Vladimir Pchelin
Assignment #4
git@bitbucket.org:ph7vov/sta242.git

Notations and introduction. In this report I compare various functions that move cars in a BML matrix. Here I use the following names for these function:

- Rv1 - is a simple function in R
- Rv2 - is a function in R that uses vectorization extensively
- Cv1 - is a C analogue of Rv1, all important pieces of code are written in C
- Cv2 - is an "improved" Cv2, also almost completely written in C
- Cv3 - is identical to Cv1 apart from the following piece of code:

to wrap matrices in Cv3 I used

```
m1[,1]=m1[,c+1]
m1[,c+2]=m1[,2]
```

while in Cv1 the same operation is written in C (see below).

```
IntegerMatrix wrapRightInC(IntegerMatrix m){
            int r=m.nrow();
            int c=m.ncol();
            for(int i=0; i<r; i++){
            m(i,0)=m(i,c-2);
            m(i,c-1)=m(i,1);
            }
            return m;
            }
```

One would expect that this operation should be quite fast in R. Unexpectedly, the assignment `m1[,1]=m1[,c+1]` is many times slower in R than the above code in C.

In Cv2 I essentially replaced

```
if(m(i,j)==0 & m(i-1,j)==2){
            m(i,j)=2;
            m(i-1,j)=4;
            }else{
            if(m(i,j)==4){m(i,j)=0;}
```

from the subfunction `moveRightInC` in Cv1 by

```
if( notMoved & (m(i,j)==0) & (m(i,j-1)==1) ){
            m(i,j)=1;
            m(i,j-1)=0;
            notMoved=false;
```

```
}else{
notMoved=true;}
```

The new version of `moveRightInC` is called `moveRightInCv2`. The variable `notMoved` tracks whether a car moved on the previous step or not. My belief is that CPU should keep the Boolean value `notMoved` in a register. Thus, all operation with `notMoved` should be significantly faster than with matrix entries `m(i,j)`. Surprisingly, Cv1 is a little faster than Cv2 for some matrices. I can only guess that it's something to do with memory locality, the order of evaluations or the way BML grid stabilizes. On the other hand when the density is large and `notMoved` is `false` with a considerable probability the subfunction `moveRightInCv2` is faster as one would expect.

When cars don't move performance will depend on very specific aspects of our functions. For example, it may depend on the order of logic expressions in the `if else` statements. That's why we will mainly examine the regime when density is small.

Profiling Cv1 with `summaryRprof`. We will mainly test our functions on grids with small densities because when cars don't move performance may depends Profiling with `createBMLGrid(0.2,r=1000,c=1000)` and `crunBMLGrid(g,10000)` gives

```
$by.total
              total.time total.pct self.time self.pct
"crunBMLGrid"     100.84    100.00      0.04     0.04
"<Anonymous>"     100.68     99.84    100.68    99.84
"moveRightInC"     57.28     56.80      0.00     0.00
"moveUpInC"        42.96     42.60      0.00     0.00
"wrapUpInC"         0.44      0.44      0.04     0.04
"addTwo"            0.06      0.06      0.04     0.04
"wrapRightInC"      0.06      0.06      0.02     0.02
"matrix"            0.02      0.02      0.02     0.02
```

Obviously, operations on columns and on rows take different amounts of time. It's because of how matrices are represented in memory. Overall this profiling indicates that our program works fine, most of the time is spent on moving cars not on the other operations.

Compare Cv1 vs Cv2 vs Cv3. I profile Cv1, Cv2 and Cv3 with different densities, linear size is 500, `crunBMLGrid(g,100)`. The graph below shows dependency between `sampling.time` and density.

