Homework 3

Stephanie Chan

February 11, 2013

Contents

1 Create C Routines

The code for the C Routines are given in the Appendix. There are three main functions: `RupdateBlueCarsCRoutine`, `RupdateRedCarsCRoutine`, `RupdateManyStepsCRoutine`. These use the `.C` function call from R.

2 Test the C Routines

I tested the C functions to check that they would give equivalent results as the methods in R. These tests were done using a randomized grid with random dimensions and random number of cars. I did this 100 times and each time compared the result from C with the result from the previous

function in R. After the first two tries, the two functions returned the same grid.

3 Check Rprofile for runtime

I used the same random function that was used before to check the runtime. I used Rprofile, and a random sampled number of steps to run each time for twenty times. Surprisingly, the R function ran faster than the C function. The C function is named `updateManyStepsWithC` and the R function is `updateManySteps`. The total amount of time spent in the C function was 17.82 while the total amount of time in the optimized R function was 10.08. The total function sampling time is 28. This may be because the `updateManySteps` function is optimized and uses mostly vector addition, subtraction and matching in order to update the cars.

```
> summaryRprof("Rprof.out")
$by.self
                      self.time self.pct total.time total.pct
".C"                      17.82    63.64      17.82     63.64
"match"                    2.10     7.50       2.22      7.93
"updateRed"                1.52     5.43       4.08     14.57
"updateBlue"               1.00     3.57       5.82     20.79
"loadMethod"               0.84     3.00       1.48      5.29
"assign"                   0.64     2.29       0.64      2.29
"standardGeneric"          0.62     2.21       9.88     35.29
"=="                       0.38     1.36       0.38      1.36
"slot<-"                   0.32     1.14       1.24      4.43
"dim"                      0.32     1.14       0.32      1.14
"as.integer"               0.28     1.00       0.28      1.00
"%in%"                     0.22     0.79       2.40      8.57
".getClassFromCache"       0.22     0.79       0.22      0.79
"checkSlotAssignment"      0.18     0.64       0.92      3.29
"el"                       0.18     0.64       0.18      0.64
"*"                        0.14     0.50       0.14      0.50
"+"                        0.14     0.50       0.14      0.50
".identC"                  0.14     0.50       0.14      0.50
"@<-"                      0.12     0.43       1.36      4.86
"getClass"                 0.12     0.43       0.32      1.14
">"                        0.12     0.43       0.12      0.43
```

"c"	0.12	0.43	0.12	0.43
"-"	0.10	0.36	0.10	0.36
"updateManySteps"	0.08	0.29	10.08	36.00
"checkIfCarStuck"	0.06	0.21	2.44	8.71
"elNamed"	0.06	0.21	0.28	1.00
"%%"	0.06	0.21	0.06	0.21
"doTryCatch"	0.04	0.14	0.04	0.14
"bluecars"	0.02	0.07	0.66	2.36
"redcars"	0.02	0.07	0.52	1.86
"rev"	0.02	0.07	0.02	0.07

$by.total

	total.time	total.pct	self.time	self.pct
"FUN"	28.00	100.00	0.00	0.00
"la*pply"	28.00	100.00	0.00	0.00
"sapply"	28.00	100.00	0.00	0.00
".C"	17.82	63.64	17.82	63.64
"updateManyStepsWithC"	17.82	63.64	0.00	0.00
"updateManySteps"	10.08	36.00	0.08	0.29
"standardGeneric"	9.88	35.29	0.62	2.21
"updateBlue"	5.82	20.79	1.00	3.57
"updateRed"	4.08	14.57	1.52	5.43
"checkIfCarStuck"	2.44	8.71	0.06	0.21
"%in%"	2.40	8.57	0.22	0.79
"match"	2.22	7.93	2.10	7.50
"loadMethod"	1.48	5.29	0.84	3.00
"@<-"	1.36	4.86	0.12	0.43
"slot<-"	1.24	4.43	0.32	1.14
"checkSlotAssignment"	0.92	3.29	0.18	0.64
"bluecars"	0.66	2.36	0.02	0.07
"assign"	0.64	2.29	0.64	2.29
"redcars"	0.52	1.86	0.02	0.07
"=="	0.38	1.36	0.38	1.36
"dim"	0.32	1.14	0.32	1.14
"getClass"	0.32	1.14	0.12	0.43
"as.integer"	0.28	1.00	0.28	1.00
"elNamed"	0.28	1.00	0.06	0.21
".getClassFromCache"	0.22	0.79	0.22	0.79
"el"	0.18	0.64	0.18	0.64
"*"	0.14	0.50	0.14	0.50

```
"+"                     0.14      0.50      0.14      0.50
".identC"               0.14      0.50      0.14      0.50
">"                     0.12      0.43      0.12      0.43
"c"                     0.12      0.43      0.12      0.43
"-"                     0.10      0.36      0.10      0.36
"generateBMLgrid"       0.10      0.36      0.00      0.00
"initialize"            0.08      0.29      0.00      0.00
"new"                   0.08      0.29      0.00      0.00
"validObject"           0.08      0.29      0.00      0.00
"%%"                    0.06      0.21      0.06      0.21
"doTryCatch"            0.04      0.14      0.04      0.14
"try"                   0.04      0.14      0.00      0.00
"tryCatch"              0.04      0.14      0.00      0.00
"tryCatchList"          0.04      0.14      0.00      0.00
"tryCatchOne"           0.04      0.14      0.00      0.00
"rev"                   0.02      0.07      0.02      0.07
"getClassDef"           0.02      0.07      0.00      0.00
"is"                    0.02      0.07      0.00      0.00

$sample.interval
[1] 0.02

$sampling.time
[1] 28
```

