Jeremy Kang
GTID: 903136703
GT Name: jkang315

#1. One common theme I found unique to using R was the suggestion to avoid using loops. This seems counterintuitive to those who have some familiarity with other programming languages. This isn't specific to a data analytical/visual language (i.e., R) either, as languages like python still retains efficiency when running loops.

```
a = 1:10
# compute sum of squares using a for loops
c=0
for (e in a) c = c + e^2
 c
 ## [1] 385
 # same operation using vector arithmetic
sum(a^2)
## [1] 385
 # time comparison with a million elements
a = 1:1000000; c = 0
system.time(for (e in a) c = c+e^2)
 ## user system elapsed
 ## 0.381 0.006 0.386
system.time(sum(a^2)
) ## user system elapsed
 ## 0.007 0.003 0.010
```

The code above shows that the CPU time is about 54 times faster using the built-in function for computing the sum of squares as opposed to a custom function using a for loop. This is quite significant and can especially make a impactful difference when working with large data sets.

The reading suggest an interesting alternative built-in function to using loops: sapply().

```
a = seq(0, 1 ,length.out = 10) b=0 c=0 for (e in a) {
b = b + exp(e) }
b
## [1] 17.33958
c = sum(sapply(a, exp))
c
## [1] 17.33958
# sapply with an anonymous function f(x)=exp(x^2)
```

```
sum(sapply(a, function(x) {return(exp(x^2))}))
## [1] 15.07324
# or more simply
sum(sapply(a, function(x) exp(x^2)))
## [1] 15.07324
```

After playing with some of R's built-in functions, I came up with an sapply() solution to the sum of squares problem from before. Note, it's essentially the same as using the built in function, however, since sapply() returns a list of the same length as the given input (a = 1:10), I needed to find a way to select a specific element. Below is the code:

```
round(system.time(sapply(sum(a^2), `[[`, 1)), digits=5)
##  user  system elapsed
##  0.001   0.000   0.000
```

This is quite fast, which isn't surprising as it uses the same built-in function but just uses sapply() and selects the first element.

#5. I ran into many issues trying to increase the n (iterations) to an appropriate amount. After changing the recommended expressions to 500,000, playing around with the system stack size limits, and many other solutions that didn't lead to much success, the largest N my machine ran was to 1705 iterations. This includes counts of 0's. My best guess is that these 0's represent the for loop through the N's for each of the functions and **not** representing execution times for the three functions themselves.

After removing 0's, the observations went down to 627, which seemed really low compared to what other classmates have posted about on Piazza/Slack. The cleanest plot I found was of N=1130 with 0's and N=416 without 0's. Thus, I will include multiple plots for my analysis to hopefully give the best report possible.

Summary Table of For Loop, Recursive, and Built-in Execution Times		
timeFor	timeRec	timeR
Min. :0.0040	Min. :0.008	Min. :0.001000
1st Qu.:0.1722	1st Qu.:0.896	1st Qu.:0.001000
Median :0.4735	Median :2.821	Median :0.002000
Mean :0.5633	Mean :3.473	Mean :0.001995
3rd Qu.:0.9123	3rd Qu.:5.911	3rd Qu.:0.003000
Max. :1.4820	Max. :9.642	Max. :0.004000

Table 1: Summary statistics for all three functions using N=1130.