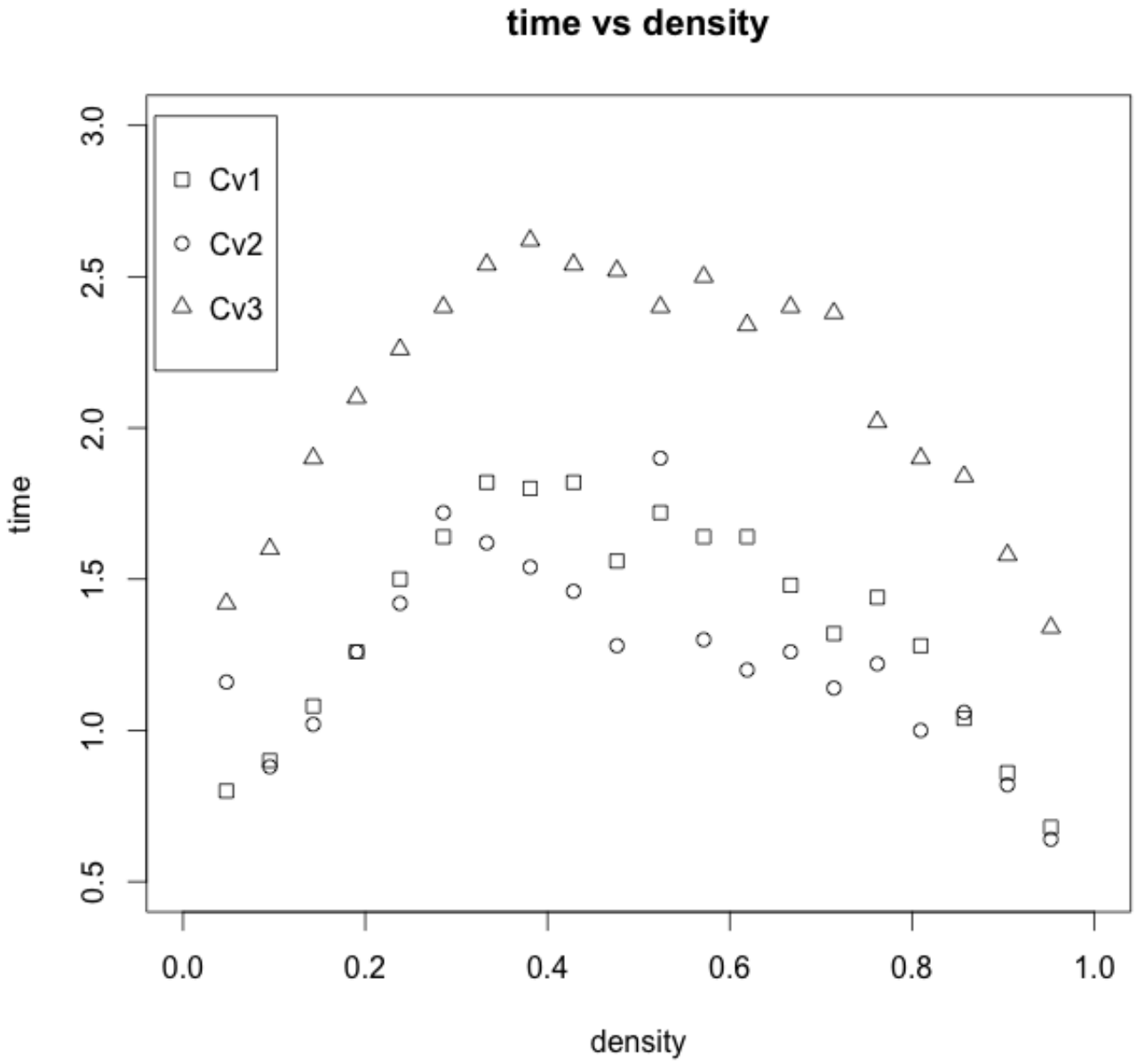

Figure 1

The function Cv3 is ultimately slower than the other two. But relation between Cv1 and Cv2 is quite unpredictable and can't be explained easily.

Compare Rv2 vs Cv1 vs size. The next plot gives times of Rv2 and Cv1*100 for different sizes, density is 0.2. I perform only 30 moves, `runBMLGrid(g,30)`, for each setting. That's why the points corresponding Cv1*100 don't fit a smooth curve.

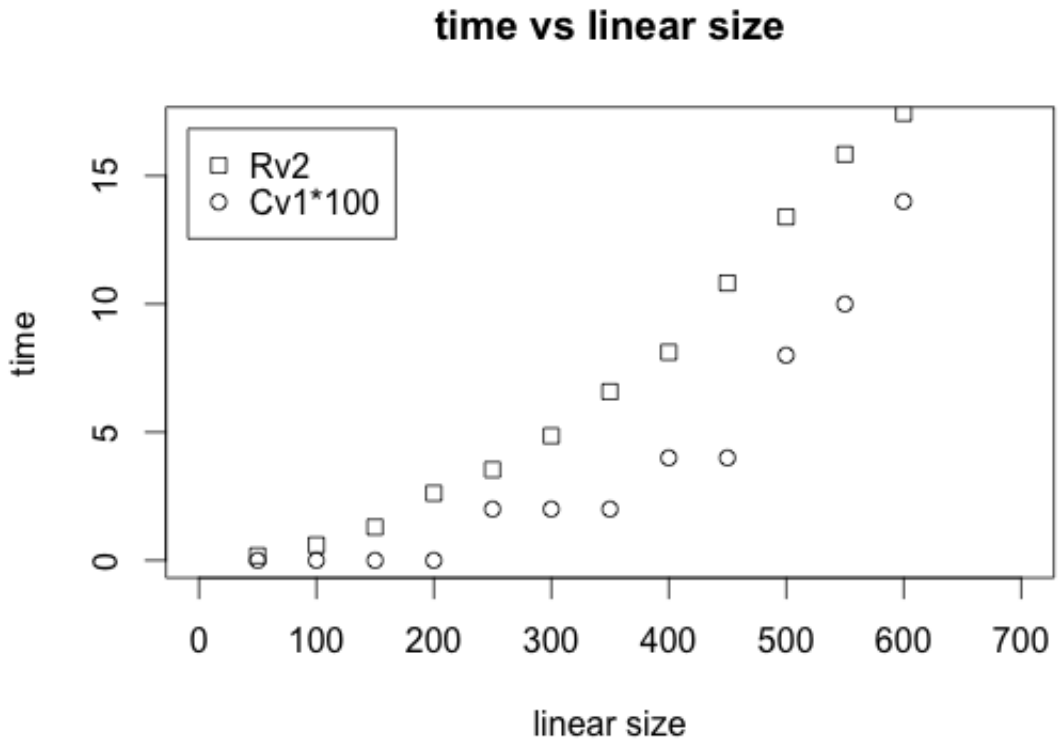

Figure 2

Both functions have similar patterns and Cv1 is always approximately 100 times faster.

