# Biham-Middleton-Levine traffic model[1]

University of California, Davis
Qiwei Li

## Background

The Biham–Middleton–Levine (BML) traffic model is a self-organizing structure. It consists some blue cars, some red cars, and a grid that cars move on.

At t = 0, all cars are randomly put on the grid. In our configuration, the blue cars move at time periods t = 1, 3, 5, ... and the red cars move at time periods t = 2, 4, 6, ..., i.e., they alternate in time. The “blue” cars move vertically upward. The “red” cars move horizontally rightwards. When a car reaches to the edge, it “warps” around in the same row or column. A car cannot move to cell if that cells is already occupied by another car (of any color).

BML traffic model is interesting to study because it is one of the simplest self-organizing structure that exhibits a phase-transition. Define the density to be the proportion of cars to the grid size. At lower density, the randomly put cars will self-organize into a free flow stage. However, as the density increases, suddenly the behavior of cars change. This is density threshold is called a phase-transition point.

## Clarification on Moving Rules

The main rule of how cars move is “*A car cannot move to cell if that cells is already occupied by another car (of any color)*”. The definition is unclear in the following situation.

Let a vector represent a row in the grid. Let 0 represent empty space. Let 1 represent represent a red car. The question is what’s the result of moving [0 1 1 0]? Two results can be seen.

- The first result is [0 0 1 1] if you consider cars move simultaneously where one car’s movements can make space for the other.
- The second result is [0 1 0 1] if you consider cars move non- simultaneously. This is a more natural representation of a traffic model. In reality, cars stacked together do not move simultaneously.

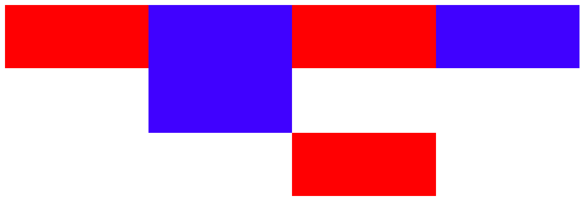
In section “Transition Phase”, we will see that the simultaneous method will not allow BML self organize as often as non-simultaneous method. Simultaneous method makes cars jam very often. The non-simultaneous method is more natural and commonly studied. We will compare the difference between the two. However, we will focus on the simultaneous method.

[1] Bitbucket repository: `https://qiwei_li@bitbucket.org/qiwei_li/sta242-assignment2.git`

## Approach and Speed

A natural representation of a grid in R is a matrix. Also, let's define 0 represent empty space, 1 represent red cars, and 2 represent blue cars.

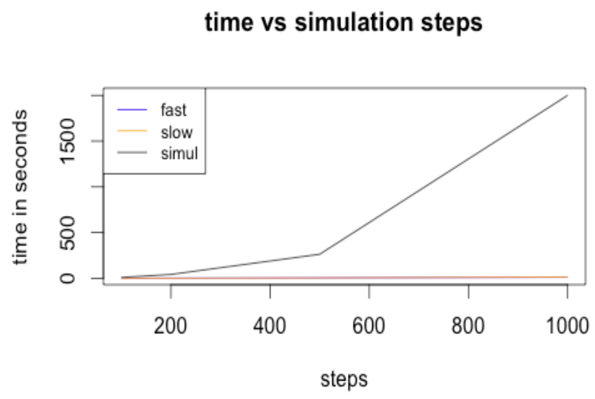
| | | | |
|---|---|---|---|
| 1 | 2 | 1 | 2 |
| 0 | 2 | 0 | 0 |
| 0 | 0 | 1 | 0 |

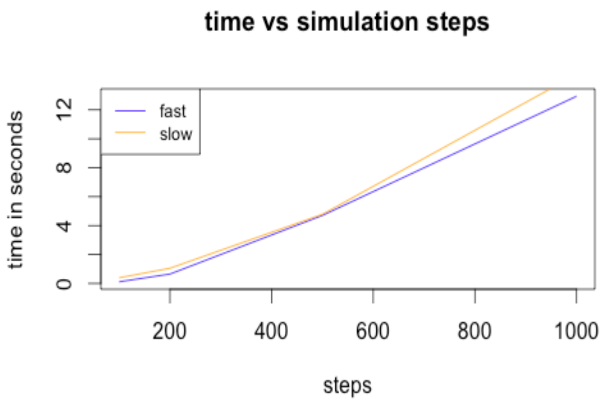


Let's consider the simultaneous method, since we need to consider if the moving of a car will make space for the other when cars are stacked together, we are using previous iteration results in each iteration. The only way to implement this is a for loop. In my package "BML", **moveBlue.simul()** and **moveRed.simul()** implement this approach.

