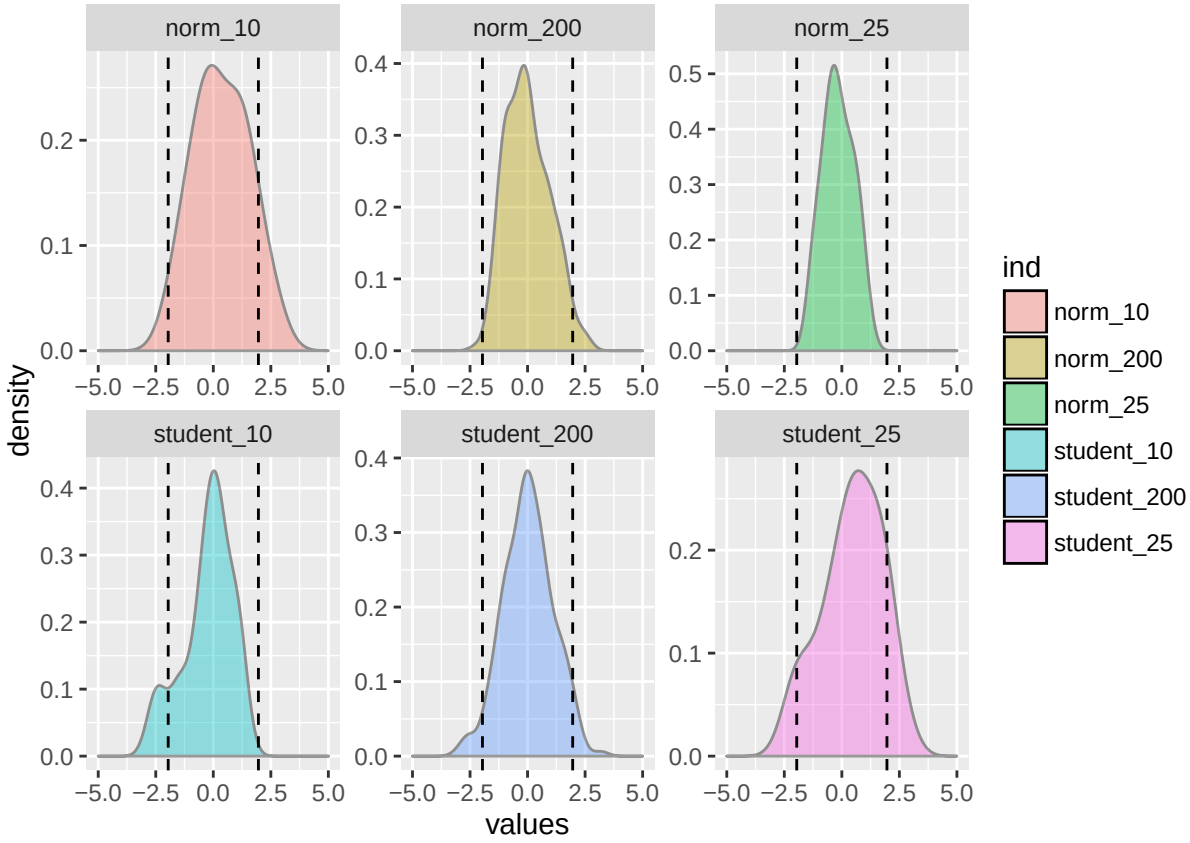
Mohammad Jawad Habib

March 18, 2016

Student-t

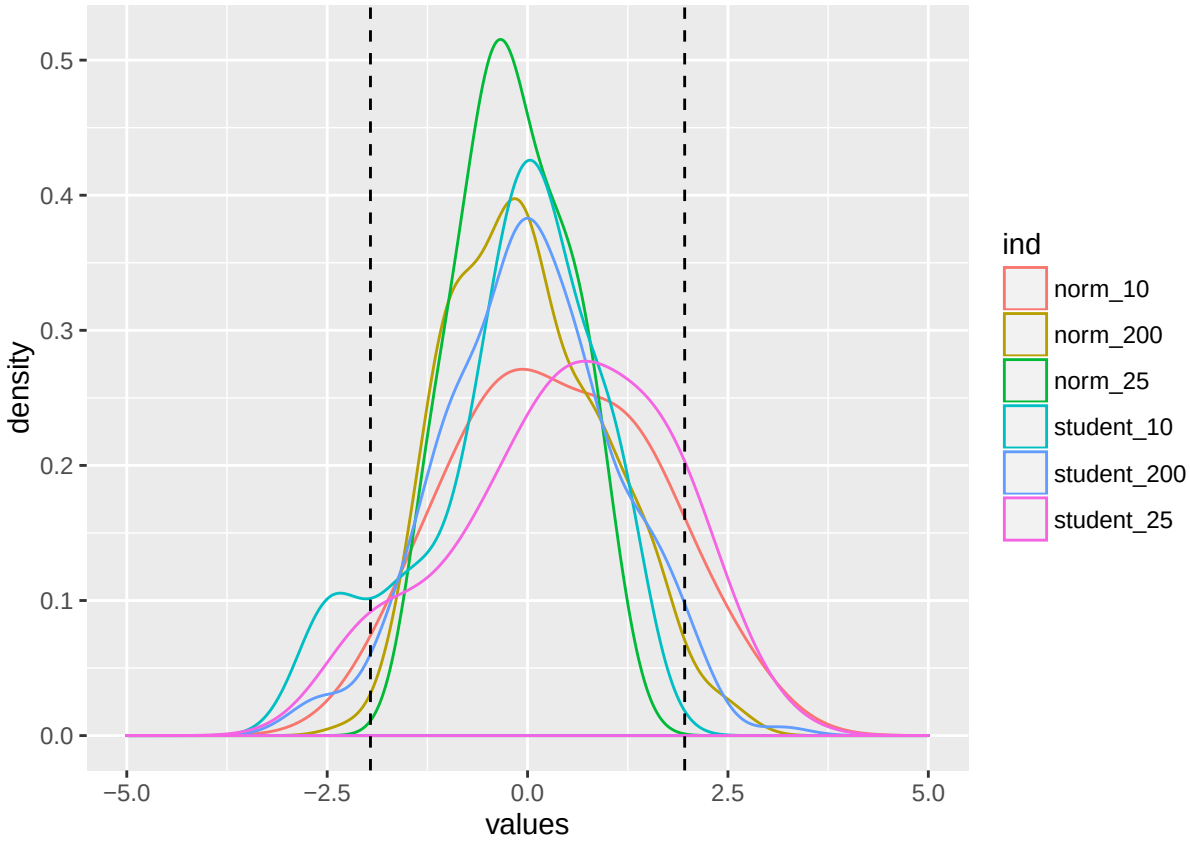
We will create graphs for normal and student-t distributions with sample sizes of 10, 25 and 200.

```
library(ggplot2)
library(plyr)

set.seed(0)

dat <- list(data.frame(rnorm(10)), data.frame(rnorm(25)), data.frame(rnorm(200)),
            data.frame(rt(10, df = 9)), data.frame(rt(25, df = 24)),
            data.frame(rt(200, df = 199)))

dat <- do.call(rbind.fill, dat)
colnames(dat) <- c("norm_10", "norm_25", "norm_200",
                   "student_10", "student_25", "student_200")
dat <- stack(dat)
dat <- dat[complete.cases(dat),]

dat.plot <- ggplot(data = dat, aes(x = values)) +
  geom_density(aes(group = ind, fill = ind), alpha = 0.4) +
  scale_x_continuous(limits = c(-5, 5)) +
  geom_vline(aes(xintercept = -1.96), linetype = "dashed") +
  geom_vline(aes(xintercept = 1.96), linetype = "dashed") +
  facet_wrap(~ ind, scales = "free")

dat.plot
```



We can draw all distributions on one plot too but it won't look too good.

```
ggplot(data = dat, aes(x = values)) +
  geom_density(aes(group = ind, colour = ind)) +
  scale_x_continuous(limits = c(-5, 5)) +
  scale_fill_brewer(palette = "Set1") +
  geom_vline(aes(xintercept = -1.96), linetype = "dashed") +
  geom_vline(aes(xintercept = 1.96), linetype = "dashed")
```



We can see from the graph above, with lines drawn at -1.96 and 1.96 that the different sample-size distributions have visibly different areas outside of the lines. The 1.96 critical value corresponds to an alpha level of 0.05 for a two-tailed test on a normal distribution because we are interested in both sides of the distribution.