Cv1 performance vs size. Now I plot times of Cv1 vs linear size for different densities. I also added fitted quadratic regression lines. As one can see, the points lies almost exactly on parabolas for all four densities.

Conclusion. To surpass quadratic relationship between the execution time and matrix size one should parallelize it's code. Implementation of the BML algorithm in C was indeed beneficial. There are no good reasons to write the grid creation in C since this part is not critical for the performance.

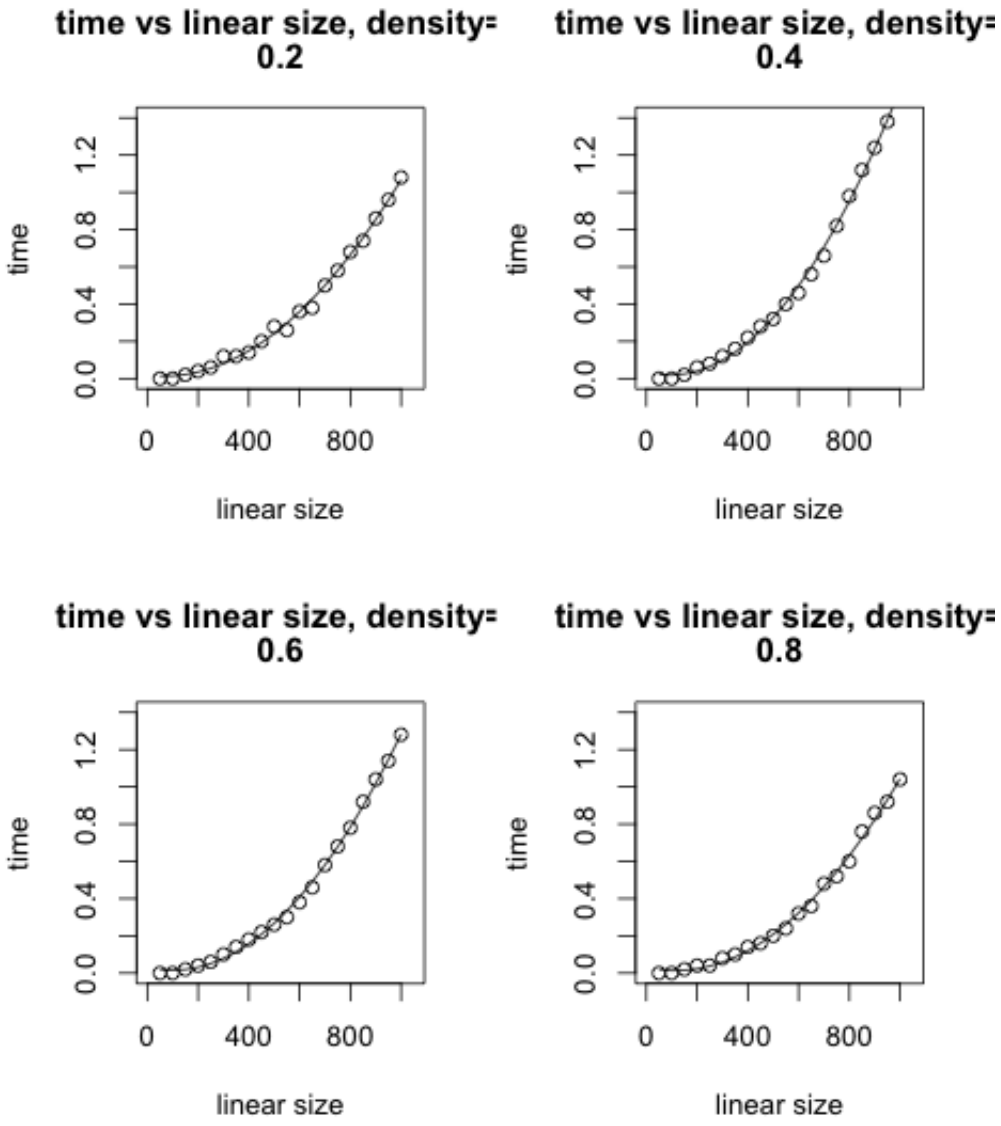
time vs linear size, density= 0.2
time vs linear size, density= 0.4
time vs linear size, density= 0.6
time vs linear size, density= 0.8
time
linear size
0.0
0.4
0.8
1.2
0
400
800

Figure 3