UNIVERSITY OF CALIFORNIA,DAVIS
STA242
SPRING 2015

Assignment 4

Junxiao Bu
999452701
SSH: git@bitbucket.org:shingtime/sta242-project2.git

May 14, 2015

PART I: Purpose and Organization

In this project, the C version of *runBMLGrid* is implemented. The main purpose is comparing the running speeds' difference between R version and C version. Specificlly, the different grid size and density will be discussed separately. Besides, some verifications are present, which are related the consistency of R and C code.

Part II: Algorithm Introdcution

CrunBMLGrid, the C version of *runBMLGrid*, is implemented. The algorithm is same as what I implemented using R.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. All cars must move simultaneously. 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.

To verify if the simualation results are same through *runBMLGrid* and *CrunBMLGrid*. An 3*3 grid with 4 steps of movements are shown.
The initial grid is shown first.

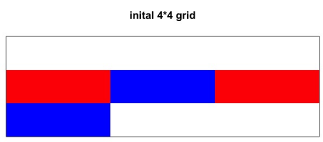

Figure 1: inital Grid of 4*4

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

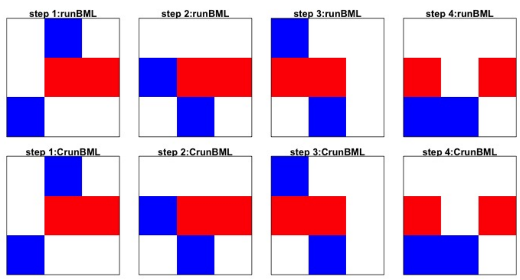

Figure 2: Movement via R and C

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 results are same through *CrunBMLGrid* or *runBMLGrid*.

For further confirmation, I simulated the process of 128*128 grid with density 0.5. The iteration is 100 times. The plots are shown below.

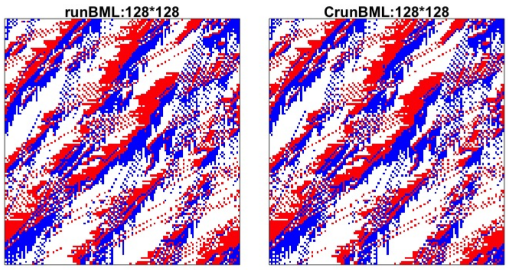

Figure 3: 128*128 grid via R and C

Part III: Prodfiling Code

As mentioned before, my R code is slower than the C code. The profiling tables of different are shown below. As the benchmark, I use a150*150 grid with 3000 number cars for each color. The iteration will be 100 times.

Table 1: Profiling of loop version in R

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

Table 2: Profiling of vectorized version in R

Time Type(seconds)	self.time	self.pct	total.time	total.pct
"move_cars"	0.08	19.05	0.42	100.00
".change_color"	0.08	19.05	0.1	23.81
"which"	0.04	9.52	0.08	19.05
"cbind"	0.04	9.52	0.04	9.52
"car_coordinate"	0.02	4.76	0.2	47.62

The comparison of system time is shown below.

Table 3: Profiling of vectorized version in R

Time Type(seconds)	self.time	self.pct	total.time	total.pct
".C"	0.02	100	0.02	100

Table 4: Time for running different code

Time Type(seconds)	User Time	System Time	Elapsed Time
before vectorization	69.117	12.023	81.312
after vectorization	0.496	0.082	0.578
C version	0.010	0.001	0.011

From the above tables, by comparing the user time , the loop version of R code is definitely the slowest. The C code is fastest. There is a very significant speedup from the loop-version of the code.

So we get a speedup of 50 times by using C code. The C code significantly outperforms the vectorized version written in R. I This doesnt mean R is a bad language. One possible reason is that my vectorized version is not that efficient. The other one is that R's dynamic nature, which makes it easy to express computations, often leads to slower code than with compiled languages like C.

