STAT40780 Data Programming with C (online)

Lab Sheet 7 (Solutions)

Dr Marie Galligan

Summer 2015

This week's lab requires you to write some C++ functions making use of the Rcpp and inline R packages. There are 3 parts to this lab sheet.

1 A simple function with Rcpp

Write a C++ function (inlined in R code) with Rcpp that accepts two integer scalar arguments passed from R, and returns the first value, divided by the second value. Compile, link, and load this function into R using `cxxfunction()`, then call the function from R.

A possible solution

Run the following R script.

divideInt

```
1  library(Rcpp)
2  library(inline)
3
4  #the function body is defined in this R character string
5  body_divideInt <- '
6    int val1 = as<int>(x1);
7    int val2= as<int>(x2);
8    return wrap( static_cast<double>(val1) / val2 );
9  '
10
11 #compile, link and load the C++ function
12 divideInt <- cxxfunction(
13   signature(x1 = "integer", x2 = "integer"),
14   body = body_divideInt,
15   plugin = "Rcpp"
16 )
17
18 #call the compiled C++ function from R
19 divideInt(as.integer(4), as.integer(3))
20 divideInt(as.integer(8), as.integer(2))
```

2 Practice with `IntegerVector`

With the help of Rcpp, write a C++ version of the which.min() function in R, that accepts as input an integer vector from R. Note: you can assume for now that no missing values will be passed to this function.

If you are unsure what the `which.min()` function does, you can get help on this in R by running

```
?which.min
```

A possible solution

The function below accepts as input an integer vector x from R, searches for the minimum value, and returns the position of that element in the vector. e.g. if x = (9, 8, 2, 3), the minimum element is 2, and is the 3rd element in the array, hence the function should return 3. If there are ties (more than one minimum), return the index of the first minimum value encountered.

whichMinCpp

```
#A C++ equivalent of the which.min function
library(Rcpp)
library(inline)

#create the function body as an R character string
body_whichMinCpp <- '
  IntegerVector xx(x); //convert to IntegerVector object
  int minval = xx[ 0 ]; //current minimum set to first element
  int min_index = 1; //current minimum is first element

  for( int i = 1 ; i < xx.size(); i++ )
  {
    //check if element indexed by i is
    //less than the current minimum
    if( xx[ i ] < minval )
    {
      //update minimum value
      minval = xx[ i ];
      //update position of minimum element
      min_index = i+1; //because element indexed by i is the (i+1)th element
    }
  } //end of for loop

  return wrap( min_index );
'

#compile, link, load
whichMinCpp <- cxxfunction(
  signature(x = "integer"),
  body = body_whichMinCpp,
  plugin = "Rcpp"
)

#create some data to test it on
x <- sample(1:20, size = 10, replace = TRUE)

#call the compiled C++ function
whichMinCpp( x )
```

3 Bubblesort function

The C++ code inlined in the R script shown below is the implementation of the bubblesort algorithm from Lesson 4.2, which sorts an `IntegerVector` passed from R. Improve this function as follows:

1. On pass k over the vector, the top k-1 elements are already sorted. Modify the bubblesort function so that on pass k, it does not pass over the top k-1 elements.

2. A vector with n elements might be sorted in less than n-1 passes. Improve the bubblesort algorithm so that it does not continue to pass over a vector that is already sorted.

Benchmark your modified version against this one, to compare performance. Note that it is possible to optimize the bubblesort function further. How?

Bubblesort algorithm

```
#body of the C++ bubblesort function
#stored in an R character string
body_bubblesort2 <- '
 IntegerVector xx = clone(x); //use of clone()
 int n = xx.size(); //no. of elements
 int temp; //temporary storage of swap value
 for( int k = 1; k <= n - 1; k++ ){ //for pass k
  //loop over pairs of elements
  for( int i = 0; i < n - 1; i ++ ){
    if( xx[ i ] > xx[ i+1 ] ){
      temp = xx[ i + 1 ];
      xx[ i + 1 ] = xx[ i ];
      xx[ i ] = temp;
    } //end of if
  } //end of loop over array pairs
 } //end of loop over passes
 return(wrap(xx));
'

#compile, link, load
bubblesort2 <- cxxfunction(signature( x = "integer" ),
                           body = body_bubblesort2,
                           plugin = "Rcpp")

#create an R integer vector to input to bubblesort
x2 <- as.integer( sample(1:100, size = 100, replace = FALSE) )

#call
bubblesort2(x2) #returns sorted x2
x2 #original x2 is not sorted
```

A possible solution

A possible solution is shown here, where modifications 1 and 2 have been made. On Blackboard, I'm uploading some notes on the first modification.

Modified bubblesort algorithm

```
#MODIFICATION 1: On pass k, this algorithm does not pass over the top
#k-1 elements of  the vector
#MODIFICATION 2: the algorithm has been modified so that it stops passing
#over the vector if no swaps were made during the previous pass
body_bubblesortOpt2 <- '

 IntegerVector xx = clone(x); //use of clone()
 int n = xx.size(); //no. of elements
 int temp; //temporary storage of swap value
 int wasSwap = 1; //to test whether a swap was made on pass k
 //set wasSwap to 0? if no swap was made, 1 otherwise

 //for pass k, continue if k <= n-1
 //AND if a swap was made on previous pass
 for( int k = 1; (k <= n - 1) && (wasSwap == 1 ); k++ ){
 wasSwap = 0; //so far, no swap has been made on this pass, so set to zero

  //loop over pairs of elements
  //modified i < n-1 to i < n-k to
  //avoid passing over elements that
  //are already sorted
  for( int i = 0; i < n - k ; i ++ ){
   if( xx[ i ] > xx[ i+1 ] ){
    temp = xx[ i + 1 ];
    xx[ i + 1 ] = xx[ i ];
    xx[ i ] = temp;
    wasSwap = 1;  // indicate that a swap was made
   }

  } //end of loop over array pairs
 } //end of loop over passes
 return(wrap(xx));
'

#compile, link, load
bubblesortOpt2 <- cxxfunction(signature( x = "integer" ),
                              body = body_bubblesortOpt2,
                              plugin = "Rcpp")

#create some data
x <- as.integer( sample(1:100, size = 100, replace = FALSE) )
#call the modified bubblesort function
sorted_x <- bubblesortOpt2(x) #sorts x
sorted_x #sorted_x is sorted
x #x is not sorted
```

Benchmarking bubblesort

```

x <- as.integer( sample(1:100, size = 100, replace = FALSE) )
benchmark(bubblesort2(x),  bubblesortOpt2(x),
          order = "relative",
          replications = 1000)
```

Optimizing bubblesort further: On pass k over the vector, **at least** the top k-1 elements will already be sorted - but it is possible that more than k-1 top elements will already be sorted. After a pass over the array, all elements after the last swap during that pass are already sorted. So, if after a pass over a vector with 10 elements, the 3th and 4th elements were swapped, but none above this were swapped, then there would be no need to check the top 6 elements on subsequent passes over the array.