UC Davis STA 242 2015 Spring Assignment 4 [1]

Wenhao Wu, 9987583

May 10, 2015

1 Implementation of `crunBMLGrid()`

In order to highlight the performance difference between R and C++ implementation of BML simulation, we implement 2 versions of the BML simulator routine:

- `crunBMLGrid1()`: rewrite the entire `runBMLGrid()` in C++.
- `crunBMLGrid2()`: rewrite only the 2 key routines to get the next location of cars (`idx_right()` and `idx_up()`) with C++.

The overall algorithm design for both functions are very similar to the original R function `runBMLGrid()`. However, the R's vectorized operations are rewritten with C++ for-loops. As suggested by the example "R vectorisation vs. C++ vectorisation" in [2], one advantage of C++ for-loop implementation over the R vecotrized operation is that it might need to create less intermediate vector variables. Since our original 2 routines, `idx_right()` and `idx_up()`, both contains a few vectorized mathematic operations, we expect to achieve some performance gain with `crunBMLGrid2()`. Another main advantage of C++ over R is that it allows more efficient memory management: in our original `runBMLGrid()` when ever the grid `g` or the cars' locations `red` and `blue` is updated, R creates a new object instead of modifying in-place (as verified with function `address()` from package 'pryr'). This might results in a lot of redundant memory management operations under the hood. In `crunBMLGrid1()`, we use in-place modification whenever possible, which is mainly achieved by using reference inputs for self-defined functions. To interface our C++ functions and R we make use of package 'Rcpp'.

2 Verification

We first verify qualitatively the behavior of BML model computed with our new C++ routines. As in our previous assignment, We pick a $r = 100$, $c = 99$ grid in which the number of blue cars and red cars are the same. After $N = 10000$ steps, the final states of the grid for $\rho = 0.2, 0.33, 0.38, 0.43, 0.5$ returnd by `crunBMLGrid1()` are plotted in Fig. 1, where we observe the same chaotic

phenomenon as the results computed with our original `runBMLGrid()` routine. `crunBMLGrid2()` returns a similar results and is thus omitted.
We further verify that `crunBMLGrid1()`, `crunBMLGrid2()` and `runBMLGrid()` return identical results given the same inputs with package 'testthat'. Besides the degenerated cases, in a $r = 100$, $c = 99$ grid we test 6 different cases where there are different numbers of red and blue cars. For each case, we randomly generate 5 instances of initial grid `g`. Our tests make sure that both `crunBMLGrid1()` and `crunBMLGrid2()` return identical results after $N = 10000$ steps for all 30 instances. We have also manually checked that, upon early break from the outer for-loop due to grid lock when ρ is large, the number of steps executed before breaking are the same for all 3 routines.

3 Running Time Comparison with R's Vectorized Operation

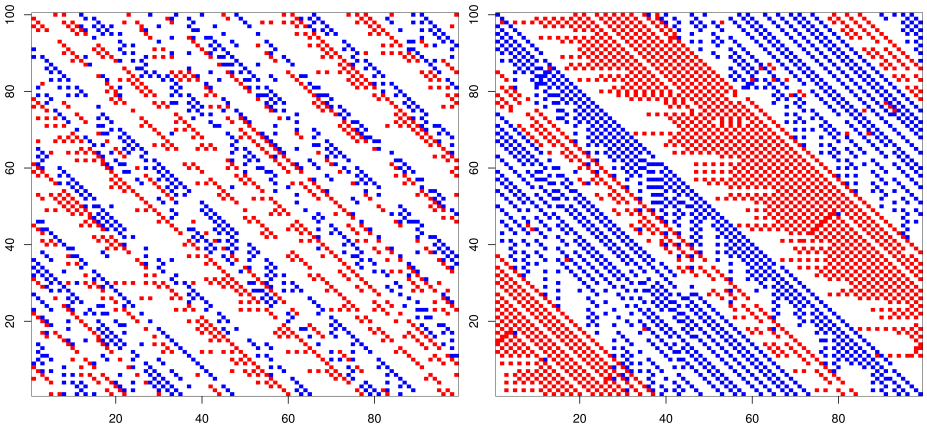
To compare the performance of `crunBMLGrid()` and `runBMLGrid()`, we measure the running time of both functions for $r = c = 128, 256, 512, 1024$ and $\rho = 0.1, 0.2, 9.3, 0.4, 0.5, 0.6, 0.7$. For each of the $4 \times 7 = 28$ settings we randomly generate 10 initial grids and apply `crunBMLGrid1()`, `crunBMLGrid2()` and `runBMLGrid()` on them and record the average running time. We also fix the number of steps to $N = 10000$ and have the same number of red and blue cars in the grid. Again we run this test on a Dell Precision T1700 workstation equipped with 16GB DDR3 RAM and a Core i7-4790K CPU in Ubuntu 14.04 OS. The average running time in seconds and the relative speed up from `runBMLGrid()` to `crunBMLGrid1()` and `crunBMLGrid2()` are plotted in Fig. 2 and Fig. 3, repectively. The original data of Fig. 2 is also provided in the Appendix A. The main results are summarized as follows:

- For `crunBMLGrid2()` where only the key routines are rewritten in C++, there is a limited speed up of less than 2x, and this speed up remains relatively constant for different ρ and l.
- For `crunBMLGrid1()` which is entirely rewritten in C++, there is a significant speed up from 4x to 9x. In general, the larger the edge length, the smaller the speed up is. The speed up for cases where there is no grid lock detected is significantly higher than the cases where there is grid lock and early breaks. Surprisingly, the peak of running time for both `runBMLGrid()` and `crunBMLGrid2()` appears at $\rho = 0.3$, yet that for `crunBMLGrid1()` appears at $\rho = 0.4$ (In our running time test we have also verified that all 3 functions returned the same results). The above two observations probably indicate that early break from the outer for-loop in `crunBMLGrid1()` comes with a significant price.

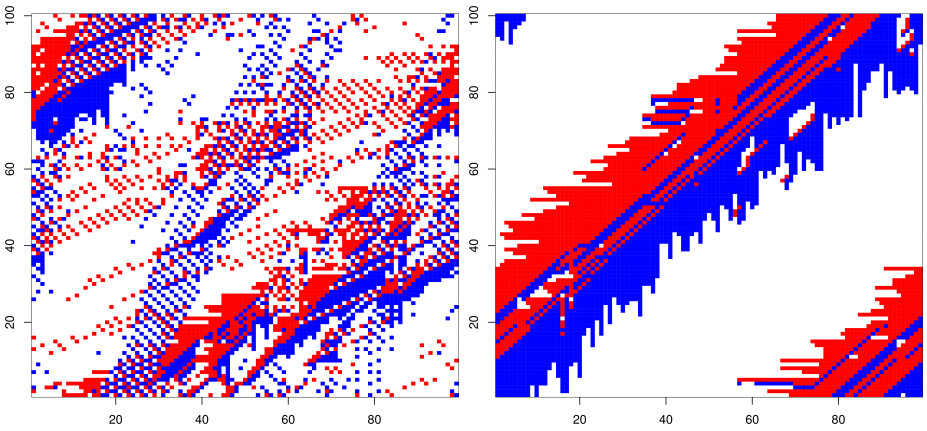
4 Conclusion

Based on the above results, our conclusion is

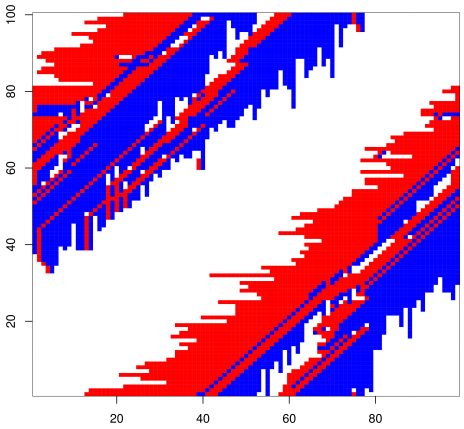
- In this assignment, it is not worth rewriting the key routines in C++ as in `crunBMLGrid2()`, since the performance improvement is very limited.
- Although `crunBMLGrid2()` offers considerable speed up, in my opinion it is still not worth rewriting the entire routine in C++. First of all, unless we are dealing with really time-consuming simulation, R's advantage of agile development will outweigh the greater running speed of C++ routine. Also it is more difficult for C++ routine to return differen types of outputs given different inputs. Finally, it is more difficult to add other functionalities (such as animation) to the C++ routines not being able to use the abundant R packages.
- It is not worth implementing the grid creation function in C since this is not a compuation-intensive routine.