For non-simultaneous method, we can take advantage of R vectorization. I have two implementations for this method: **moveBlue.slow(), moveRed.slow()** and **moveBlue.fast(), moveRed.fast()**.[2]

We run our three approach on grid size of 100 x 100, 200 x 200, 500 x 500, 1000 x 1000. We can see that there is a huge efficiency different.





Obviously from the left plot, the for loop implementation for simultaneous method is big $O(n^2)$ This is due to the nested for loops for checking each car one by one in each row or column and in each column or row.

Let's zoom in to compare two vectorize implementations of non-simultaneous approach. We can that the fast approach is slightly faster than the slow approach. Let's take a look where do the functions spend their time using **Rprof()**.

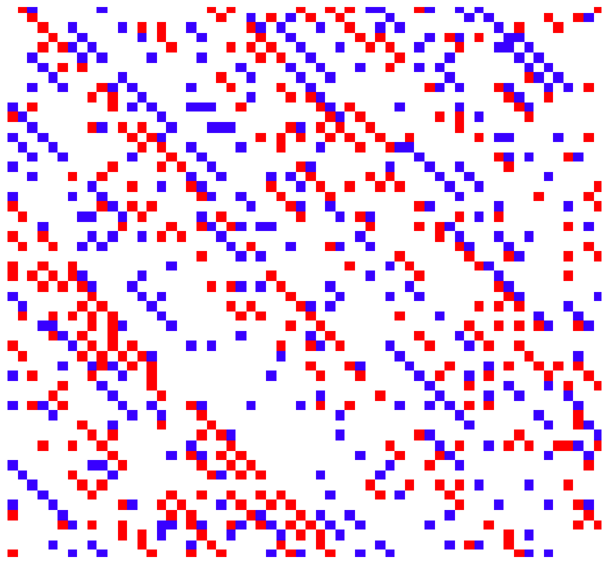
[2] See Appendix I for codes

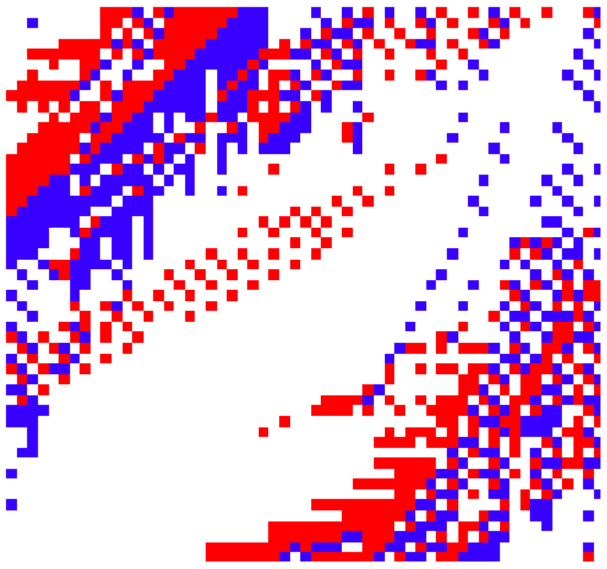
| Rprof: move.slow | Rprof: move.fast |
|---|---|
| <pre>            self.time self.pct<br>"moveBlue"       3.44    23.47<br>"which"          3.44    23.47<br>"moveRed"        3.42    23.33<br>"=="             1.48    10.10<br>"[<-"            1.20     8.19<br>"["              0.86     5.87<br>"-"              0.24     1.64<br>"+"              0.18     1.23</pre> | <pre>             self.time self.pct<br>"t.default"        1.62    31.64<br>"moveBlue"         1.28    25.00<br>"=="               1.04    20.31<br>"moveRed"          0.78    15.23<br>"which"            0.14     2.73<br>"%%"               0.06     1.17<br>"+"                0.04     0.78</pre> |

For the same BML grid, we can see move.fast is faster than move.slow mainly because the function **which()** spends less time. Because in my move.slow implementation, I only vectorize on each row/column. In move.fast, I vectorize on the entire matrix. Note that these two implementations' speed will not be affected by the density of cars. This is due to our vectorized useage.

## Transition Phase

There are typically two stages in a BML traffic model. Using **plot()**[3], we obtained the following plots. One is called a "free flow" stage (on the left). This occurs when most blue and red car move freely in a nice pattern. The other is called "jammed" stage (on the right). This occurs when cars are jammed in various degree locally or globally. Transition phase point is the density which change the BML long term behavior from "free flow" to "jammed".



