UC Davis STA 242 2015 Spring Assignment 4 [1]

Wenhao Wu, 998587583

May 11, 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 [3]: 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 `crunBMLGrid1()` and `crunBMLGrid2()` with `runBMLGrid()`, we measure the running time of all the three 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 20 initial grids and apply `crunBMLGrid1()`, `crunBMLGrid2()` and `runBMLGrid()` on them and record the 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 and error bar of the running time in seconds and the relative speed up of the average running time from `runBMLGrid()` to `crunBMLGrid1()` and `crunBMLGrid2()` are plotted in Fig. 2 and Fig. 3, repectively. We have also plotted the running time for different number of steps of N for $\rho = 0.3, 0.4, 0.5$ and $r = c = 256$ in Fig. 4. 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, l and N.

- For `crunBMLGrid1()` which is entirely rewritten in C++, there is a significant speed up from 3x 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). In Fig. 4 we further verified that when $\rho = 0.4$, $N = 1000$, even though for all the 20 initial grids the early break has not been detected, the relative speed up ($\sim$ 4x) has already been much lower than that when $\rho = 0.3$. This indicates that the sharp drop of speed up in Fig. 3 has nothing to do with the grid-detection mechanism in our algorithm. Our best guess is that it has something to do with R's or Rcpp's memory management schemes.

- From Fig. 2 and Fig. 4, we notice that when the BML model is in ordered state ($\rho < 0.4$), the variances of the running time of all 3 routines are very small among different initial states. When the BML model is in the critical state ($\rho \approx 0.4$), the variances of the running time are very large since different initial states of the grid may lead to significantly different early break time (if there is grid lock detected). When the BML model is in the stale state ($\rho > 0.4$), as ρ grows, the variances of running time begin to decrease since most initaial grid states causes the grid to enter the grid lock mode after only a few steps.

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.

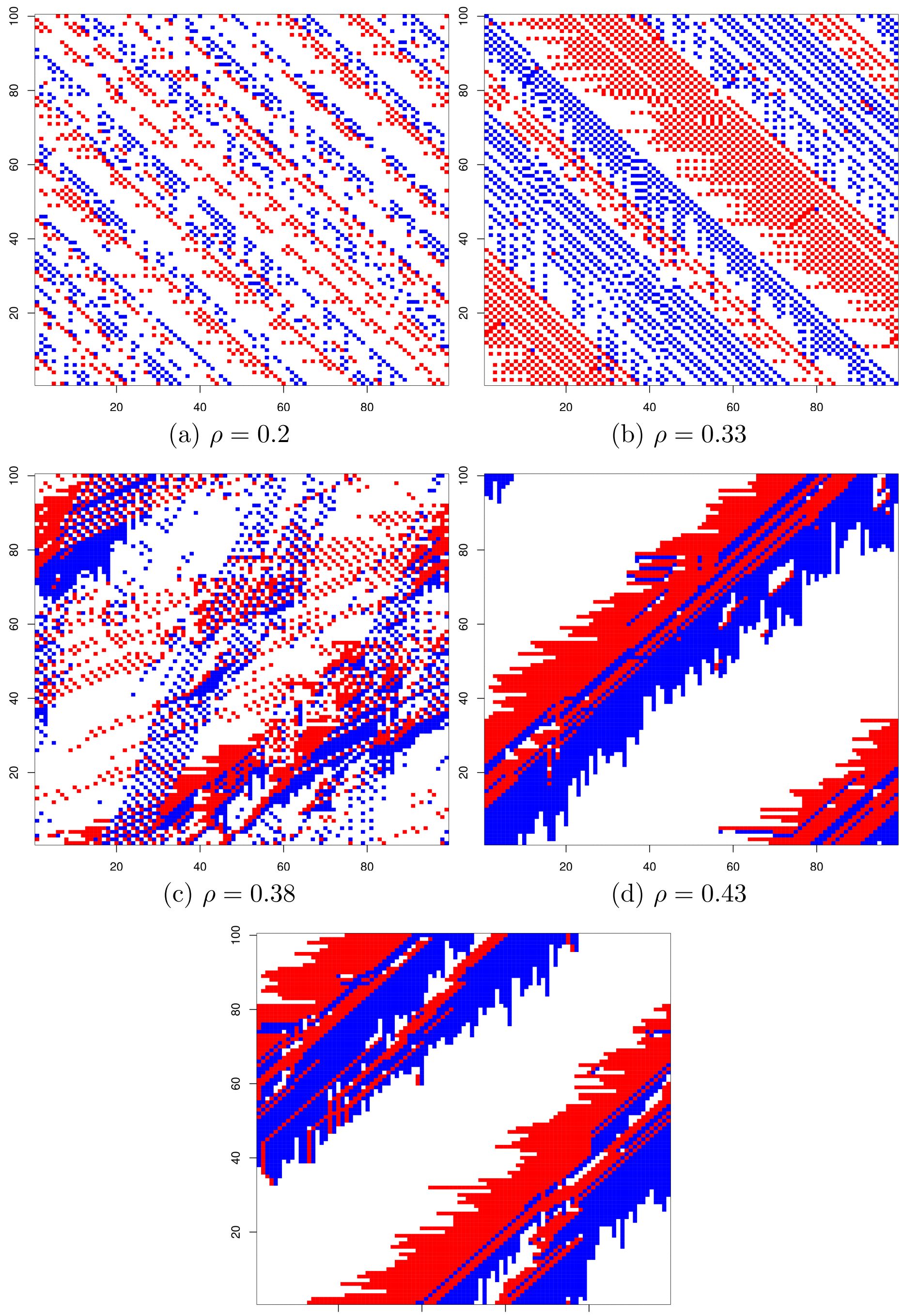
- It is not worth implementing the grid creation function in C since this is not a compuation-intensive routine.