4 Appendix

4.1 RMoveCars.R

This file contains the three new functions that called the C routines to update the positions of the cars.

```
dyn.load("RMoveCars.so")
is.loaded("RupdateBlueCarsCRoutine")

## function to updateBlue using C function
updateBlueWithC = function(obj) {
    ## get values
    dim = getdim(obj)
    objblue = as.integer(bluecars(obj))
```

```
    objred = as.integer(redcars(obj))
    numBlue = length(objblue)
    numRed = length(objred)

    ## call C
    newBluecars = .C("RupdateBlueCarsCRoutine",
      dim[1],
      dim[2],
      numBlue,
      objblue,
      numRed,
      objred,
      ans = integer(numBlue))$ans

    ## # for debugging purposes
    ## print("oldblue")
    ## print(objblue)
    ## print("newblue")
    ## print(newBluecars)

    ## update object
    obj@bluecars = newBluecars

    return(obj)
}


## function to updateRed using C function
updateRedWithC = function(obj) {
    ## get values
    dim = getdim(obj)
    objblue = as.integer(bluecars(obj))
    objred = as.integer(redcars(obj))
    numBlue = length(objblue)
    numRed = length(objred)

    ## call C
    newRedcars = .C("RupdateRedCarsCRoutine",
      dim[1],
      dim[2],
```

```
      numBlue,
      objblue,
      numRed,
      objred,
      ans = integer(numRed))$ans

    ## # for debugging purposes
    ## print("oldred")
    ## print(objred)
    ## print("newred")
    ## print(newRedcars)

    ## update object
    obj@redcars = newRedcars

    return(obj)
}


## function to updateManySteps using C function
updateManyStepsWithC = function(obj,numSteps) {
    ## get values
    dim = getdim(obj)
    objblue = as.integer(bluecars(obj))
    objred = as.integer(redcars(obj))
    numBlue = length(objblue)
    numRed = length(objred)

    ## update with C
    updatedObj = .C("RupdateManyStepsCRoutine",
      dim[1], dim[2],
      numBlue, objblue,
      numRed, objred,
      numSteps,
      newBlue = integer(numBlue),
      newRed = integer(numRed))

    newBlue = updatedObj[['newBlue']]
    newRed = updatedObj[['newRed']]
```

```
    ## update object
    obj@bluecars = newBlue
    obj@redcars = newRed

    return(obj)
}
```

4.2 RMoveCars.c

This file contains the wrappers for the C routines that were called by R.

```
/*
  This is the file of wrappers for all the functions in order to pass the correct valu
 */
#include "moveCars.h"

void
RupdateBlueCarsCRoutine(int *dimRow,
                        int *dimCol,
                        int *numBluecars,
                        int *bluecars,
                        int *numRedcars,
                        int *redcars,
                        int *ans)
{

    updateBlueCarsCRoutine(*dimRow,*dimCol,
                           *numBluecars,bluecars,
                           *numRedcars,redcars,ans);

}



void
RupdateRedCarsCRoutine(int *dimRow,
                       int *dimCol,
                       int *numBluecars,
                       int *bluecars,
                       int *numRedcars,
```

```
                      int *redcars,
                      int *ans)
{

    updateRedCarsCRoutine(*dimRow,*dimCol,
                          *numBluecars,bluecars,
                          *numRedcars,redcars,ans);

}


void
RupdateManyStepsCRoutine(int *dimRow, int *dimCol,
                         int *numBluecars, int *bluecars,
                         int *numRedcars, int *redcars,
                         int *numMoves,
                         int *newBluecars, int *newRedcars)
{

    updateManyStepsCRoutine(*dimRow, *dimCol,
                            *numBluecars, bluecars,
                            *numRedcars, redcars,
                            *numMoves,
                            newBluecars, newRedcars);

}
```

