URL: https://yinzhang@bitbucket.org/yinzhang/stat242.git

Assignment 4

Yin Zhang

BML model with C code

Abstract:

First, I evaluate the functions in R and find which one should have the C version. Then, I build C code which have similar algorithms with R and check whether they have the same answer. Based on the C code and R "for loop" version and vectorize version, I evaluate the running time in three aspects: different number of time steps, different dimension of grid and different proportion of cars, and also evaluate the stability of the three methods. The results indicate that C is the fast and also stable for the running time, while "for loop" in R is slow and not stable.

1 function chosen and organization

1.1 Brief introduction for C code

According to the assignment 2, the parts of the code that take the most time is: In the "runBMLGrid" function, the command "ifelse" , "==" and "move " function that take 81.95% out of all time. It is because the "for loop" which calls the move function several times. Furthermore, in the "move" function, it is command "ifelse" and "which" that cost 50% out of all time totally. According to it, if I run it in C, it will spend little time, especially for the "for" loop and command "which", [1]. So I change those corresponding parts to C code which is similar to the "runBMLGrid" and "move" function in R.

For C code, there are two functions, one is "move", the other "crunBMLGrid" function. The "move" function's algorithms are similar to "moveslow"(for loop in R, because in C, it can not be vectorized) and "crunBMLGrid" to "runBMLGrid" function in R respectively. It makes the C code comparable to R code. Then in the Rcode, "crunBMLGrid" function is used to call the C code.

1.2 Evaluate of C code

Table1: running time of C routine

	self.time	self.pct	total.time	total.pct
".C"	7.96	99.75	7.96	99.75
"+"	0.02	0.25	0.02	0.25

Table1 shows the result of the running time of the "crunBMLGrid" function, with the gird(500*500), with the proportion of cars 0.4 after 10000 times. The results indicate the .C cost most of time. So it is reasonable to use the system time of "cunBMLGrid" to stands for the running time in C.

1.3 Not worth for the grid creation function in C

I run a big grid for the "createBMLGrid" function in R . then I will evaluate how much the time it will cost in different parts. I create a 1024*1024 grid with 0.7 proportion of cars, and the result time of "Rprof" shows below:

Table2: running time of "createBMLgrid"

	self.time	self.pct	total.time	total.pct
"sample"	0.06	75	0.06	75
"class"	0.02	25	0.02	25

(1)In "creatBMLGrid" function in R, the "sample" function costs lots of time (75% out of all time), which will also costs time in C programming[2];

(2) There is no for loop in "creatBMLGrid" and the elapse time is small enough when create a large grid (1024*1024 with 0.7 proportion of the cars costs 0.23 time and in out research, it is the largest grid and proportion that we need).

Based (1) and (2), it is not worth to use C to create BMLGrid.

2 Verify the C code and R have the same result

In the test folder, first, I check whether the R gives the right result, then I verify the C code and R code after one step and two steps to see that whether they have the same results. (Only need to check those two steps because blue cars run in the odd steps and the red car run in even steps)

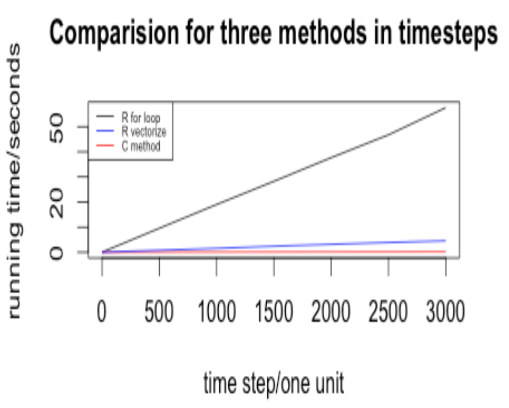
3 Comparison of C code and R code

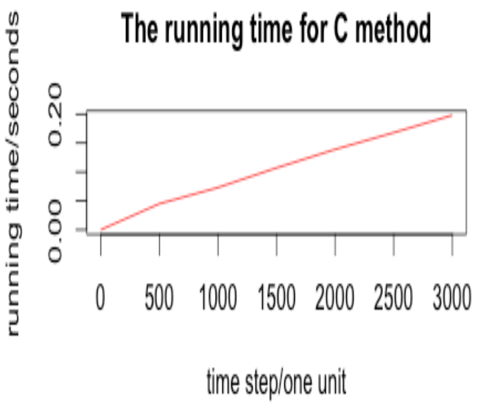
3.1 Running time comparison

There are three main factors that influence on the behavior of the Biham-Midleton-Levine (BML car) model simulation[2], which is the number of steps, dimension of the grid, the proportion of cars in the grid.