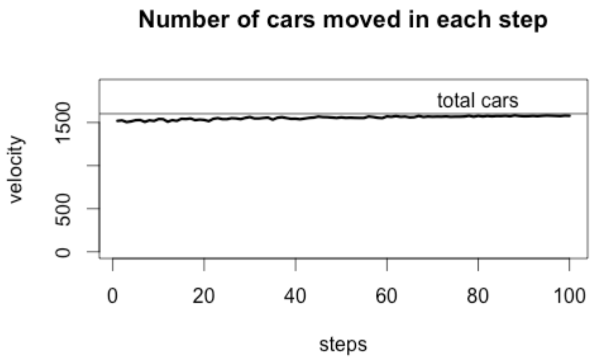
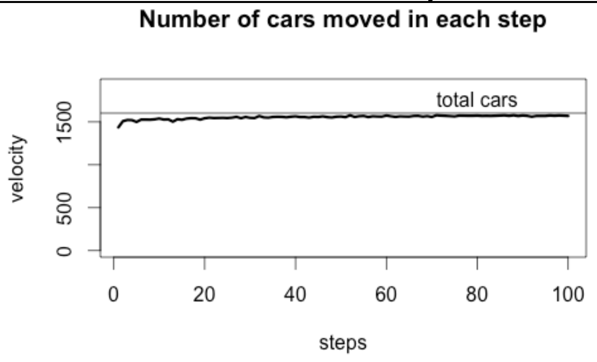
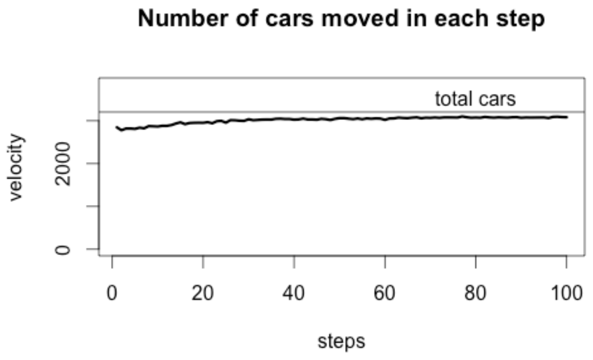
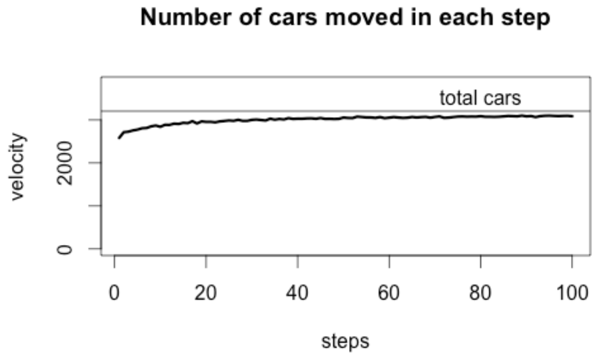