4.3 moveCars.h

This is the header file that contained the functions in `moveCars.c`.

```
/* C Routine to update the blue cars.
   Called by RupdateCarsCRoutine

   Each blue car moves up by one space.
   INPUTS: int dimRow:       num of rows
           int dimCol:       num of columns
           int numBluecars:  num of bluecars
           int *bluecars:    pointer to the bluecars
```

```
           int numRedcars:   num of redcars
           int *redcars:     pointer to the redcars
           int *newBluecars: pointer to the newset of bluecars


 */
void updateBlueCarsCRoutine(int dimRow,
                            int dimCol,
                            int numBluecars,
                            int *bluecars,
                            int numRedcars,
                            int *redcars,
                            int *newBluecars);

/* C Routine to update the red cars.
   Called by RupdateredCarsCRoutine.

   Each red car moves up by one space.
   INPUTS: int dimRow:       num of rows
           int dimCol:       num of columns
           int numBluecars:  num of bluecars
           int *bluecars:    pointer to the bluecars
           int numRedcars:   num of redcars
           int *redcars:     pointer to the redcars
           int *newRedcars: pointer to the new set of redcars

 */
void updateRedCarsCRoutine(int dimRow,
                           int dimCol,
                           int numBluecars,
                           int *bluecars,
                           int numRedcars,
                           int *redcars,
                           int *newRedcars);

/* C Routine to update red and blue cars for multiple steps
   Called by RupdateManyStepsCarsCRoutine.

   Will alternately move blue and red cars  for numMoves number of moves
   INPUTS: int dimRow:       num of rows
```

```
           int dimCol:       num of columns
           int numBluecars:  num of bluecars
           int *nluecars:    pointer to the bluecars
           int numRedcars:   num of redcars
           int *redcars:     pointer to the redcars
           int numMove:      num of moves to take
           int *newBluecars: pointer to the new set of bluecars
           int *newRedcars:  pointer to the newset of redcars

 */
void
updateManyStepsCRoutine(int dimRow,       int dimCol,
                        int numBluecars,  int *bluecars,
                        int numRedcars,   int *redcars,
                        int numMoves,
                        int *newBluecars, int *newRedcars);



/* C Routine to move single blue car and return new position.
   Called by updateBlueCarsCRoutine.
*/
int moveOneBlueCar(int oldPosition,
                   int dimRow,
                   int numBluecars,
                   int *bluecars,
                   int numRedcars,
                   int *redcars);

/* C Routine to move single red car and return new position.
   Called by updateRedCarsCRoutine.
 */
int moveOneRedCar(int oldPosition,
                  int dimRow,
                  int dimCol,
                  int numBluecars,
                  int *bluecars,
                  int numRedcars,
                  int *redcars);
```

```
/* Check if there is currently a car in the given position.
*/

int checkIsCarPositionTaken(int positionToCheck,
                            int numCarsInArray,
                            int *carArray);
```

