University of California,Davis
STA242
Spring 2015

Project 2

Junxiao Bu
999452701

April 29, 2015

Purpose and Organization

In this project, the first part is about the simulation algorithm design and the exploration of the BML traffic model process. The second part is about profile. In the second part, I will present some of the code refinements to speed up my code including vectorization and parallel computing.

Part I: Algorithm and Package Introduction

The package to simulate the BML traffic model is called **BMLsimulation**. In this simuation process, the blue cars will move upwards first. Then the red cars will move to the right. If cars move to the borders of the grid, red cars will jump back to the left border of the same row. Blue cars will jump back to the bottom border of the same column. The process can simulated in the next plots. Here, the time step I defined in the package is a little bit different. In my definition, each time step means blue cars and red cars move once in their own orders.

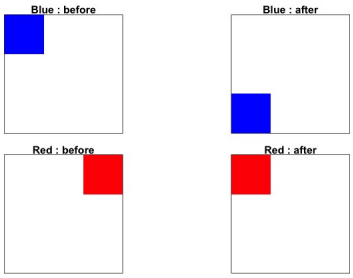

Figure 1: Different cars' moving pattern

If the next position of one color's car is occupied, then this particular car cannot move. The red car cannot move if its right cell has a car. The blue car cannot move if its upper cell has a car.

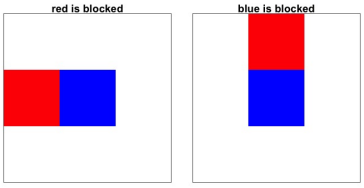

Figure 2: Different cars' moving pattern

Next, in order to verify if the simulation process is correct using **BMLsimulation** package. A 4*4 grid with 4 red cars and 4 blue cars with 4 steps of movements for each color is simulated. The initial grid is shown below.

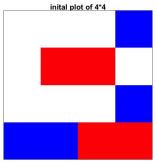

Figure 3: Initial plot of 4*4 grid with 4 red cars and 4 blue cars

All the plots that show the moving steps are shown below.

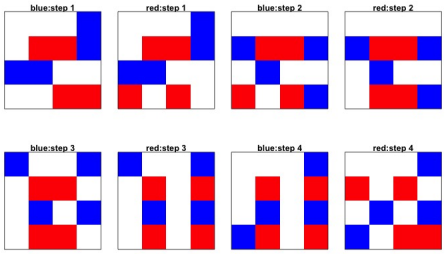

Figure 4: Moving process for a 4*4 grid with 4 steps

From the plots, one can observe the moving process by eyes and verify the result step by step. In this 4*4 grids, the moving result is correct.

Part II: Stochastic Process

In this BML traffic model process, different mode of car moving may lead to different results. In my definition, I choose to let cars move simultaneously rather than sequentially.The behaviour of the model depends on the exact dimensions of the grid as well as the initial density of the cars. I simulated the process of 128 *128, 256*256, 512*512 and 1028*1028 grids. In order to find the transition points. I use the density from 0.2-0.7. (I suppose that the numbers of red cars are equal to the number of blue cars.) Here I only present the result from 256*256 grid. To balance the efficiency and accuracy, the grid is run for 5000 times (red cars: 5000 times; blue cars: 5000 times). All the grids are represented below.
If initialized with a low enough density of cars, the system eventually self-organizes into a configuration where all cars can move at each time step (each car has asymptotic velocity equal to unity). From figure 5, in the top-left, we have a free-flowing grid with a density of 0.2 in which the cars arrange themselves into diagonal lines.