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].

[3] Efficient Ragged Arrays in R and Rcpp. `http://helmingstay.blogspot.com/2014/07/computational-speed-is-common-complaint.html`, 2014. [Online; accessed 11-May-2015].

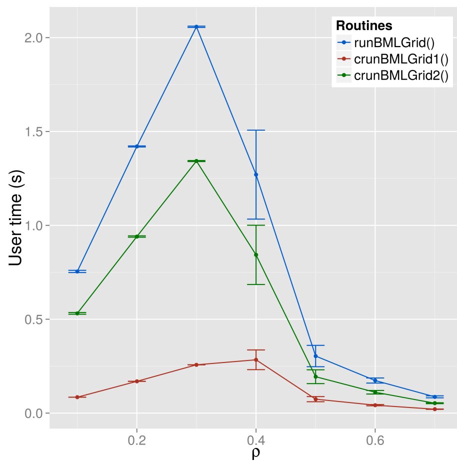


(a) $\rho = 0.2$ (b) $\rho = 0.33$

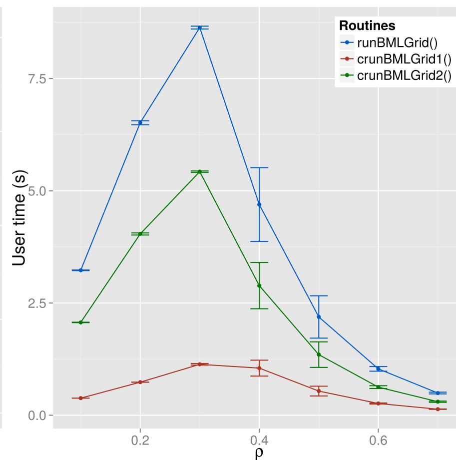
(c) $\rho = 0.38$ (d) $\rho = 0.43$

(e) $\rho = 0.5$

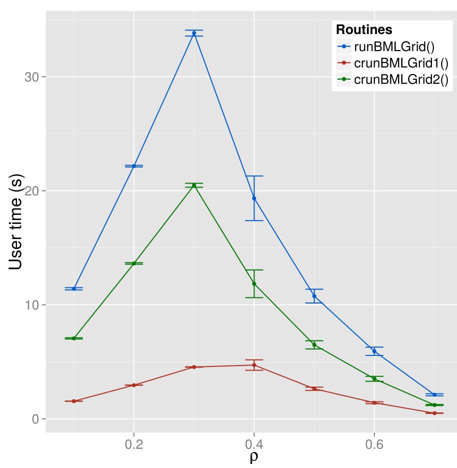
Figure 1: Final state of a 100×99 grid with equal number of blue and red cars after 10000 steps for different car density ρ.