(a) $\rho = 0.2$ (b) $\rho = 0.33$



(c) $\rho = 0.38$ (d) $\rho = 0.43$



(e) $\rho = 0.5$

Figure 1: Final state of a 100×99 grid with equal number of blue and red cars after 10000 steps for different car density ρ.

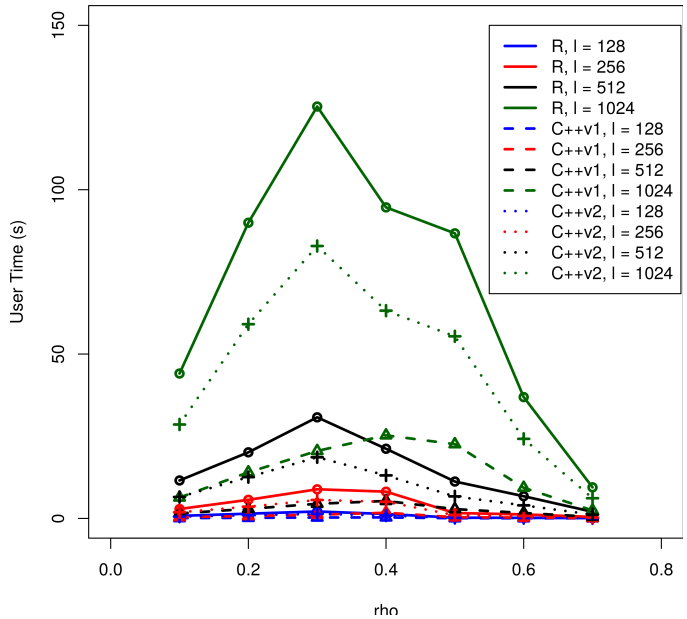


Figure 2: User time of `crunBMLGrid(g, 10000)` `runBMLGrid(g, 10000)` averaged over 10 repetitions for $\rho = 0.1, 0.2, 9.3, 0.4, 0.5, 0.6, 0.7$ and $r = c = 128, 256, 512, 1024$.

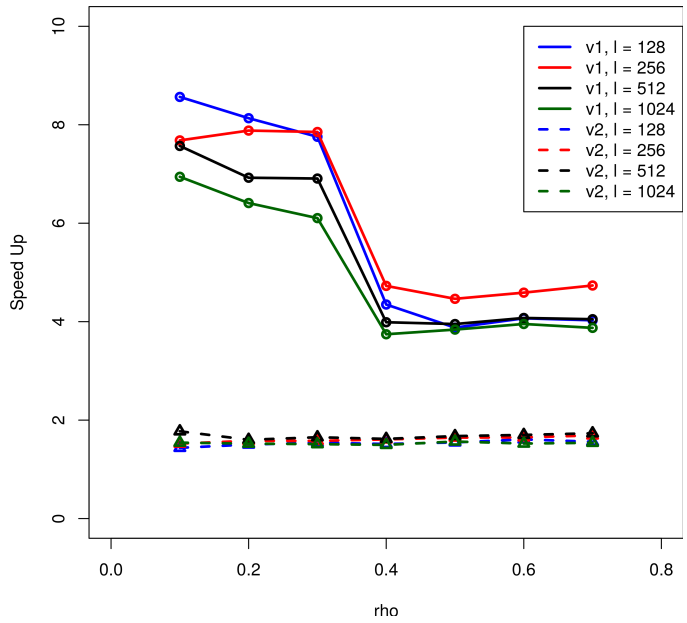


Figure 3: Relative speed up by rewriting the R routine with C++.

References

[1] Wenhao Wu. STA 242 Assignment 4: R Package 'BMLGrid'. `git@bitbucket.org:shasqua/stat242_2015_assignment3.git`, 2015. [Online; accessed 10-May-2015].

[2] Hadley Wickham. Advanced R. `http://adv-r.had.co.nz/`, 2014. [Online; accessed 10-May-2015].

