STAT 33A Homework 6

CJ HINES (3034590053)

Nov 19, 2020

This homework is due **Nov 19, 2020** by 11:59pm PT.

Homeworks are graded for correctness.

As you work, write your answers in this notebook. Answer questions with complete sentences, and put code in code chunks. You can make as many new code chunks as you like.

Please do not delete the exercises already in this notebook, because it may interfere with our grading tools.

You need to submit your work in two places:

- Submit this Rmd file with your edits on bCourses.
- Knit and submit the generated PDF file on Gradescope.

If you have any last-minute trouble knitting, **DON'T PANIC**. Submit your Rmd file on time and follow up in office hours or on Piazza to sort out the PDF.

Background: Discrete Distributions

A *discrete probability distribution* is a table of mutually exclusive outcomes and their associated probabilities.

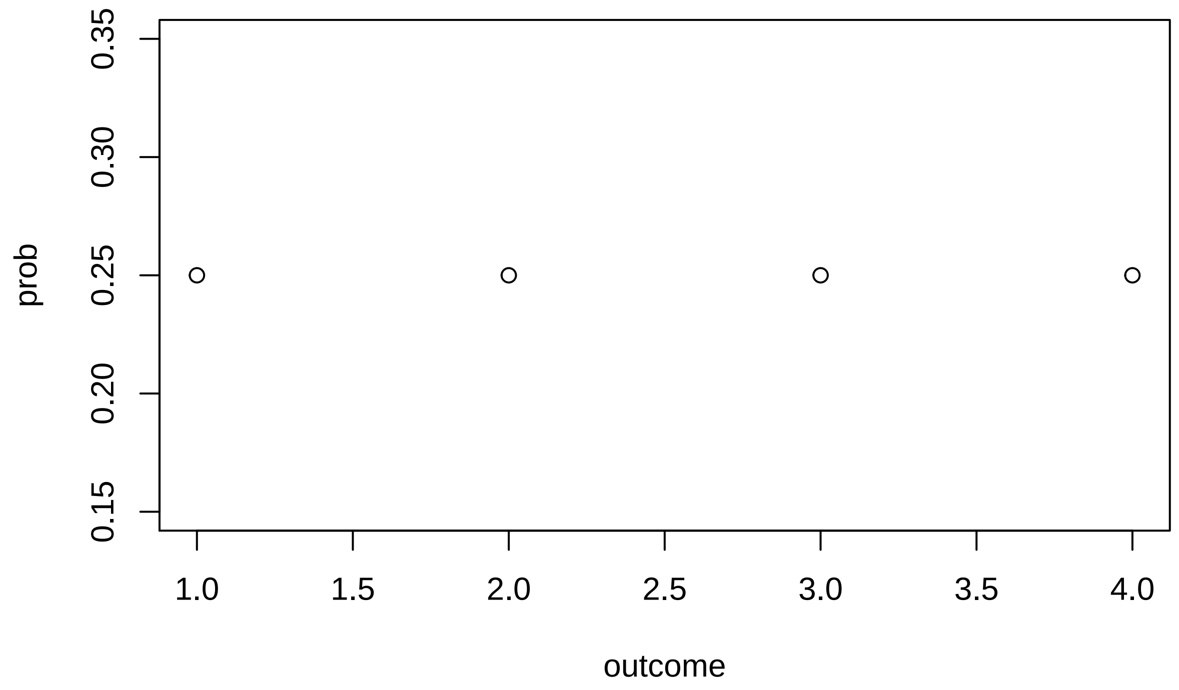
For example, the *discrete uniform distribution* assigns equal probability to each outcome. So the discrete uniform distribution on the integers 1-4 is:

Outcome	Probability
1	0.25
2	0.25
3	0.25
4	0.25

A fair coin toss is another instance of a discrete uniform distribution, where each of the two outcomes (heads and tails) has probability 0.5.