3.1.1 Cost time in each methods

Because the "for loop" in C which has similar algorithms with the first R version. So I compared C, the first version of R("for loop") and the vectorlize version of R. According to the assignment 2 of the running time in part 2. I choose the grid 100*100 and the proportion of cars is 0.4, and the time step is from 0 to 3000 by 100 increasing every step, which the running time is not too much or not too small. It makes easy and intuitive to compare the three methods:

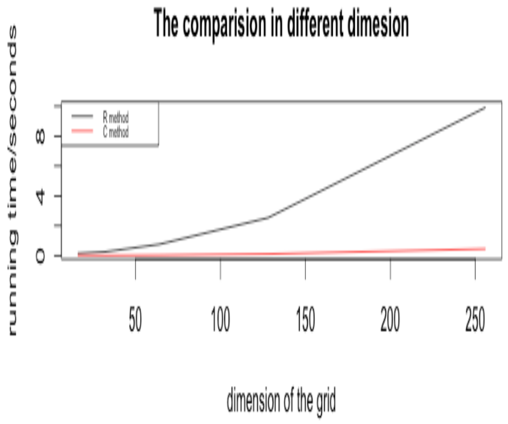


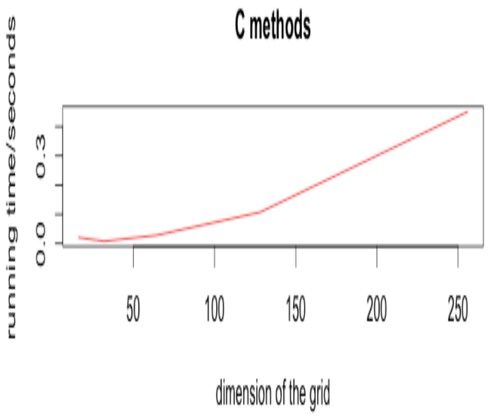


The plots show that the running time is increasing with time step increasing and the increases are linear for all the three methods. So with the time step increasing, the differences among the three methods are more different with each other. The plot in the right side shows that the C method is actually not a horizontal line, but with a little slop. By using linear regression, the slop for R for loop version, for R vectorize version and the C methods are 0.01895, 0.001555 and 6.804e-05. They are 279, 23 times slower than the C methods, respectively, i.e., "for loop" and similar algorithm in R is about 279 times slower than the similar methods in C. When use vectorize in R, it will develop the speed about 12 times then the "for loop" version. So C is more faster. The plot also shows that when the time steps are small, using R or C methods are both good because the running times are not much different in this case, but when the time steps are big, it is better to use C.

3.1.2 Dimension vs running time

According to the Assignment 2 in part 2 for the comparison of different dimension. I use the same dimension here, which the grid is: 16*16, 32*32, 64*64, 128*128, 256*256 to compare the running time. Because in 3.2.1, it shows that R version for for loop is not good, I only compare the R vectorize version and C version.