If initialized at slightly higher density, the cars are blocked by other cars, until eventually all cars end up participating in one large global jam, where no car can move (asymptotic velocity equal to zero).At a higher density of 0.4, I see a

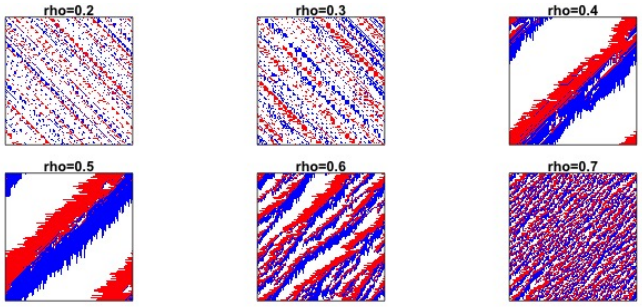

Figure 5: BML grids with different density

structure with some deadlocked cars in two parallel groups running diagonally We also see bands of red and blue cars moving. These are intermediate phases that combine jammed and free-flowing phases, in either a periodic or disordered manner. Some cars will move freely, while in the 0.5 picture the cars have arranged themselves into several diagonal bands which avoid each other. At a higher density (larger than 0.4), I see additional groups but the same pattern of moving cars. For higher densities, I see more parallel groups but the same deadlock.

From figure 6, it seems that the phase transition point between traffic-free to intermediate phase (some cars are deadlocked) occurs when the density is around 0.3. The phase transition point between intermediate phase and deadlocked phase occurs when the density is around 0.4. Two plots with densities around 0.3 are the diagonal lines are already starting to emerge. But most of the cars can still move freely. Two plots with densities around 0.5 show the patterns of deadlocked cars into two parallel groups. Due to the length of paper, not all the plots for phase transition densities are shown here.There are more details in the following parts to discuss the transition phase.

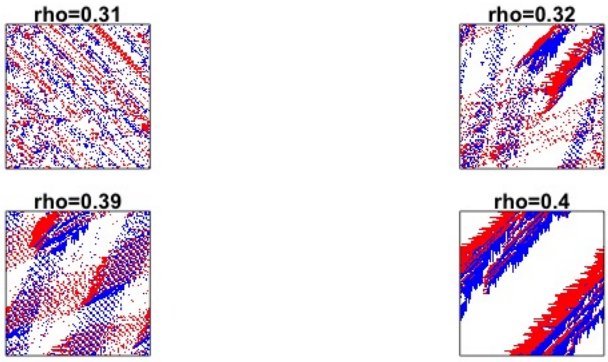

Figure 6: BML grids to explore transition points

Different Grid Size

Next, the impacts of different grid size on the behavior of the model are going to explored.From previous plots, the intermediate phases seems to be different. For grids with coprime dimensions, the intermediate states are self-organized bands of jams and free-flow with detailed geometric structure, that repeat periodically in time. In non-coprime rectangles, the intermediate states are typically disordered rather than periodic.[1]

Choosing the grid size to be 144*289, the plots of 5500,6000, 6500 ,7000,7500 and 8000 iteration times are shown below. The density is 0.38.

From the plots in Figure 7, the coprime grid size tends to produce periodic intermediate phases near the transition density. The periodic images are quite clear. The red cars and blue cars gather into two groups. The red group moves from topleft to bottomright. The blue group moves from left to right. The states are self-organized bands of jams and free-flow with detailed geometric structure, that repeat periodically in time.

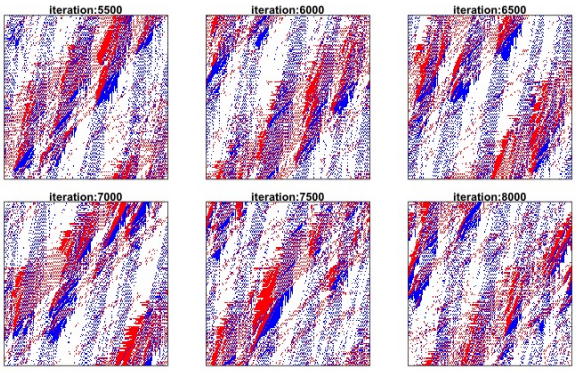

Figure 7: Averge velocity vs. density (1000 times)

Different Density

Next, some explorations about different car numbers are presented. In figure 8, the first situation is numbers of red cars are more than the numbers of blue cars. The grid is still 256*256.I suppose the ratio of red cars vs. blue cars is firstly 7:3 and then 3:7. The overall density is still in the range of 0.3-0.5.
In the density = 0.5 picture there is a single jam spanning the entire grid, while in the density = 0.3 picture the cars have arranged themselves into wide diagonal bands which avoid each other. In the density = 0.4 picture (red: blue=7:3), all cars move some of the time and wait some of the time, and this is achieved by semi-regular geometric patterns of jams feeding into each other. This is the intermediate phase.