PART IV: Performance over Different Grid sizes

Next, the impacts of different grid size on the performance of the code are going to explored. The vectorized version R code and C code's performance will be discussed. First, the performance of the code over different grid sizes will be discussed. Here I use the grid sizes of 4*48*8, 16*16, 32*32, 64*64, 128*128, 256*256 and 512*512,and 1024*1024. And I choose three densities to check the difference. The first density is 0.2(traffic flow). The second density is 0.35(transition phase). The third density is 0.7(jam phase). The iteration will be 100 times. The plot is shown below.

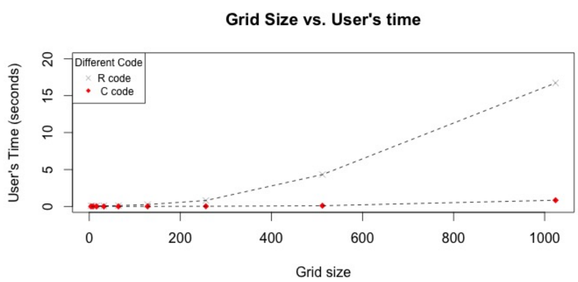

Figure 4: Density:0.2

For R code, these plot show that as the number of grid size, and the total number of cars, doubles, the time taken approximately doubles. But for C code, as grid sizes

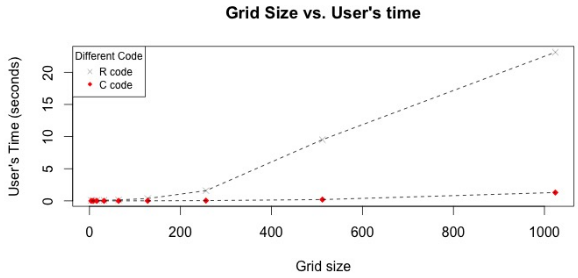

Figure 5: Density:0.35

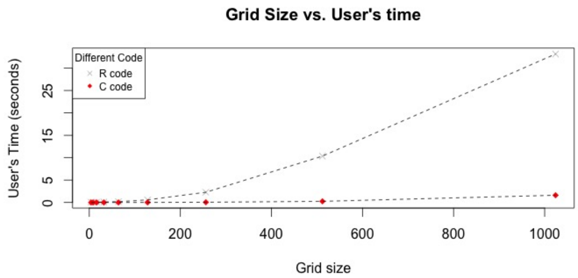

Figure 6: Density:0.7

increasing, the time taken approximately increases far less than grid sizes. The user time of C code are always significantly shorter than R code's. I calculate the ratio of different code's user time. It seems that the ratio is decreasing as the grid sizes increases. As grid sizes increases, this ratio may unchanged in a range. Due to the limitation of computing source, further computations are ignored.

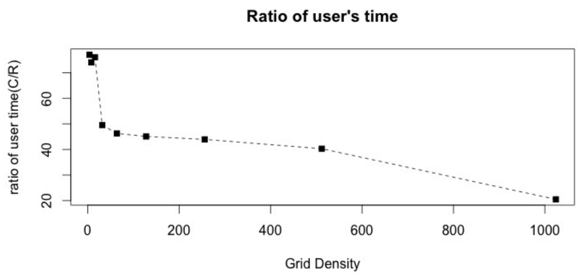

Figure 7: Density:0.7

PART V: Performance over Different Densities

Second, the performance of the code over different densities will be discussed. Here I use the grid size 64*64,128*128 and 256*256. The density is in the range of 0.1 to 0.7.(equally split in 13 blocks). The iteration will also be 100 times. The plots are shown below.