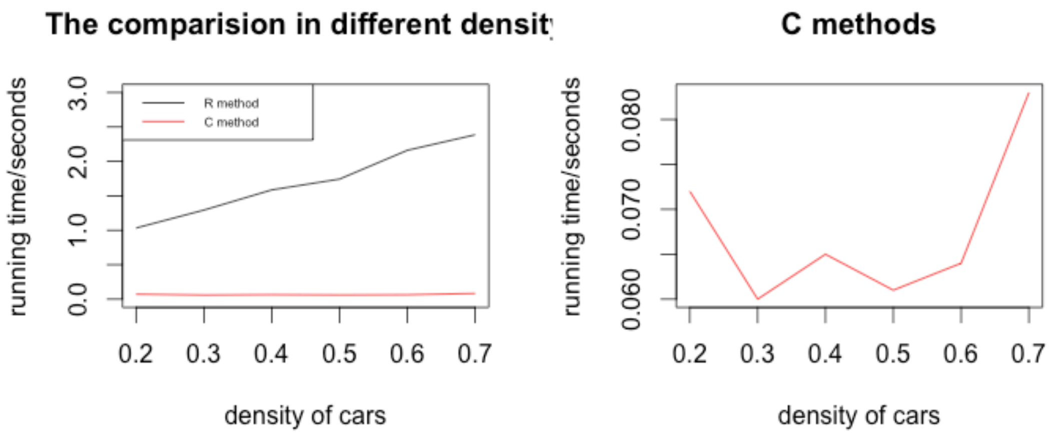




The plot shows that the running time is similar quadratic increasing with the dimension increasing. Similarly, the running time cost in C is much faster than running in R vectorize, and the different are bigger and bigger with the grid increasing. So if the gird is big, it is much better to use the C method than the R verctorize method for the BMLGrid system. Especially when the dimension of the grid is big, the differences between the two methods are big.

3.2.3 Density vs running time

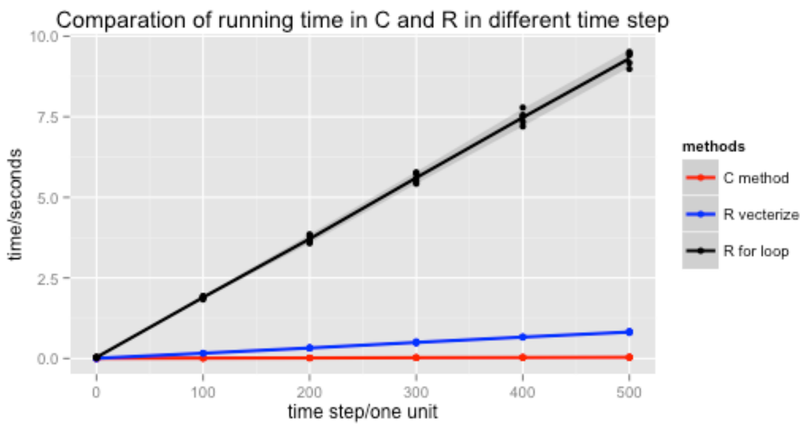
According to the assignment 2, I use the density from 0.2 to 0.7 by 0.1 increasing every step.



The plots above show that the system running time change with the proportion of cars in the grid from 0.2-0.7. The plot on the left hand indicates that C is faster than R vectorized version, but it is not much different with the proportion increasing like the first (time step) part and second (dimension) part. However, we can see that system running time in the C code does not increase with the proportion of cars increase. So it indicates that the proportion is not a key factor that influence on the system running time for C, since the system varies a lot for the same density and grid of cars. But in R, there is still a trend of increse with the proportion of cars in the grid in in crease.

3.2 The stability of the two methods:

I use the same grid 100*100 and with 0.4 proportion of cars in the grid to illustrate the stability of the three methods. Because every time steps I will repeat 5 times, so I only use the time step from 0 to 500 by 100 steps increasing every step to illustrate.



The grey shades are the standards variance for each step. From the plot above, it shows that the grey shades of “for loop” are wider than the other two shades which are not obvious. It indicate that when use “for loop” in R, the system running is not stationary for each step to run a “runBMLGrid” function, i.e. there are much different cost running time in same grid and same time step. On the other hand, the vectorize in R and “for loop” in C are more stable which means the running time are almost similar with each other when running the same grid and same time steps.

4 Conclusion:

Based on the C code and R “for loop” version and vectorize version, I evaluate the running time in different number of time steps, different dimension of grid and different proportion of cars respectively. The running time are linear increasing with number of time steps for all the three version, and with different dimension of grid is quadratic increasing. Though system running time in the C code does not increase with the proportion of cars increase, C code are also faster then the vectorize version of R. I also evaluate the stability of the three methods. The results indicate that C and vectorize version of R are stable for the running time, while “for loop” in R is not. At the same time, how to evaluate the system running time for command in C and the cost time of the result in R to pass to a C routine in .C()[4,5].

Reference

*[1] Learn C Programming, A short C Tutorial, http://www.loirak.com/prog/*ctutor*.php*

[2] Deborah Nolan, Duncan Temple Lang. Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving, 346-351

[3] Why Call C or Fortran from R, http://users.stat.umn.edu/~geyer/rc/#why

[4] R’s C interface http://adv-r.had.co.nz/C-interface.html

[5] Roger D. Peng, Jan de Leeuw, An Introduction to the .C Interface to R, 2002

R code:

#Some clarification of the first package and the second package

#In order to compare three versions of the BML grid, I add to run the slowest function in the code.