Given the above knowledge, we can calculate the alpha level (or Type I error rate) for each distribution for critical values of -1.96 and 1.96 for a two-tailed test.

```
dat.norm <- dat[dat$ind %in% c("norm_10", "norm_25", "norm_200"), ]

means <- data.frame(tapply(dat.norm$values, dat.norm$ind, FUN = mean))
sds <- data.frame(tapply(dat.norm$values, dat.norm$ind, FUN = sd))
sizes <- data.frame(tapply(dat.norm$values, dat.norm$ind, FUN = length))

# our original samples had means of zero
# so z-scores for the sample means will be
z.scores <- means/(sds/sqrt(sizes))

dat.norm2 <- cbind(c("norm_10", "norm_200", "norm_25"),
              means, sds, z.scores)
colnames(dat.norm2) <- c("sample.type", "means", "stdev", "z.scores")
```

For normal distribution, the two-tailed alpha values at +/- 1.96 will be as follows.

```
2*pnorm(1.96, mean = dat.norm2$means[1:3], sd = dat.norm2$stdev[1:3], lower.tail = FALSE)
```

```
##     norm_10    norm_200     norm_25
## 0.184071419 0.041376116 0.001486761
```

For student-t distribution, the two-tailed alpha values at +/- 1.96 will be as follows. We'll assume there are 10-1, 25-1, and 200-1 degrees of freedom in our samples.

```
sample.typ <- c("student_10", "student_25", "student_200")
t.alphas <- pt(1.96, df = c(9, 24, 199), lower.tail = FALSE)

cbind(sample.typ, t.alphas)
```

```
##      sample.typ    t.alphas
## [1,] "student_10"  "0.0408222027302083"
## [2,] "student_25"  "0.0308530119112674"
## [3,] "student_200" "0.0256959169520801"
```

Bootstrapping

The function MedianBootstrap returns a vector of medians calculated for the samples taken from our input. It also returns a 95% confidence interval for the resulting vector of medians.

```
MedianBootstrap <- function(input.sample, NBS = 1000) {
  median.list <- replicate(NBS,
                           median(sample(input.sample, size = 30, replace = TRUE)))

  return(list(median.list, quantile(median.list, c(0.025, 0.975))))
}
```

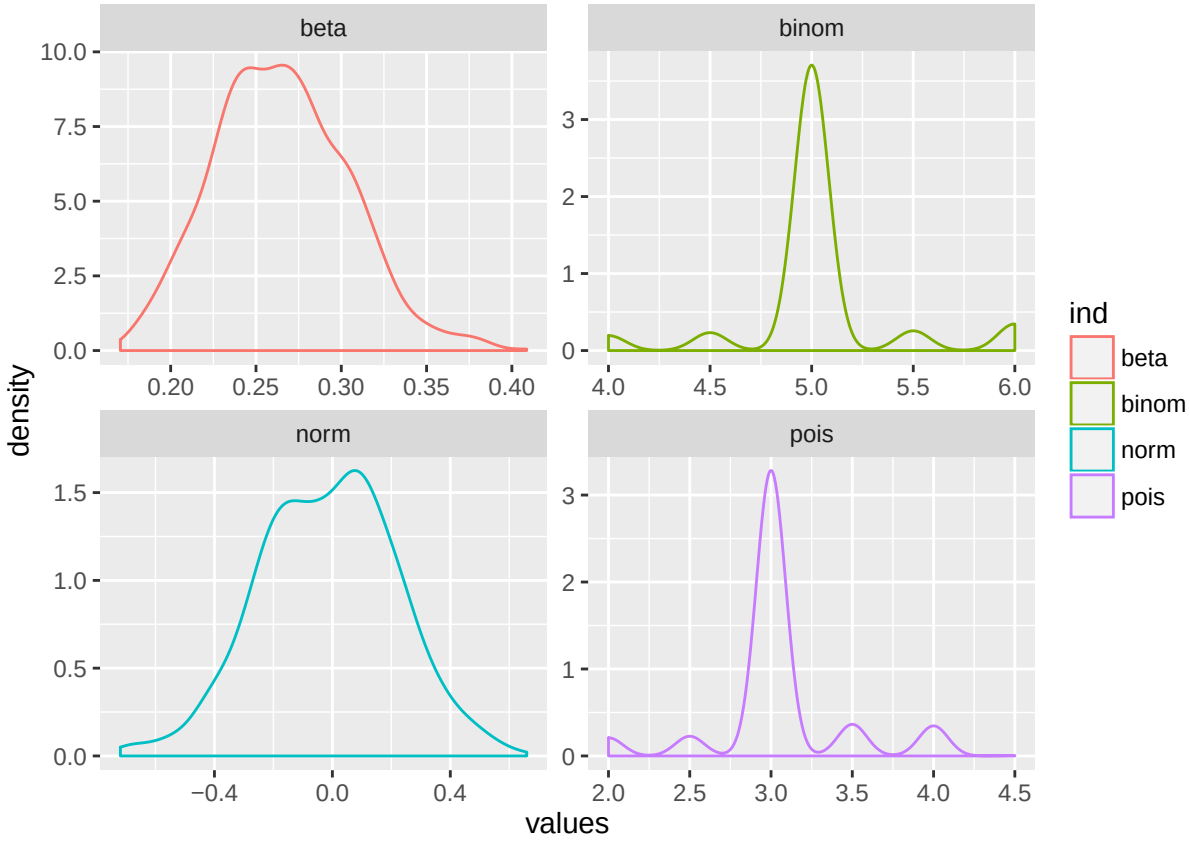
I was not sure if we were to return a 95% confidence interval for each of the samples taken inside the function or just for the overall median result so I did both.

```
MedianBootstrap2 <- function(input.sample, NBS = 1000) {
  median.list <- c()
  conf.min.list <- c()
  conf.max.list <- c()
  for(i in 1:NBS) {
    s <- sample(input.sample, size = 30, replace = TRUE)
    median.list <- append(median.list, median(s))
    conf.min.list <- append(conf.min.list, quantile(s, c(0.025)))
    conf.max.list <- append(conf.max.list, quantile(s, c(0.975)))
  }

  df <- data.frame(median.list, conf.min.list, conf.max.list)
  colnames(df) <- c("sample_median", "conf_int_min", "conf_int_max")

  return(df)
}
```