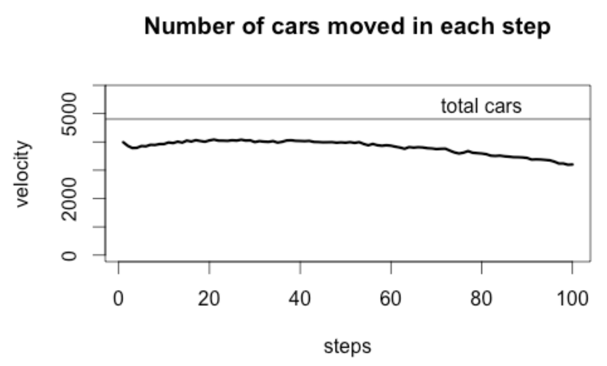
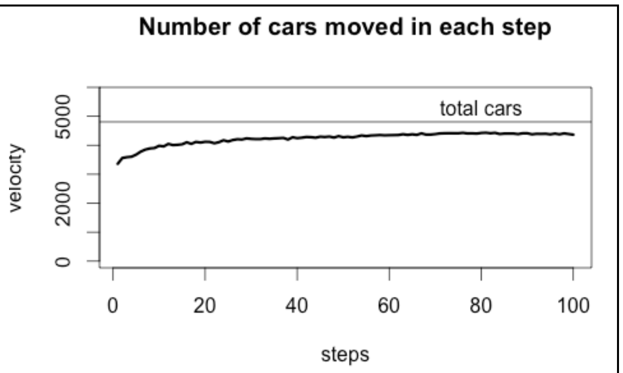
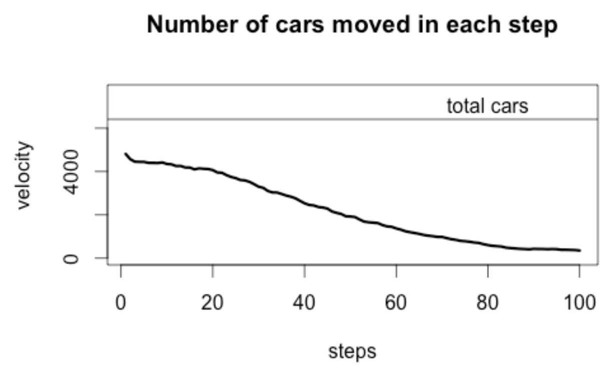
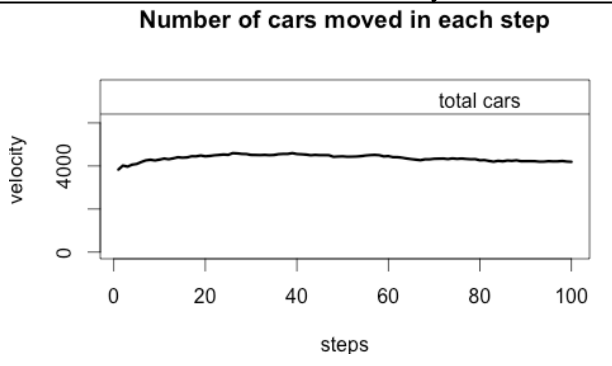
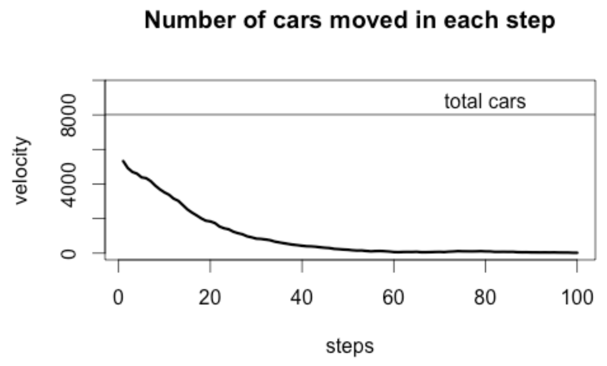
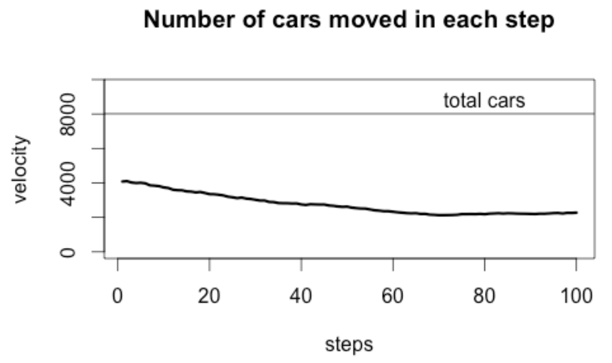
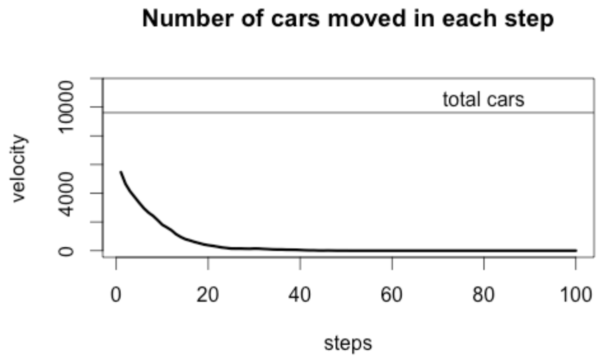
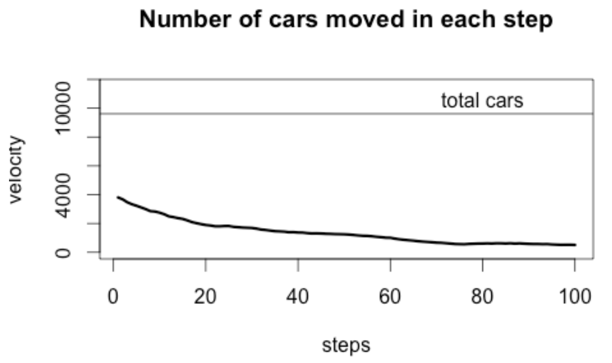
[3] See Appendix II for help page

It is difficult to understand the stages by looking at the simulation without animation. Therefore, we define **velocity** to be the amount of cars move in each steps. By checking velocity, we can understand where transition point occurs.