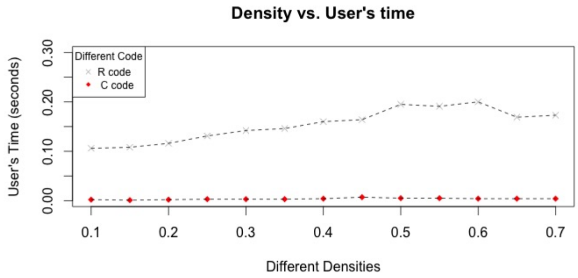

Figure 8: 64*64 Grid with different densities

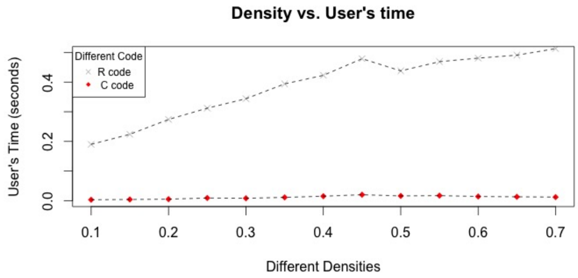

Figure 9: 128*128 Grid with different densities

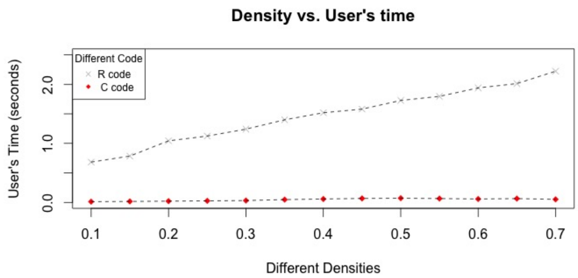

Figure 10: 256*256 Grid with different densities

For C code, these plot show that as the density increases, the time taken increases(not too significant). For R code, as densities increasing, the time taken approximately increases significantly. The performance of C code is always better than R code.

.

PART VI: Performance over Different iteration times

Next, the obvious relation between iteration time and user's time is explored. For both of C and R code, the user's time should increase as grid sizes increases. Here I choose 128*128 grid with density 0.35. The iteration times is in the range from 1000 to 10000 with interval 1000.

Obviously , the iteration time is linearly increasing with increasing in iteration

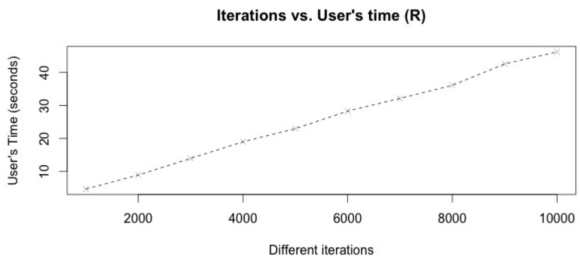

Figure 11: R's performance via different iteration times

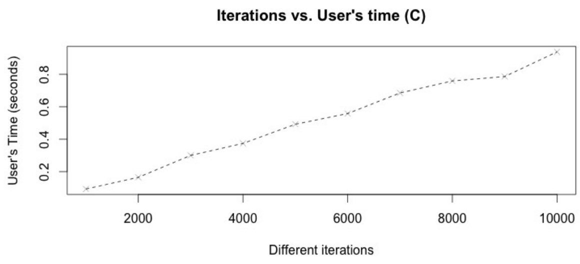

Figure 12: C's performance via different iteration times

times for both code. But C code's performance is far more efficient that R code's performance.

PART VII: Conclusion

This report explores the different performance of C code and R code implementing same algorithm. The first part verifies that the two sets of codes can generate same result.(Same grid). The next several parts discuss the performance between two methods. The possible reasons, about why R implementation is slowerr relatively to C version, are also discussed. In this case, writing C code is worth.

Reference

1. classnotes
2. Piazza: 268, 277,281.