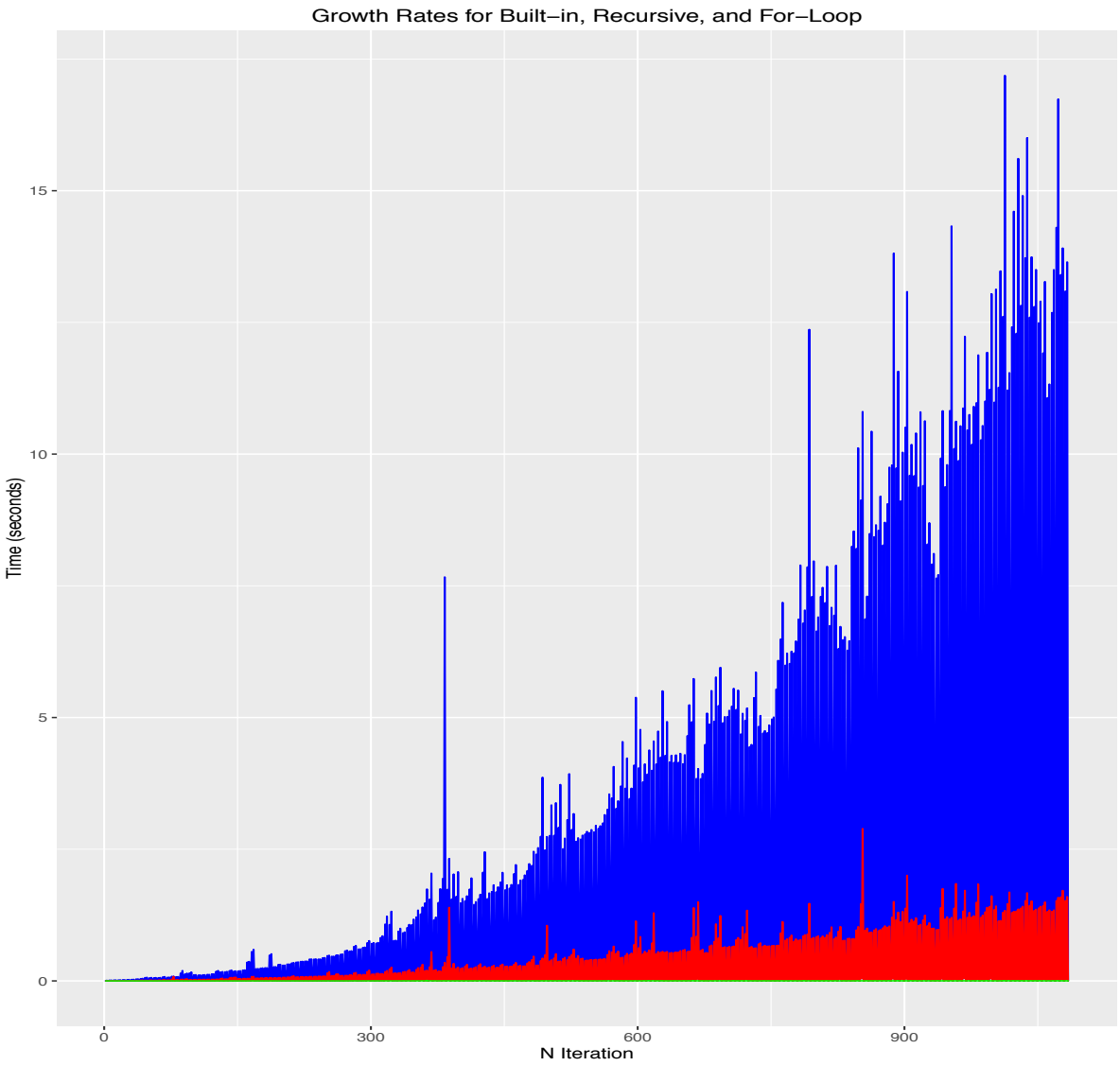

Figure 1. All functions overlayed in one plot. Blue is the recursive, red is the for-loop implementation, and green (as seen as the line below the red) is the built-in function. N=1085.