#So I change the runBMLGrid which can not only run the slowest version but also the quick

#version in R (add a parameter “slow” in the runBMLGrid function, the default value is FALSE,

#which means that it will run the quick version; if its value is TRUE, which means that it will run

#the lowest version).

```
#C code:
#include <stdio.h>
/*
There are two functions in this code, the first one is "move", which returns three results: the grid
that move in one step, and the corresponding velocity (parameter "v"), the number of moved cars
(parameter "carmove"); the second function is "crunBMLGrid", which returns the grid that move
several steps which can be decided by the users, and the velocity in every step, and
corresponding number of cars moved in every step.
*/

/*
move function: grid move in one step, store the the grid which move one step, and the
corresponding velocity, the number of cars
that move

parameter:
grid - the initial grid before the step
nrow - the number of rows of the grid; ncol - the number of column in the grid
nblue - the number of blue cars in the grid; nred - the number of red cars in the grid
step - the time step, i.e which time step to run. For example, the seond step, means when the time
=2
v – velocity of the corresponding step
carmove - the number of car moves in this step
*/
```

```
void move(int *grid, int *nrow, int *ncol, int *nblue, int *nred, int step, double *v, int *carmove)
{
/*
totcar - the number of cars in the grid
length - the length of the grid. Because the grid is a array, I should initalize the length of it.
j - the red cars index; k - the blue cars index; t - for the cars index;
movecar - store the number of cars that moves, the initial value is 0. I do not use the carmove to
directly calculate the number of moved cars,
            because it easily to be wrong

Arrays:
cur_long - the length of it is the same as the grid, but when there is a car, it turns to be 1; if not,
turns to be 0
cur_have - instore the position which have cars
next_blue - the next positions for the blue cars after the "step"th time;
next_red - the next positions for the red cars after the "step"th time;
*/

  int numrow = *nrow, numcol = *ncol, totcar = (*nblue) + (*nred), length = (*nrow) * (*ncol), i, j
= 0, k = 0, t = 0, movecar = 0;
  int cur_long[length], cur_have[totcar], next_blue[*nblue], next_red[*nred];

/* get the position of the none car and the position which have a car*/
 for ( i = 0; i < length; i++)
 {
  if (grid[i] != 0)   {
      cur_long[i] = 1; /*if it is not "", the coresponding position means there is a car, give the
value 1; otherwise, give the value 0*/
      cur_have[t] = i; /*store the position in the grid*/
      t++;
    }else cur_long[i] = 0;
 }
```

```
/* first, consider for the blue car */
        if ( (step%2) == 1) /* in odd step*/
        { for ( i = 0; i < totcar; i++) /* every cars' position is in the totcar, so make a for loop in it can
decrease the times of loop */
            {
            if ( grid[cur_have[i]] == 1) /* if the position have blue cars */
                  {

                  /* if the blue car in the fist row, it will wraps, if not, the position will -1 */
                  if( (cur_have[i] % numrow) == 0) next_blue[k] = cur_have[i] + numrow - 1;
                  else next_blue[k] = cur_have[i] - 1;

                /* means there is a car in the current position, so the car can not move */
                   if ( cur_long[next_blue[k]] == 1 ) next_blue[k] = cur_have[i];
                   else movecar++;

              /* change the original position to be no car, and the next position to be blue can*/
              grid[cur_have[i]] = 0;
              grid[next_blue[k]] = 1;

            /* calculate the velocity of the car, "*0.1" means it is double */
              v[step-1] = movecar*1.0/(*nblue);

                k++;
            }
          }
      }

    /* consider for the red car */
    if ( (step%2) == 0 ) /* for the even step */
        { for ( i = 0; i < totcar; i++)
            {
            if ( grid[cur_have[i]] == 2) /*find the current position which is red car*/
            {
                  if ( (cur_have[i] / numrow) == numcol - 1 ) next_red[j] = (cur_have[i]) % numrow;
                  else next_red[j] = cur_have[i] + numrow;

                      /* means there is a car in the current position, so the car can not move */
                  if ( cur_long[next_red[j]] == 1 )    next_red[j] = cur_have[i];
                  else movecar++;

               /* same as the blue car case, change the original position to be no car, and the next
position to be blue can*/
               grid[cur_have[i]] = 0;
               grid[next_red[j]] = 2;

               v[step-1] = movecar*1.0/(*nred);

               j++;
          }
       }
   }

   carmove[step-1] = movecar; /* give the calculated number of moved cars to the variable
carmove, because carmove is a array, so there should be "-1"*/
}
```