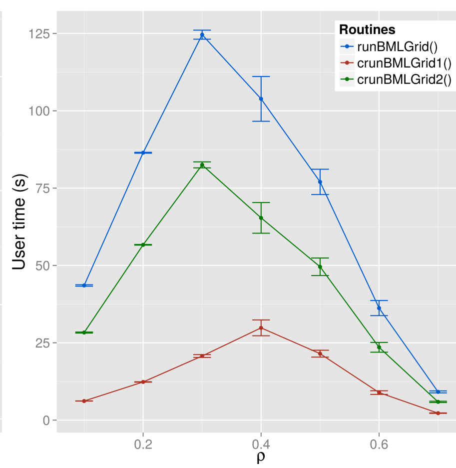
(a) $l = 128$



(b) $l = 256$



(c) $l = 512$



(d) $l = 1024$

Figure 2: User time comparison between `runBMLGrid(g, 10000)`, `crunBMLGrid1(g, 10000)` and `crunBMLGrid2(g, 10000)` over 20 random intial grids for $\rho = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7$ and $r = c = 128, 256, 512, 1024$.

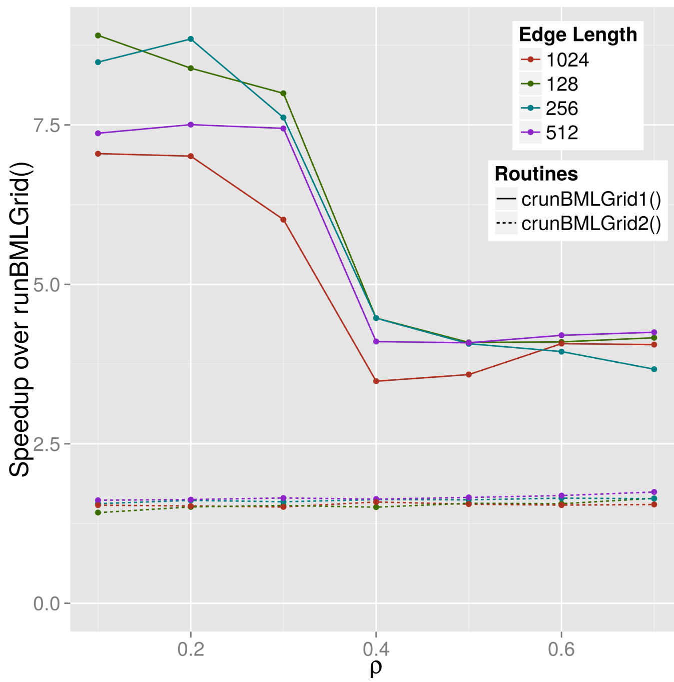
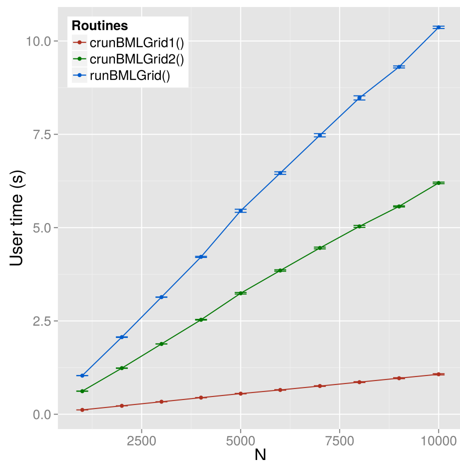
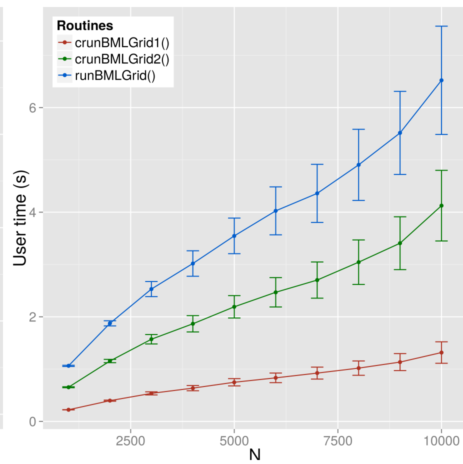