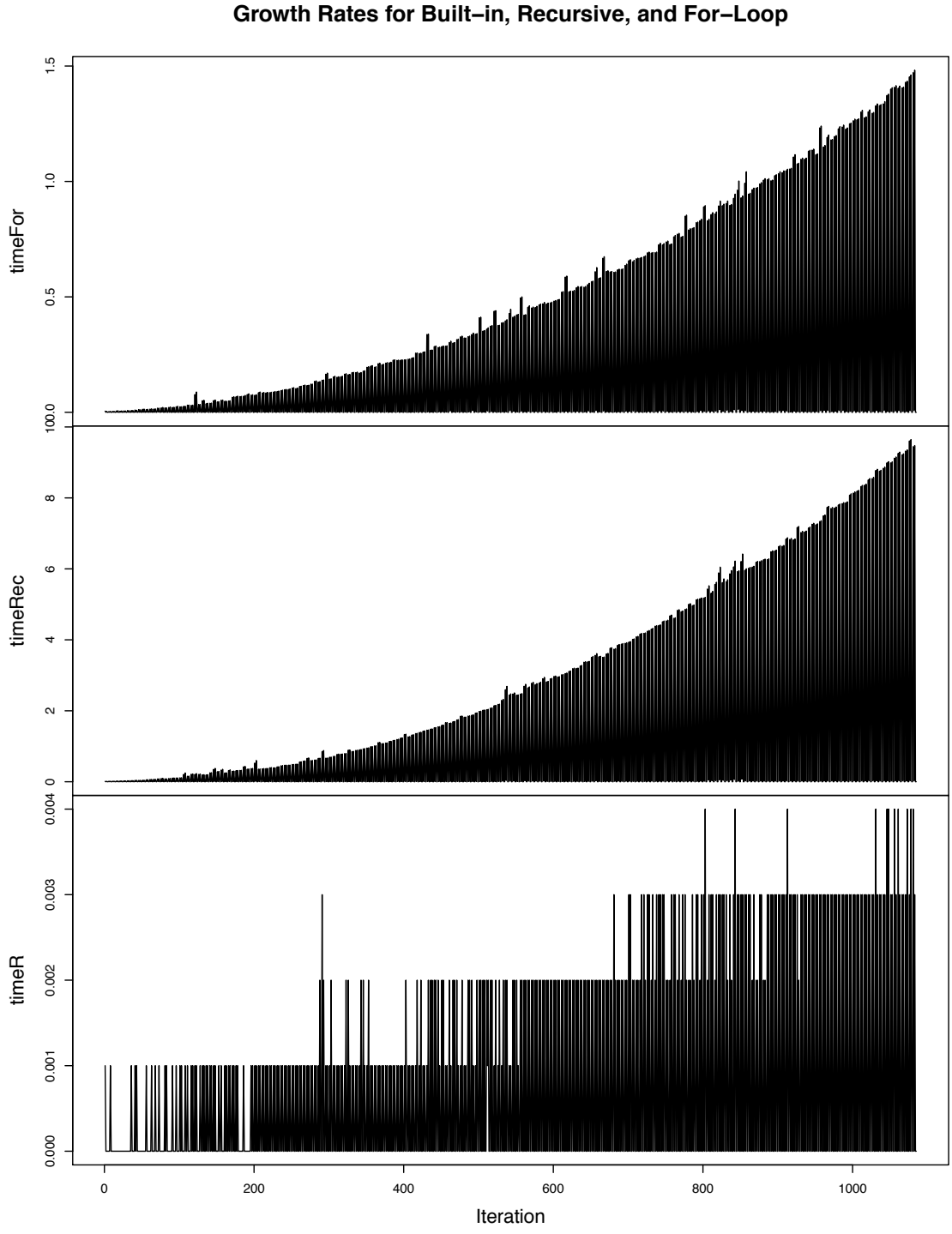

Figure 2. Cleanest iteration found. N = 1130.

You can see from Table 1 and Figure 1 that the built-in function was significantly faster at an average CPU time of 0.002, compared to the recursive function average time at 3.473 and the for loop implementation at an average of 0.5633. The fastest time for the recursive function at 0.008 is still 2 times slower than the slowest time of the built-in function at 0.004. The standard deviation for the built-in is 0.0008, whereas the standard deviation for the recursive function is 1.9246 – which suggests high variability by using the recursive function. The standard deviation for the for loop implementation was 0.2960.

The for loop implementation is the second fastest and is shown in Figure 1 to have a linear growth. The recursive function has a clear logarithmic growth as N increases (see Figure 1). The shape of the built-in function seems to be a linear/step function (see Figure 1 and 2). It appears that at about N=240, the execution time jumps from 0.001 to 0.002, and again jumps another +0.001 at about N=400, to 0.003, and again to 0.004 at about N=700. The built-in functions are not without an increase in execution time over large iterations. Regardless, it appears that the R implementation is highly optimized for the given task.