4.4 moveCars.c

This is the file that contains the code to move the cars.

```
////////////////////////////////////////////////////////////////////////////
// This file contains the following functions to move the cars
// void updateBlueCarsCRoutine
// void updateRedCarsCRoutine
// int moveOneBlueCar
// int moveOneRedCar
// int checkIsCarPositionTake
// void setArray
// void updateManyStepsCRoutine

#include "moveCars.h"
#include <stdbool.h>

////////////////////////////////////////////////////////////////////////////
// header files for private functions

void
setArray(int arraySize, int *arrayToUpdate, int *arrayWithValues);


////////////////////////////////////////////////////////////////////////////
// Beginning of code


/*  The function will change the values inside newBluecars to reflect
    the changes of the new blue cars.
*/
void
```

```
updateBlueCarsCRoutine(int dimRow,
                       int dimCol,
                       int numBluecars,
                       int *bluecars,  // pass in reference to bluecars
                       int numRedcars,
                       int *redcars,   // pass in reference to redcars
                       int *newBluecars)  // reference to place in memory for blue car

{

    /*  Inital try using just to make sure that values can be moved
    // set the new blue cars to be the old blue car spaces
    for (int i=0; i<numBluecars; i++) {
        newBluecars[i]=bluecars[i];
    }
    */


    for(int i=0; i<numBluecars; i++) {
        newBluecars[i] = moveOneBlueCar(bluecars[i],dimRow,
                                        numBluecars,bluecars,numRedcars,redcars);
    }


}


/* This function will update all the redcars and place the new
   locations of the cars in the array newRedcars.
*/

void
updateRedCarsCRoutine(int dimRow,
                      int dimCol,
                      int numBluecars,
                      int *bluecars,  // pass in reference to bluecars
                      int numRedcars,
                      int *redcars,   // pass in reference to redcars
                      int *newRedcars)  // reference to place in memory for blue cars
```

```
{

    for(int i=0; i<numRedcars; i++) {
        newRedcars[i] = moveOneRedCar(redcars[i],dimRow,dimCol,
                                      numBluecars,bluecars,numRedcars,redcars);
    }


}



/* function to move a single blue car according to the rules.
   This function returns the new location of the car as an integer.
 */
int
moveOneBlueCar(int oldPosition,
               int dimRow,
               int numBluecars,
               int *bluecars,
               int numRedcars,
               int *redcars)
{
    int newPosition;

    // move the car up one space
    newPosition = oldPosition-1;

    // if the car is at the top row, then cycle and move car to next
    if( (newPosition % dimRow) == 0) {
        newPosition = newPosition + dimRow;
    }

    // if the new position is taken by a car, then set it to the old
    // position.

    // check blue spaces
    if( checkIsCarPositionTaken(newPosition, numBluecars, bluecars) ) {
        newPosition = oldPosition;
    }
```

```
    // check red spaces
    if( checkIsCarPositionTaken(newPosition, numRedcars, redcars) ) {
        newPosition = oldPosition;
    }

    return(newPosition);
}



/* function to move a single red car according to the rules.  This
   function returns the new location of the car as an integer.
   This function is called by updateRedCarsCRoutine.
*/
int
moveOneRedCar(int oldPosition,
              int dimRow,
              int dimCol,
              int numBluecars,
              int *bluecars,
              int numRedcars,
              int *redcars)
{
    int newPosition;

    // move the car one space to the right
    newPosition = oldPosition + dimRow;

    // if the car position is at the right-most row, cycle back to the left.
    // do this by subtracting if total position is greater than than
    //the total size of the grid.

    int totalSize = dimRow * dimCol;
    if( newPosition > totalSize ) {
        newPosition = newPosition - totalSize;
    }

    // if the new position is taken by a car, then set it to the old
    // position.
```

```
    // check blue spaces
    if( checkIsCarPositionTaken(newPosition, numBluecars, bluecars) ) {
        newPosition = oldPosition;
    }

    // check red spaces
    if( checkIsCarPositionTaken(newPosition, numRedcars, redcars) ) {
        newPosition = oldPosition;
    }

    return(newPosition);
}


/* checkIsCarPositionTaken:
   This function checks to see if there is currently a car in the position.
*/
int
checkIsCarPositionTaken(int positionToCheck,
                        int numCarsInArray,
                        int *carArray)
{

    /* loop through all the cars in the array, and return TRUE if any
       of them is in the position we want.  If none are taken, then
       return FALSE
    */
    for(int i; i<numCarsInArray; i++) {
        if(positionToCheck==carArray[i]) {
            return(true);
        }
    }

    return(false);

}

/* setArray: This is a helper function that is used to set the values
   from one array to another array.
```

```
*/
void
setArray(int arraySize, int *arrayToUpdate, int *arrayWithValues)
{
    for(int i=0; i < arraySize; i++) {
        arrayToUpdate[i] = arrayWithValues[i];
    }

}




/* updateManyStepsCRoutine: This function will just loop over
   updateBlueCarsCRoutine and updateRedCarsCRoutine for numsteps
   times.  It will start with the bluecars first each time.
*/

void
updateManyStepsCRoutine(int dimRow, int dimCol,
                        int numBluecars, int *bluecars,
                        int numRedcars, int *redcars,
                        int numMoves,
                        int *newBluecars, int *newRedcars)
{

    // At each time step, the previous car locations will be stored at
    // tempBluecars and tempRedcars, and the newly updated locations
    // will be stored at newBluecars and newRedcars

    int tempBluecars[numBluecars], tempRedcars[numRedcars];

    // initialize all the car arrays
    setArray(numBluecars, newBluecars, bluecars);
    setArray(numRedcars, newRedcars, redcars);
    setArray(numBluecars, tempBluecars, bluecars);
    setArray(numRedcars, tempRedcars, redcars);

    // loop over numMoves times
    for(int i=0; i < numMoves; i++) {
```

```
        // if i is even, so it's an odd time step (because i starts at 0 not 1)
        // update the blue cars
        if( (i % 2) == 0 ) {

            updateBlueCarsCRoutine(dimRow, dimCol, numBluecars, tempBluecars,
                                   numRedcars, tempRedcars, newBluecars);

            setArray(numBluecars, tempBluecars, newBluecars);
        }
        // i is not even, update the red cars
        else {

            updateRedCarsCRoutine(dimRow, dimCol, numBluecars, tempBluecars,
                                  numRedcars, tempRedcars, newRedcars);

            setArray(numRedcars, tempRedcars, newRedcars);
        }

    }

    // finish!

}
```