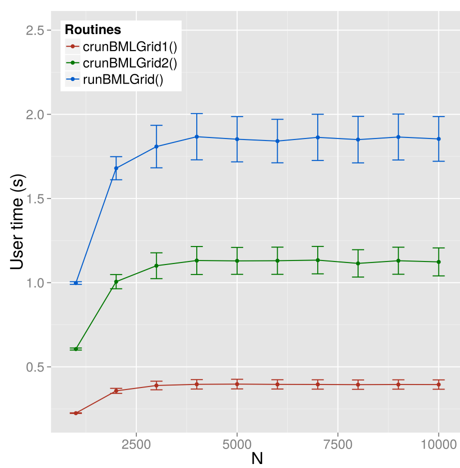
Figure 3: Speed up from `runBMLGrid(g, 10000)` to `crunBMLGrid1(g, 10000)` and `crunBMLGrid2(g, 10000)` over 20 random intial grids for $\rho = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7$ and $r = c = 128, 256, 512, 1024$.



(a) $\rho = 0.3$



(b) $\rho = 0.4$



(c) $\rho = 0.5$

Figure 4: User time comparison between `runBMLGrid(g, N)`, `crunBMLGrid1(g, N)` and `crunBMLGrid2(g, N)` over 20 random intial grids for $\rho = 0.3, 0.4, 0.5$ and $r = c = 256$.