These patterns were apparent in most, if not all, of the plots (see Figure 2, 3, 4) – with the exception of outliers in the other plots shown below. These outliers most likely represent random noise with my machine's CPU resources. Overall, this suggests R's built-in function for calculating the sum of log gamma is very efficient, using a for-loop is somewhat acceptable in some cases, and to only use recursive functions when necessary.

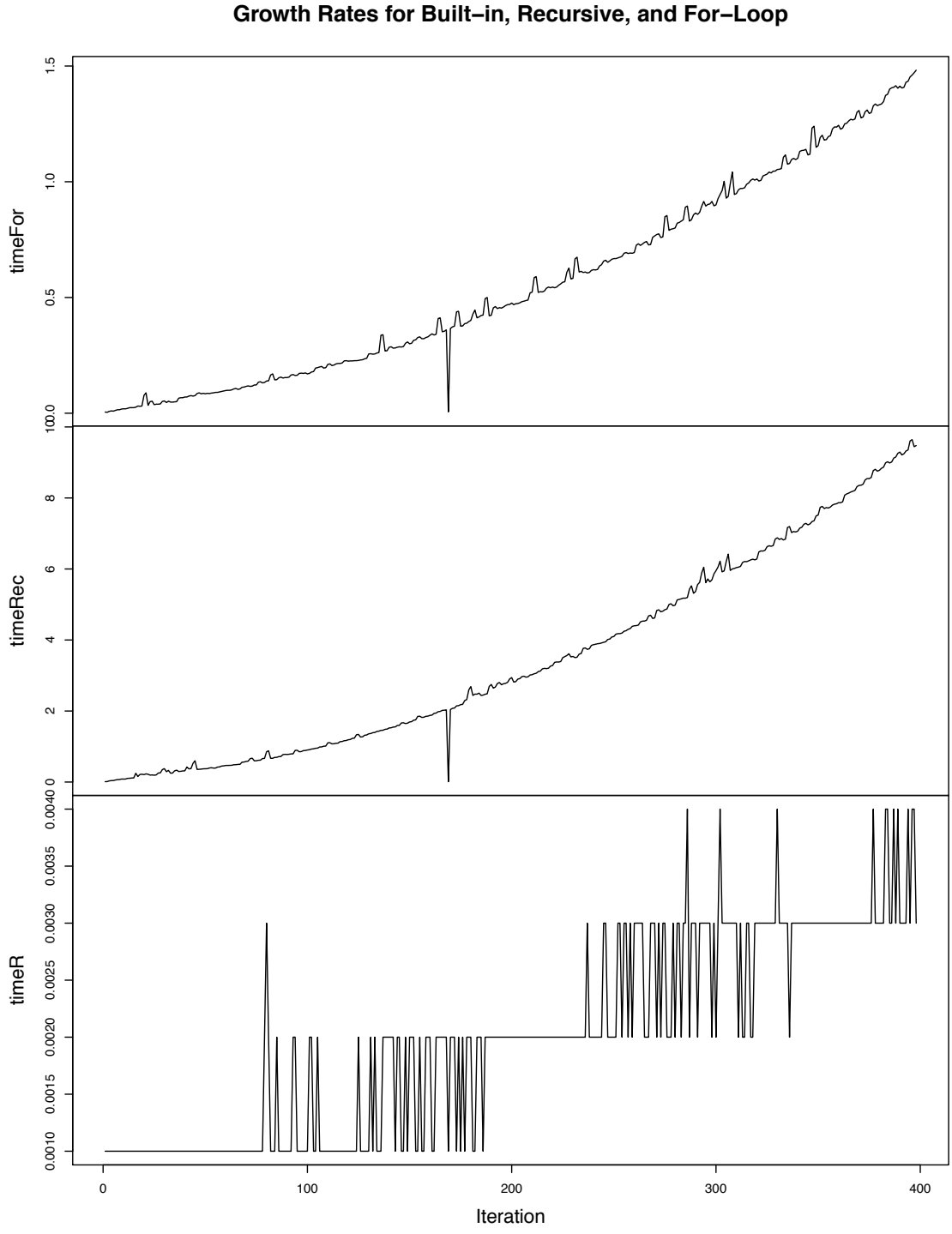

Figure 3. Clean graph without 0's. N = 416.