4.5 testC.R

This file contains all the code I used to test the package and the C functions.

```
## source("RMoveCars.R")

library(schanBMLgrid)

#############################################################################
## First set of tests to see if the bluecars and redcars work with the
## C updates.
firstTest = FALSE

if (firstTest) {
    set.seed(100)
```

```
    x = generateBMLgrid(5,5,5,5)
    print(bluecars(x))

    par(mfrow=c(2,2))
    plot(x)

    y1 = updateRedWithC(x)
    print(redcars(y1))
    plot(y1)

    y2 = updateBlueWithC(y1)
    print(bluecars(y2))
    plot(y2)

    y3 = updateRedWithC(y2)
    print(redcars(y3))
    plot(y3)
}


##############################################################################
## Here is a funtion to test if the C routine returns the same values
## as the R function.  It updates blue and red each two times and
## checks if each result is equivalent for R and C
testRandomGrid = function(seed) {

    ## generate a random grid
    ## print(seed)
    set.seed(seed)
    dim = sample(2:100,2)
    size = prod(dim)
    numCars = sample(1:(size/2),2)

    grid = generateBMLgrid(dim[1],dim[2],numCars[1],numCars[2])
    ## summary(grid)

    ## update the grid using R
    gridR1 = updateBlue(grid)
    gridR2 = updateRed(gridR1)
    gridR3 = updateBlue(gridR2)
```

```
    gridR4 = updateRed(gridR3)

    ## update the grid using C
    gridC1 = updateBlueWithC(grid)
    gridC2 = updateRedWithC(gridC1)
    gridC3 = updateBlueWithC(gridC2)
    gridC4 = updateRedWithC(gridC3)

    allgrids = list(grid,gridR1,gridC1,gridR2,gridC2,gridR3,gridC3,gridR4,gridC4)

    ## par(mfrow=c(3,3))

    ## for(i in 1:9) {
    ##     plot(allgrids[[i]],main = i)
    ## }


    ## print(paste("R1=C1",identical(gridR1,gridC1)))
    ## print(paste("R2=C2",identical(gridR2,gridC2)))
    ## print(paste("R3=C3",identical(gridR3,gridC3)))
    ## print(paste("R4=C4",identical(gridR4,gridC4)))

    ## check if the R and C grids are the same
    identical(gridR1,gridC1) & identical(gridR2,gridC2) & identical(gridR3,gridC3) & i

}

testRandomResult =sapply(1:100,testRandomGrid)
## Show that the results from updating with C is equivalent to the
## results from updating in R.


######################################################################
## Here is a function to test if the multistep grid function for C
## works.
testRandomGridManySteps = function(seed) {

    ## generate a random grid
    set.seed(seed)
    numSteps = sample(1:1000,1)
```

```
    ## print(numSteps)

    dim = sample(2:100,2)
    size = prod(dim)
    numCars = sample(1:(size/2),2)

    grid = generateBMLgrid(dim[1],dim[2],numCars[1],numCars[2])

    ## update the grid manySteps
    gridUpdateR = updateManySteps(grid,numSteps)
    gridUpdateC = updateManyStepsWithC(grid,numSteps)

    ## check if the grids are the same
    identical(gridUpdateR,gridUpdateC)
}

testRandomGridManyStepsResult = sapply(1:100,testRandomGridManySteps)

## do profiling
Rprof("Rprof.out")
profileRandomGridManySteps = sapply(1:20,testRandomGridManySteps)
Rprof(NULL)
summaryRprof("Rprof.out")
```