You can plot the distribution above with the code:

```r
outcome = seq(1, 4)
prob = rep(0.25, 4)
plot(outcome, prob)
```



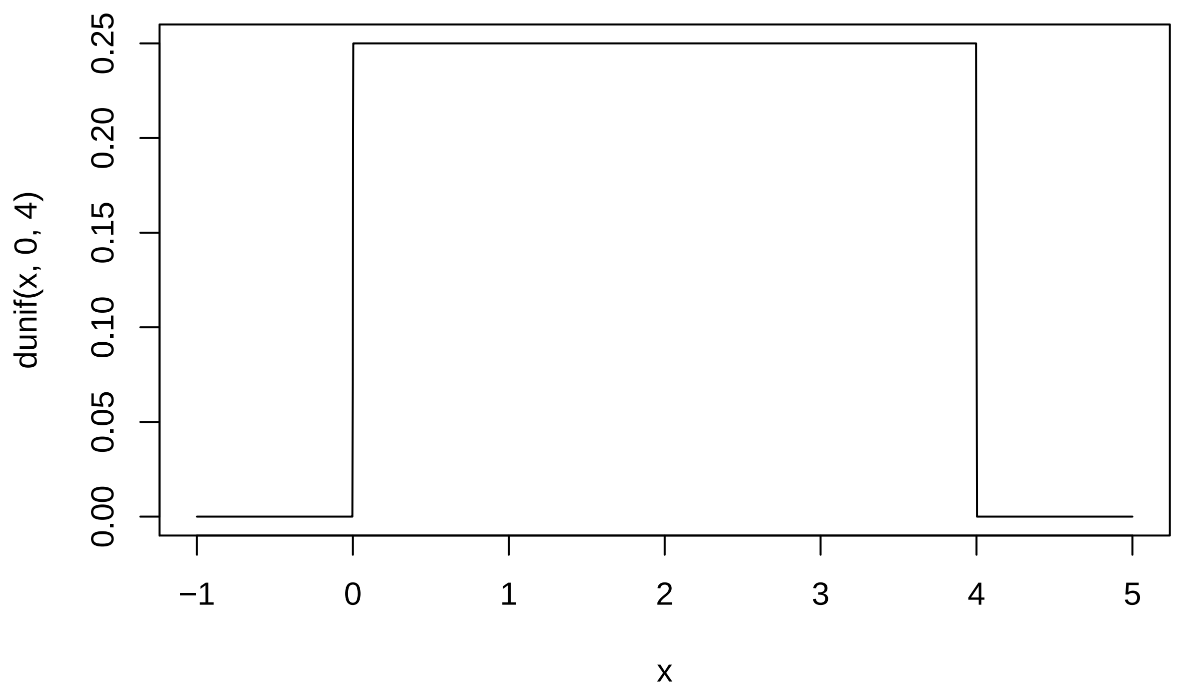
Because the probabilities are all equal, a plot of a uniform distribution is always flat, regardless of the number of outcomes.

Background: Continuous Distributions

The idea of a uniform distribution can also be extended to continuous values. For example, suppose we want to select a decimal value between 0 and 4 (inclusive), with equal probability for any value in the interval. This is a *countinuous* uniform distribution.

You can represent this idea visually with the plot:

```
curve(dunif(x, 0, 4), -1, 5, n = 1000)
```



This is a density plot, similar to the density plots we learned about before. The curve is called the *density curve* (or simply the density) for the distribution.

In a density plot, the y-axis no longer represents probability. Instead, the probability of points in any interval is the total area between the curve and the line $y = 0$ over the interval. For instance, in the distribution above, the probability of the outcome being a point between 1 and 2 (inclusive) is 0.25.

R provides functions to compute the density of well-known distributions. For example, the code above uses the `dunif` function (pronounced "dee you-niff" and short for "density uniform"). The function has parameters to control the interval the distribution covers.

R also provides functions to generate random samples from well-known distributions. For example, the `runif` function (pronounced "are you-niff" and short for "random uniform") generates a random sample from a continuous uniform distribution. You can sample 10 values from the continuous uniform distribution from -1 to 1 with the code:

```r
runif(10, -1, 1)
```

```
##  [1] -0.4821973  0.1015996 -0.2418831 -0.4075129 -0.1959255 -0.1815559
##  [7] -0.8346138 -0.6757137 -0.1610089 -0.1925333
```

Since these values are sampled randomly, they will be different each time you run the code.

Exercise 1

The *Gaussian* or *normal* distribution is a continuous probability distribution that plays an important role in statistics and the natural sciences. The density of the Gaussian distribution is often described as a "bellshaped curve". The distribution is typically parameterized by its mean (center) and standard deviation (how spread out it is).

