Implementation of Sorting Model Algorithms in R and C++

Jon Minton

3 Feb 2014

# Purpose of report

This short report describes how Rcpp has been used to develop structured code which operates sorting model algorithms in C++, but which can be operated ‘at arm’s length’, from within R, and without requiring any C++ expertise. The code is currently work-in-progress, but has been structured and developed in such a way that it should be relatively scalable, easy to understand, easy to debug, and easy to maintain.

# Background

Sorting models are in important new development in spatial economic and econometrics. Applying them to real-world data can involve computationally intensive calculations. Currently, bespoke sorting model algorithms have been written by Prof. Chris Timmins in FORTRAN.

FORTAN is a low level statistical language of similar age to C. As a low level language it runs quickly, but also requires specialist programming expertise which few economists and social scientists possess. This severely limits the potential uptake and popularity of econometric methods which are currently only implemented using this language.

R is a high level statistical programming language whose popularity has been increasing almost exponentially for many years. It is open source, and thousands of libraries of functions have been created by statisticians and programmers in order to make implementing new statistical and econometric methods very straightforward.

R is an interpreted rather than compiled language. It encourages interaction between user and software, allowing easy testing and iterative development of code. However, this interaction comes at the expense of processing speed, meaning for large scale computationally intensive procedures it is not optimal.

R is a variant of S, and S is written in the low level programming language C. Many computationally intensive functions, such as numerical optimisation algorithms, within R work by calling C code. The process of developing C based functions and operating them through R requires a high level of expertise with C, R and the methods used to link between them. C code is a relatively ‘flat’ rather than structured language, meaning that understanding the logic and processes implemented in the language can be very difficult.

C++, also known as ‘C with classes’, is an object-oriented programming (OOP) language, which allows code to be written in a way that is more modular and accessible than using C alone. This more modular programming system makes code easier to develop, maintain, and link with other objects and software platforms without requiring extensive re-writing.

Within C++, **objects** are created as instantiations (realisations) of **classes**. Classes are defined by the programmer to possess other objects known as **fields**, and functions which operate on these objects known as **methods**. Both fields and methods can be defined as either **private** or **public.** A public field/method can be accessed and acted on by something **external** to the class, but a private field/member cannot. By restricting access to a class’ fields and members, the risk of one section of code having an unintended side effect on another section of code is greatly reduced.

Rcpp is a series of tools developed which make it much easier for R programmers to make use of the speed benefits of C++. It includes both code written in R and code written in C++. The R code library makes it more convenient to pass objects from R to C++, and makes it possible to access functions and objects written in C++ as if they are R objects. The C++ library makes it easier to access and operate on objects passed to it from R, and also defines a number of classes in C++ which operate in a similar way to objects in R, meaning that the C++ implementation of an algorithm can look very similar to its R implementation, while still running on average between one and three orders of magnitude faster.

# The need for structured data management

Sorting models require a large number of inputs. These inputs include **data** and **model parameters**. These input objects are of different data types and dimensions. For example, data includes the proportion of stayers, and population counts by areal unit over two or more time periods. The proportion of stayers is a positive non integer. The population counts could either be represented as a N by 2 matrix, or as two N length vectors, in either case containing integers.

For handling ‘ragged’ data structures like this, R uses **list** objects. List objects are vectors which can be of any length. Each element of the vector can contain another object, which can itself be of any length or structure. List objects can even contain as list objects other lists. Each element in a list object can be located either by location, or by name. (For example, if the second element is named “B”, then the object it contains can be accessed as either x[[“B”]], or as x[[2]].)

The use of list objects makes the management of data much clearer and easier to understand. The structure of the list object I currently use as input to the C++ sorting model code is shown diagrammatically in Figure 1. Its implementation in R is shown in Table 1.