Appendix

C code

```c

/* Appendix: C source code.
 *file: CrunBML.c
 * this file is the c version of moving cars simultaneously.
 */

#include <stdio.h>
#include "runBML.h"
#include <string.h>


/* private function PROTOTYPE*/
void move_cars(double *grid, int *dimension, int direction, int *location, int *number);


/* main routine*/
void crunBMLGrid (double *grid, int *dimension , int *red_loc, int *blue_loc,int *nred , int *nblue, int *numsteps)

{

    int i;
    /* store color */
    int dir;

    for (i = 0; i< *numsteps ; i++){
      /* move blue cars*/
        dir = 2;
        move_cars(grid, dimension,dir,blue_loc,nblue);
        /* move red cars*/
        dir = 1;
        move_cars(grid, dimension,dir,red_loc,nred);
    }
    return;
}


void move_cars(double *grid,int *dimension, int direction, int *location, int *number)
{
    int color = direction;
    int i,t,k;

    /* generate a array to store index of array represent of the original grid */
    int new_loc[2*number[0]];

    for (i=0; i < number[0]; i++){
        if (color == 2) { // move blue cars
        /* for blue cars, move upwards */
            new_loc[i]  = location[i]-1 ;
        /* column won't change */
            new_loc[i + number[0]] = location[i + number[0]];
          /* border: move from top to bottom */
            if (new_loc[i]<1) new_loc[i] = dimension[0];
        }

        else {  /* move red cars*/
            new_loc[i]  = location[i] ;
            /* move to the right */
            new_loc[i + number[0]] = location[i + number[0]]+1;
            /* border : move from left to right */
```

```
            if ( new_loc[i + number[0]] > dimension[1]) new_loc[i + number[0]] = 1;
        }

    }

    /* Check if a car can be moved, if it can be moved, change it's value to -1 indicates it can be moved. */

    for(k=0; k < number[0];k++) {

        /* Create an index to change location x row into grid index*/


        int indice = location[k]-1+dimension[0]*(location[k+number[0]]-1);
        int indice_new =new_loc[k]-1+dimension[0]*(new_loc[k+number[0]]-1) ;

        /* Change the value of cars that can be moved into -1*/

        if (grid[indice_new] == 0) grid[indice] = -1;
    }

    /* based on the index, move the cars */

    for (t=0; t < number[0];t++) {
        if ( grid[location[t]-1+dimension[0]*(location[t+number[0]]-1)] == -1)
        {
          /* change the corresponding cars */
            grid[new_loc[t]-1+dimension[0]*(new_loc[t+number[0]]-1)] = color;
            grid[location[t]-1+dimension[0]*(location[t+number[0]]-1)] = 0;
            location[t] = new_loc[t];
            location[t+number[0]] = new_loc[t+number[0]] ;
        }
    }
}
```

