Appendix: Code

1.C functions and R functions changed

1.1 C functions

1.1.1 First method of moving cars at one time period

```c
/**
 * Function: move
 * ------------------------------------------------------------
 * move different kind of cars in one step(by color)
 * parameters:
 *              *r: the number of rows of a grid
 *              *c: the number of columns of a grid
 *              *grid: the content of grid matrix (stored by columns)
 *              *color: which kind of car need to move(1 means red and 2 means blue)
 *              *velocity: the velocity of cars in this moving step
 */
void move(int *r, int *c, int *grid, int *color, double *velocity)
{

    int tot = r[0] * c[0];    /*the total number of elements in grid*/


    /*get the number of cars in that color*/
    int numColor = 0;
    for(int j = 0; j < tot; j++){
      if(grid[j] == *color )    {
           numColor = numColor + 1;
        }
    }

    /*If there is no car in that color, set velocity to 0 and no need to move*/
    if(numColor == 0){
        *velocity = (double)0;
        return;
    }

    /*record current position and next position of cars which can move*/
    int* curPos = (int *) malloc ( sizeof(int) * numColor );
    int* nexPos = (int *) malloc ( sizeof(int) * numColor );

    if(curPos == NULL || nexPos == NULL){
        printf("Memory allocation failed");
        return;
    }else{
        for(int i = 0; i < numColor; i++){
            curPos[i] = 0;
```

```
            nexPos[i] = 0;
        }
    }



    int k = 0;   /*the number of moveable cars*/
    int nexPosI = 0;


    if(*color == 1){
        /*move red cars to the right*/
        for(int i = 0; i < tot; i++){
            if(grid[i] == color[0]){

                /*compute the next position*/
                nexPosI = i + r[0];
                if(nexPosI >= tot)
                    nexPosI = nexPosI - tot;

                /*if the car can move, record current position and next position*/
                if(grid[nexPosI] == 0)
                {
                    curPos[k] = i;
                    nexPos[k] = nexPosI;
                    k = k + 1;
                }
            }
        }

    }else if(*color == 2){
        /*move blue cars upwards*/
        int numColum = 0;
        int numRow = 0;
        for(int i = 0; i < tot; i++){
            if(grid[i] == color[0]){

                /*compute the next position*/
                numColum = i / r[0];
                numRow = i % r[0] ;
                nexPosI = numRow + 1;
                if(nexPosI >= r[0])
                    nexPosI = nexPosI - r[0];
                nexPosI = numColum * r[0] + nexPosI;

                /*if the car can move, record current position and next position*/
                if(grid[nexPosI] == 0)
                {
                    curPos[k] = i;
                    nexPos[k] = nexPosI;
                    k = k + 1;
                }
```

```
            }
        }

    }else{
        *velocity = (double)0;
        return;
    }

    velocity[0] = (double)k / (double)numColor; /*update velocity*/

    /*change positions in grid*/
    for(int i = 0; i < k; i++){
        grid[curPos[i]] = 0;
        grid[nexPos[i]] = *color;

    }

    free(curPos);
    free(nexPos);
    curPos = NULL;
    nexPos = NULL;

}
```

1.1.2 Second method of moving cars at one time period

```

/**
 * Function: move2
 * ---------------------------------------------------------
 * another method of moving different kind of cars in one step(by color)
 * parameters:
 *              *r: the number of rows of a grid
 *              *c: the number of columns of a grid
 *              *grid: the content of grid matrix (stored by columns)
 *              *color: which kind of car need to move(1 means red and 2 means blue)
 *              *velocity: the velocity of cars in this moving step
 */
void move2(int *r, int *c, int *grid, int *color, double *velocity)
{

    int tot = r[0] * c[0];  /*the total number of elements in grid*/

    /*copy content ofgrid to nexGrid*/
    int* nexGrid = (int *) malloc ( sizeof(int) * tot );
    if(nexGrid == NULL){
        printf("Memory allocation failed");
        return;
    }else{
        for(int i = 0; i < tot; i++){
            nexGrid[i] = 0;
        }
    }
```

```
    memcpy(nexGrid, grid, tot * sizeof(int));


    int numColor = 0; /*the number of cars in that color*/
    int numMove = 0; /*the number of moveable cars*/
    int nexPosI = 0; /*inner variable to record next postion of each car*/

    if(color[0] == 1){
        /*move red cars to the right*/
      for(int i = 0; i < tot; i++){
            if(nexGrid[i] == color[0]){
                numColor = numColor + 1;

                /*compute new position*/
                nexPosI = i + r[0];
                if(nexPosI >= tot)
                    nexPosI = nexPosI - tot;

                /*if the car can move, update grid directly*/
                if(nexGrid[nexPosI] == 0){
                    grid[nexPosI] = color[0];
                    grid[i] = 0;
                    numMove = numMove + 1;
                }
            }
        }
    }else if(color[0] == 2){
        /*move blue cars upwards*/
        int numColum = 0;
        int numRow = 0;

        for(int i = 0; i < tot; i++){
            if(nexGrid[i] == color[0]){
                numColor = numColor + 1;

                /*compute new position*/
                numColum = i / r[0];
                numRow = i % r[0] ;
                nexPosI = numRow + 1;
                if(nexPosI >= r[0])
                    nexPosI = 0;
                nexPosI = numColum * r[0] + nexPosI;

                /*if the car can move, update grid directly*/
                if(nexGrid[nexPosI] == 0){

                    grid[nexPosI] = color[0];
                    grid[i] = 0;
                    numMove = numMove + 1;
                }
            }
        }