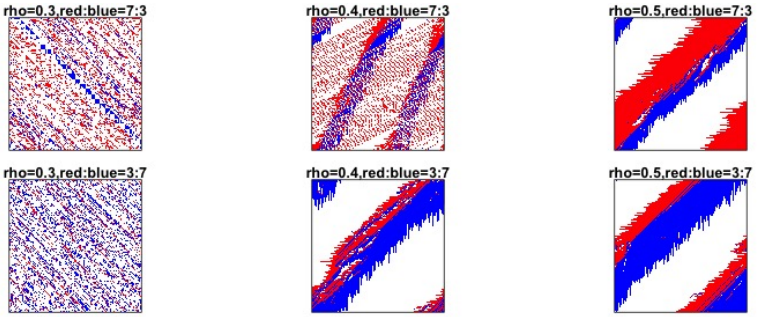

Figure 8: BML grids with different numbers of cars in each color

Comparing with the previous cases, the parallel group patterns are still present in these two cases. Obviously, most of the deadlock cars are still in two parallel groups running diagonally. But the widths of the color bands are different now. The majority colors bands are wider. If the density of the grid increases, there would be more parallel groups as shown in the previous cases (red cars = blue cars) but with different widths.

Next, the average velocities of cars are present when giving different grid size and density range. Here the average velocity is defined as:

$$\frac{the\ number\ of\ cars\ that\ can\ move}{total\ number\ of\ cars}$$

Notice that the numbers of cars that can move were computed when all the cars have moved for given time steps. The density was evenly split into 50 blocks in the range of 0.2 to 0.7. There would be two sets of plots with different iteration times. One is 500 times, another one is 1000 times. The grid sizes are 64*64, 128*128, 256*256. The two sets of plots are shown below.

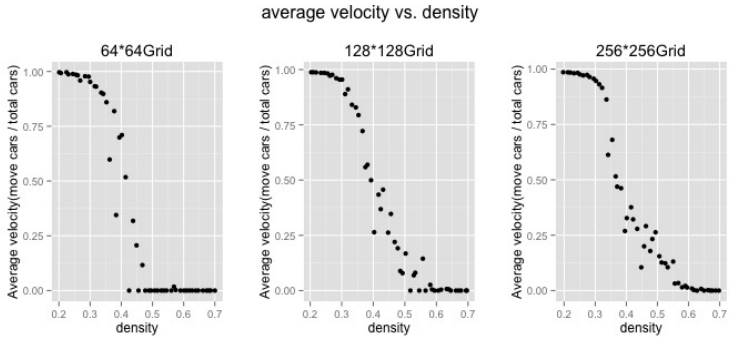

Figure 9: Averge velocity vs. density (500 times)

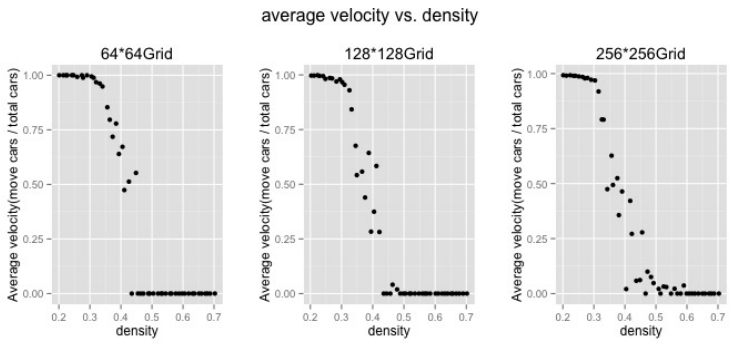

Figure 10: Averge velocity vs. density (1000 times)

For small size systems like 64*64 I observe the predicted behavior of a relatively sharp transition from freely flowing to dead-lock phase. when we implement systems with larger tha 64*64, we observe a bifurcation where the traffic free-flow phase and dead-lock phase stat to coexist, as we go from low to intermediate values of density. The plots show that there are different points of transition phase, not simply free-flowing or deadlocked.