/*

"crunBMLGrid" function used to run the grid for several time steps, and calculate the velocity in every step, the number of moved cars in every step

Parameter:

grid, nrow, ncol, nblue, nred - have the same meanings as the move function

numSteps - how many step should be run

vstep - the initial velocity for every step. The initial value is 0 for the length of step

```
carmovestep - the moved cars for every step. The initial value is 0 for the length of step
*/

void crunBMLGrid(int *grid, int *nrow, int *ncol, int *nblue, int *nred, int *numSteps, double
*vstep, int *carmovestep)
{ int i, step = *numSteps;

  for( i = 1; i <= step; i++ )
  move(grid, nrow, ncol, nblue, nred, i, vstep, carmovestep);
}
```

#R code

```
#"crunBMLGrid" is a function which can call the function in C. The input and
#output is the same as the runBMLGrid except there is no "slow" parameter i
#n the input
  crunBMLGrid = function (grid, numSteps)
  { #get the input parameter in C functions
    nrow=nrow(grid)
    ncol=ncol(grid)
    grid=match(grid, c("", "blue", "red"))-1 #change the charactor to be the numeric, because in
C, the 1 is represent for blue, and 2 is represent for red, 0 is represent for no car, so there is
minus 1
    nblue=sum(grid == 1)
    nred=sum(grid == 2)
    cresult=.C("crunBMLGrid", grid = as.integer(grid), nrow=as.integer(nrow),
ncol=as.integer(ncol),nblue=as.integer(nblue),nred = as.integer(nred),numSteps =
as.integer(numSteps), vstep = as.double(rep(0,numSteps)), carmovestep =
as.integer(rep(0,numSteps))) #give the parameter to C to run
    cgrid = matrix(c('','blue','red')[cresult$grid+1], nrow=nrow)
    # change the numeric to be charactor again, 0,1,2 to be "","blue","red" respectively, "+1"
#means change the 0,1,2 to be 1,2,3 in order to give the charactor
```

```
        class(cgrid)=c("BML",class(cgrid))
        list(grid=cgrid, vstep=cresult$vstep,movestep=cresult$carmovestep)
    }



#check whether they are same
g=createBMLGrid(sample(x = 100,size = 1),sample(x = 100,size = 1),0.4)
c_result = crunBMLGrid(g,100)
r_result = runBMLGrid(g,100)
any(c_result$grid!=r_result$grid)
any(c_result$vstep!=r_result$vbystep)
any(c_result$movestep!=r_result$movestep)

# "runBMLGrid" parameter: "g" stands for the initial grid, and "numSteps" stands
#for number of steps, "slow" means if run the slow version of move function. The
#output is a list, the first one grid is the grid after "numSteps". The second one is
#"vbystep" which is a vector and means the velocity of each steps. The third is
#"movestep" which is a vector and means the number of cars moves in every step
runBMLGrid = function ( g, numSteps, slow = FALSE)
{ vstep=numeric(numSteps) #velocity in every step
    movestep=integer(numSteps) #number of cars move
    for (i in 1:numSteps)
    { if (slow == TRUE) result=moveslow(g,i)
        else result=move(g,i)
        g=result$grid
        vstep[i]=result$v #the blue car velocity
        movestep[i]=result$carmove
    }
    list(grid=g,vbystep=vstep, movestep=movestep)
}

#whether it is worth to build a function in C to createBMLGrid
g = createBMLGrid(1024,1024,0.4)
```

```
Rprof("move.out")
move = move(g, 1)
Rprof(NULL) #turn it off
summaryRprof("move.out")

system.time(createBMLGrid(1024,1024,0.7))
Rprof("createBMLGrid.out")
gird = createBMLGrid(1024,1024,0.7)
Rprof(NULL) #turn it off
summaryRprof("createBMLGrid.out")


#evaluate the crunBMLGrid function, to see if run the crunBMLGrid, the cost time in each part
g_testc = createBMLGrid(500,500,0.4)
Rprof("crunBMLGrid.out")
result = crunBMLGrid(g_testc, 10000)
Rprof(NULL) #turn it off
summaryRprof("crunBMLGrid.out")


#run grid 100*100, the number of red cars and the number of blue cars are 2000 seperatly