Appendix A: Running Time Results

Table 1: Average running time of `runBMLGrid()` for different edge length (l) and car density (ρ). The number of red and blue cars are equal and the average running time is measured with 10 random initial grids for each case for $N = 10000$ steps.

l	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.3$	$\rho = 0.4$	$\rho = 0.5$	$\rho = 0.6$	$\rho = 0.7$
128	0.7726	1.4485	2.1080	1.3458	0.2242	0.1729	0.0862
256	2.8686	5.6584	8.8592	8.1394	1.6388	1.2199	0.4517
512	11.5545	20.1091	30.7582	21.1879	11.2239	6.7548	2.0129
1024	44.0835	89.9521	125.3059	94.6253	86.7233	36.9050	9.4589

Table 2: Average running time of `crunBMLGrid1()`.

l	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.3$	$\rho = 0.4$	$\rho = 0.5$	$\rho = 0.6$	$\rho = 0.7$
128	0.0902	0.1781	0.2718	0.3095	0.0578	0.0425	0.0214
256	0.3734	0.7179	1.1281	1.7220	0.3671	0.2658	0.0954
512	1.5264	2.9042	4.4529	5.3124	2.8402	1.6575	0.4970
1024	6.3493	14.0341	20.5300	25.2712	22.5889	9.3361	2.4415

Table 3: Average running time of `crunBMLGrid2()`.

l	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.3$	$\rho = 0.4$	$\rho = 0.5$	$\rho = 0.6$	$\rho = 0.7$
128	0.5375	0.9632	1.3688	0.8878	0.1449	0.1075	0.0552
256	1.8832	3.5817	5.6120	5.0610	0.9962	0.7396	0.2674
512	6.5171	12.5623	18.6154	13.0864	6.6969	3.9744	1.1615
1024	28.5689	59.0848	82.8642	63.2015	55.4016	24.2087	6.1441