These are the intermediate states. This conclusion validates what I observed from previous grid plots with different densities.

For different iteration times, there are no big differences between two sets of plots for different grid sizes.The only difference is that there are less points in the intermediate phase when iteration times increase. This is reasonable because cars are more easily to be blocked when their move times increase. If the iteration time is larger enough,i.e. 100000 times. There wil be fewer points in the interemediate phase. Most of the points(different densities) will have value of average velocity 0 or 1.

If drawing three grids' average velocities on one plot, there exists a trend the

density at which average velocity decreasing occurs decreases as the grid size increases.

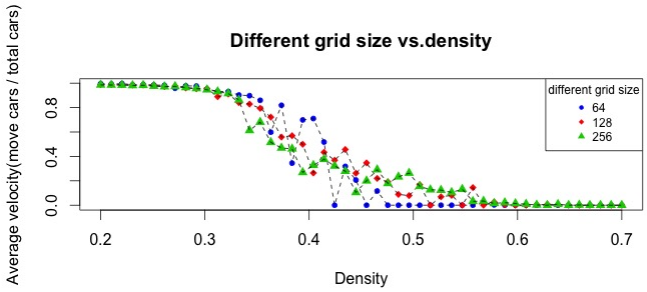

Figure 11: Averge velocity vs. density (500 times)

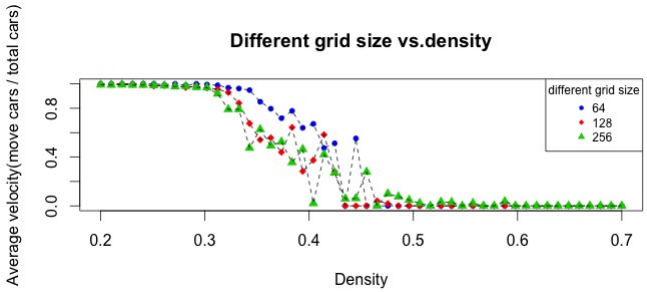

Figure 12: Averge velocity vs. density (1000 times)

We see that the density at which deadlock occurs (average velocities are equal to zero) are roughly same except for the 64*64 grid. At the lower density level, almost all of the cars can move all the times (rho = 0.1 or 0.2). The average velocities for both cars are significant decreasing when density is larger than 0.3, which implies that the transition points density is around 0.3(slightly larger than 0.3). This result validates what I discuss above. When observing the grid with density around 0.3, the diagonal lines are already starting to emerge.

Part III: Evaluating the performance of the code

In order to speed up my code, my refinements include two parts. The first part is to use vectorization method and matrix subset instead of for loop. The second part is parallel computing for iterating over time steps. As the benchmark, I use a150*150 grid with 3000 number cars for each color. The iteration will be 100 times.

For the first part, in my function to move the cars, I use **for** loop and **if else** simulate the process. I use profiling to find out where the computations spend most of their time. The result is in the following tables.

Table 1: Time for running the slower version of code

Time Type(seconds)	User Time	System Time	Elapsed Time
	69.117	12.023	81.312

Table 2: Profiling to show each part's time

Time Type(seconds)	self.time	self.pct	total.time	total.pct
"movecars1"	63.76	85.45	74.58	99.95
"[[.data.frame"	2.08	2.79	6.44	8.63
"match"	1.38	1.85	2.08	2.79
"Anonymous"	1.14	1.53	1.36	1.82
"$"	0.88	1.18	8.76	11.74

The user time is roughly 70 seconds. As I expect, my function movecar1is taking a large proportion of the overall time, since I havent vectorized my functions. More specifically, I checked the profiling for movecar1 function.