Let's use the first function to generate a few medians for various samples.

```
mb.rnorm <- MedianBootstrap(rnorm(1000))
mb.rpois <- MedianBootstrap(rpois(1000, pi))
mb.rbinom <- MedianBootstrap(rbinom(1000, 10, 0.5))
mb.rbeta <- MedianBootstrap(rbeta(1000,2,5))
```

```
mb <- data.frame(mb.rnorm[[1]], mb.rpois[[1]], mb.rbinom[[1]], mb.rbeta[[1]])
colnames(mb) <- c("norm", "pois", "binom", "beta")

mb.long <- stack(mb)
mb.plot <- ggplot(data = mb.long, aes(x = values)) +
  geom_density(aes(group = ind, colour = ind)) +
  facet_wrap(~ ind, scales = "free")
mb.plot
```



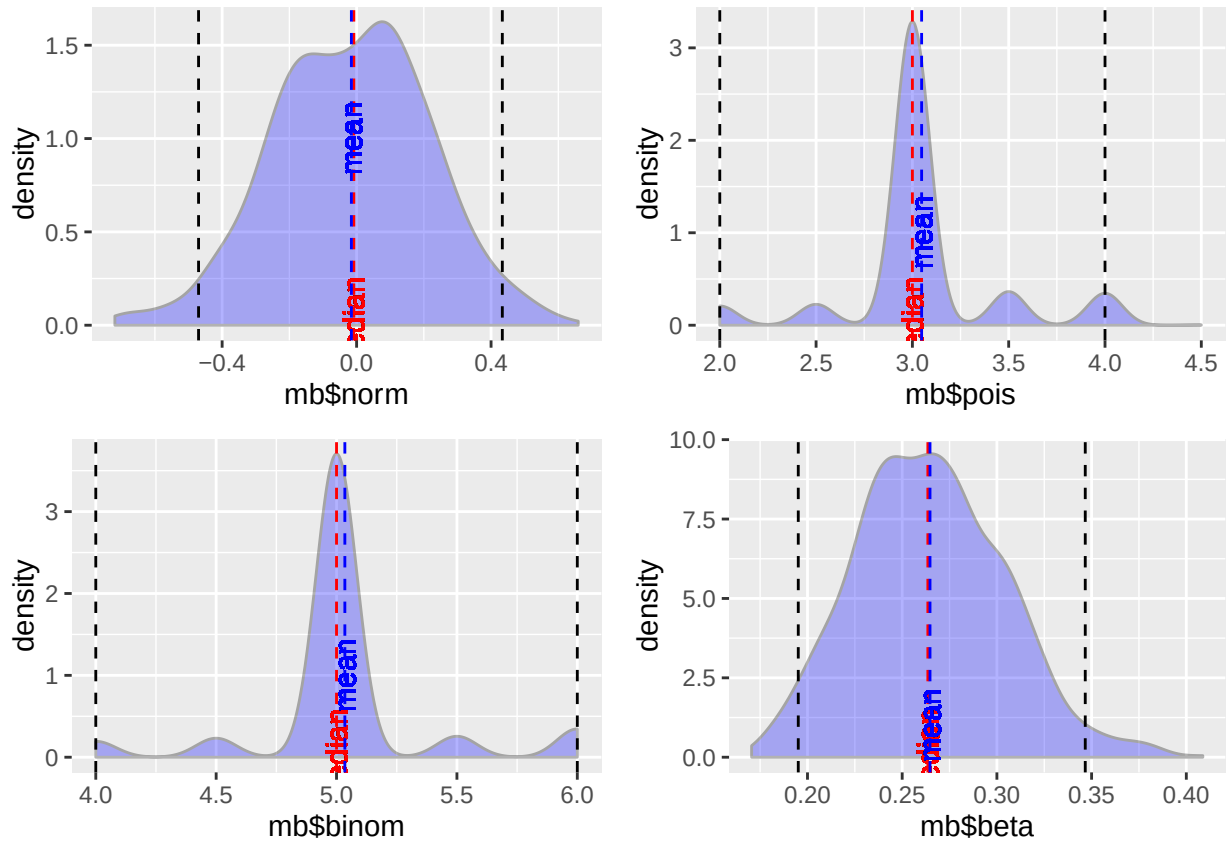
Since I don't know how to add individual medians, means and other x-intercept lines to individual facets, I will just create four graphs and combine them into one grid for display.

```
norm.plot <- ggplot(data = mb, aes(x = mb$norm)) + geom_density(fill = "blue", alpha = 0.3) +
  geom_vline(aes(xintercept = median(mb$norm)), linetype = "dashed", colour = "red") +
  geom_text(aes(x=median(mb$norm), label="median", y=0), colour="red", angle=90) +
  geom_vline(aes(xintercept = mean(mb$norm)), linetype = "dashed", colour = "blue") +
  geom_text(aes(x=mean(mb$norm), label="mean", y=1), colour="blue", angle=90) +
  geom_vline(aes(xintercept = mb.rnorm[[2]][1]), linetype = "dashed") +
  geom_vline(aes(xintercept = mb.rnorm[[2]][2]), linetype = "dashed")
# norm.plot

pois.plot <- ggplot(data = mb, aes(x = mb$pois)) + geom_density(fill = "blue", alpha = 0.3) +
  geom_vline(aes(xintercept = median(mb$pois)), linetype = "dashed", colour = "red") +
  geom_text(aes(x=median(mb$pois), label="median", y=0), colour="red", angle=90) +