```

```
    }

    if (numColor == 0)
    {
        velocity[0] = (double)0;
    }else{
        velocity[0] = (double)numMove / (double)numColor; /*update velocity*/
    }


    free(nexGrid);
    nexGrid = NULL;

}
```

1.1.3 Moving cars several times.

```
/**
 * Function: runSteps
 * ----------------------------------------------------------------
 * move the cars several times
 * parameters:
 *              *numSteps: number of runs
 *              *r: the number of rows of a grid
 *              *c: the number of columns of a grid
 *              *grid: the content of grid matrix (stored by columns)
 *              *velocity: the velocity of cars in each step
 */
void runSteps(int *numSteps, int *r, int *c, int *grid, double *velocity)
{
  double v = 0;
    double *p1 = &v;

    int color = 1;
    int *c1 = &color;

    for(int i = 0; i < numSteps[0]; i++){
        /*move blue cars first and then red cars and replicate*/
        if(i%2 == 0)
            *c1 = 2;
        else
            *c1 = 1;

        move(r, c, grid, c1, p1);
        velocity[i] = *p1;
    }

}
```

1.2 R function in BML package

1.2.1 cmoveCars function

```
#move cars at one time period
#color = 1 means moving red cars and 2 means moving blue cars
cmoveCars = function(g, color = 1){
  r = nrow(g)
  c = ncol(g)
  v = 0
  out = .C("move2",r = as.integer(r), c = as.integer(c),g = g,
           col = as.integer(color),v = as.numeric(v))
  return(list(grid = out$g, velocity = out$v))
}
```

1.2.2 crunBMLGrid function

```
#move cars several times
#the final grid and velocity in each step are returned
crunBMLGrid = function(g, numSteps = 10000){
  if(numSteps > 0){
    r = nrow(g)
    c = ncol(g)
    v = rep(0, numSteps)
    out = .C("runSteps", numSteps = as.integer(numSteps),
             r = as.integer(r), c = as.integer(c),g = g, v = as.numeric(v))
    return(list(grid = out$g, velocity = out$v))
  }else{
    return(list(grid = g, velocity = 0))
  }
}
```

2.Unit Test

Since it is similar when testing moveCars, cmoveCars, cmoveCars2, we will only give three unit test code here(test createBMLGrid, cmoveCars, crunBMLGrid).

2.1 Test createBMLGrid function

```
library(RUnit)

test.createBMLGrid = function(){

  #test if row number < 0
  checkException(createBMLGrid(r = -1, c = 4, 0.3))

  #test if density<0
  checkException(createBMLGrid(10, 10 , -0.5 ))

```

```
  #test if number of cars larger than grid
  checkException(createBMLGrid(10, 10, c(red = 50, blue = 51)))

  checkEquals(nrow(createBMLGrid(10,9,0.3)), 10)

  checkEquals(sum(createBMLGrid(10,9,c(red = 10, blue = 11)) > 0), 21)

}
```

