STA 242: Assignment 3

Olivia Lau

February 13, 2013

1 Introduction

To continue from our BML Traffic Model, we will shift our focus from actual implementation to computational efficiency. We know that our package works, but we want to make it better. We must identify what areas need the most improvement and what are the cost-benefits of rewriting in R or, in this case, C.

2 Motivation

As far as the first portion of implementation went, it was both efficient in one respect and inefficient in another.

For one, it was efficient considering where it began. When I first began, it did the most redundant steps, repeating similar calls. Once I got those portions to work, I then attempted to minimize for-loops, if-else statements, and the overall the number of lines. To this end, I was successful.

But on the other hand, we limited our simulation to a small grid and few time steps. This personal limitation was due to the overall timeliness of processing. This was, in large part, due to the making of .gif files from each simulation. This portion could have been left out, especially considering that this was evident from the `Rprof()` summary. It just looked cool, but unfortunately ate up a large portion of the time.

3 Implementing C

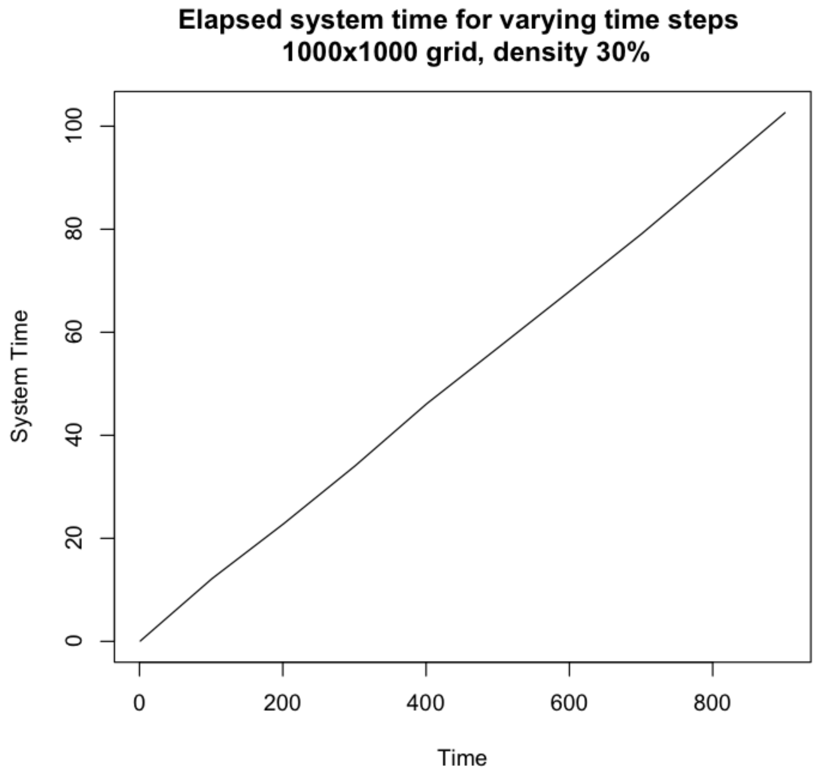
Initially, my first thought was to trash the entire package and write it all in C. As I began writing the function to generate a grid, I realized that the investment was not worthwhile. It was taking a good 15 lines, whereas in R it spanned 5 lines and worked quickly. It became clear that some aspects of this would be better done in R and some better done in C.

