The method described here works well for simulations that can be summarized by a single number. If you need to store more information for each simulation, things will need to be adjusted.

As an example, let’s suppose we want to use simulations to see that \( X \sim \text{Norm}(\mu, \sigma) \)

**Step 1: Write a function that does one simulation.**

As a first step, you might outline your simulation function focusing on the arguments (things you might adjust in the simulation) and the main steps of the simulation.

```r
oneSim <- function(n, mu, sigma) {
  # 1. sample n values from Norm(mu, sigma) population
  # 2. compute and return the mean of the simulated data
    
#} 
```

Now we can fill in the outline

```r
oneSim <- function(n, mu, sigma) {
  # 1. sample n values from Norm(mu, sigma) population
  x <- rnorm(n, mu, sigma)
  # 2. compute and return the mean
  mean(~ x)
}
```

At this point you can test your function to see that it appears to be working.

```r
# should get value near 500
oneSim(10, 500, 100)
```

```r
## [1] 572.5825
```

```r
# second time should give a slightly different result
oneSim(10, 500, 100)
```

```r
## [1] 545.4686
```

**Step 2: Set up a parameter space for multiple simulations**

Here we mean parameters of the simulation, which may or may not be parameters of the distribution. `expand.grid()` is very useful for this. It returns a data frame with every combination of its arguments on a separate row.
SimSetUp <-
expand.grid(
  n = c(10, 25),
  mu = c(10, 100),
  sigma = c(1, 10)
)

## n mu sigma
## 1 10 10 1
## 2 25 10 1
## 3 10 100 1
## 4 25 100 1
## 5 10 10 10
## 6 25 10 10
## 7 10 100 10
## 8 25 100 10

Step 3: Run the simulation for each setting of the parameters

Now we can run all 8 simulations in one go using the `mutate()` function from the `dplyr` package.

```r
SimSetUp %>%
  mutate(x.bar = oneSim(n=n, mu=mu, sigma = sigma))
```

## n mu sigma x.bar
## 1 10 10 1 57.55033
## 2 25 10 1 57.55033
## 3 10 100 1 57.55033
## 4 25 100 1 57.55033
## 5 10 10 10 57.55033
## 6 25 10 10 57.55033
## 7 10 100 10 57.55033
## 8 25 100 10 57.55033

Hmm, that’s not working

We have a problem here. We should have the same result in every row, but we do. (We also didn’t save our results, but since they aren’t good anyway, that’s not a big loss just yet.) The `mutate()` function requires a vectorized function – one that can process multiple values for its arguments at once and return multiple values. The problem is that our simulation function is not vectorized.

```r
oneSim(n=c(10, 25), mu = c(10, 100), sigma = c(2, 5))
```

## [1] 57.28772

Vectorizing your function

Fortunately, this is really easy to fix:
oneSimV <- Vectorize(oneSim)
oneSimV(n=c(10, 25), mu = c(10, 100), sigma = c(2, 5))


# the above is equivalent to
oneSim(n=10, mu = 10, sigma = 2)  # use first value of each

## [1] 10.18651

oneSim(n=25, mu = 100, sigma = 5)  # use second value of each

## [1] 101.2701

Because R does recycling, we can also do things like

oneSimV <- Vectorize(oneSim)
oneSimV(n=c(10, 25), mu = c(10, 100), sigma = 2)

## [1] 9.461104 100.267383

# the above is equivalent to
oneSim(n=10, mu = 10, sigma = 2)  # use first value of each

## [1] 10.5064

oneSim(n=25, mu = 100, sigma = 2)  # use second value of each

## [1] 99.54073

Note: there are options to Vectorize() that let you vectorize only some of the arguments. In some situations that can be very important.