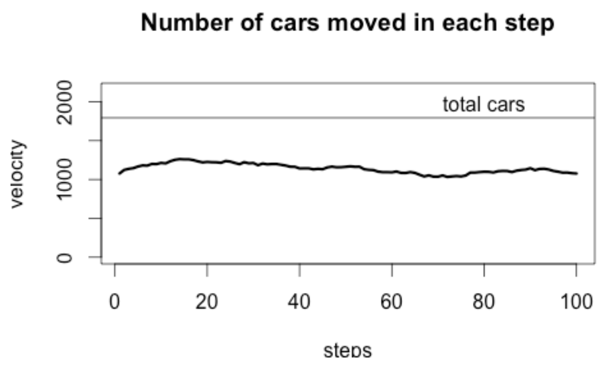
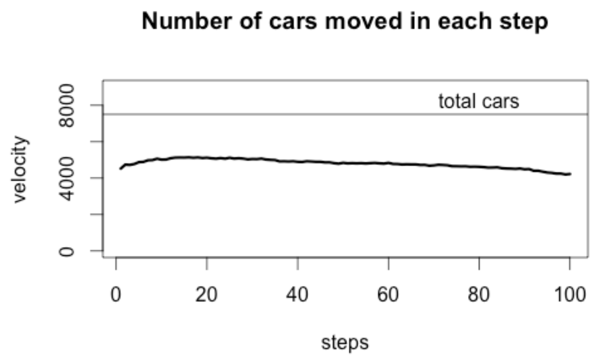
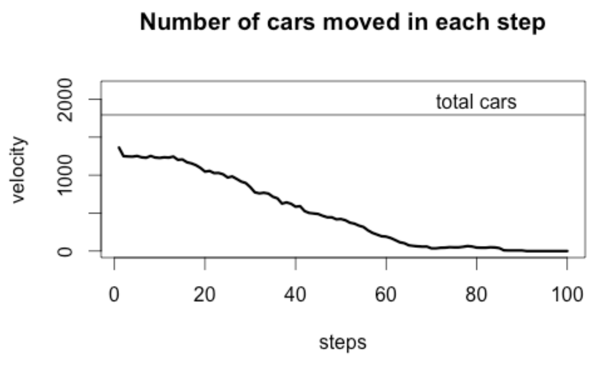
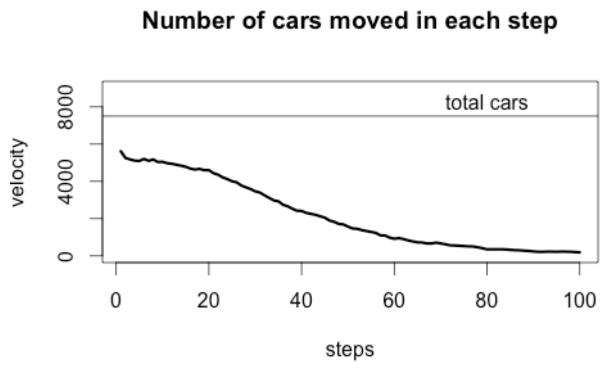
Recall the definition of transition phase. It is the density which change the BML long term behavior from “free flow” to “jammed”. Free flow means nearly all cars are moving, so the velocity should be close to total number of cars in a free flow stage. On the other hand, in a jammed stage, we should see the velocity decreases.

Now, let’s detect the phase transition point.
For a fix grid size (117 x 137), the **simultaneous** method and **non-simultaneous** method for BML simulation exhibits different long term behavior. We can that cars starts to jam globally after density is increased to 0.3 for simultaneous method. For non-simultaneous method, cars only starts to jam globally after 0.5

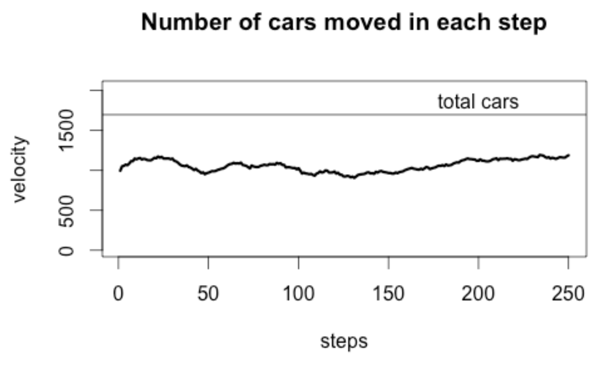
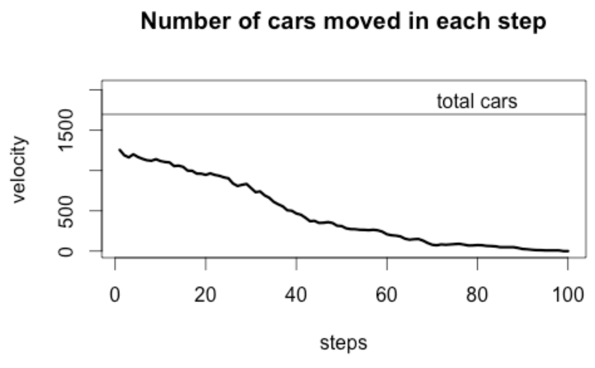
| Simultaneous, density = 0.1 | Non-simultaneous, density = 0.1 |
|---|---|
|  |  |
| Simultaneous, density = 0.2 | Non-simultaneous, density = 0.2 |
|  |  |
| Simultaneous, density = 0.3 | Non-simultaneous, density = 0.3 |