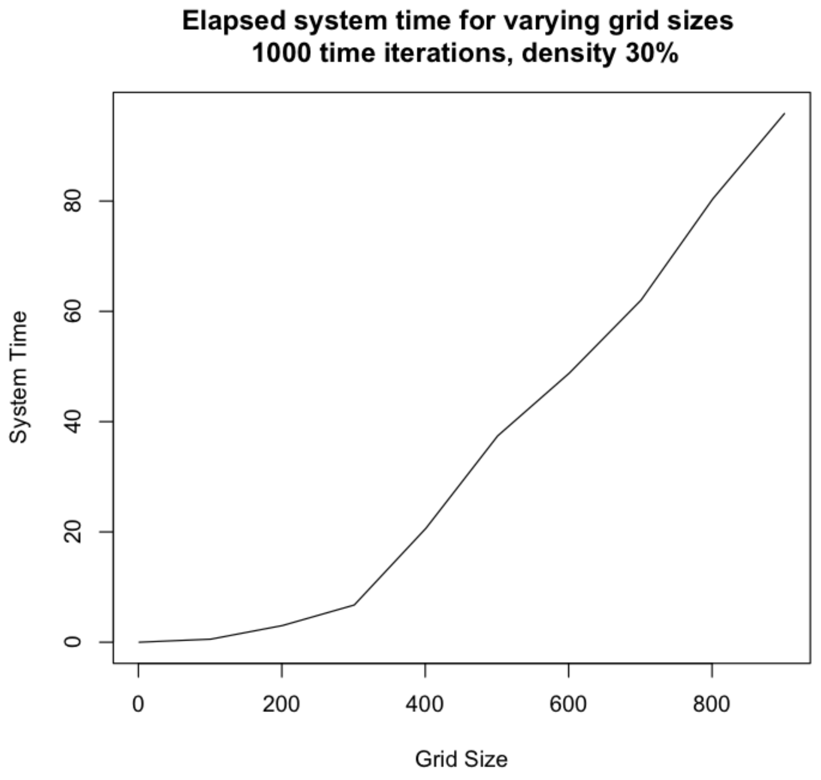
Taking the aforementioned motivations into consideration, two major changes were made. For one, and most easily, the creation of .gif files were removed all together. It didn't contribute much, and seemed to overshadow some other areas that could be improved. After doing this, `Rprof()` immediately identified our `swap()` function to be the main time-eater. This function in essence takes (x, y) coordinates of current car positions (red or blue) and potential car positions (right and up, respectively) and swaps values if the potential position is a 0, or empty. Admittedly, this function was more involved than it needed to be. There was an `apply()`, a `which()`, a for-loop, and an if-else statement. Thus, we went ahead with implementing this function in C.

The actual writing of the code structure was similar to that of R, with some small and, frankly, annoying nuances that made it more challenging. Without the real-time feedback from each line of code, it was difficult to see if I was headed in the right direction. The type and structure of variables differed slightly, but it was manageable. For example, there are matrices in R but only arrays in C, arrays started at 0 in C and 1 in R. Being able to convert back and forth between the two was key. Similarly, being able to simply pass variables between the two was a new concept entirely.

4 Speeding-Up

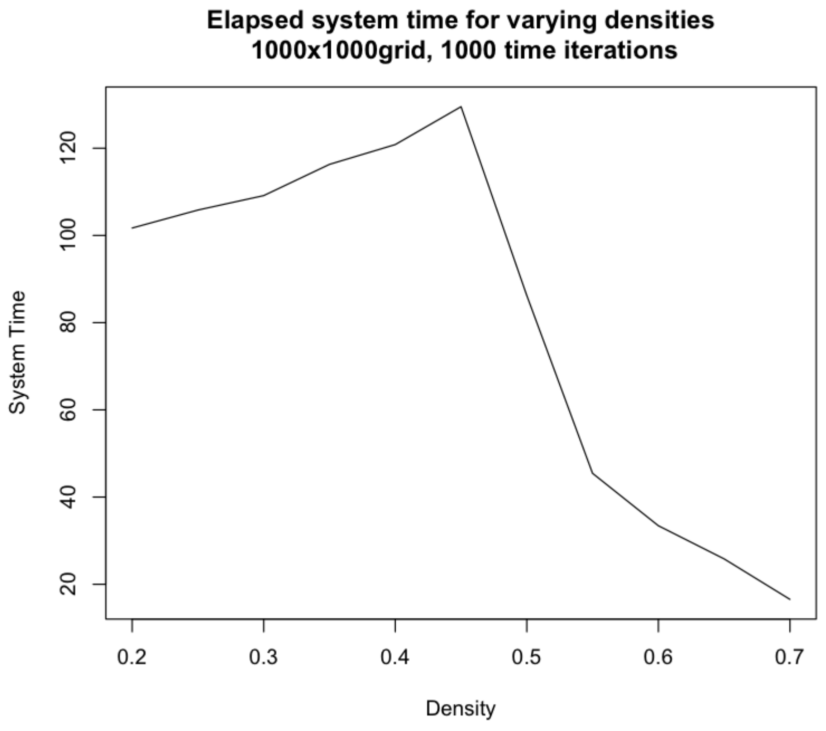
To quantify the difference before and after implementation in C, the summary from `Rprof()` is displayed in Table 1. This was run for a 512 x 512, 30% density grid for 500 time steps. After the function swap() is moved into C, the self.time are all less than 5.00. That's almost 2% of the original time! Overall, the sampling time before was 436.92. After, the sample time was 18.74. Sitting in from of the computer, the difference was noticeable and surprising. The new swap function doesn't even appear in the top 10 of the most time consuming functions!