  geom_vline(aes(xintercept = mean(mb$pois)), linetype = "dashed", colour = "blue") +
```

```
  geom_text(aes(x=mean(mb$pois), label="mean", y=1), colour="blue", angle=90) +
  geom_vline(aes(xintercept = mb.rpois[[2]][1]), linetype = "dashed") +
  geom_vline(aes(xintercept = mb.rpois[[2]][2]), linetype = "dashed")
# pois.plot

binom.plot <- ggplot(data = mb, aes(x = mb$binom)) + geom_density(fill = "blue", alpha = 0.3) +
  geom_vline(aes(xintercept = median(mb$binom)), linetype = "dashed", colour = "red") +
  geom_text(aes(x=median(mb$binom), label="median", y=0), colour="red", angle=90) +
  geom_vline(aes(xintercept = mean(mb$binom)), linetype = "dashed", colour = "blue") +
  geom_text(aes(x=mean(mb$binom), label="mean", y=1), colour="blue", angle=90) +
  geom_vline(aes(xintercept = mb.rbinom[[2]][1]), linetype = "dashed") +
  geom_vline(aes(xintercept = mb.rbinom[[2]][2]), linetype = "dashed")
# binom.plot

beta.plot <- ggplot(data = mb, aes(x = mb$beta)) + geom_density(fill = "blue", alpha = 0.3) +
  geom_vline(aes(xintercept = median(mb$beta)), linetype = "dashed", colour = "red") +
  geom_text(aes(x=median(mb$beta), label="median", y=0), colour="red", angle=90) +
  geom_vline(aes(xintercept = mean(mb$beta)), linetype = "dashed", colour = "blue") +
  geom_text(aes(x=mean(mb$beta), label="mean", y=1), colour="blue", angle=90) +
  geom_vline(aes(xintercept = mb.rbeta[[2]][1]), linetype = "dashed") +
  geom_vline(aes(xintercept = mb.rbeta[[2]][2]), linetype = "dashed")
# beta.plot

require(gridExtra)
```

```
## Loading required package: gridExtra
```

```
grid.arrange(norm.plot, pois.plot, binom.plot, beta.plot)
```