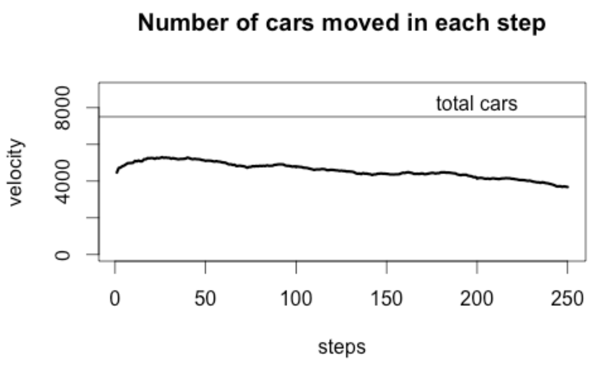
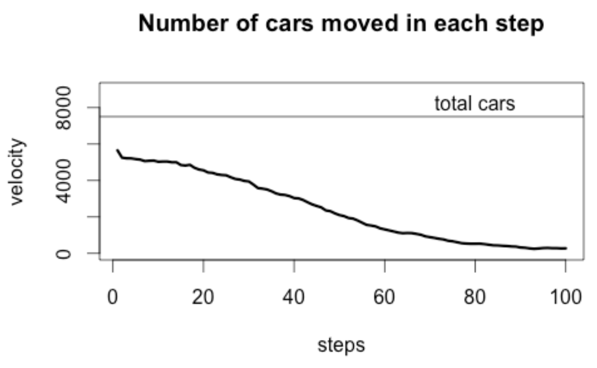
| | |
|---|---|
|  |  |
| Simultaneous, density = 0.4 | Non-simultaneous, density = 0.4 |
|  |  |
| Simultaneous, density = 0.5 | Non-simultaneous, density = 0.5 |
|  |  |
| Simultaneous, density = 0.6 | Non-simultaneous, density = 0.6 |
|  |  |

Now, dimensions have two component, size and shape.
First, let's fix shape and detect if **grid size** affect simulation behavior given the same density. We can see that size does not change the long term behavior of velocity. However, we do notice as grid size increases, the velocity is much smoother.

| Simultaneous, 67 x 67 | Non-simultaneous, 137 x 137 |
|---|---|
|  |  |
| Simultaneous, 67 x 67 | Non-simultaneous, 137 x 137 |
|  |  |

For a fix density (0.4), let's see the velocity plot for two **grid shape** (square-ish and rectangle-ish). Both simultaneous method and non-simultaneous method for BML simulation's long term behavior are affected by the grid shape. We can see that the velocity behavior for square-ish grid is smoother than rectangle shape.

| Simultaneous, rectangle | Non-simultaneous, rectangle |
|---|---|
|  |  |

| Simultaneous, square | Non-simultaneous, square |
|---|---|
|  |  |

## Limitations

- Even though our vectorized implementation takes linear time, it still carries a large factor on the overhead. We can improve this by programming in C or interfacing C within R.
- It's worth to mention that the plotting function **image()** in R is really slow. The reason is R is plotting the entire plot after every move. Ideally, we would have a interactive graphic device such that an element on the graph can update by itself.
- In our report, we just explore the phase transition point by looking at the velocity plot. Ideally, we want to have a function that can detect if the long term behavior for a BML grid is free flow or jamming.

## Conclusion

In this report, we studied the BML traffic model in R. We introduced different implementations to do the simulation. We demonstrated that R vectorized approach is much better than nested for loops. We also explored the phase transition property of the BML traffic model. We also discussed some limitations of our current approach. Even though this report is about BML, one can relate the process step by step to any other simulation problems.