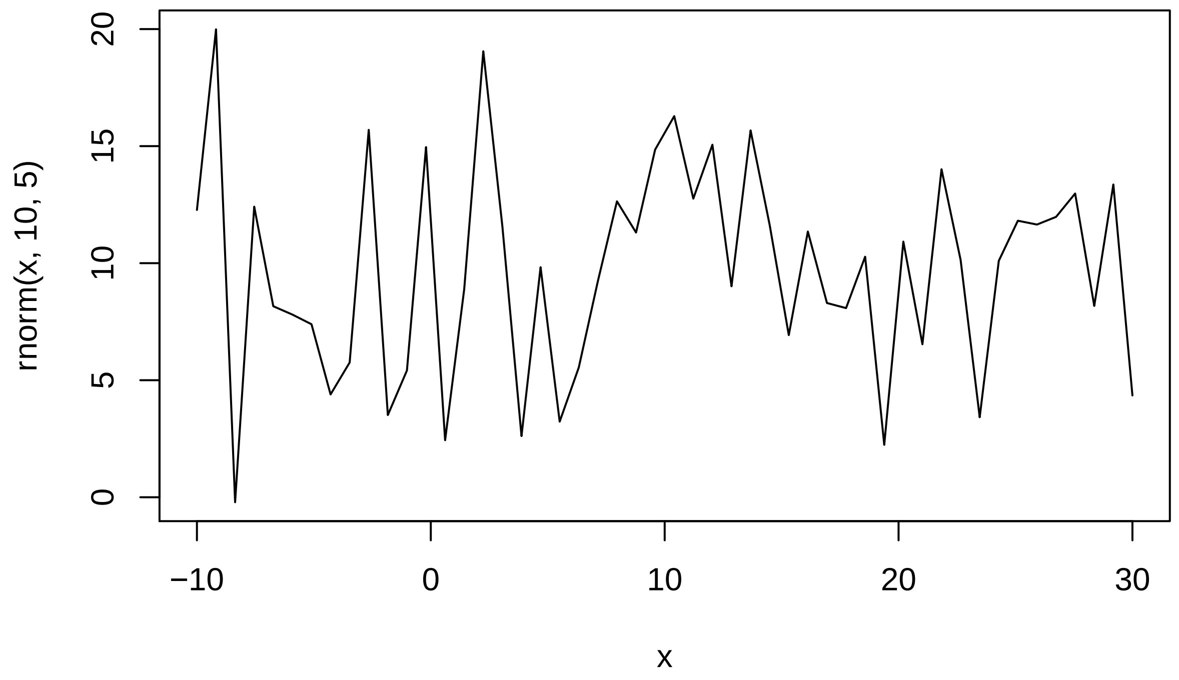
The R functions `dnorm` and `rnorm` for the Gaussian distribution are analogous to the `dunif` and `runif` functions for the uniform distribution.

1. Use the `curve` function to make a density plot of the Gaussian distribution with mean 10 and standard deviation 5. Set the x-axis to span from -10 to 30.
2. Skim the documentation for `rnorm`. Then create a variable called `samp` that contains 10 random values from a normal distribution with mean 10 and standard deviation 5.
3. Compute the mean and standard deviation of `samp`. Comment briefly on how much these differ from the true mean 10 and true standard deviation 5.

   What happens if you run your code a second time? Are the sample mean and standard deviation the same? Are the sampled points the same?

YOUR ANSWER GOES HERE:

```
#1
curve(rnorm(x, 10, 5), -10, 30, n = 50)
```



```
#2
samp = rnorm(n = 10, 10, 5)

#3
samp
```

```
##  [1] 13.135555 11.742934 12.896584  9.296142  9.096588  7.611019 10.032925
##  [8]  6.964178  6.909902  8.774550
```

```r
mean(samp)
```

```
## [1] 9.646038
```

```r
sd(samp)
```

```
## [1] 2.292339
```

> The mean of samp is 10.32 which is a little more than the mean of 10. The standard deviation of samp is 4.99 which is about a little less than the standard deviation of 5. When I ran the code a second time, the points, mean, and sd were different.

Random Number Generation

R's functions for random sampling use a *pseudo-random number generator* (PRNG). The numbers produced by a PRNG are not truly random, but satisfy conditions that make them close enough to random for most scientific computing tasks.

An advantage of PRNGs is that they are deterministic. Given the same initial parameter, called a **seed**, a PRNG will generate the same sequence of "random" numbers every time. When you start R, the seed is automatically assigned value from your computer that varies (e.g., the system time in milliseconds). This ensures that you get a different sequence of random numbers in each R session.

You can the PRNG seed with the `set.seed` function. Setting the seed makes your results reproducible. That is, other people can run your code and get the same results even if the code generates random numbers.