We can visually see that the distributions above do not look normal. This might be due to the sample size of our input to MedianBootstrap. We can run the shapiro-wilk test.

```
sapply(mb, shapiro.test)
```

```
##           norm                           pois
## statistic 0.9976122                      0.7009164
## p.value   0.1553136                      8.381782e-39
## method    "Shapiro-Wilk normality test" "Shapiro-Wilk normality test"
## data.name "X[[i]]"                       "X[[i]]"
##           binom                          beta
## statistic 0.6430748                      0.9926412
## p.value   2.512362e-41                   7.199832e-05
## method    "Shapiro-Wilk normality test" "Shapiro-Wilk normality test"
## data.name "X[[i]]"                       "X[[i]]"
```

We see from above that the median seems to be normally distributed for the sample drawn from `rnorm`. The other three are not normal.

Numerical Optimization

We will use `optim` to numerically find the kth root of a positive number. Optim will minimize the `objf` function for a given `r` and `number`. We will stop when the default max iterations run out.

Given more time, I would write my own ln function using a Taylor Series expansion and my own integer exponentiation function for use in the ln.

```
objf <- function(r, number, k) {
  # we will suppress warnings because
  # optim might try to calculate
  # log of a negative number
  suppressWarnings(abs(sum(number - exp(k*log(r)))))
}

rootk <- function(number, k, start = NULL) {
  # we will only deal with positive bases
  if (!all(number > 0)) {
    stop("Negative number provided as input.")
  }

  if (length(number) != length(start) & !is.null(start)) {
    stop("Number and start must have the same length")
  }

  # make a guess for r if not given as input
  if (is.null(start)) {
    start <- rep.int(1, length(number))
  }
  start <- abs(start)

  # call optim
  root <- suppressWarnings(sapply(seq_along(number), function(i) {
    optim(start[[i]], objf, number = number[[i]], k = k, method = "BFGS",
          control = list(reltol = sqrt(.Machine$double.eps)))$par
    }))
  return(root)
}
```

Let's call this for a few values.

```
rootk(c(100, 1000, 9, 27), 3, c(1, 2, 3, 4))
```

```
## [1]  4.641589 10.000000  2.080083  3.000000
```

```
# compare with ^ operator
print(c(100^(1/3), 1000^(1/3), 9^(1/3), 27^(1/3)))
```

```
## [1]  4.641589 10.000000  2.080084  3.000000
```

```
rootk(c(100, 1000, 9, 27), 2, c(10, 20, 3, 5))
```

```
## [1] 10.000000 31.622777  3.000000  5.196152
```

```
# compare with ^ operator
print(c(100^(1/2), 1000^(1/2), 9^(1/2), 27^(1/2)))
```

```
## [1] 10.000000 31.622777  3.000000  5.196152
```

```r
rootk(8, 3)
```

```
## [1] 1.999999
```

```r
# compare with ^ operator
print(c(8^(1/3)))
```

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

```r
rootk(8, -5)
```

```
## [1] 0.6597578
```

```r
# compare with ^ operator
print(c(8^(-1/5)))
```

```
## [1] 0.659754
```

```r
rootk(c(27, 50, 75), -3, c(1, 1, 1))
```

```
## [1] 0.3333385 0.2714487 0.2371263
```

```r
# compare with ^ operator
print(c(27^(-1/3), 50^(-1/3), 75^(-1/3)))
```

```
## [1] 0.3333333 0.2714418 0.2371262
```

We can see from the above tests that the results are pretty close.