timestep=seq(from = 0 ,to = 3000,by = 500 )
g=createBMLGrid(100,100,0.4)
timecostc=sapply(timestep, function(x) system.time(crunBMLGrid(g,x))[[3]])
timecostr=sapply(timestep, function(x) system.time(runBMLGrid(g,x))[[3]])
timecostsl=sapply(timestep, function(x) system.time(runBMLGrid(g,x, slow=TRUE))[[3]])
plot(timecostsl~timestep, type = "l", main="Comparision for three methods in timesteps", xlab =
"time step/one unit", ylab = "running time/seconds")
lines(timecostc~timestep, col = "red")
lines(timecostr~timestep, col = "blue")
legend("topleft", legend=c("R for loop", "R vectorize", "C method"),col=c("black", "blue", "red"),
lty=1, cex=0.5)
plot(timecostc~timestep, col = "red", type = "l", main="The running time for C method", xlab =
"time step/one unit", ylab = "running time/seconds")
lm(timecostsl~timestep-1) #-1 means the intercept is 0 when regression, because running time is 0 when the timestep is 0
lm(timecostc~timestep-1)
lm(timecostr~timestep-1)

#The stationary of the three method
timestep=seq(from = 0 ,to = 500,by = 100 )
g=createBMLGrid(100,100,0.4)
rep_c = replicate(5, sapply(timestep, function(x) system.time(crunBMLGrid(g,x))[[3]])) #3 means I choose elapse time to evaluate the performance
rep_r = replicate(5, sapply(timestep, function(x) system.time(runBMLGrid(g,x))[[3]]))
rep_rf = replicate(5, sapply(timestep, function(x) system.time(runBMLGrid(g,x, slow = TRUE))[[3]]))
timedata = as.data.frame(cbind(timestep = rep(timestep, 5), c = as.vector(rep_c), r = as.vector(rep_r), r_f = as.vector(rep_rf)))#5 is the number of times for replicate

library(reshape)
library(ggplot2)
library(Hmisc)
#get the right form of the data in order to use ggplot
timedata= melt(timedata, id ='timestep', variable_name="methods")
#plot the corresponding mean time in 5 times and the variance
ggplot(timedata, aes(x=timestep, value, colour=factor(methods))) +
  stat_summary(fun.data="mean_cl_normal", geom="smooth", size=1) +
  geom_point()+xlab("time step/one unit")+ylab("time/seconds")+ggtitle("Comparation of running time in C and R in different time step")+scale_colour_manual(values=c("c"="red", "r"="blue", "r_f" = "black"), name="methods",breaks=c("c", "r","r_f"),labels=c("C method", "R vecterize", "R for loop"))

##dimention vs time
dim=c(16, 32, 64, 128, 256) # the dimension of the gird to be use
g=lapply(dim, function(x) createBMLGrid(x,x,0.4))
```

```
dim_timecost_c=sapply(g, function(x) system.time(crunBMLGrid(x,1000))[[3]])
dim_timecost_r=sapply(g, function(x) system.time(runBMLGrid(x,1000))[[3]])
plot(dim_timecost_r~dim, type = "l", main="The comparision in different dimesion", xlab =
"dimension of the grid", ylab = "running time/seconds")
lines(dim_timecost_c~dim, col = "red")
legend("topleft", legend=c("R method", "C method"),col=c('black', 'red'), lty=1, cex=0.5)
plot(dim_timecost_c~dim, col = "red", type = "l", main="C methods", xlab = "dimension of the
grid", ylab = "running time/seconds")

##density vs time
dens = seq(0.2,0.7,0.1) #density from 0.2 to 0.7
#have the corresponding gird with different dencity
grid_dens=lapply(dens, function(x) createBMLGrid(100,100,x))
#calculate the system time for each methods
dens_timecost_c=sapply(grid_dens, function(x) system.time(crunBMLGrid(x,1000))[[3]])
dens_timecost_r=sapply(grid_dens, function(x) system.time(runBMLGrid(x,1000))[[3]])
#give a plot
plot(dens_timecost_r~dens, type = "l", main="The comparision in different density", xlab =
"density of cars", ylab = "running time/seconds",ylim=c(0,3))
lines(dens_timecost_c~dens, col = "red")
legend("topleft", legend=c("R method", "C method"),col=c('black', 'red'), lty=1, cex=0.5)
plot(dens_timecost_c~dens, col = "red", type = "l", main="C methods", xlab = "density of cars",
ylab = "running time/seconds")
```