Let’s try that again with a vectorized simulator function

SimSetUp %>%
  mutate(x.bar = oneSimV(n=n, mu=mu, sigma = sigma))

##    n mu sigma x.bar
## 1   1 10   10     1 9.590007
## 2   2 25   10     1 9.891578
## 3   3 10  100   100.082819
## 4   4 25 1000  100.228172
## 5   5 10 1000   5.479423
## 6   6 25 1000   9.030942
## 7   7 10 1000  94.595006
## 8   8 25 1000 100.639398

That looks much better.
Running multiple simulations at each setting

Now we want to do 1000 of each of these (and save the results). Here are two ways to do that.

```r
Results1 <-
  do (1000) * {
    SimSetUp %>%
      mutate(x.bar = oneSimV(n=n, mu=mu, sigma = sigma))
  }
head(Results1)
```

```r
## n mu sigma x.bar .row .index
## 1 10 10 1 9.991704 1 1
## 2 25 10 1 10.334676 2 1
## 3 10 100 1 100.493632 3 1
## 4 25 100 1 99.728777 4 1
## 5 10 10 10 5.132639 5 1
## 6 25 10 10 11.695477 6 1
```

```r
SimSetUp2 <-
  expand.grid(
    n = c(10, 25),
    mu = c(10, 100),
    sigma = c(1, 10),
    sim = 1:1000
  )
Results2 <-
  SimSetUp2 %>%
    mutate(x.bar = oneSimV(n=n, mu=mu, sigma = sigma))
head(Results2)
```

```r
## n mu sigma sim x.bar
## 1 10 10 1 1 10.462183
## 2 25 10 1 1 9.933109
## 3 10 100 1 1 100.233344
## 4 25 100 1 1 99.765019
## 5 10 10 10 1 13.397015
## 6 25 10 10 1 11.302127
```

Step 4: Summarize the results

Now we can use numerical or graphical summaries to inspect our results.

```r
freqpolygon(~ x.bar | mu + sigma, groups = n, data = Results1)
```
freqpolygon((x.bar - mu) / (sigma/sqrt(n))) | mu + sigma, 
groups = n, data = Results1)

mean(- x.bar | n + mu + sigma, data = Results1)

## 10.10.1 25.10.1 10.100.1 25.100.1 10.10.10 25.10.10
## 9.993282 9.992578 100.016522 99.996921 10.149178 10.113141
## 10.100.10 25.100.10
## 100.145972 100.045056
sd(~ x.bar | n + mu + sigma, data = Results)

## 10.10.1 25.10.1 10.100.1 25.100.1 10.10.10 25.10.10 10.100.10
## 0.2995625 0.1960524 0.3132013 0.2062556 3.2865907 1.9791106 3.0919232
## 25.100.10
## 2.0192448

The plant density example

Here is how this might look for the plant density problem

```r
simTestStat <- Vectorize(function( n, lambda = 1, area = 1) {
  # simulate data according to the model
  y <- rweibull(n, 2, 1 / sqrt(lambda * pi))
  # compute the MLE and return it
  n / sum(pi * y^2)
})

# set up some parameters
SimSetUp <- expand.grid(
  A = c(10, 100),
  lambda = c(.1, 1),
  n = c(100, 1000),
  simid = 1:1000
)

Results <- SimSetUp %>%
  mutate(testStat = simTestStat(n = n, lambda = lambda, area = A))

# look at the results
freqpolygon(~ (testStat / lambda) | paste("lambda =", lambda) + paste("area =", A),
groups = paste("sample size =", n),
data = Results,
auto.key = TRUE)
```
You can, of course, be much cleverer about setting up the parameter space to simulate and creating interesting summaries of the results.

**Side Notes**

**Setting the random seed**

If you want to get the same results each time you run your simulation (useful for debugging, or if you need to discuss the results in text and don’t want them to keep changing), you can set the random seed with

```r
set.seed(1234) # choose any number you like here.
```

**Caching results**

RMarkdown provides the option of caching results so that they don’t have to be rerun each time you knit the document. This can greatly speed things up if your simulations are time-consuming. Simply put `cache = TRUE` into either an individual chunk’s options or near the top of the document (to set the default for all chunks).

```r
# put this near the top of your RMarkdown document
classic(require(knitr)
classic$opts_chunk$set(
classic  fig.align = "center", # center the figures
classic  cache = TRUE, # cache results
classic  tidy = FALSE, # display code as typed
classic  size = "small" # slightly smaller font for code
)
```

When the code in a code chunk changes, the code will be re-executed. Otherwise, cached data is used to save time.