Generally, you should only set the seed once at the beginning of an R script or notebook, because it affects all subsequent calls that use the PRNG.

Exercise 2

1. Set the seed to 93.
2. Create a variable called `samp` that contains 1000 random values from a normal distribution with mean 10 and standard deviation 5.
3. Compute the mean and standard deviation of `samp`.

   What happens if you run your code a second time? Are the sample mean and standard deviation the same? Are the sampled points the same? Explain how step 1 affects this.
4. Compute the number of values in `samp` that are greater than 8. *Hint: there is a fast, simple way to count `TRUE` values using implicit coercion.*

YOUR ANSWER GOES HERE:

```r
#1
set.seed(93)

#2
samp = rnorm(1000, 10, 5)

#3
head(samp)
```

```
## [1]  8.390873  6.659145 13.604346 14.659495 14.200569  4.199102
```

```r
mean(samp)
```

```
## [1] 10.1505
```

```r
sd(samp)
```

```
## [1] 5.127733
```

```r
#4
x = samp
sum((x > 8) == TRUE)
```

```
## [1] 668
```

3. The mean of samp is 10.15 which is a little above the actual mean. The standard deviation of samp is 5.13 which is a little above the actual standard deviation. When I ran the code a second time, the points, mean, and sd were different.

Rejection Sampling

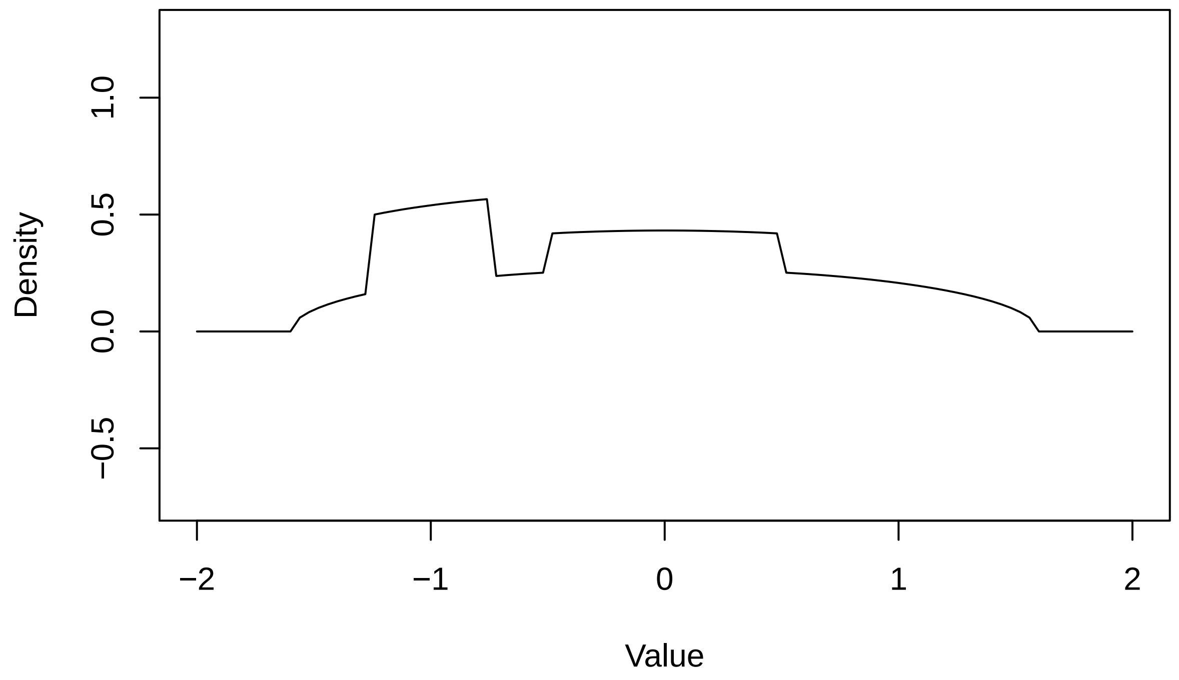
What if you want to sample from a distribution that's not well-known?

For example, suppose you want to sample from this distribution on -1.6 to 1.6:

```r
dplat = function(x) {
  y = numeric(length(x))
  i = -1.6 < x & x < 1.6
  y[i] = sqrt(2.56 - x[i]^2) + dunif(x[i], -1.25, -0.75) +
    dunif(x[i], -0.5, 0.5)

  y / 6.021239
}

curve(dplat, -2, 2, xlab = "Value", ylab = "Density", asp = 1)
```



Let's call this distribution the "plateau" distribution, since it has two plateaus.