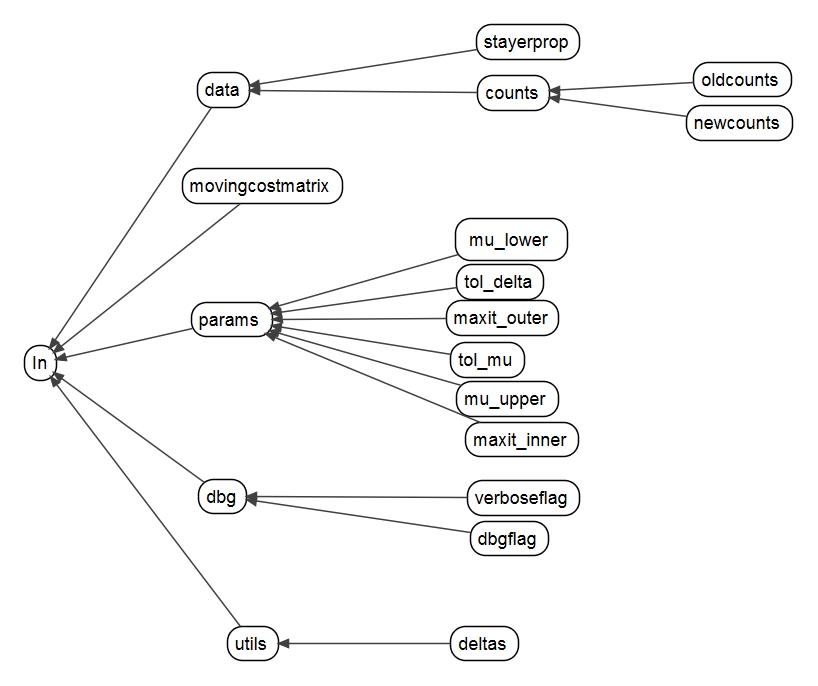


Figure 1 Graphical representation of Input data structure

|  |
| --- |
| In <- list(  data = list(  stayerprop= 0.4174,  counts = list(  oldcounts=countsold,  newcounts=countsnew  )  ),  movingcostmatrix = movingcostmatrix,  utils = list(  deltas = rep(0, length(countsold))  ),  params = list(  tol\_delta = 10^-2,  tol\_mu = 10^-2,  mu\_upper = 0.10,  mu\_lower = -0.15,  maxit\_outer = 10,  maxit\_inner = 100  ),  dbg = list(  dbgflag= T,  verboseflag=T  )  ) |

Table Implementation of In data structure in R

# Defining a class within C++

The basic template for a class definition within C++ is shown below. The name of the class follows the keyword **class.** The names and broad definitions of fields and methods within the class are defined within a pair of curly brackets { } which follows. (The characters // are used to indicate comments: symbols which appear to the right of these characters are ignored by the compiler.)

|  |
| --- |
| class Name\_of\_class{  //Private members and fields  public:  //Public members and fields  } |

The contents of a field/method can be specified either inside the class, or elsewhere. If it is defined elsewhere, then the full definition needs to use the scope operator :: in order to declare its membership to the class. For example:

|  |
| --- |
| class class\_name{  //Private members and fields  public:  int class\_method(int input);  }  int class\_name::class\_method(int input){  return (2 \* input);  } |

Classes within C++ contain **constructor** functions, which are methods which define the properties of objects of their class when they are first created. Similarly, **destructor** functions are implicitly called when an object of a given class is destroyed. The constructor function is the function that a call from R will run implicitly when it instantiates a class object. Constructor functions have the same name as the class, and do not specify a return value. Destructor functions also have the same name as the class, but use a ~ prefix. For example:

|  |
| --- |
| class class\_name{  //Private members and fields  public:  class\_name();  ~class\_name();  }  class\_name::class\_name(){  //This is the constructor (called on creation)  }  Class\_name::~class\_name()  {  //This is the destructor (called on exit)  } |

Within my implementation of the sorting model, a constructor function is used to populate private fields of the class with inputs from the R In object. A much simplified example of this is as follows:

|  |
| --- |
| class class\_name{  int private\_field;  public:  class\_name(int input);  }  class\_name::class\_name(int input){  private\_field = input;  } |

Although R cannot access private\_field directly, it can populate it by instantiating an object of the class class\_name. The input to the constructor function input is then passed to private\_field. It is this form of action-at-a-distance that allows R to use C++ classes as a black box, and with reduced danger of unintended side effects.