Appendix B: Source Files

BMLGrid.R

```
#' Constructor for S3 class BMLGrid
#'
#' @param r A non-negative integer, the number of rows of the grid.
#' @param c A non-negative integer, the number of columns of the grid.
#' @param ncars A named vector of 2 non-negative integers where \code{ncars['red']}, \code{ncars['blue']} represent the number of red/blue cars in the grid, respectively.
#' @return A BMLGrid class object which is essentially a matrix.
#' @examples
#' library(BMLGrid)
#' g = createBMLGrid(r = 100, c = 99, ncars = c(red = 100, blue = 100))
#' @export
createBMLGrid <- function(r, c, ncars) {
  cars <- sample(1 : (r * c), ncars['red'] + ncars['blue']) # The vector index of the cars in the grid
  red <- sample(cars, ncars['red'])  # The vector index? of the red cars in the grid
  blue <- setdiff(cars, red)  # The vector index of the blue cars in the grid
  grid <- matrix(0L, nrow = r, ncol = c) # The matrix representing the cars, 0 indicates no cars on that grid
  grid[red] <- 1L # 1 indicates a red car on the grid
  grid[blue] <- 2L # 2 indicates a blue car on the grid

  class(grid) <- 'BMLGrid'
  return(grid)
}

#' plot method for BMLGrid class object
#'
#' Plot the cars on the grid as red/blue squares over a white background.
#'
#' @param x A BMLGrid class object.
```

```
28 #' @param ... Other input arguments are simply ignored
     .
29 #' @examples
30 #' library(BMLGrid)
31 #' g = createBMLGrid(r = 100, c = 99, ncars = c(red =
     100, blue = 100))
32 #' plot(g)
33 #' @export
34 plot.BMLGrid <- function(x, ...) {
35   colormap <- c("white", "red", "blue")
36   image(seq_len(ncol(x)), seq_len(nrow(x)), t(x), col
       = colormap, xlab = '', ylab = '')
37 }
38
39 #' summary method for BMLGrid class object
40 #'
41 #' The summary includes information on the grid size
     and the number of red and blue cars in the grid.
42 #'
43 #' @param object A BMLGrid class object.
44 #' @param ... Other input arguments are simply ignored
     .
45 #' @examples
46 #' library(BMLGrid)
47 #' g = createBMLGrid(r = 100, c = 99, ncars = c(red =
     100, blue = 100))
48 #' summary(g)
49 #' @export
50 summary.BMLGrid <- function(object, ...) {
51   lines <- c("BMLGrid␣class␣object.", paste(c("␣-",
       toString(nrow(object)), 'rows,', toString(ncol(
       object)), 'columns'), collapse = '␣'), paste(c("␣
       -", toString(sum(object == 1)), 'red,', toString(
       sum(object == 2)), 'blue.\n'), collapse = '␣'))
52   return(cat(paste(lines, collapse = '\n')))
53 }
54
55 #' Simulator for Biham-Middleton-Levine Traffic Model.
56 #'
57 #' The function that actuall runs the Biham-Middleton-
     Levine Traffic Model from an initial state by a
     given number of steps.
58 #'
59 #' @import animation
60 #' @param g A BMLGrid class object representing the
     initial state of the grid.
```

```
61 #' @param numSteps Number of moves/periods.
62 #' @param movieName If specified as a non-NULL string,
       functions from package 'animation' will be used to
       record the BML process as a movie.
63 #' @param recordSpeed The flag value indicating
       whether to record and return the average speed of
       the red and blue cars ar each step.
64 #' @return If recordSpeed is unspecified or specified
       as \code{FALSE}, returns a \code{BMLGrid} object
       representing the final state of the simulation;
       otherwise return a list where the first element is
       the final-state grid object and the 2nd and 3rd
       elements record the average speed of red cars and
       blue cars, respectively.
65 #' @examples
66 #' library(BMLGrid)
67 #' g = createBMLGrid(r = 100, c = 99, ncars = c(red =
       100, blue = 100))
68 #' g.out = runBMLGrid(g, numSteps = 10000)
69 #' plot(g.out)
70 #' g.out = runBMLGrid(g, numSteps = 50, movieName = '
       movieBMLGrid', recordSpeed = TRUE)
71 #' plot(g.out$g)
72 #' summary(g.out$v.blue)
73 #' summary(g.out$v.red)
74 #' @export
75 runBMLGrid <- function(g, numSteps, movieName = NULL,
       recordSpeed = FALSE) {
76   r <- nrow(g)
77   c <- ncol(g)
78   red <- which(g == 1L) # Get the initial locations of
         red and blue cars
79   blue <- which(g == 2L)
80   white <- which(g == 0L)
81
82   if (length(red) + length(blue) + length(white) != r
       * c || typeof(g) != "integer")
83   {
84     stop("Wrong␣grid␣format:␣values␣should␣be␣0,␣1,␣2␣
         only!")
85   }
86   if (!is.numeric(numSteps) || numSteps < 0 || as.
       integer(numSteps) != numSteps){
87     stop("numSteps␣must␣be␣an␣integer␣greater␣than␣or␣
         equal␣to␣0!")
88   }
```

```
89    if (r == 0 || c == 0 || (length(red) + length(blue))
          == 0) { # Degenerate cases, return immediatly
90      warning('Degenerate␣BMLGrid␣object!')
91      flush.console()
92      return(g)
93    }
94
95    flag_movie <- !is.null(movieName)
96    if (flag_movie){
97      par(bg = "white") # ensure the background color is
            white
98      plot(c, r, type = "n")
99      animation::ani.record(reset = TRUE) # clear
            history before recording
100     plot(g) # Plot the initial g
101     animation::ani.record() # record the current frame
102   }
103
104   if (recordSpeed) {
105     nmoved <- rep(0, numSteps + 1) # Record the number
            of cars moved at each step
106     nmoved[1] <- get_nmoved(g, r, c, blue, 'up')
107   }
108   movable_any <- TRUE
109   for (step in seq_len(numSteps)) {
110     if (step %% 2 == 0) { # Red cars move to right by
            1 grid
111       red_right <- idx_right(red, r, c) # The vector
              index of the right grids to current red cars