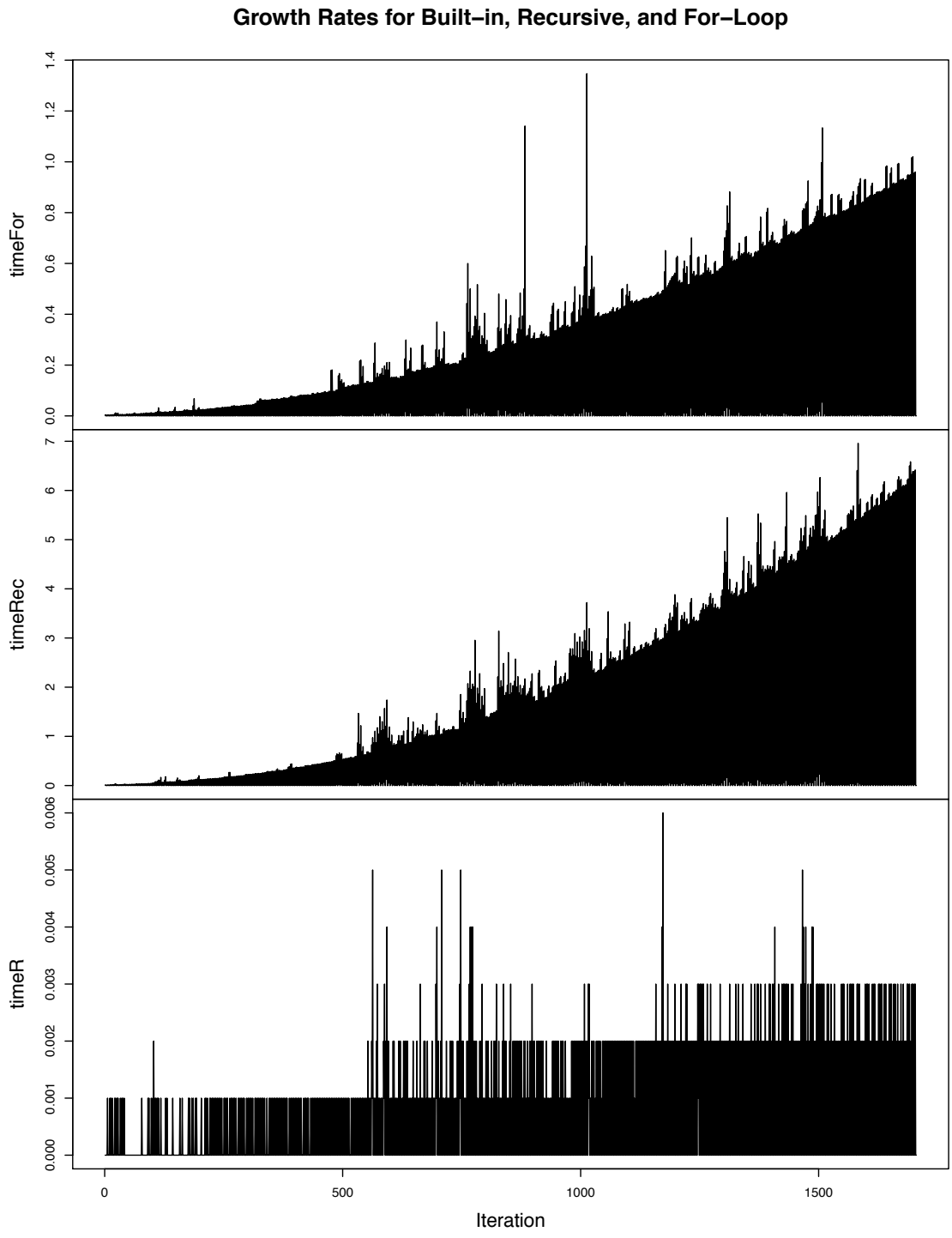

Figure 4. Graph with 0's. N = 1705.

