Biham-Middleton-Levine Traffic Model

Shuhua Liang

February 2, 2013

The Biham-Middleton-Levine (BML) Traffic Model mimics the movements of cars and traffic jams by simulating blue and red grids -used to represent two types of cars - on a rectangular plane. On that plane, blue cars can only move up by one grid at each time point, and red cars can only move to the right by one grid. If a car is blocked by one of the opposite color, it will have to skip a move at that particular time point. Also, when a car is at the edge of the plane, it will wrap around the plane and start the moving process again from the other end of the plane.

Given different dimensions of the plane and different proportions of grid to fill, the movement of cars will vary, and as the proportion of cars increase, the cars will jam and cluster. Below are five figures that illustrates the traffic situation at different time points.

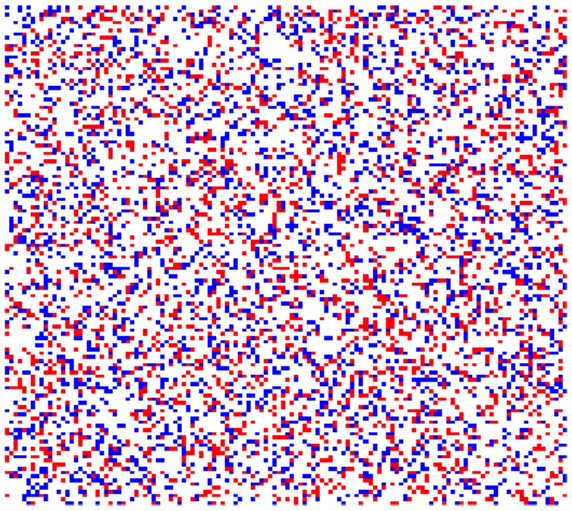


Figure 1: time $= 0$, $\rho = 0.3$

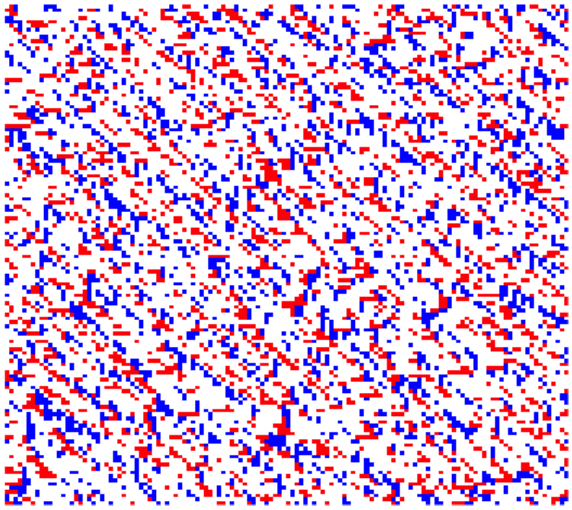


Figure 2: time = 50, $\rho = 0.3$

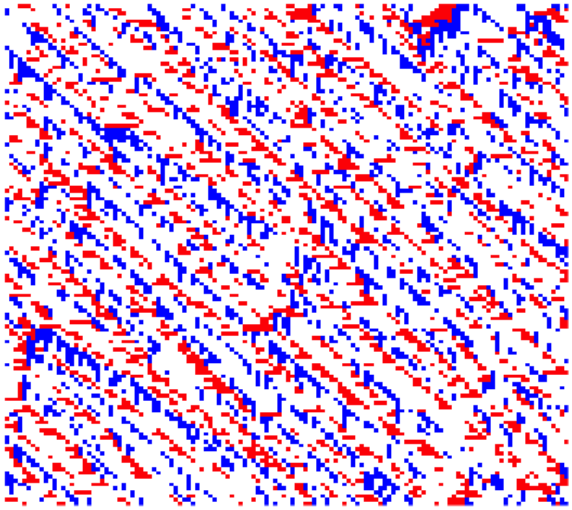