R Soruce Code of Package

```

################ STA 242 HW2 final version #################
### Appendix C: R source code

#####Note: In the source file , all the 0 mean white color on the grid. All the 1
#### represent red cars. All the 2 represent blue cars. These three constant are using
### to represent color.

##########  function to create initial grid  ############
createBMLGrid =function(r,c, ncars){
  ## check if the cars' numbers are reasonable
  if (sum(ncars)/(r*c) > 1) return ("There are too many cars.")
  else{
    ## generate the empty grid
    grid = matrix(0, r, c)
    pos = sample(1:length(grid), sum(ncars))
    ## -1:blue;  1: red
    grid[pos] = sample(rep(c(2, 1), ceiling(ncars)))[seq(along = pos)]
  }
  ## s3 method to define grid's class
  class(grid) = c("BMLGrid", "matrix")
  grid
```

```
}

########## function to store the cars' coordinates ##########
## In this algorithms, all the cars will move simultaneouly
car_coordinate = function(matrix){
  ## check cars' location
  index = which(matrix!=0)
  ## row number with cars
  row_indice = row(matrix)[index]
  ## column number with cars
  col_indice = col(matrix)[index]
  ## each car's position
  position = t(rbind(row_indice, col_indice))
  ## cars' coordinates on the initial grid
  color = matrix[position]
  data.frame(row_indice=row_indice,col_indice=col_indice,colors = color)
}

########## function to move cars ####################
### the arguments color specifies what color cars we want to move
.move_cars = function (matrix,color){
  ## call car_coodinate function to extract cars coordinates
  car_info = car_coordinate(matrix)
  ## those color's index we want
  indice = which(car_info$colors==color)
  ## create a matrix to store specific color's cars' location
  current_location = as.matrix(cbind(car_info$row_indice[indice],car_info$col_indice[indice]))
  colnames(current_location) = c("row_indice","col_indice")
  ## duplicate the current location matrix and use it to compute the next location
  possible_next_location = current_location
  if(color==1){
    ## add one to all red cars' column indices-- move to the right
    possible_next_location[,"col_indice"] = possible_next_location[,"col_indice"] + 1
    ## check if the red cars move to the right border of the grid, if yes, change row indices to one
    possible_next_location[,"col_indice"][possible_next_location[,"col_indice"] > ncol(matrix)] = 1
  }
  ## blue cars:move upwards
  else{
    ## minus one to all blue cars' row indices-- move upwards
    possible_next_location[,"row_indice"] = possible_next_location[,"row_indice"] - 1
    ## check if the blue cars move to the upper border of the grid, if yes, change row indices to the
    ## number of columns of the initial grid.
    possible_next_location[,"row_indice"][possible_next_location[,"row_indice"] < 1] = nrow(matrix)
  }
  ## call change_color function to move the cars
  .change_color(current_location,possible_next_location,color,matrix)
}

.change_color = function (current_location,possible_next_location,color,matrix){
  ## use all the new indices for the cars to subset the original grid
  new_indice = matrix[possible_next_location]
  ## check if the new positions are blank. If yes, change those blanks to the corresponding colors
  matrix[possible_next_location[which(new_indice == 0),,drop=FALSE ]] = color
  ## change those cars' current locations' colors to blank
  matrix[current_location[which(new_indice == 0),,drop=FALSE ]] = 0
  matrix
```

```
}

########## function to get the final grid
## need to export all global functions and global variables
rumBMLGrid= function(numSteps,grid){
  # Register cluster
  for(i in 1:numSteps){
    ##move blue car
    grid = .move_cars(grid,2)
    ## move red car
    grid = .move_cars(grid,1)
  }
  grid
}

######### C vrsion to move cars and get the final grid ##############

CrunBMLGrid = function(grid,numsteps){
  location = car_coordinate(grid)
  ### red cars indices
  red = as.matrix(location[which(location$colors==1),1:2])
  ### blue cars indices
  blue = as.matrix(location[which(location$colors==2),1:2])
  result = .C("crunBMLGrid",grid,dim(grid),red,blue,nrow(red),
              nrow(blue),as.integer(numsteps))
  ### return the final grid
  result[[1]]

}
######## plot function: S3 method ###########
plot.BMLGrid = function(x,...){
  image_grid = matrix(match(x, c(0, 1, 2)),
                      nrow(x), ncol(x))
  ## the row number should be opposite
  image_grid = image_grid[c(nrow(image_grid):1),]