2.2 Test cmoveCars function

```
test.cmoveCars = function(){

  #case 1
  g = matrix(as.integer(c(0, 1, 0,2,1,0)),2,3)
  class(g) = c('BML', class(g))

  redg = matrix(as.integer(c(1, 1, 0,2,0,0)),2,3)
  class(redg) = c('BML', class(redg))

  blueg = matrix(as.integer(c(0, 1, 2,0,1,0)),2,3)
  class(blueg) = c('BML', class(blueg))

  #check move red cars
  checkEquals(cmoveCars(g,1)$grid, redg)
  checkEquals(cmoveCars(g,2)$grid, blueg)


  #case 2: only one kind of car
  g = matrix(as.integer(c(1, 1, 0,0,1,0)),2,3)
  class(g) = c('BML', class(g))

  redg = matrix(as.integer(c(0, 0, 1,1,1,0)),2,3)
  class(redg) = c('BML', class(redg))

  blueg = matrix(as.integer(c(1, 1, 0,0,1,0)),2,3)
  class(blueg) = c('BML', class(blueg))

  #check move red cars
  checkEquals(cmoveCars(g,1)$grid, redg)
  checkEquals(cmoveCars(g,2)$grid, blueg)


  #case 3: only one row
  g = matrix(as.integer(c(0, 1, 2,0,1,0)),1,6)
  class(g) = c('BML', class(g))

  redg = matrix(as.integer(c(0, 1, 2,0,0,1)),1,6)
  class(redg) = c('BML', class(redg))

  blueg = matrix(as.integer(c(0, 1, 2, 0, 1, 0)),1,6)
  class(blueg) = c('BML', class(blueg))
```

```
  #check move red cars
  checkEquals(cmoveCars(g,1)$grid, redg)
  checkEquals(cmoveCars(g,2)$grid, blueg)
}
```

2.3 Test crunBMLGrid function

```
test.crunBMLGrid = function(){

  #case 1: no car is blocked in the end
  g = createBMLGrid(10, 10, 0.3)
  checkEquals(crunBMLGrid(g,1000), runBMLGrid(g, 1000))
  checkEquals(crunBMLGrid(g,1000), crunBMLGrid2(g, 1000))

  #case 2: all cars are blocked in the end
  g = createBMLGrid(100, 100, 0.5)
  checkEquals(crunBMLGrid(g,1000), runBMLGrid(g, 1000))
  checkEquals(crunBMLGrid(g,1000), crunBMLGrid2(g, 1000))

  #case 3: only one kind of car
  g = createBMLGrid(100, 100, c(red = 500, blue = 0))
  checkEquals(crunBMLGrid(g,1000), runBMLGrid(g, 1000))
  checkEquals(crunBMLGrid(g,1000), crunBMLGrid2(g, 1000))

  #case 4: only one column
  g = createBMLGrid(100, 1, c(red = 20, blue = 20))
  checkEquals(crunBMLGrid(g,1000), runBMLGrid(g, 1000))
  checkEquals(crunBMLGrid(g,1000), crunBMLGrid2(g, 1000))

  #case 5:
  g = matrix(as.integer(c(0, 1, 0,2,1,0)),2,3)
  class(g) = c('BML', class(g))

  g2 = matrix(as.integer(c(1,0,2,1,0,0)),2,3)
  class(g2) = c('BML', class(g2))

  #check move red cars
  checkEquals(crunBMLGrid(g,10)$grid, g2)
}
```

3. Performance comparision

3.1 Get running time with different settings

```
library(reshape)

#density
density = seq(0.1,0.8,by = 0.1)
#size
```

```r
size = c(10,20,50,100,200,500)
nruns = 10000

v1 = matrix(rep(0,6*8),6,8) #store result of r method
v2 = matrix(rep(0,48),6, 8) # store result of C1 code
v3 = matrix(rep(0,48),6, 8) # store result of C2 code

for(i in 1:length(size)){
  for(j in 1:length(density)){
    g = createBMLGrid(size[i],size[i],density[j])
    v1[i,j] = unname(system.time(runBMLGrid(g, 5000))[1])
    v2[i,j] = unname(system.time(crunBMLGrid(g, 5000))[1])
    v3[i,j] = unname(system.time(crunBMLGrid2(g, 5000))[1])
  }
}

#merge them together
tmpv1 = as.data.frame(t(v1))
names(tmpv1) = size
tmpv1$density = density
tmpv1$method = 'R'
tmpv2 = as.data.frame(t(v2))
names(tmpv2) = size
tmpv2$density = density
tmpv2$method = 'C1'
tmpv3 = as.data.frame(t(v3))
names(tmpv3) = size
tmpv3$density = density
tmpv3$method = 'C2'
v = rbind(tmpv1,tmpv2)
v = rbind(v, tmpv3)
v = melt(v, id = c('density','method'), variable_name = "Size")
```

3.2 Plot functions

```r
#plot running time with different density(bar chart)
require(ggplot2)
qplot(x=density, y=value, fill=method, data=v[v$Size == '200',], geom="bar",
      stat="identity", xlab = "Density", ylab = "User Time(seconds)",
      position="dodge",
      main = "Running time with different density(200*200 Grid, runSteps = 10000)")

#running time with different grid size
qplot(x=Size, y=log(value), fill=method, data=v[v$density == '0.3',], geom="bar",
      stat="identity", xlab = "Size", ylab = "log(User Time(in seconds))",
      position="dodge",
      main = "Running time with different size (density =0.3, runSteps = 10000)")
```