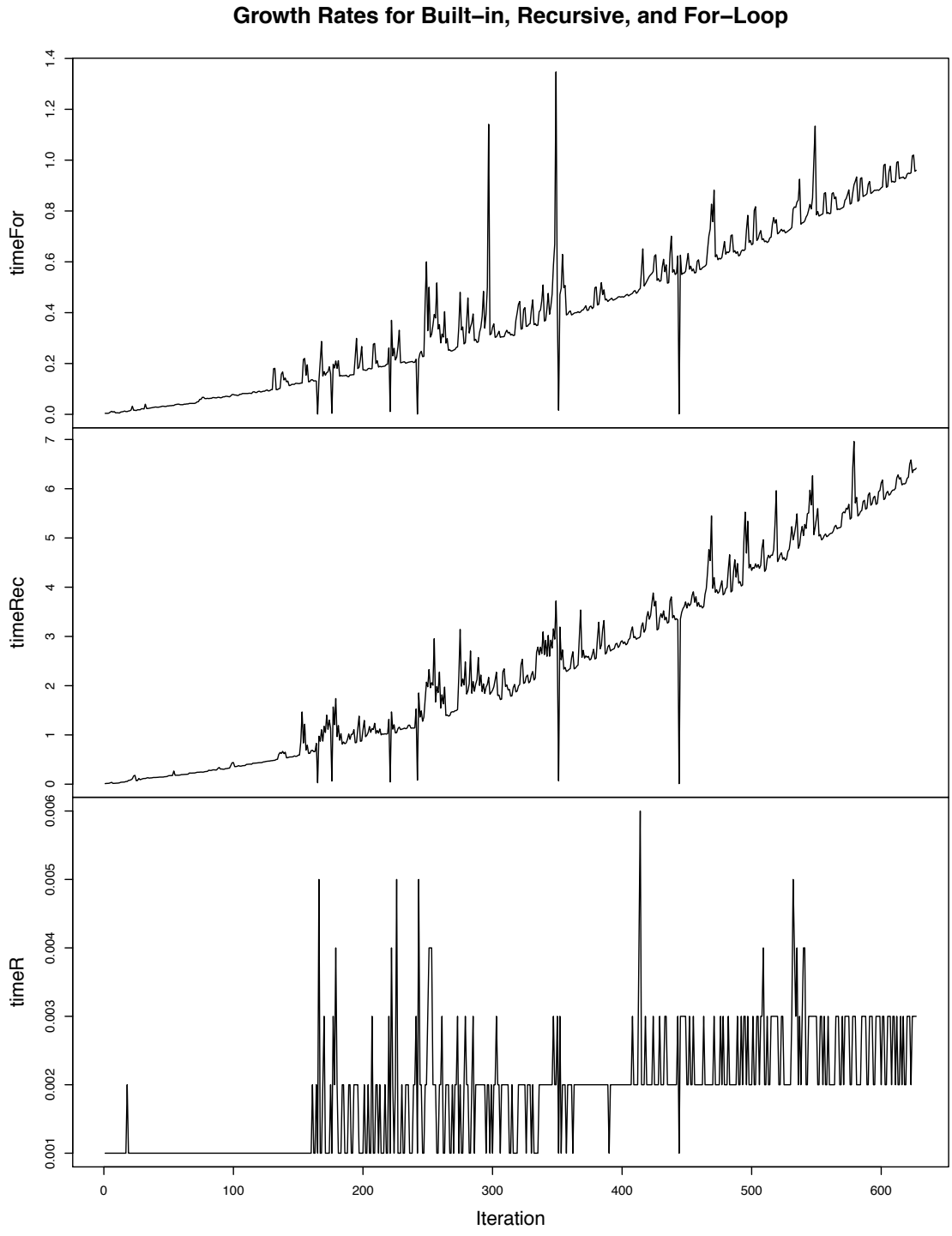

Figure 5. Clean graph without 0's. N = 627.