Figure 3: time = 100, $\rho = 0.3$

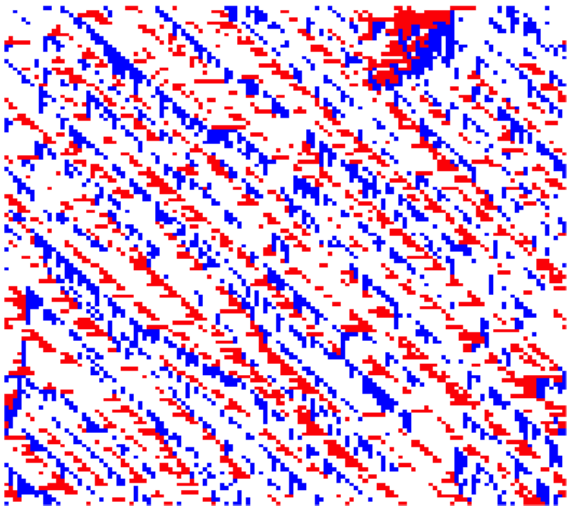


Figure 4: time = 150, $\rho = 0.3$

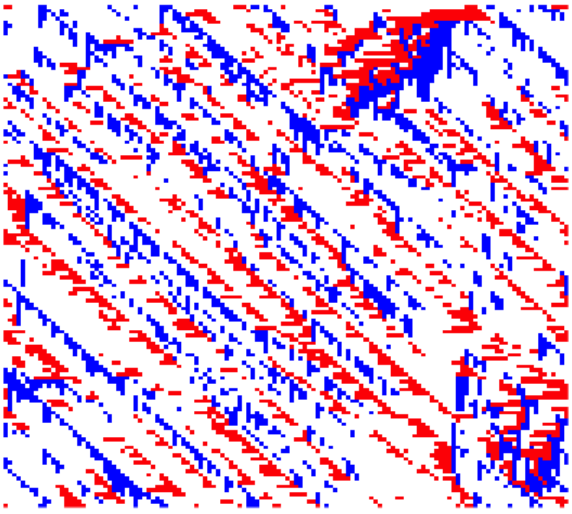


Figure 5: time = 200, $\rho = 0.3$

In the example, the plane has dimension 130 by 130 and 30% of the grids are filled with cars. When time equals 0, data is generated randomly with the given information; it is the originals state where no cars are moved. At time one (not shown), unblocked blue cars will all moved up by one grid, and then at time two, the unblocked red cars will move to the right by one grid. This pattern will continue on for a given number of time periods. The first plane shows the model at time zero, where all the cars are scattered. At time 50 (Figure 2), cars start to block each other, and more white spaces show. Compared to the first two figure, Figure

3 shows that the cars gather diagonally. Then at time 150 and 200, cars start to cluster, and the spread of cars did not change much.

At each time point, the number of cars moved or unmoved can be calculated, and they explained by the following plots.