Elapsed system time for varying time steps
1000x1000 grid, density 30%
System Time
Time
0
20
40
60
80
100
0
200
400
600
800


Elapsed system time for varying grid sizes
1000 time iterations, density 30%
System Time
Grid Size
0
20
40
60
80
0
200
400
600
800

function	self.time	self.pct	function	self.time	self.pct
”swap”	269.40	61.66	”t”	4.68	24.97
”FUN”	52.20	11.95	”matrix”	2.94	15.69
”[”	37.16	8.50	”==”	2.30	12.27
”[<-”	33.32	7.63	”cbind”	2.24	11.95
”apply”	17.32	3.96	”simTraffic”	2.06	10.99
”lapply”	9.26	2.12	”t.default”	1.30	6.94
”t”	3.38	0.77	”as.double”	0.98	5.23
”-”	2.54	0.58	”arrayInd”	0.68	3.63
”cbind”	2.46	0.56	”next.pos”	0.54	2.888
”arrayInd”	2.30	0.53	”which”	0.54	2.88
⋮	⋮	⋮	⋮	⋮	⋮
”sample”	0.02	0.00	”dim”	0.02	0.11

Table 1: Before/After implementing swap function in C



This incredible speed up from such a small effort enabled us to look at larger grids, for longer periods of time. Thus, we can quickly look also at the speed differences with different parameters such as grid sizes, time periods, and densities.

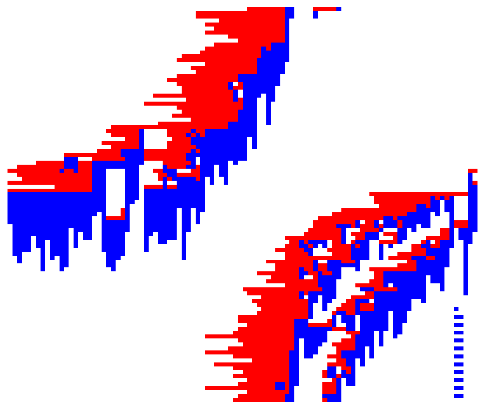
With increasing time, the association appears to be very strictly linear in an almost one-to-one proportional manner (Figure 1). Conversely with increasing grid sizes, the elapsed system time increases faster with a larger grid (Figure 2). And lastly, the variation in grid density displays an interesting fact (Figure 3). The elapsed time increases for densities up to about 45%. Thereafter, it falls quickly and tapers. This fact is especially since I built a stopping mechanism: if the grid has no movement for 5 time periods, it stops. Thus, with an increasing density the grid comes to a halt faster and faster.

5 Testing

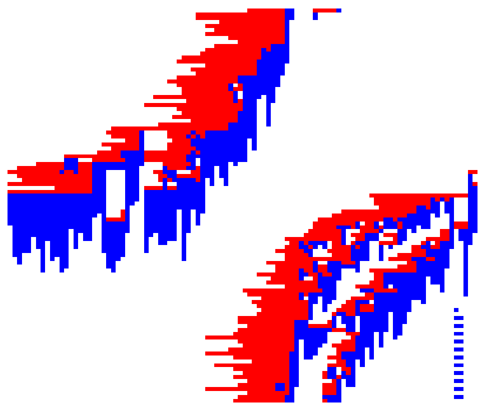
All this speed is definitely a good thing, but it means nothing if the answer is wrong. Thus, it is essential that we confirm our output between our new and improved package and our old one.

A quick fire way (of sorts), was simply to compare the output from the two packages.

```
> library("LauBML", lib.loc = "~/Documents/STA 242/Assignment3/Working")
> set.seed(2218)
> grid = genGrid(100, 100, 0.3)
> myVelocty = simTraffic(grid, 1000, TRUE)
```



```
> library("LauBML", lib.loc = "~/Documents/STA 242/Assignment2/Final")
> set.seed(2218)
> grid2 = genGrid(100, 100, 0.3)
> myVelocty2 = simTraffic(grid, 1000, TRUE)
```



The plots look the same, but this is very rough. We're not completely sure. It is hard to match up each pixel individually. Thus, we may go down and test out the `swap()` function directly. We define our `swapR()` as our old swap function, and `swapC()` as our new swap function.

```
> grid = genGrid(1000, 1000, 0.3)
> blue = which(grid == 1, arr.ind=TRUE)
> next.blue = next.pos(blue, nrow(grid))
> gridC = swapC(blue, next.blue, grid)
> gridR = swapR(blue, next.blue, grid)
> all(gridC == gridR)
```

```
[1] FALSE
```

For a single time point, the resulting grids are not alike. With more investigation, this fact is not surprising. Our `swapR()` takes a snap shot of all the values before changing anything whereas `swapC()` runs through the grid iteratively. That is, if a blue car is in the way of another but moves first, the subsequent blue is allowed to move. For this project, it is alright: for higher densities, the grid will still gridlock, just perhaps at an earlier time point, for lower densities, the grid is in a constant flow. Regardless, this difference between the two functions is important to note.