One way to sample from distributions that are not well-known by using a statistical technique called *rejection sampling.*

The idea is to choose a rectangle that completely encloses the target density curve, and then uniformly sample points within the rectangle. If a point falls below the density curve, then the point is accepted and its x-coordinate is a new sample value. If a point falls above the density curve, then it is rejected (and discarded). This produces the correct distribution because relatively more points will be accepted in places where the density curve is taller.

The bottom side of the enclosing rectangle should always be on the line `y = 0`.

The exact steps in rejection sampling are:

1. Randomly sample a uniform x coordinate in the rectangle.
2. Randomly sample a uniform y coordinate in the rectangle.
3. Test whether the y coordinate is below the target density curve. If it is, save the x coordinate as a sample value. If it isn't, discard the x coordinate.
4. Repeat steps 1-3 until the desired number of sample values is reached.

Exercise 3

What's an appropriate enclosing rectangle for the plateau distribution's density curve?

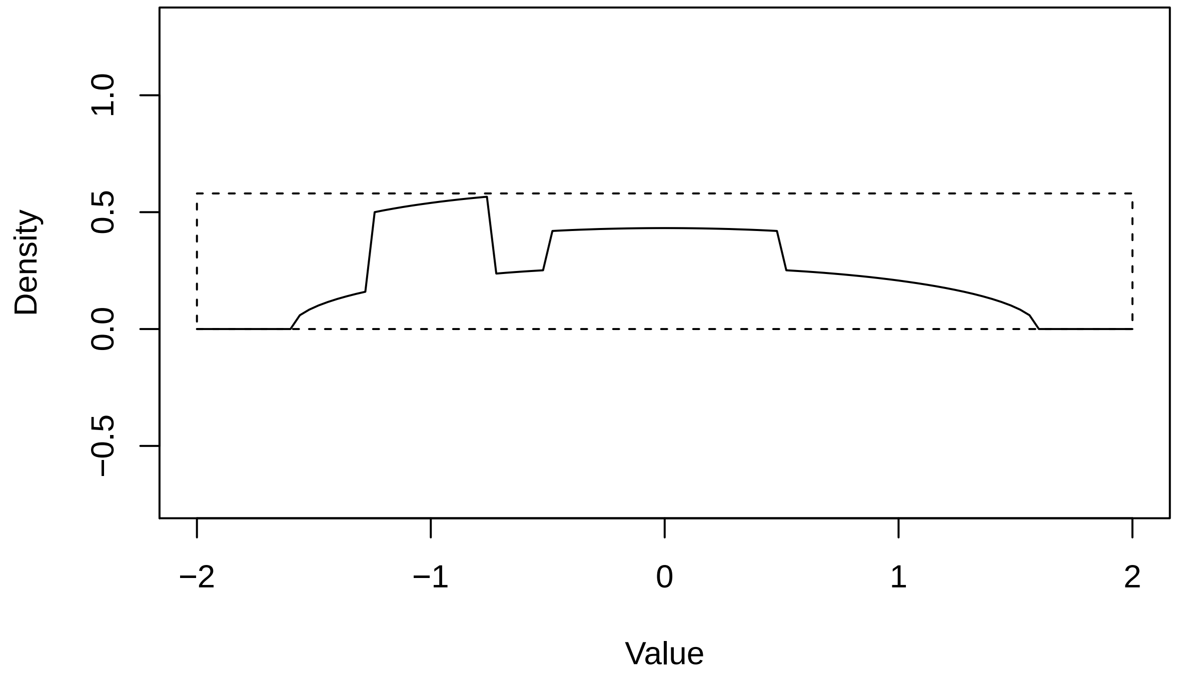
Use the call to `curve` above, followed by a call to `rect` (a base R plotting function), to plot the density curve with the enclosing rectangle superimposed. Use the argument `lty = "dashed"` in the call to `rect` so that the rectangle is visually distinct from the curve.

There are many possible answers to this question, but smaller rectangles are more efficient for rejection sampling.

YOUR ANSWER GOES HERE:

> An appropriate enclosing rectangle for this plateau distribution's density curve is `-2, 0, 2, 0.3` which is 0.3 tall and 1 wide from -2 to 2.

```r
curve(dplat, -2, 2, xlab = "Value", ylab = "Density", asp = 1)
rect(-2, 0, 2, 0.58, lty = "dashed")
```



Exercise 4

Write a function `rplat` that uses rejection sampling to return a vector of `n` samples from the plateau distribution. Your function should have a parameter `n` with default argument `100`.