  ## get the correct order
  image(t(image_grid), col = c("white", "red", "blue"),axes=FALSE,...)
  box()

}

###### summary function : S3 method ###########
#### this function exports the given color's grid's some summary
    statistics.
#### 1. how many cars for this color
#### 1. how many cars can move
#### 2. how many cars are blocked
#### 3. average velocity

summary.BMLGrid = function(object,...){
  car_info_blue = car_coordinate(object)
  output_blue = .blue_summary(object,car_info_blue)
  ### red cars
  grid_red = .move_cars(object,2)
  car_info_red = car_coordinate(grid_red)
  output_red = .red_summary(grid_red,car_info_red)
  output = c(output_blue,output_red)
  names(output) = c("total-blue","move-blue","block-blue","velocity-
    blue","total-red","move-red","block-red","velocity-red")
  options(digits=4)
  output
}
######### function to summary blue car
.blue_summary = function(grid,car_info){
  indice = which(car_info$colors == 2)
  num_car = length(indice)
  current_row = car_info$row_indice[indice]
  current_col = car_info$col_indice[indice]
```

```
  next_col = current_col
  next_row = current_row - 1
  next_row[ next_row < 1 ]  = nrow(grid)
  new_possible_position = cbind(next_row,next_col)
  move = sum(grid[new_possible_position] == 0)
  block = num_car - move
  velocity = move/num_car
  output = c(num_car,move,block,velocity)
  output
}
######## function to summary red car
.red_summary = function(grid,car_info){
  indice = which(car_info$colors == 1)
  num_car = length(indice)
  current_row = car_info$row_indice[indice]
  current_col = car_info$col_indice[indice]
  next_row = current_row
  next_col = current_col + 1
  next_col[ next_col > ncol(grid) ]  = 1
  new_possible_position = cbind(next_row,next_col)
  move = sum(grid[new_possible_position] == 0)
  block = num_car - move
  velocity = move/num_car
  output = c(num_car,move,block,velocity)
  output
}
########## function to plot a given series of densities' grids plot

.reproduce_plot = function(r,c, density, numSteps){
  car_num = sapply(density, function(x) round(r*c*x/2))
  initial = lapply(array(car_num),function(x)createBMLGrid(r, c, c(red = car_num[[x]], blue = car_num[[x]])))
  final = lapply(1:length(density), function(x) rumBMLGrid(numSteps, initial[[x]]))
  par(mfrow = c(2, 3), mar = c(1,1,1,1), pty = "s")
  sapply(1:length(density), function(x) plot(final[[x]], main = paste("density = ", density[x])))
}


######## this function is to get the average velocity over different density ##########
velocity_density = function(r,c,density,numSteps){
  car_num = sapply(density, function(x) round(r*c*x/2))
  initial = lapply(1:length(density),function(x)createBMLGrid(r, c,
                                                                c(red = car_num[[x]], blue = car_num[[x]])))
  final = lapply(1:length(density),function(x){
    rumBMLGrid(numSteps,initial[[x]])
  })
  veo_summary = lapply(1:length(density),function(x){
    summary(final[[x]])
  })
  ## list to store red cars and blue cars' average velocity across different density
  veo_summary = do.call(rbind.data.frame,veo_summary)
  velocity = cbind((veo_summary[,2]+veo_summary[,6])/(veo_summary[,1]+veo_summary[,5]),density)
  colnames(velocity) = c("velocity","density")
  as.data.frame(velocity)
}


######### SLOWER VERSION OF move_cars & rumBMLGrid ################################

.move_cars1 = function(matrix){
  car_info = car_coordinate(matrix)
```

```
  red_indice = which(car_info$colors==1)
  blue_indice = which(car_info$colors==2)
  ## red cars: move to the right
  ## check if we can move the red cars
  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
    ## change the position
    if(matrix[next_row,next_col] ==0){
      matrix[next_row,next_col]=1
      matrix[current_row,current_col]=0
    }
  }
  ## blue cars:move upwards
  for (j in blue_indice){
    current_row = car_info$row_indice[j]
    current_col = car_info$col_indice[j]
    next_col = current_col
    if(current_row==1) next_row = nrow(matrix)
    else next_row = current_row - 1
    ## change the position
    if(matrix[next_row,next_col]==0){
      matrix[next_row,next_col]=2
      matrix[current_row,current_col]=0
    }
  }
  matrix
}

.rumBMLGrid1= function(initial_grid,numSteps){
  for (i in 1:numSteps){
    update_grid = .move_cars1(initial_grid)
    update_grid = .move_cars1(update_grid)
    initial_grid = update_grid
  }
  initial_grid
}
```