Appendix: Source Files

BMLGrid.R

```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.
```

```r
#' @param ... Other input arguments are simply ignored.
#' @examples
#' library(BMLGrid)
#' g = createBMLGrid(r = 100, c = 99, ncars = c(red = 100, blue = 100))
#' plot(g)
#' @export
plot.BMLGrid <- function(x, ...) {
  colormap <- c("white", "red", "blue")
  image(seq_len(ncol(x)), seq_len(nrow(x)), t(x), col = colormap, xlab = '', ylab = '')
}

#' summary method for BMLGrid class object
#'
#' The summary includes information on the grid size and the number of red and blue cars in the grid.
#'
#' @param object A BMLGrid class object.
#' @param ... Other input arguments are simply ignored.
#' @examples
#' library(BMLGrid)
#' g = createBMLGrid(r = 100, c = 99, ncars = c(red = 100, blue = 100))
#' summary(g)
#' @export
summary.BMLGrid <- function(object, ...) {
  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 = ' '))
  return(cat(paste(lines, collapse = '\n')))
}

#' Simulator for Biham-Middleton-Levine Traffic Model.
#'
#' The function that actuall runs the Biham-Middleton-Levine Traffic Model from an initial state by a given number of steps.
#'
#' @import animation
#' @param g A BMLGrid class object representing the initial state of the grid.
```

```r
#' @param numSteps Number of moves/periods.
#' @param movieName If specified as a non-NULL string, functions from package 'animation' will be used to record the BML process as a movie.
#' @param recordSpeed The flag value indicating whether to record and return the average speed of the red and blue cars ar each step.
#' @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.
#' @examples
#' library(BMLGrid)
#' g = createBMLGrid(r = 100, c = 99, ncars = c(red = 100, blue = 100))
#' g.out = runBMLGrid(g, numSteps = 10000)
#' plot(g.out)
#' g.out = runBMLGrid(g, numSteps = 50, movieName = 'movieBMLGrid', recordSpeed = TRUE)
#' plot(g.out$g)
#' summary(g.out$v.blue)
#' summary(g.out$v.red)
#' @export
runBMLGrid <- function(g, numSteps, movieName = NULL, recordSpeed = FALSE) {
  r <- nrow(g)
  c <- ncol(g)
  red <- which(g == 1L) # Get the initial locations of red and blue cars
  blue <- which(g == 2L)
  white <- which(g == 0L)

  if (length(red) + length(blue) + length(white) != r * c || typeof(g) != "integer")
  {
    stop("Wrong grid format: values should be 0, 1, 2 only!")
  }
  if (!is.numeric(numSteps) || numSteps < 0 || as.integer(numSteps) != numSteps){
    stop("numSteps must be an integer greater than or equal to 0!")
  }
```

```r
  if (r == 0 || c == 0 || (length(red) + length(blue)) == 0) { # Degenerate cases, return immediatly
    warning('Degenerate BMLGrid object!')
    flush.console()
    return(g)
  }

  flag_movie <- !is.null(movieName)
  if (flag_movie){
    par(bg = "white") # ensure the background color is white
    plot(c, r, type = "n")
    animation::ani.record(reset = TRUE) # clear history before recording
    plot(g) # Plot the initial g
    animation::ani.record() # record the current frame
  }

  if (recordSpeed) {
    nmoved <- rep(0, numSteps + 1) # Record the number of cars moved at each step
    nmoved[1] <- get_nmoved(g, r, c, blue, 'up')
  }
  movable_any <- TRUE
  for (step in seq_len(numSteps)) {
    if (step %% 2 == 0) { # Red cars move to right by 1 grid
      red_right <- idx_right(red, r, c) # The vector index of the right grids to current red cars
      movable <- (g[red_right] == 0L)
      g[red[movable]] <- 0L # Update grid
      g[red_right[movable]] <- 1L
      red <- c(red_right[movable], red[!movable])
      if (recordSpeed) {
        nmoved[step + 1] <- get_nmoved(g, r, c, red, 'up')  # Record the number of cars moved at each step
      }
    } else { # Blue cars move upward by 1 grid
      blue_up <- idx_up(blue, r, c) # The vector index of the right grids to current red cars
      movable <- (g[blue_up] == 0L)
      g[blue[movable]] <- 0L # Update grid
      g[blue_up[movable]] <- 2L
      blue <- c(blue_up[movable], blue[!movable])
      if (recordSpeed) {
```

```
        nmoved[step + 1] <- get_nmoved(g, r, c, blue, 'right')  # Record the number of cars moved at each step
      }
    }
    if (!movable_any && !any(movable)) {
      #warning('fuckyou!')
      warning(paste('Grid lock detected at step', toString(step)))
      flush.console()
      break # We have entered a grid lock, no need to continue
    } else {
      movable_any <- any(movable)
    }
    if (flag_movie){
      plot(g) # Plot g
      animation::ani.record() # record the current frame
    }
  }
  if (flag_movie){
    oopts = animation::ani.options(interval = 1)
    animation::saveHTML(animation::ani.replay(), img.name = toString(movieName)) # export the animation to an HTML page
  }
  if (recordSpeed) {
    n_moved_blue <- nmoved[seq(1, numSteps, by=2)]
    n_moved_red <- nmoved[seq(2, numSteps, by=2)]
    return(list(g = g, v.blue = n_moved_blue / length(blue), v.red = n_moved_red / length(red)))
  }
  return(g)
}

#' Simulator for Biham-Middleton-Levine Traffic Model, with key operations written in C++.
#'
#' The function that actuall runs the Biham-Middleton-Levine Traffic Model from an initial state by a given number of steps.
#'
#' @import animation
#' @param g A BMLGrid class object representing the initial state of the grid.
```

```r
#' @param numSteps Number of moves/periods.
#' @return a \code{BMLGrid} object representing the final state of the simulation.
#' @examples
#' library(BMLGrid)
#' g = createBMLGrid(r = 100, c = 99, ncars = c(red = 100, blue = 100))
#' g.out = crunBMLGrid2(g, numSteps = 10000)
#' plot(g.out)
#' @export
crunBMLGrid2 <- function(g, numSteps) {
  r <- nrow(g)
  c <- ncol(g)
  red <- which(g == 1L) # Get the initial locations of red and blue cars
  blue <- which(g == 2L)
  white <- which(g == 0L)

  if (length(red) + length(blue) + length(white) != r * c || typeof(g) != "integer")
  {
    stop("Wrong grid format: values should be 0, 1, 2 only!")
  }
  if (!is.numeric(numSteps) || numSteps < 0 || as.integer(numSteps) != numSteps){
    stop("numSteps must be an integer greater than or equal to 0!")
  }
  if (r == 0 || c == 0 || (length(red) + length(blue)) == 0) { # Degenerate cases, return immediatly
    warning('Degenerate BMLGrid object!')
    flush.console()
    return(g)
  }

  movable_any <- TRUE
  for (step in seq_len(numSteps)) {
    if (step %% 2 == 0) { # Red cars move to right by 1 grid
      red_right <- cidx_right(red, r, c) # The vector index of the right grids to current red cars
      movable <- (g[red_right] == 0L)
      g[red[movable]] <- 0L # Update grid
      g[red_right[movable]] <- 1L
      red <- c(red_right[movable], red[!movable])
```

```r
    } else { # Blue cars move upward by 1 grid
      blue_up <- cidx_up(blue, r) # The vector index of the right grids to current red cars
      movable <- (g[blue_up] == 0L)
      g[blue[movable]] <- 0L # Update grid
      g[blue_up[movable]] <- 2L
      blue <- c(blue_up[movable], blue[!movable])
    }
    if (!movable_any && !any(movable)) {
      #warning('fuckyou!')
      warning(paste('Grid lock detected at step', toString(step)))
      flush.console()
      break # We have entered a grid lock, no need to continue
    } else {
      movable_any <- any(movable)
    }
  }
  return(g)
}

# Vectorized function to get the vector index of the right grid right to the current grid
idx_right <- function(idx, r, c) {
  return((idx + r - 1) %% (r * c) + 1)
}

# Vectorized function to get the vector index of the right grid right to the current grid
idx_up <- function(idx, r, c) {
  return(idx %% r + 1 + ((idx - 1) %/% r) * r)
}

# 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
get_nmoved <- function(grid, r, c, cars, direction) {
  if (direction == 'up') {
    return(sum(grid[idx_up(cars, r, c)] == 0))
  } else {
    return(sum(grid[idx_right(cars, r, c)] == 0))
  }
}
```