6 Lessons Learned

During the course of this assignment, several topics were touched upon that I had not encountered before. Although I have used C before, interfacing between R and C was an novel endeavor. This really forced a good understand of what goes on in both programs, in terms of memory management and general calculations and manipulation. More specifically, the use of pointers. Adding asterisk here and there merited no valuable solution. It became essential to understand its use, when to use and when not to use.

Another aspect that I had never really used before was the R profiling. I had used it in passing to see what areas needed small tweaking, but no changes were quite as significant as the one evidenced here.

7 Conclusion

I had taken ECS 30 and was successful in using C – three years ago. It by no means made this assignment a cinch. Without frequent use, my knowledge and dexterity was sparse. Grasping the concept of pointers was most difficult. But once I passed that hurdle, it was fairly straightforward.

Evidently, having R interface some portion to C directly has major advantages. Less time is wasted for it to chug through, but also enables us to play around with larger parameters and its resulting outcomes.

8 Appendix: Code

8.1 swap.c

```c
#include <stdio.h>

void swap(int *len, int *currX, int *currY,
            int *nextX, int *nextY,
            int *M, double *grid){

  int currVal, nextVal;
  for(int i = 0; i < len[0]; i++){
    currVal = grid[((currY[i] - 1) * M[0]) + (currX[i]-1)];
    nextVal = grid[((nextY[i] - 1) * M[0]) + (nextX[i]-1)];
    if(nextVal == 0){
      grid[((currY[i] - 1) * M[0]) + (currX[i]-1)] = 0;
      grid[((nextY[i] - 1) * M[0]) + (nextX[i]-1)] = currVal;
    }
  }
}
```

8.2 Rswap.c

```c
#include "swap.h"
#include <stdio.h>

void Rswap(int *len, int *currX, int *currY,
           int *nextX, int *nextY,
           int *M, double *grid){
  swap(len, currX, currY, nextX, nextY, M, grid);
}
```

8.3 swap.h

```c
void swap(int *len, int *currX, int *currY,
            int *nextX, int *nextY,
            int *M, double *grid);
```

8.4 swap.R

```r
swap = function(car.pos, potential, grid){
  matrix(.C('swap', as.integer(nrow(car.pos)), as.integer(car.pos[,1]), as.integer(car.pos[,2]), as.integer(
      potential[,1]), as.integer(potential[,2]), as.integer(nrow(grid)), myGrid = as.double(grid))$myGrid, nrow(
      grid), ncol(grid))
}
```