R Soruce Code to Generate Figures and Tables in the Report

```




### Appendix B
########## source code to reproduce the plot in the report.


#### figure1: 4*4 grid
dev.off()
par(mfrow=c(1,1))
g = createBMLGrid(3,3,c(2,2))
plot(g,main="inital 4*4 grid")
par(mfrow=c(2,4),mar=c(1,1,1,1))

#### figure 2: moving 4 steps
dev.off()
par(mfrow=c(2,4),mar=c(1,1,1,1))
plot(rumBMLGrid(1,g),main="step 1:runBML")
plot(rumBMLGrid(2,g),main="step 2:runBML")
plot(rumBMLGrid(3,g),main="step 3:runBML")
plot(rumBMLGrid(4,g),main="step 4:runBML")


```

```
plot(CrunBMLGrid(g,1),main="step 1:CrunBML")
plot(CrunBMLGrid(g,2),main="step 2:CrunBML")
plot(CrunBMLGrid(g,3),main="step 3:CrunBML")
plot(CrunBMLGrid(g,4),main="step 4:CrunBML")

####figure 3: 128*128 density=0.5 timestep=100
dev.off()
par(mfrow=c(1,2),mar=c(1,1,1,1))
g = createBMLGrid(128,128,c(128*128*0.5/2,128*128*0.5/2))
plot(rumBMLGrid(100,g),main="runBML:128*128")
plot(CrunBMLGrid(g,100),main="CrunBML:128*128")

##### table r profiling

Rprof("/tmp/r_BML.prof")
g = createBMLGrid(150,150,c(3000,3000))
ss = rumBMLGrid(100,g)
Rprof(NULL)
head(summaryRprof("/tmp/r_BML.prof")$by.self, 5)

Rprof("/tmp/C_BML.prof")
g = createBMLGrid(150,150,c(3000,3000))
ss = CrunBMLGrid(g,100)
Rprof(NULL)
head(summaryRprof("/tmp/C_BML.prof")$by.self, 10)
system.time(rumBMLGrid(100,g))
system.time(CrunBMLGrid(g,100))


###### figure 4:



######## performance with different grid size


### grid size with user's time
#### suppose the density is 0.2 ,0.35 , 0.7
#### run 100 hundered times

N = 2^(2:10)
timings =sapply(N,
          function(n) {
            print(n)
            g = createBMLGrid(n,n,c(n^2*0.2/2,n^2*0.2/2))
            system.time(rumBMLGrid(100,g))
          })

timings_C =
  sapply(N,
          function(n) {
            print(n)
            g = createBMLGrid(n,n,c(n^2*0.2/2,n^2*0.2/2))
            system.time(CrunBMLGrid(g,100))
          })

dev.off()
plot(N, timings[1,], type = "p",
      xlab = "Grid size" , ylim = c(0,20) ,
      ylab = "User's Time (seconds)",pch=4,col=24,main="Grid Size vs.
     User's time")
lines(N,timings[1,],lty=2)
points(N,timings_C[1,],pch=18,col=58)
lines(N,timings_C[1,],lty=20)

legend("topleft",title="Different Code",c("R code"," C code"),
        pch=c(4,18),col=c(24,58),cex=0.75)

```

```r


timings_0.35 = sapply(N,
                      function(n) {
                        print(n)
                        g = createBMLGrid(n,n,c(n^2*0.35/2,n^2*0.35/2))
                        system.time(rumBMLGrid(100,g))
                      })

timings_C_0.35 =
  sapply(N,
         function(n) {
           print(n)
           g = createBMLGrid(n,n,c(n^2*0.35/2,n^2*0.35/2))
           system.time(CrunBMLGrid(g,100))
         })


plot(N, timings_0.35[1,], type = "p",
     xlab = "Grid size",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Grid Size vs. User's time")
lines(N,timings_0.35[1,],lty=2)
points(N,timings_C_0.35[1,],pch=18,col=58)
lines(N,timings_C_0.35[1,],lty=20)
legend("topleft",title="Different Code",c("R code","C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)



timings_0.7 = sapply(N,
                     function(n) {
                       print(n)
                       g = createBMLGrid(n,n,c(n^2*0.7/2,n^2*0.7/2))
                       system.time(rumBMLGrid(100,g))
                     })

timings_C_0.7 =
  sapply(N,
         function(n) {
           print(n)
           g = createBMLGrid(n,n,c(n^2*0.7/2,n^2*0.7/2))
           system.time(CrunBMLGrid(g,100))
         })

plot(N, timings_0.7[1,], type = "p",
     xlab = "Grid size",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Grid Size vs. User's time")
lines(N,timings_0.7[1,],lty=2)
points(N,timings_C_0.7[1,],pch=18,col=58)
lines(N,timings_C_0.7[1,],lty=20)
legend("topleft",title="Different Code",c("R code","C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)

ratio = timings_0.7[1,]/timings_C_0.7[1,]


plot(N,ratio,xlab = "Grid Density",ylab = "ratio of user time(C/R)",, type="p",pch=15,
     main = "Ratio of user's time")
lines(N,ratio,lty=2)

######## different density

density = seq(from=0.1,to=0.7,length.out = 13)