112       movable <- (g[red_right] == 0L)
113       g[red[movable]] <- 0L # Update grid
114       g[red_right[movable]] <- 1L
115       red <- c(red_right[movable], red[!movable])
116       if (recordSpeed) {
117         nmoved[step + 1] <- get_nmoved(g, r, c, red, '
                up')  # Record the number of cars moved at
                each step
118       }
119     } else { # Blue cars move upward by 1 grid
120       blue_up <- idx_up(blue, r, c) # The vector index
              of the right grids to current red cars
121       movable <- (g[blue_up] == 0L)
122       g[blue[movable]] <- 0L # Update grid
123       g[blue_up[movable]] <- 2L
124       blue <- c(blue_up[movable], blue[!movable])
125       if (recordSpeed) {
```

```
126           nmoved[step + 1] <- get_nmoved(g, r, c, blue,
                  'right')  # Record the number of cars moved
                   at each step
127         }
128       }
129       if (!movable_any && !any(movable)) {
130         #warning('fuckyou!')
131         warning(paste('Grid lock detected at step',
                  toString(step)))
132         flush.console()
133         break # We have entered a grid lock, no need to
                  continue
134       } else {
135         movable_any <- any(movable)
136       }
137       if (flag_movie){
138         plot(g) # Plot g
139         animation::ani.record() # record the current
                  frame
140       }
141     }
142     if (flag_movie){
143       oopts = animation::ani.options(interval = 1)
144       animation::saveHTML(animation::ani.replay(), img.
                name = toString(movieName)) # export the
                animation to an HTML page
145     }
146     if (recordSpeed) {
147       n_moved_blue <- nmoved[seq(1, numSteps, by=2)]
148       n_moved_red <- nmoved[seq(2, numSteps, by=2)]
149       return(list(g = g, v.blue = n_moved_blue / length(
                blue), v.red = n_moved_red / length(red)))
150     }
151     return(g)
152   }
153
154   #' Simulator for Biham-Middleton-Levine Traffic Model,
          with key operations written in C++.
155   #'
156   #' The function that actuall runs the Biham-Middleton-
        Levine Traffic Model from an initial state by a
        given number of steps.
157   #'
158   #' @import animation
159   #' @param g A BMLGrid class object representing the
        initial state of the grid.
```

```
160 #' @param numSteps Number of moves/periods.
161 #' @return a \code{BMLGrid} object representing the
      final state of the simulation.
162 #' @examples
163 #' library(BMLGrid)
164 #' g = createBMLGrid(r = 100, c = 99, ncars = c(red =
      100, blue = 100))
165 #' g.out = crunBMLGrid2(g, numSteps = 10000)
166 #' plot(g.out)
167 #' @export
168 crunBMLGrid2 <- function(g, numSteps) {
169   r <- nrow(g)
170   c <- ncol(g)
171   red <- which(g == 1L) # Get the initial locations of
        red and blue cars
172   blue <- which(g == 2L)
173   white <- which(g == 0L)
174
175   if (length(red) + length(blue) + length(white) != r
      * c || typeof(g) != "integer")
176   {
177     stop("Wrong␣grid␣format:␣values␣should␣be␣0,␣1,␣2␣
        only!")
178   }
179   if (!is.numeric(numSteps) || numSteps < 0 || as.
      integer(numSteps) != numSteps){
180     stop("numSteps␣must␣be␣an␣integer␣greater␣than␣or␣
        equal␣to␣0!")
181   }
182   if (r == 0 || c == 0 || (length(red) + length(blue))
       == 0) { # Degenerate cases, return immediatly
183     warning('Degenerate␣BMLGrid␣object!')
184     flush.console()
185     return(g)
186   }
187
188   movable_any <- TRUE
189   for (step in seq_len(numSteps)) {
190     if (step %% 2 == 0) { # Red cars move to right by
        1 grid
191       red_right <- cidx_right(red, r, c) # The vector
          index of the right grids to current red cars
192       movable <- (g[red_right] == 0L)
193       g[red[movable]] <- 0L # Update grid
194       g[red_right[movable]] <- 1L
195       red <- c(red_right[movable], red[!movable])
```

```
196       } else { # Blue cars move upward by 1 grid
197         blue_up <- cidx_up(blue, r) # The vector index
                of the right grids to current red cars
198         movable <- (g[blue_up] == 0L)
199         g[blue[movable]] <- 0L # Update grid
200         g[blue_up[movable]] <- 2L
201         blue <- c(blue_up[movable], blue[!movable])
202       }
203       if (!movable_any && !any(movable)) {
204         #warning('fuckyou!')
205         warning(paste('Grid lock detected at step',
                toString(step)))
206         flush.console()
207         break # We have entered a grid lock, no need to
                continue
208       } else {
209         movable_any <- any(movable)
210       }
211     }
212     return(g)
213   }
214
215   # Vectorized function to get the vector index of the
        right grid right to the current grid
216   idx_right <- function(idx, r, c) {
217     return((idx + r - 1) %% (r * c) + 1)
218   }
219
220   # Vectorized function to get the vector index of the
        right grid right to the current grid
221   idx_up <- function(idx, r, c) {
222     return(idx %% r + 1 + ((idx - 1) %/% r) * r)
223   }
224
225   # Function to compute the number of cars that moved
        given the index of cars we would like to move, the
        current grid and whether we would like to move up
        or right
226   get_nmoved <- function(grid, r, c, cars, direction) {
227     if (direction == 'up') {
228       return(sum(grid[idx_up(cars, r, c)] == 0))
229     } else {
230       return(sum(grid[idx_right(cars, r, c)] == 0))
231     }
232   }
```