8.5 The rest of it...

```r
######################################################################
##This function generates a grid of red and blue cars       ##
##Inputs: number of rows (grid.r), number of columns (grid.c),    ##
##  proportion of grid filled with cars (rho)           ##
##Output: a grid.r by grid.c matrix randomly filled with 0, 1, 2's  ##
##  where the 1 = blue, 2 = red. Note that the proportion of 1's and##
##  2's are 1:1.                              ##
######################################################################

genGrid = function(grid.r, grid.c, rho = .5){
  if(rho > 1 | rho < 0){
    cat("Warning: This is not a valid proportion (rho). \n Proceeding with default rho = 0.5. \n")
    rho = 0.5
  }
  if(rho == 0){
    cat("Cool, you made an empty parking lot...\n\n")
  }
  if(rho == 1){
    cat("Looks like LA rush hour. \n\n")
  }

  ###SIMPLE CALCS###
  ncars = round(rho * grid.r * grid.c, 0) ##total number of cars
  totcells = grid.r * grid.c  ##total number of cells

  ###GENERATE GRID###
  colors = c(rep(0, totcells-ncars), rep(1, round(ncars/2)), rep(2, round(ncars/2)))
  grid = matrix(sample(colors, totcells), nrow = grid.r)
  class(grid) = "Grid"
  return(grid)
}
```

```
##This function creates an S3 method to plot of class "Grid"    ##
##Inputs: a matrix of type "Grid".                    ##
##Output: plots the grid as an image.                 ##

plot.Grid = function(x, y, ...){
  rows = nrow(x)
  image(t(x[rows:1,]), axes = FALSE, col = c("white", "blue", "red"), ...)
  }
```

```
######################################################################
##This function creates an S3 method to print the summary of an   ##
##  object of class "Grid"                         ##
##Inputs: a matrix of type "Grid".                     ##
##Output: prints the current state, dimensions, total number of   ##
##  spaces, number of blue cars, number of red cars, number of open ##
##  spaces.                                  ##
######################################################################

summary.Grid = function(object, ...){
  cat("Current grid state:\n")
  print(object)
  cat("Grid dimensions:", dim(object), "\n")
  cat("Total number of spaces", length(object), "\n")
  cat("Number of blue cars:", sum(object==1), "\n")
  cat("Number of red cars:", sum(object==2), "\n")
  cat("Number of open spaces:", sum(object==0), "\n")
}
```

```
######################################################################
##This function simply changes the (x, y) coordinate to move up.  ##
##  Also checks if next space is out of the grid. If it is, it     ##
##  loops to the other side.                       ##
##Inputs: a matrix of x, y coordinates to be moved up. Also the   ##
##  number of rows in the grid, for checking if out of the grid   ##
##Output: a matrix of x, y coordinates moved up one row.      ##
######################################################################

next.pos = function(ij, n.row){
  up = cbind((ij[,1]-1), ij[,2])  ##change coords
  up[up[ , 1] == 0, 1] = n.row  ##check coords
  return(up)             ##return coords
}
```

```
######################################################################
##This function moves the colored cars according to time and thereby##
##  color. Red uses the same functions as Blue, only on a transposed##
##  version of the grid.                       ##
##Inputs: original grid, and time                  ##
##Output: returns given grid with moved cars at time.       ##
######################################################################

move = function(grid, time){
  d = dim(grid)
  if(time %% 2 == 0){   ##red cars
    grid = t(grid[ , ncol(grid):1])
    car.pos = which(grid == 2, arr.ind = TRUE)
    potential = next.pos(car.pos, nrow(grid))
    new.grid = swap(car.pos, potential, grid)
    new.grid = t(new.grid[nrow(new.grid):1,])
  }
  else{   ##blue cars
    car.pos = which(grid == 1, arr.ind = TRUE)
    potential = next.pos(car.pos, nrow(grid))
    new.grid = swap(car.pos, potential, grid)
  }
  return(new.grid)
}
```

```
####################################################################
##This function simulates traffic movement for a given grid, from  ##
##  time = 1 until user specified time = t.              ##
##Inputs: a given grid (grid) and time (t)               ##
##Output: Creates a .gif of the movement from time 1 through t in  ##
##  current directory titled "myMovie.gif", plots grid at time t,  ##
##  and returns a vector of velocities at each time point     ##
####################################################################

simTraffic = function(grid, t = 100, plotLast = TRUE){
  velocity = integer(t)
  for(i in 1:t){
    new.grid = move(grid, i)
    velocity[i] = length(grid) - sum(new.grid == grid)
    grid = new.grid
    class(grid) = "Grid"
    if(i > 5){
      if(all(velocity[(i - 5):i] == 0)){
      cat("No movement for 5 time periods. \n Stopping now at time = ", i, "\n\n")
      break}
    }
  }
  if(plotLast == TRUE){
    plot(grid)}
  return(velocity)
}
```

8.6 Some other stuff...

```
grid = genGrid(1000, 1000, 0.3)
time = seq(1, 1000, 100)
systime = numeric(10)
for(i in 1:10){
  systime[i] = system.time(simTraffic(grid, time[i], FALSE))[3]
}
plot(time, systime, type = "l", xlab = "Time", ylab = "System Time", main = "Elapsed system time for varying time
     steps \n 1000x1000 grid, density 30%")
size = seq(1, 1000, 100)
systime = numeric(10)
for(i in 1:10){
  grid = genGrid(size[i], size[i], 0.3)
  systime[i] = system.time(simTraffic(grid, 1000, FALSE))[3]
}
plot(size, systime, type = "l", xlab = "Grid Size", ylab = "System Time", main = "Elapsed system time for varying
     grid sizes \n 1000 time iterations, density 30%")
dense = seq(.2, .7, .05)
systime = numeric(11)
for(i in 1:11){
  grid = genGrid(1000, 1000, dense[i])
  systime[i] = system.time(simTraffic(grid, 1000, FALSE))[3]}
plot(dense, systime, type = "l", xlab = "Density", ylab = "System Time", main = "Elapsed system time for varying
     densities\n 1000x1000grid, 1000 time iterations")
```