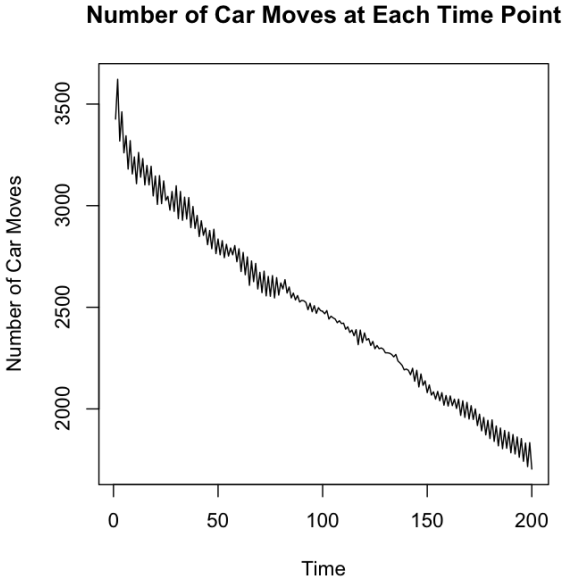


Figure 6: Number of Cars Moved

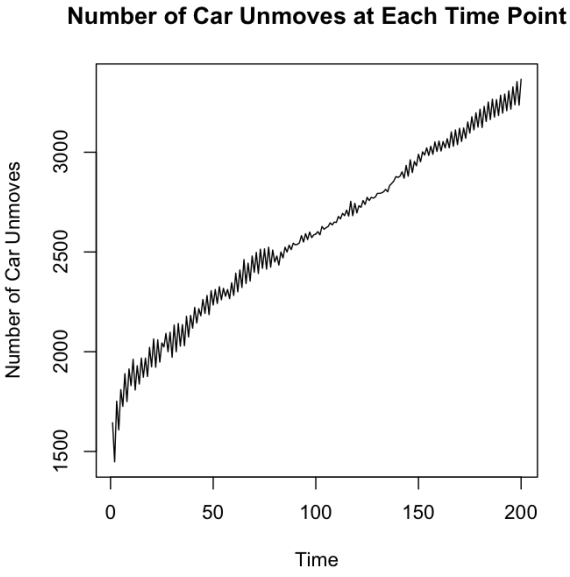


Figure 7: Number of Cars Unmoved

In the beginning, most cars are able to move. Figure 6 shows that out of 9000 cars in the plane of 30000 grids, almost 3500 blue cars moved at time 1. As time increases, the number of cars moved decreases. At time 200, less than 500 cars were able to move. On the other hand, Figure 7 shows the count of cars that are not moved, and it shows an exactly opposite story than FIgure 6. As time goes, the number of cars unmoved decreased. In conclusion, the traffic jam gets more serious as time increases, and cars cluster if they are only limited to a certain aera.

In a different example of the same size plane but 70% of girds filled, the cars will stop at an earlier time period (See Figures 8-10). Compare the first plane at time zero with $\rho = 0.3$ to the first plane where $\rho = 0.7$, the latter plane is much more crowded. Also, when $\rho = 0.7$, the cars stop moving at time 30, which can easily see that the planes for time 30 and time 50 are almost identical.