BMLGrid.h

```
1  #ifndef BMLGRID
2  #define BMLGRID
3
4  #include <Rcpp.h>
5  using namespace Rcpp;
6
7  // Enable C++11 via this plugin (Rcpp 0.10.3 or later)
8  // [[Rcpp::plugins(cpp11)]]
9
10 //' Simulator for Biham-Middleton-Levine Traffic Model
       written in c++.
11 //'
12 //' The function that actually runs the Biham-
       Middleton-Levine Traffic Model from an initial
       state by a given number of steps.
13 //' @param g A BMLGrid class object representing the
       initial state of the grid.
14 //' @param numSteps Number of moves/periods.
15 //' @examples
16 //' library(BMLGrid)
17 //' g = createBMLGrid(r = 100, c = 99, ncars = c(red =
       100, blue = 100))
18 //' g.out = crunBMLGrid1(g, 10000)
19 //' plot(g.out)
20 //' @export
21 // [[Rcpp::export]]
22 IntegerMatrix crunBMLGrid1(IntegerMatrix g, int
       numSteps);
23
24 // Function to locate in grid 'g' all cars of 'color',
       equivalent to which()
25 IntegerVector locateColor(const IntegerVector& g, int
       color);
26
27 // Function to move cars of a certain color to their
       next location if possible
28 //
29 // The function that actuall runs the Biham-Middleton-
       Levine Traffic Model from an initial state by a
       given number of steps.
30 // @param g A BMLGrid class object representing the
       current state of the grid.
31 // @param loc The location of cars of a certain color
       that we would like to move.
```

```
32 // @param nextLoc The function to use to get the next
       location of a car, either nextLocUp() for blue car
       or nextLocRight() for red car.
33 // @param buffer_loc_next The vector that holds the
       intermediate variable of the next locations.
       Supplied to function to avoid repeated construction
       /destruction.
34 // @param buffer_movable The vector that holds the
       intermediate bool variable indicating whether a car
        is movable or not.
35 bool moveCars(IntegerMatrix& g, IntegerVector& loc,
       std::function<int(int,int,int)> nextLoc,
       IntegerVector& buffer_loc_next, std::vector<bool>&
       buffer_movable);
36
37 // Function to return the location (cyclicly) above
       the current location
38 int nextLocUp(int loc, int r, int c);
39
40 // Function to return the location (cyclicly) above
       the current location
41 int nextLocRight(int loc, int r, int c);
42
43 //' Function to get the vector index of the grid right
        to the current grid.
44 //'
45 //' c++ implementation of the idx_right() fucntion
46 //' @param idx Current locations (vector index in the
       grid) of cars of a certain color.
47 //' @param r numbers of rows
48 //' @param c number of columns
49 // [[Rcpp::export]]
50 IntegerVector cidx_right(IntegerVector idx, int r, int
        c);
51
52 //' Function to get the vector index of the grid above
        the current grid.
53 //'
54 //' c++ implementation of the idx_up() fucntion
55 //' @param idx Current locations (vector index in the
       grid) of cars of a certain color.
56 //' @param r numbers of rows
57 // [[Rcpp::export]]
58 IntegerVector cidx_up(IntegerVector idx, int r);
59 #endif
```