# Accessing C++ classes, and selected public fields and methods, within R

Rcpp provides a number of different approaches to allowing C++ functions, classes and objects to be used within an R session, ‘as if’ they are R objects and functions. The approach I have adopted uses Rcpp **modules**, which require a few lines of additional code within the C++ script, but almost no additional code within R.

A very simple example of the code required to access C++ classes within R is shown below. First the C++ code:

|  |
| --- |
| class my\_class{  int private\_field;  public:  class\_name(int input);  int return();  }  my\_class::my\_class(int input){  private\_field = 2 \* input;  }  int my\_class::return(){  return private\_field;  }  RCPP\_MODULE(ExposedClass){  using namespace Rcpp;    class\_<c> ("my\_class")    .constructor<int>()  .method("return", &my\_class::return)  ;  } |

To create a C++ object of class my\_class in R, and then return the private field from it, only the following would be required:

|  |
| --- |
| as\_if\_R\_object <- new(“my\_class”, 2)  as\_if\_R\_object$return() |

The first line creates a C++ object, of C++ class my\_class, which is called as\_if\_R\_object. The second argument of the new() function passes the value ‘2’ to the constructor function within my\_class. Rcpp correctly interprets this object as an integer, and so an appropriate argument for my\_class constructor. The my\_class constructor passes twice this value to private\_field, and so the value of private\_field is 4.

The second line accesses the return method of the my\_class object as\_if\_R\_object. The $ symbol is used within R much as the . symbol is used in many other languages, to indicate a subset, method or field of an object. (The line .method(“return”, &my\_class::return) in the C++ part of the declaration exposes the method to R.)

The result of as\_if\_R\_object$return() is ‘4’.

# General description of approach used to implement the Sorting Model algorithm using Rcpp

A class, Contraction, was implemented within C++ to handle the computationally intensive contraction mapping algorithms, in which mu (marginal disutility per unit of moving cost) is estimated using the method of bisection, and deltas (the ‘area specific utilities’) are estimated using a contraction mapping approach.

The constructor function takes as its input the R list object In described above, which is first converted into an equivalent Rcpp object with the same structure. Elements of this list object are then loaded into lower-level C++ objects, such as doubles and integers, which are private members of the Contraction class. A method of Contraction, called RunContraction, is exposed to R through the Rcpp module, which initiates the contraction mapping process. It repeatedly calls a non-exposed method called RunOuter, which performs the method of bisection on mu. RunOuter itself calls another non-exposed method, RunInner, which performs the contraction method on each of the delta estimates.

As all methods are members of the same class, they have read and write access to the same private members, including the double scalar mu and the double vector delta, which is repeatedly updated until the tolerance conditions for delta (specified by tol\_delta) and mu (tol\_mu) are met. RunInner and RunOuter also have access to a number of helper functions which perform different parts of the calculations involved in the contraction mapping process.

Once the processes set in motion by calling run() have completed, the final estimates of mu and delta can be accessed in R by a call to an exposed function ExtractEstimates, which is mapped to the term extractests in R.

Therefore, from the R user’s perspective, the only commands required are as follows:

|  |
| --- |
| SortingObject <- new(“Contraction”, In)  SortingObject$run()  SortingObject$extractests() |

# Next steps

The coding structures and conventions described above have only very recently been developed. The contraction algorithms have not been fully tested and debugged, but the specification of class membership and structure within C++, and interfaces between R and C++, are all operational. From this structural base the modularisation and testing of code, and implementation of alternative versions of the contraction mapping algorithms and sorting models, should be *comparatively* straightforward, because the structure developed above allows for highly modularised testing and development of the C++ class. Methods which are computationally or conceptually very different could be implemented in very similar ways from the R user’s perspective. For example, the run() method could take a flag or string as its argument, which would determine which of a choice of contraction algorithms would be implemented within C++. From the R user’s perspective, however, the only difference between using the two methods would be very slight, as suggested below:

|  |
| --- |
| SortingObA <- new(“Contraction”, In)  SortingObA$run(“MethodA”)  estsA <- SortingObA$extractests()  SortingObB <- new(Contraction”, In)  SortingObB$run(“MethodB”)  estsB <- SortingObB$extractests()  estsB$mu – estsA$mu # compare difference in mu estimates  estsB$deltas – estsA$deltas # compare difference in delta estimates |