After some refinement including vectorization and matrix subset in the function, the system time is significant lower than before. The new time is in the following table. As the table shows, the old users time is over 200 times of new user time, which means my code, is over 200 times faster than before.

Table 3: Time for running the faster version of code – vectorization

Time Type(seconds)	User Time	System Time	Elapsed Time
	0.319	0.082	0.401

Some code examples are presented here. (vectorization method)

```
for (i in red_indice){
    current_row = car_info$row_indice[i]
    current_col = car_info$col_indice[i]
    next_row = current_row
    if(current_col==ncol(matrix)) next_col = 1
    else next_col = current_col +1
    }
```

This slower version of code is to calculate each red cars current coordinates and next coordinates on the grid. If the coordinates are blank, then this read car would move to the blank space. Here I use for to loop over all the red cars coordinates. For each car, I calculated the new coordinates and checked if this car can move. This version is very slow. I found that some parts can be substitute by vectorized methods. For example, I can calculate all the cars coordinates once and calculate the new coordinates once. The faster version is as following:

```
current_location = as.matrix(cbind(car_info$row_indice[indice],
                                   car_info$col_indice[indice]))
  colnames(current_location) = c("row_indice","col_indice")
  possible_next_location = current_location
  if(color==red){
    ## check if we can move the red cars
   possible_next_location[,"col_indice"]
       = possible_next_location["col_indice"] + 1
   possible_next_location[,"col_indice"][possible_next_location[,"col_indice"]
                        > ncol(matrix)] = 1
  }
```

Here I use vectorization method to calculate red cars coordinates. First, I calculated all the red cars current coordinates. Then I added one to all their row coordinates, since red cars would move to the right. At last, I checked if the new row coordinates are out of the grid. If yes, these new row coordinates would be changed to 1.

The second example here is the matrix subset. The first code chunk is the slower version. First, the code computed the coordinates of red cars new position. For each pair of coordinates, the code checked if the corresponding location in the matrix is available. If yes, red cars coordinates would be updated.

```
if(matrix[next_row,next_col] ==white){
        matrix[next_row,next_col]=red
        matrix[current_row,current_col]=white
}
```

The second version is using the matrix subset to achieve the same goalupdate red cars coordinates. The matrix *possible_next_location* includes all the red cars right spots coordinates. The code uses this matrix to subset the original grid and then checks if the new locations are available. If yes, red cars coordinates would be updated.

```
new_indice = matrix[possible_next_location]
matrix[possible_next_location[which(new_indice == white),,drop=FALSE ]]
        = color
matrix[current_location[which(new_indice == white),,drop=FALSE ]]
        = white
matrix
```

The users time now is roughly 0.319 seconds, which seems not too large. But I only iterated time steps 100 times. The users time will increase when iteration times increases. The reason is the vectorized method cannot be applied when iterating over time steps, since each step depends on the previous steps moving result. In order to speed up, I choose to use **Parallel** and **doParallel** package in **R** for parallel computing. Specifically, I created cluster with desire number of cores and registered cluster using doParallel package. Then I use foreach in **Parallel** package to write the iterations for time steps. The new system time is in the following table.

Table 4: Time for running the faster version of code – vectorization

Time Type(seconds)	User Time	System Time	Elapsed Time
	0.098	0.035	0.304

Note that the user time now is only 0.098 seconds, which means my code is 3.3 times faster when using parallel computing. Some further discussions are here. R is an interpreted language, which is significant lower than compile language. If the all the functions are written in C/C++, the speed will be far faster than the current version. Due to the time constraint, this part is not included in this assignment.

Reference

1. http://en.wikipedia.org/wiki/Biham

2. classnotes

3. Piazza: 168,181,183.