BMLGrid.cpp

```
1  #include "BMLGrid.h"
2  using namespace Rcpp;
3
4  IntegerMatrix crunBMLGrid1(IntegerMatrix gInput, int
       numSteps)
5  {
6    IntegerMatrix g = clone(gInput); // Copy the input
       matrix so that it won't be affected by the
       undefined behavior of modifying inputs
7    int r = g.nrow();
8    int c = g.ncol();
9    if (0 == r || 0 == c || numSteps < 0)
10   {
11     //Rcout << "Degenerate BMLGrid object." << std::
         endl;
12     return(g);
13   }
14
15   IntegerVector red = locateColor(g, 1);
16   IntegerVector blue = locateColor(g, 2);
17   IntegerVector white = locateColor(g, 0);
18
19   int buffer_size = ((red.size() > blue.size()) ? red.
       size() : blue.size());
20   IntegerVector buffer_loc_next(buffer_size);
21   std::vector<bool> buffer_movable(buffer_size);
22
23   if (red.size() + blue.size() + white.size() != r * c
       )
24   {
25     stop("Wrong␣grid␣format:␣values␣should␣be␣0,␣1,␣2␣
         only!");
26   }
27   if (0 == red.size() + blue.size())
28   {
29     //Rcout << "Degenerate BMLGrid object." << std::
         endl;
30     return(g);
31   }
32
33   bool movable_last = true;
34   bool movable;
35   for (int step = 0; step < numSteps; step++)
36   {
```

```
37      checkUserInterrupt();
38      if (0 == step % 2)
39      {
40        movable = moveCars(g, blue, nextLocUp,
              buffer_loc_next, buffer_movable);
41      }
42      else
43      {
44        movable = moveCars(g, red, nextLocRight,
              buffer_loc_next, buffer_movable);
45      }
46      if (!movable_last && !movable)
47      {
48        Rcout << "Grid␣lock␣detected␣at␣step␣" << step +
              1 << std::endl;
49        break;
50      }
51      else
52      {
53        movable_last = movable;
54      }
55    }
56    return g;
57  }
58
59  IntegerVector locateColor(const IntegerVector& g, int
      color)
60  {
61    IntegerVector v = seq(0, g.size() - 1);
62    return v[g == color];
63  }
64
65  bool moveCars(IntegerMatrix& g, IntegerVector& loc,
      std::function<int(int,int,int)> nextLoc,
      IntegerVector& buffer_loc_next, std::vector<bool>&
      buffer_movable)
66  {
67    int r = g.nrow();
68    int c = g.ncol();
69    bool movable_any = false;
70
71    IntegerVector::iterator itr_loc, itr_loc_next;
72    std::vector<bool>::iterator itr_movable;
73
74    // The first loop: compute the next locations and
        identify the movable cars
```

```
75    itr_loc_next = buffer_loc_next.begin();
76    itr_movable = buffer_movable.begin();
77    for (itr_loc = loc.begin(); itr_loc != loc.end();
          itr_loc++, itr_loc_next++, itr_movable++)
78    {
79      *itr_loc_next = nextLoc(*itr_loc, r, c);
80      if (g[*itr_loc_next] == 0) // The next location is
            not occupied, move the car
81      {
82        *itr_movable = true;
83        movable_any = true;
84      }
85      else
86      {
87        *itr_movable = false;
88      }
89    }
90
91    // The second loop: move cars according to the
          result from the first loop
92    itr_loc_next = buffer_loc_next.begin();
93    itr_movable = buffer_movable.begin();
94    for (itr_loc = loc.begin(); itr_loc != loc.end();
          itr_loc++, itr_loc_next++, itr_movable++)
95    {
96      if (*itr_movable) // The next location is not
            occupied, move the car
97      {
98        g[*itr_loc_next] = g[*itr_loc];
99        g[*itr_loc] = 0;
100       *itr_loc = *itr_loc_next;
101     }
102   }
103   return movable_any;
104 }
105
106 int nextLocUp(int loc, int r, int c)
107 {
108   return((loc + 1) % r + (loc / r) * r);
109 }
110
111 int nextLocRight(int loc, int r, int c)
112 {
113   return((loc + r) % (r * c));
114 }
115
```

```
116 IntegerVector cidx_right(IntegerVector idx, int r, int
        c)
117 {
118   IntegerVector idx_next(idx.size());
119   IntegerVector::iterator itr_idx, itr_idx_next;
120   itr_idx_next = idx_next.begin();
121   for (itr_idx = idx.begin(); itr_idx != idx.end();
         itr_idx++, itr_idx_next++)
122   {
123     *itr_idx_next = nextLocRight(*itr_idx - 1, r, c) +
           1;
124   }
125   return(idx_next);
126 }
127
128 IntegerVector cidx_up(IntegerVector idx, int r)
129 {
130   IntegerVector idx_next(idx.size());
131   IntegerVector::iterator itr_idx, itr_idx_next;
132   itr_idx_next = idx_next.begin();
133   for (itr_idx = idx.begin(); itr_idx != idx.end();
         itr_idx++, itr_idx_next++)
134   {
135     *itr_idx_next = nextLocUp(*itr_idx - 1, r, 0) + 1;
136   }
137   return(idx_next);
138 }
```