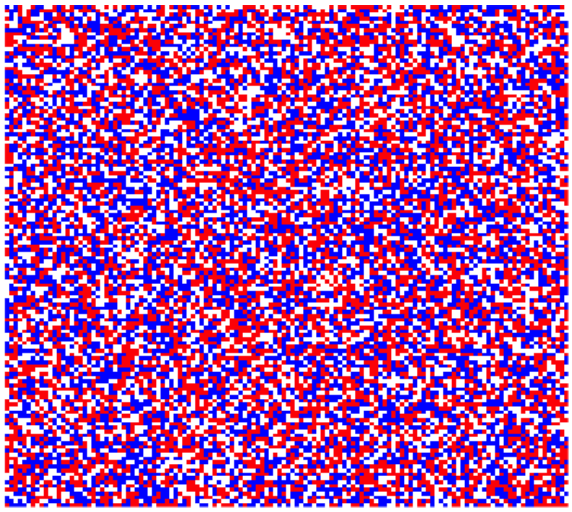


Figure 8: time = 0, $\rho = 0.7$

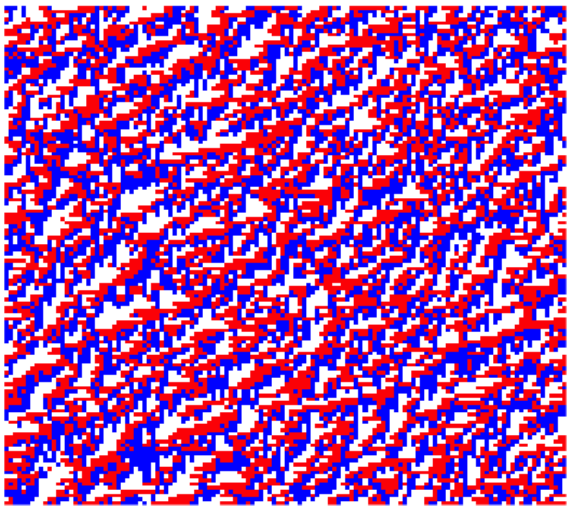


Figure 9: time = 30, $\rho = 0.7$

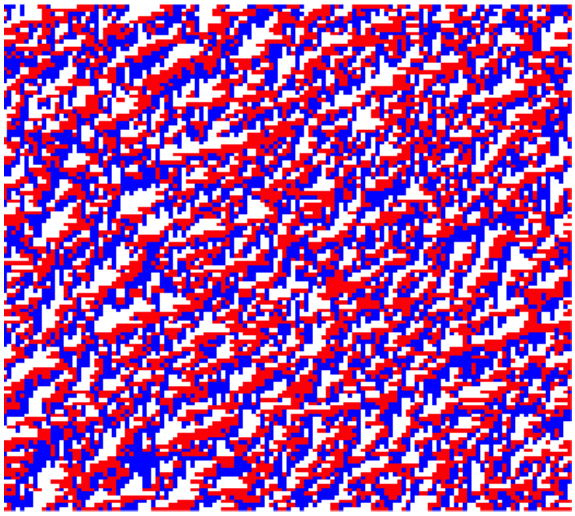
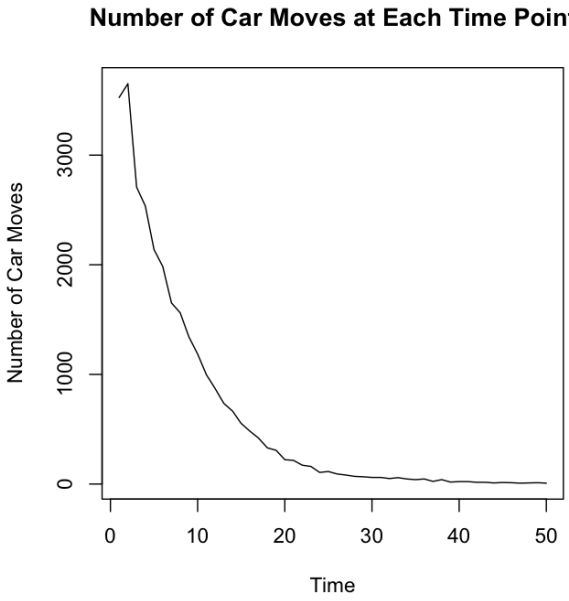
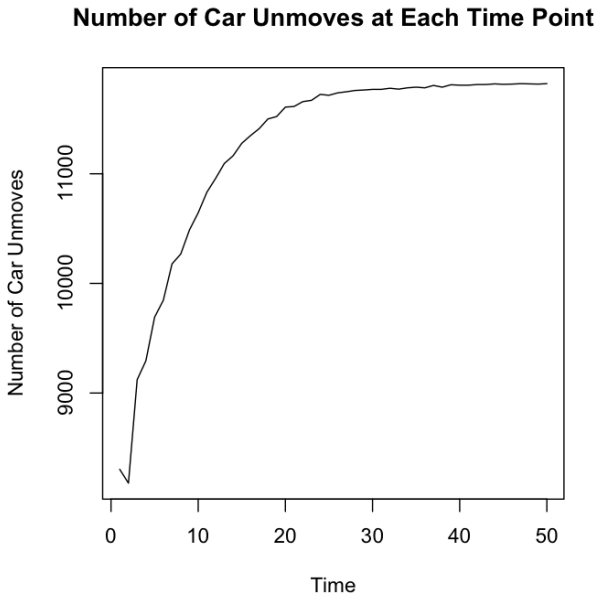


Figure 10: time = 50, $\rho = 0.7$

The plots below shows an exponential distribution for the count of car moves, and these plot are smoother compared to the ones for $\rho = 0.3$ because there are much smaller spaces for cars to more in a denser plane.





One way to present the Biham-Middleton-Levine Traffic Model is to generate a plane of grids and record its changes at each time point. Then one can convert those planes into a .gif file and see the movements of cars in a sequence (GIF file attached in e-mail).