```

```r
timing_density_128 =
  sapply(density,
         function(n) {
           print(n)
           g = createBMLGrid(128,128,c(red = ceiling(128^2*n/2),blue = ceiling(128^2*n/2)))
           system.time(rumBMLGrid(100,g))
         })


timings_density_C_128 =
  sapply(density,
         function(n) {
           print(n)
           g = createBMLGrid(128,128,c(red = ceiling(128^2*n/2),blue = ceiling(128^2*n/2)))
           system.time(CrunBMLGrid(g,100))
         })


plot(density, timing_density_128[1,], type = "p",
     xlab = "Different Densities",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Density vs. User's time",ylim = c(0,0.5))
lines(density,timing_density_128[1,],lty=2)
points(density,timings_density_C_128[1,],pch=18,col=58)
lines(density,timings_density_C_128[1,],lty=20)

legend("topleft",title="Different Code",c("R code"," C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)

##### 64*64 grid size

timing_density_64 =
  sapply(density,
         function(n) {
           print(n)
           g = createBMLGrid(64,64,c(red = ceiling(64^2*n/2),blue = ceiling(64^2*n/2)))
           system.time(rumBMLGrid(100,g))
         })

timings_density_C_64 =
  sapply(density,
         function(n) {
           print(n)
           g = createBMLGrid(64,64,c(red = ceiling(64^2*n/2),blue = ceiling(64^2*n/2)))
           system.time(CrunBMLGrid(g,100))
         })


plot(density, timing_density_64[1,], type = "p",
     xlab = "Different Densities",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Density vs. User's time",ylim = c(0,0.3))
lines(density,timing_density_64[1,],lty=2)
points(density,timings_density_C_64[1,],pch=18,col=58)
lines(density,timings_density_C_64[1,],lty=20)
legend("topleft",title="Different Code",c("R code"," C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)

##### 256*256 grid size

timing_density_256 =
  sapply(density,
         function(n) {
           print(n)
```

```
            g = createBMLGrid(256,256,c(red = ceiling(256^2*n/2),blue =
      ceiling(256^2*n/2)))
            system.time(rumBMLGrid(100,g))
          })

timings_density_C_256 =
  sapply(density,
          function(n) {
            print(n)
            g = createBMLGrid(256,256,c(red = ceiling(256^2*n/2),blue =
      ceiling(256^2*n/2)))
            system.time(CrunBMLGrid(g,100))
          })

plot(density, timing_density_256[1,], type = "p",
     xlab = "Different Densities",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Density vs.
    User's time",ylim = c(0,2.5))
lines(density,timing_density_256[1,],lty=2)
points(density,timings_density_C_256[1,],pch=18,col=58)
lines(density,timings_density_C_256[1,],lty=20)
legend("topleft",title="Different Code",c("R code"," C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)


############## different iteration times

iteration = seq(1000,10000,1000)

timings_iteration =
  sapply(iteration,
          function(n) {
            print(n)
            g = createBMLGrid(128,128,c(red = ceiling(128^2*0.35/2),
    blue = ceiling(128^2*0.35/2)))
            system.time(rumBMLGrid(n,g))
          })


timings_iteration_C =
  sapply(iteration,
          function(n) {
            print(n)
            g = createBMLGrid(128,128,c(red = ceiling(128^2*0.35/2),
    blue = ceiling(128^2*0.35/2)))
            system.time(CrunBMLGrid(g,n))
          })

plot(iteration, timings_iteration[1,], type = "p",
     xlab = "Different iterations",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Iterations vs.
    User's time (R)")
lines(iteration,timings_iteration[1,],lty=2)

plot(iteration, timings_iteration_C[1,], type = "p",
     xlab = "Different iterations",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Iterations vs.
    User's time (C)")
lines(iteration,timings_iteration_C[1,],lty=2)
```