BMLGrid.h

```
#ifndef BMLGRID
#define BMLGRID

#include <Rcpp.h>
using namespace Rcpp;

// Enable C++11 via this plugin (Rcpp 0.10.3 or later)
// [[Rcpp::plugins(cpp11)]]

//' Simulator for Biham-Middleton-Levine Traffic Model written in c++.
//'
//' The function that actually runs the Biham-Middleton-Levine Traffic Model from an initial state by a given number of steps.
//' @param g A BMLGrid class object representing the initial state of the grid.
//' @param numSteps Number of moves/periods.
//' @param warningGridLock bool value indicating whether to prompt to Rcout when grid lock is detected. Default value is false.
//' @examples
//' library(BMLGrid)
//' g = createBMLGrid(r = 100, c = 99, ncars = c(red = 100, blue = 100))
//' g.out = crunBMLGrid1(g, 10000)
//' plot(g.out)
//' @export
// [[Rcpp::export]]
IntegerMatrix crunBMLGrid1(IntegerMatrix g, int numSteps, bool warningGridLock = false);

// Function to locate in grid 'g' all cars of 'color', equivalent to which()
IntegerVector locateColor(const IntegerVector& g, int color);

// Function to move cars of a certain color to their next location if possible
//
// The function that actuall runs the Biham-Middleton-Levine Traffic Model from an initial state by a given number of steps.
```

```
// @param g A BMLGrid class object representing the current state of the grid.
// @param loc The location of cars of a certain color that we would like to move.
// @param nextLoc The function to use to get the next location of a car, either nextLocUp() for blue car or nextLocRight() for red car.
// @param buffer_loc_next The vector that holds the intermediate variable of the next locations. Supplied to function to avoid repeated construction/destruction.
// @param buffer_movable The vector that holds the intermediate bool variable indicating whether a car is movable or not.
bool moveCars(IntegerMatrix& g, IntegerVector& loc, std::function<int(int,int,int)> nextLoc, IntegerVector& buffer_loc_next, std::vector<bool>& buffer_movable);

// Function to return the location (cyclicly) above the current location
int nextLocUp(int loc, int r, int c);

// Function to return the location (cyclicly) above the current location
int nextLocRight(int loc, int r, int c);

//' Function to get the vector index of the grid right to the current grid.
//'
//' c++ implementation of the idx_right() fucntion
//' @param idx Current locations (vector index in the grid) of cars of a certain color.
//' @param r numbers of rows
//' @param c number of columns
// [[Rcpp::export]]
IntegerVector cidx_right(IntegerVector idx, int r, int c);

//' Function to get the vector index of the grid above the current grid.
//'
//' c++ implementation of the idx_up() fucntion
//' @param idx Current locations (vector index in the grid) of cars of a certain color.
//' @param r numbers of rows
```

```
// [[Rcpp::export]]
IntegerVector cidx_up(IntegerVector idx, int r);
#endif
```