Lastly, the functions that are used to calculate and mimic this process are converted to an R package called BMLpkg (also attached in e-mail). This allows users to simply install the package, apply the functions to any given information, and observe the changes in BML model.

```
## http://arxiv.org/pdf/0709.3604v3.pdf
## http://mae.ucdavis.edu/dsouza/bml.html

## http://www.personality-project.org/R/makingpackages.html

## Set.seed() before applying functions!!!!

## Blue up, t = 1,3,... (Move first!)
## Red right, t = 2,4,...
setwd("/Users/shuhualiang/Documents/Davis MS/STA 242")

options(error = recover)
library(grDevices)

## Move Up (blue):
################################################################################
## Generate the number of blue and red cars, then to them in a plane:
gen.cars = function(hor.grids, ver.grids, rho){
    total.grids = hor.grids*ver.grids
    red <- blue <- rho/2
    gen = sample(0:2, size=total.grids, replace=TRUE, prob=c((1-rho), red, blue))
    plane = matrix(gen, nrow=ver.grids, ncol=hor.grids)
}

#class(x) = "BML"
plot.BML = function(x,...){
    image(x, col=c("white","red","blue"),axes=FALSE)
}
    ## try: gen.cars(120,170,.3)
    ## 2 -> Blue, 1 -> Red

## Adjust directions, so matrix follows the direction of the plane:
turn.mat.90left = function(gen.cars){
    apply(gen.cars,1,rev)
}

turn.back.90right = function(mat){
    list.back = tapply(mat,rep(1:ncol(mat),each=nrow(mat)),function(i)i)
    revert = lapply(list.back,rev)
    matrix(unlist(revert),byrow = TRUE,nrow=length(revert))
}
## For partial vectors, move up one and make the last one zero:
one.up = function(vec){
    if(length(vec) <= 1) return(vec)
    else c(vec[2:length(vec)],0)
}

## Use when have a partial vector that comes after a 1:
two.zero.and.more = function(part){
    if(all(part==2) || length(part) <= 1) return(part)
    else {
        first.zero = min(which(part == 0))
```

```
        part[first.zero:length(part)] = one.up(part[first.zero:length(part)])
        return(part)
    }
}

## Rotate the entire vector by one element up:
rot <- function(x) (1:x %% x) +1
rotvec <- function(vec){vec[rot(length(vec))]}

## Operate the sequence in between two ones:
run.two = function(vec, ones){
    for(i in 1:(length(ones)-1)){
        if(0 %in% vec[(ones[i]):(ones[i+1])]){
            vec[(ones[i]+1):(ones[i+1]-1)] = two.zero.and.more(vec[(ones[i]+1):(ones[i
+1]-1)])}
    }
    return(vec)
}

### Move up function when there is more than one 1:
move.up.more1 <- function(vec){
    ones = which(vec==1)
    if(vec[1]==1 && vec[length(vec)]==1){
        vec = run.two(vec,ones)
        return(vec)
    }
    if(vec[1]==1 && vec[length(vec)]!=1){
        vec[(ones[length(ones)]+1):(length(vec))] =
two.zero.and.more(vec[(ones[length(ones)]+1):(length(vec))])
        vec = run.two(vec,ones)
        return(vec)
    }
    if(vec[1]!=1 && vec[length(vec)]==1){
        vec[1:(ones[1]-1)] = two.zero.and.more(vec[1:(ones[1]-1)])
        vec = run.two(vec,ones)
        return(vec)
    }
    if(vec[1]==2 && vec[length(vec)]!=1){
        vec[(ones[length(ones)]+1):(length(vec))] =
two.zero.and.more(vec[(ones[length(ones)]+1):(length(vec))])
        if(vec[length(vec)]==0){
        vec[1]=0
        vec[1:(ones[1]-1)] = two.zero.and.more(vec[1:(ones[1]-1)])
        vec[length(vec)]=2}
        vec = run.two(vec,ones)
        return(vec)
    }
    else{   #(vec[1]!=1 && vec[length(vec)]!=1)
        vec[(ones[length(ones)]+1):(length(vec))] =
two.zero.and.more(vec[(ones[length(ones)]+1):(length(vec))])
        vec[1:(ones[1]-1)] = two.zero.and.more(vec[1:(ones[1]-1)])
        vec = run.two(vec,ones)
```

```
        return(vec)
    }
}

## More a vector up 1 if there is only one 1 in the vector:
move.up.one1 = function(vec){
    one = which(vec==1)
    if(one==1){
        vec[2:length(vec)] = two.zero.and.more(vec[2:length(vec)])
        return(vec)}
    if(one==length(vec)){
        vec[1:(length(vec)-1)] = two.zero.and.more(vec[1:(length(vec)-1)])
        return(vec)}
    if(one!=1 && one!=length(vec)){
        if(vec[1]==2){
            vec[1]=0
            vec[1:(one-1)] = two.zero.and.more(vec[1:(one-1)])
            vec[(one+1):length(vec)] = two.zero.and.more(vec[(one+1):length(vec)])