## Appendix I - Functions

```
# s3 methods
plot.BML = function(grid, ifadd=TRUE){
  image(t(grid)[,nrow(grid):1], axes = FALSE, add=ifadd, col = c("white", "red", "blue"))
}

summary.BML = function(grid){
  r = nrow(grid)
  c = ncol(grid)
  size = r*c
  nRed = sum(grid==2)
  nBlue = sum(grid==1)
  d = as.data.frame(matrix(c(round(nBlue/size, digits=2), round(nRed/size, digits=2), nBlue,
nRed), nrow=2, ncol=2))
  colnames(d) = c("Density", "CarCount")
  rownames(d) = c("Blue", "Red")
  cat(paste0("This BML grid has ", r, " rows and ", c, " columns.\n"))
  cat(paste0("The size of the grid is ", size, ".\n"))
  cat("\n")
  print(d)
}

# setup
setX11 = function(r, c){
  ratio = r/c
  screen = 8/12
  if(ratio >= screen)
    x11(width=8*(c/r), height=8)
  else
    x11(width=12, height=12*(r/c))
}

createBMLGrid = function(r, c, ncars=c(red=0, blue=0), density=0){
  if(r<=0 | c<=0)
    stop("Error: grid dimension must be positive")
  if(density > 1 | density < 0)
    stop("Error: density should be between 0 and 1")
  if(density){
    nred = nblue = floor((r*c*density)/2)
    totalcar = nred + nblue
  }
  else{
    nred = ncars['red']
    nblue = ncars['blue']
    totalcar = nred + nblue
    if(totalcar > r*c | totalcar < 0)
      stop("Error: Number of cars must be between 0 and grid size")
  }
  colors = c(rep(0, r*c-totalcar), rep(1, nred), rep(2, nblue))
  structure(matrix(sample(colors, r*c), nrow = r), class=c("BML", "matrix"))
}

runBMLGrid = function(grid, numSteps, ifPlot = FALSE, method){
  numSteps = floor(numSteps/2)
  if(method == "fast"){
    moveBlue = moveBlue.fast
    moveRed = moveRed.fast
  }
  else if(method == "simultaneous"){
    moveBlue = moveBlue.simul
    moveRed = moveRed.simul
  }
  else if(method == "slow"){
    moveBlue = moveBlue.slow
    moveRed = moveRed.slow
  }
  else
    stop("Error: Invalid method")

  velocity = rep(NA, numSteps)
  r = nrow(grid)
  c = ncol(grid)
  if(ifPlot){
    setX11(r, c)
    par(mar=rep(0,4))
    plot.BML(grid, ifadd=FALSE)
    for(i in 1:numSteps){
      mb = moveBlue(grid, r, c)
      grid = mb$grid
      plot.BML(grid)
      mr = moveRed(grid, r, c)
      grid = mr$grid
      plot.BML(grid)
      velocity[i] = mb$velocity + mr$velocity
    }
    dev.off()
    par(mar=rep(5,4))
    plotVelocity(grid, velocity)
    grid
  }
  else{
    for(i in 1:numSteps){
      mb = moveBlue(grid, r, c)
      grid = mb$grid
      mr = moveRed(grid, r, c)
      grid = mr$grid
      velocity[i] = mb$velocity + mr$velocity
    }
    par(mar=rep(5,4))
    plotVelocity(grid, velocity)
    grid
  }
}

plotVelocity = function(grid, velocity){
  plot(velocity, main="Number of cars moved in each step", type='l', xlab="steps", lwd=3,
ylim=c(0, table(grid!=0)[2]*1.2))
  total = table(grid!=0)[2]
  abline(h = total)
  text(length(velocity)*0.8, total*1.1, "total cars")
}

# move functions
moveRed.fast = function(grid, r, c){
  #red is 1
  oriClass = class(grid)
  g = t(grid)
  carPos = which(g==1)
  goingPos = carPos+1
  pos = goingPos%%c==1
  goingPos[pos]=goingPos[pos]-c
  ifEmpty = g[goingPos]==0
  g[carPos[ifEmpty]] = 0
  g[goingPos[ifEmpty]] = 1
  grid = t(g)
  class(grid) = oriClass
  list(grid=grid, velocity=length(carPos[ifEmpty]))
}

moveBlue.fast = function(grid, r ,c){
  #blue is 2
  oriClass = class(grid)
  carPos = which(grid==2)
  goingPos = carPos-1
  pos = goingPos%%r==0
  goingPos[pos]=goingPos[pos]+r
  ifEmpty = grid[goingPos]==0
  grid[carPos[ifEmpty]] = 0
  grid[goingPos[ifEmpty]] = 2
  class(grid) = oriClass
  list(grid=grid, velocity=length(carPos[ifEmpty]))
}

moveRed.slow = function(grid, r, c){
  #red is 1
  oriClass = class(grid)
  count = 0
  for(lr in 1:r){
    carPos = which(grid[lr, ]==1)
    goingPos = (carPos+1)
    goingPos[goingPos>c]=1
    ifEmpty = grid[lr, goingPos]==0
    grid[lr, carPos[ifEmpty]] = 0
    grid[lr, goingPos[ifEmpty]] = 1
    count = count + length(carPos[ifEmpty])
  }
  class(grid) = oriClass
  list(grid=grid, velocity=count)
}

moveBlue.slow = function(grid, r, c){
  #blue is 2
  oriClass = class(grid)
  count = 0
  for(lc in 1:c){
    carPos = which(grid[ , lc]==2)
    goingPos = (carPos-1)
    goingPos[goingPos<1]=r
    ifEmpty = grid[goingPos, lc]==0
    grid[carPos[ifEmpty], lc] = 0
    grid[goingPos[ifEmpty], lc] = 2
    count = count + length(carPos[ifEmpty])
  }
  class(grid) = oriClass
  list(grid=grid, velocity=count)
}

moveRed.simul = function(grid, r, c){
  #red is 1
  oriClass = class(grid)
  count = 0
  r = nrow(grid)
  c = ncol(grid)
  for(lr in 1:r){
    s = tail(which(grid[lr, ]==0),1) #most right empty
    if(length(s)){
      if(s==1)
        index=c:1
      else
        index = c((s-1):1, c:s) #backwards ex.5 4 3 2 1 6

      if(grid[lr, index[1]]==1){
        grid[lr, s] = 1
        grid[lr, index[1]] = 0
        count = count+1
      }
      for(lc in 2:(length(index)-1)){
        if(grid[lr, index[lc]]==1 & grid[lr, index[lc-1]]==0){
          grid[lr, index[lc-1]] = 1
          grid[lr, index[lc]] = 0
          count = count+1
        }
      }
    }
  }
  class(grid) = oriClass
  list(grid=grid, velocity=count)
}

moveBlue.simul = function(grid, r, c){
  #blue is 1
  oriClass = class(grid)
  count = 0
  r = nrow(grid)
  c = ncol(grid)
  for(lc in 1:c){
    s = head(which(grid[ ,lc]==0),1) #most top empty
    if(length(s)){
      if(s==r)
        index=c(1:r)
      else
        index = c((s+1):r, 1:s) #forwards ex.3 4 5 1 2


      if(grid[index[1], lc]==2){
        grid[s, lc] = 2
        grid[index[1], lc] = 0
        count = count+1
      }
      for(lr in 2:(length(index)-1)){
        if(grid[index[lr], lc]==2 & grid[index[lr-1], lc]==0){
          grid[index[lr-1], lc] = 2
          grid[index[lr], lc] = 0
          count = count+1
        }
      }
    }
  }
  class(grid) = oriClass
  list(grid=grid, velocity=count)
}
```