Use the `dplat` function provided above as the target density curve.

Test that your function runs without error for `n` equal to 10, 100, and 1000.

YOUR ANSWER GOES HERE:

```r
rplat = function(n = 100) {
  count = 1
  no = numeric(0)
  while (count <= n) {
      sampley = runif(1, 0, 0.59)
      samplex = runif(1, -2, 2)
      if (sampley <= dplat(samplex)) {
         no = c(no, samplex)
         count = count + 1
      }
   }
   no
}

rplat(10)
```

```
##  [1] -0.790224543  0.169584764 -0.003655248  0.900411602 -0.384958450
##  [6]  1.474308218 -0.290482983  0.402946021 -1.177398396 -0.439391169
```

```
#rplat(100)
#rplat(1000)
```

Exercise 5

1. Use your `rplat` function to sample 100 points from the plateau distribution and plot the estimated density. Make sure to call `set.seed` first so that your result is reproducible.

   Hint: You can use base R graphics to plot an estimated density curve for a sample `x` *with* `plot(density(x), asp = 1, xlim = c(-3, 3))`*.*

2. How does the shape of your estimated density curve compare to the shape of the actual density curve for the plateau distribution (see above)?

3. Repeat Parts 1-2 with a sample of 1,000,000 points. Comment on how the new estimated density curve compares to the one from Part 1.

   Note: Sampling this many points may take 10-60 seconds. Anything substantially longer means your function is doing something inefficient.

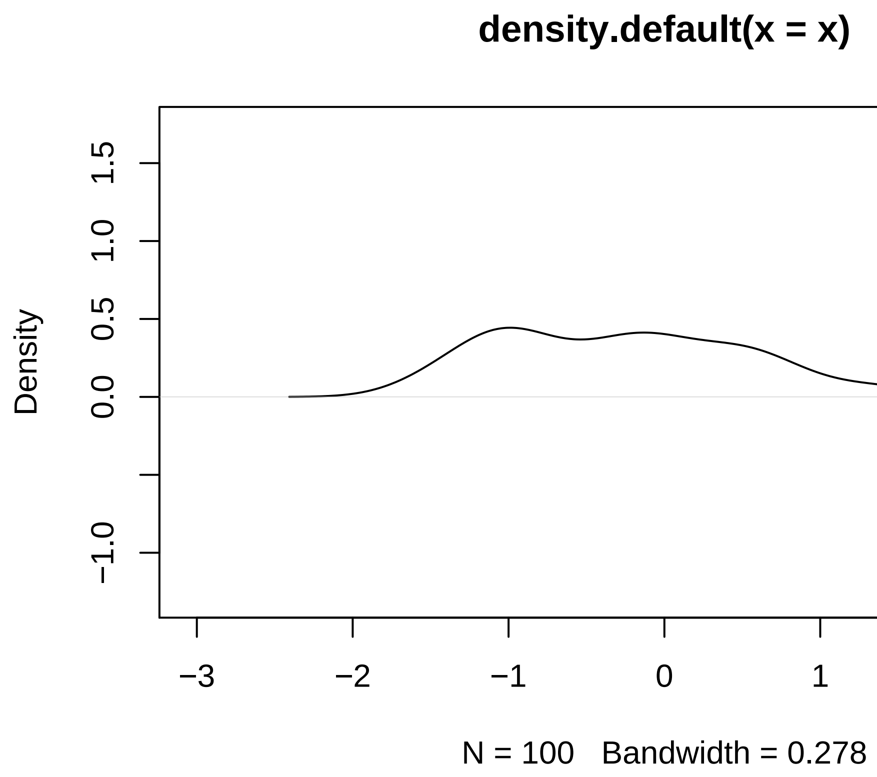
4. Based on the sample from Part 3, what are the approximate mean and standard deviation of the plateau distribution?

YOUR ANSWER GOES HERE:

2. The shape of my density curve below (4.1) is a more smooth, general shape, and is not very similar to the plateau distribution.

4. Based on the sample in (4.3), the approximate mean is 0 and standard deviation is 1.

```
#1
set.seed(5)
x = rplat(100)
plot(density(x), asp = 1, xlim = c(-3, 3))
```



```
#3
#set.seed(10)
#y = rplat(1000000)
#plot(density(y), asp = 1, xlim = c(-3, 3))
```