            if(vec[length(vec)]==0) vec[length(vec)]=2
        return(vec)
        }
        else{
        vec[1:(one-1)] = two.zero.and.more(vec[1:(one-1)])
        vec[(one+1):length(vec)] = two.zero.and.more(vec[(one+1):length(vec)])
        return(vec)
        }
    }
}

## The move up function! Given a vector, the cars move up by 1:
move.up = function(vec){
    ones = which(vec == 1)
    if(length(ones)==0){
        vec = rotvec(vec)
        return(vec)
    }
    if(length(ones) == 1){move.up.one1(vec)}
    else{move.up.more1(vec)}
}

## Input a matrix and move blues up one:
oneUp.matrix = function(mat){
    li = tapply(mat,rep(1:ncol(mat),each=nrow(mat)),function(i)i)
    sapply(li, move.up)
}

#############################################################################################

## Move Right (Red):
#############################################################################################
## Switch numbers:
swap = function(mat){
```

```
    mat[which(mat==1)] <- 3
    mat[which(mat==2)] <- 1
    mat[which(mat==3)] <- 2
    return(mat)
}

## Turn to right direction as will be plotted:
Turn2 = function(mat){
    bat = turn.mat.90left(mat)
    cat = turn.mat.90left(bat)
    return(cat)
}

TurnBack2 = function(mat){
    bat = turn.back.90right(mat)
    cat = turn.back.90right(bat)
    return(cat)
}

#############################################################################################
## Final functions:

## Function that moves blue cars when given a matrix:
Blue = function(matrix){
    rightDirection = turn.mat.90left(matrix);rightDirection
    moved.up = oneUp.matrix(rightDirection)
    rotBack = turn.back.90right(moved.up)
    return(rotBack)
}

## Function that moves red cars when given a matrix:
Red = function(matrix){
    changed.num.ratBack = swap(matrix);changed.num.ratBack
    Ready2Up = Turn2(changed.num.ratBack);Ready2Up
    Moved.right = oneUp.matrix(Ready2Up);Moved.right
    Origin.Dir = TurnBack2(Moved.right); Origin.Dir
    ready2plot = swap(Origin.Dir)
    return(ready2plot)
}

## Function that takes in time, and alternates between blue and red car movements:
Drive = function(time,hgrid, vgrid, rho,...){
    carPlane = gen.cars(hgrid, vgrid, rho)
    #mat = carPlane
    class(carPlane) = "BML"
    png(file = "Anna0")
    print(plot(carPlane))
    dev.off()

    Annas = rep("Anna",time)
    Annas = paste(Annas, 1:time, sep="")
```

```
    moved = c()
    unmoved = c()
    for(i in 1:time){
        if(i %% 2 !=0){
            plane = Blue(carPlane)
            moved[i] = length(which((carPlane == plane)==FALSE))
            unmoved[i] = hgrid*vgrid*rho - moved[i]
            carPlane = plane
            class(carPlane) = "BML"
            png(Annas[i])
            print(plot(carPlane))
            dev.off()
        }
        if(i %% 2 ==0){
            plane = Red(carPlane)
            moved[i] = length(which((carPlane == plane)==FALSE))
            unmoved[i] = hgrid*vgrid*rho - moved[i]
            carPlane = plane
            class(carPlane) = "BML"
            png(Annas[i])
            print(plot(carPlane))
            dev.off()
        }
    }
    par(mfrow=c(1,2))
    (plot(seq(time),moved,type="l",xlab = "Time",ylab="Number of Car Moves",main="Number
 of Car Moves at Each Time Point"))
    (plot(seq(time),unmoved,type="l",xlab = "Time",ylab="Number of Car
 Unmoves",main="Number of Car Unmoves at Each Time Point"))
}

## Functions that creates a GIF and cleans up the working directory:
video = list.files(pattern="Anna");video
new = paste(video,".png",sep="");new
Rename = file.rename(video, new)
GIF = system("convert -delay 1 *.png Moves.gif")

Remove = file.remove(new)

##############################################################################################
## Creating the package:
mylist = c(
"gen.cars",
"plot.BML",
"turn.mat.90left",
"turn.back.90right",
"one.up",
"two.zero.and.more",
"rot",
"rotvec",
"run.two",
"move.up.more1",
```

```
"move.up.one1",
"move.up",
"oneUp.matrix",
"swap",
"Turn2",
"TurnBack2",
"Blue",
"Red",
"Drive",
"video",
"new",
"Rename",
"GIF",
"Remove")

package.skeleton("BMLpkg", mylist, environment = .GlobalEnv, path = ".", force = TRUE)
```