## Appendix II - Package

```
# DESCRIPTION
Package: bml
Title: Biham Middleton Levine Traffic Model Toolkit
```

```
Description: Tools to simulate Biham Middleton Levine Traffic Models
       and detect phase transition point.
Version: 0.1-0
License: BSD
Author: Qiwei Li
Maintainer: Qiwei Li  <qwli@ucdavis.edu>

 # NAMESPACE
export(runBMLGrid)
export(createBMLGrid)
S3method(plot, BML)
S3method(summary, BML)

# help pacge
\name{createBMLGrid}
\alias{createBMLGrid}
\title{createBMLGrid}
\description{Create a BML simulation grid}
\usage{
createBMLGrid(r, c, ncars = c(red = 0, blue = 0), density = 0)
}
\arguments{
\item{r}{An integer representing row size of the grid}
\item{c}{An integer representing column size of the grid}
\item{ncars}{An integer vector of length 2 representing number of blue cars and red cars in
the grid (optional)}
\item{density}{An numeric value representing the proportion of cars in the grid. Blue cars and
red cars will split evenly.}
}
\value{
returns a BML grid
}
\examples{
grid = createBMLGrid(r = 67, c = 127,  density = 0.5)
grid = createBMLGrid(r = 67, c = 127,  ncars = c(red = 1000, blue = 1500))
}
\keyword{BML}

\name{runBMLGrid}
\alias{runBMLGrid}
\title{runBMLGrid}

\description{
Simulates BML model
}

\usage{
runBMLGrid(grid, numSteps, ifPlot = FALSE, method)
}
```

```
\arguments{
\item{grid}{A BML grid}
\item{numSteps}{An integer representing number of moves of the simulation}
\item{ifPlot}{An logical value representing if the animation of simulation should be plotted}
\item{method}{One of the following methods: "fast", "slow", and "simultaneous"}
}

\value{
return the final grid and plot number of moved cars (velocity) over time
}

\examples{
grid = createBMLGrid(r = 67, c = 127,  density = 0.5)
g = runBMLGrid(grid, numSteps = 1000, ifPlot=TRUE, method = "fast")
g = runBMLGrid(grid, numSteps = 1000, ifPlot=FALSE, method = "slow")
g = runBMLGrid(grid, numSteps = 1000, ifPlot=TRUE, method = "simultaneous")
}

\keyword{BML}
```