BMLGrid.cpp

```
#include "BMLGrid.h"
using namespace Rcpp;

IntegerMatrix crunBMLGrid1(IntegerMatrix gInput, int numSteps, bool warningGridLock)
{
  IntegerMatrix g = clone(gInput); // Copy the input matrix so that it won't be affected by the undefined behavior of modifying inputs
  int r = g.nrow();
  int c = g.ncol();
  if (0 == r || 0 == c || numSteps < 0)
  {
    //Rcout << "Degenerate BMLGrid object." << std::endl;
    return(g);
  }

  IntegerVector red = locateColor(g, 1);
  IntegerVector blue = locateColor(g, 2);
  IntegerVector white = locateColor(g, 0);

  int buffer_size = ((red.size() > blue.size()) ? red.size() : blue.size());
  IntegerVector buffer_loc_next(buffer_size);
  std::vector<bool> buffer_movable(buffer_size);

  if (red.size() + blue.size() + white.size() != r * c)
  {
    stop("Wrong grid format: values should be 0, 1, 2 only!");
  }
  if (0 == red.size() + blue.size())
  {
    //Rcout << "Degenerate BMLGrid object." << std::endl;
    return(g);
  }

```

```
  bool movable_last = true;
  bool movable;
  for (int step = 0; step < numSteps; step++)
  {
    checkUserInterrupt();
    if (0 == step % 2)
    {
      movable = moveCars(g, blue, nextLocUp, buffer_loc_next, buffer_movable);
    }
    else
    {
      movable = moveCars(g, red, nextLocRight, buffer_loc_next, buffer_movable);
    }
    if (!movable_last && !movable)
    {
      if (warningGridLock) {
        Rcout << "Grid lock detected at step " << step + 1 << std::endl;
      }
      break;
    }
    else
    {
      movable_last = movable;
    }
  }
  return g;
}

IntegerVector locateColor(const IntegerVector& g, int color)
{
  IntegerVector v = seq(0, g.size() - 1);
  return v[g == color];
}

bool moveCars(IntegerMatrix& g, IntegerVector& loc, std::function<int(int,int,int)> nextLoc, IntegerVector& buffer_loc_next, std::vector<bool>& buffer_movable)
{
  int r = g.nrow();
  int c = g.ncol();
  bool movable_any = false;
```

```

  IntegerVector::iterator itr_loc, itr_loc_next;
  std::vector<bool>::iterator itr_movable;

  // The first loop: compute the next locations and identify the movable cars
  itr_loc_next = buffer_loc_next.begin();
  itr_movable = buffer_movable.begin();
  for (itr_loc = loc.begin(); itr_loc != loc.end(); itr_loc++, itr_loc_next++, itr_movable++)
  {
    *itr_loc_next = nextLoc(*itr_loc, r, c);
    if (g[*itr_loc_next] == 0) // The next location is not occupied, move the car
    {
      *itr_movable = true;
      movable_any = true;
    }
    else
    {
      *itr_movable = false;
    }
  }

  // The second loop: move cars according to the result from the first loop
  itr_loc_next = buffer_loc_next.begin();
  itr_movable = buffer_movable.begin();
  for (itr_loc = loc.begin(); itr_loc != loc.end(); itr_loc++, itr_loc_next++, itr_movable++)
  {
    if (*itr_movable) // The next location is not occupied, move the car
    {
      g[*itr_loc_next] = g[*itr_loc];
      g[*itr_loc] = 0;
      *itr_loc = *itr_loc_next;
    }
  }
  return movable_any;
}

int nextLocUp(int loc, int r, int c)
{
  return((loc + 1) % r + (loc / r) * r);
}
```

```
int nextLocRight(int loc, int r, int c)
{
  return((loc + r) % (r * c));
}

IntegerVector cidx_right(IntegerVector idx, int r, int
    c)
{
  IntegerVector idx_next(idx.size());
  IntegerVector::iterator itr_idx, itr_idx_next;
  itr_idx_next = idx_next.begin();
  for (itr_idx = idx.begin(); itr_idx != idx.end();
      itr_idx++, itr_idx_next++)
  {
    *itr_idx_next = nextLocRight(*itr_idx - 1, r, c) +
        1;
  }
  return(idx_next);
}

IntegerVector cidx_up(IntegerVector idx, int r)
{
  IntegerVector idx_next(idx.size());
  IntegerVector::iterator itr_idx, itr_idx_next;
  itr_idx_next = idx_next.begin();
  for (itr_idx = idx.begin(); itr_idx != idx.end();
      itr_idx++, itr_idx_next++)
  {
    *itr_idx_next = nextLocUp(*itr_idx - 1, r, 0) + 1;
  }
  return(idx_next);
}
```