Biostatistics 615/815 Lecture 12: Interfacing C++ and R

Hyun Min Kang

October 11th, 2012

Recommended Skill Sets for Students

1. One or more of the high-level statistical language for fast and flexible implementation
   - R
   - SAS
   - Matlab
2. One or more of the scripting language for data pre/post processing
   - perl
   - python
   - ruby
   - php
   - sed/awk
   - bash/csh
3. One or more low-level languages for efficient computation
   - C/C++
   - Java

Factors to consider when developing a new method

- Personal software : Tradeoff between..
  - YOUR time cost for implementation and debugging
  - YOUR time cost for running the analysis (including number of repetitions)
  - COMPUTATIONAL cost for running the analysis
- Public software : Additional tradeoff between...
  - All three types of costs above
  - YOUR additional time cost for making your method available to others
  - YOUR time saving for letting others run the analysis on your behalf
  - Additional credit for having exposure of your method to others

Using high-level languages (such as R)

Benefits

- Implementation cost is usally small, and easy to modify
- Many built-in and third-party utilities reduces implementation burden
  - Most of the hypothesis testing procedure
  - `lm` and `glm` routines for fitting to (generalized) linear models
  - Plotting routines to visualize your outcomes
  - And many other third-party routines
- Good fit for running quick and non-repetitive jobs

Drawbacks

- R is not effcient in I/O and memory management
- Complex routines involving loops are extremely slow
- Likely slower and less user-friendly than C/C++ implementation

Interfacing your C++ code with R

- Use R for input and output handling (possibly including data visualization)
- For routines requiring computational efficiency, use C++ routines
- Load the C++ routine as a dynamically-linked library and use them inside C
- Fortran language interface is also available (will not be discussed here)

R 101

Install and run R

- Install/Download R package at *http://www.r-project.org/*
- Run R (64-bit version if available)
- Have a separate terminal available for compiling your code

Very basic commands

```
> getwd()  ## print current working directory
[1] "/Users/myid"
> setwd('/absolute/path/to/where/i/want/to/be/at'); ## move your current working c
> getwd()  ## print the new working directory
/absolute/path/to/where/i/wanted/to/be/at
> x <- c(1,2,3,4,5,6)   ## a vector of size 6
> y <- 1:6              ## x and y are identical
> z <- rep(1,6)         ## vector of size 6, filled with 1
> A <- matrix(1:6,3,2)  ## 3 by 2 matrix, first row is 1,3,5
> B <- matrix(1,3,2)    ## 3 by 2 matrix filled with 1
```

Using R - vectors and matrices

```
> u <- 1:10
> v <- rep(2,10)
> v*u      ## element-wise multiplication
 [1]  2  4  6  8 10 12 14 16 18 20
> v %*% u  ## dot product, resulting in 1x1 matrix
     [,1]
[1,]  110
> A <- matrix(1:10,5,2)
> B <- matrix(2,5,2)
> A*B      ## element-wise multiplication
     [,1] [,2]
[1,]    2   12
[2,]    4   14
[3,]    6   16
[4,]    8   18
[5,]   10   20
> t(A) %*% B  ## A'B
     [,1] [,2]
[1,]   30   30
[2,]   80   80
```

Using R - Running Fisher's exact test

```
> fisher.test( matrix(c(2,7,8,2),2,2) )

Fishers Exact Test for Count Data

data:  matrix(c(2, 7, 8, 2), 2, 2)
p-value = 0.02301
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.004668988 0.895792956
sample estimates:
odds ratio
0.08586235
```

Using R

Sorting

```
> x <- c(9,1,8,3,4)
> sort(x)
[1] 1 3 4 8 9
> order(x)
[1] 2 4 5 3 1
> rank(x)
[1] 5 1 4 2 3
```

Summary Statistics

```
> x <- c(9,1,8,3,4)
> mean(x)
[1] 5
> sd(x)
[1] 3.391165
> var(x)
[1] 11.5
```

Using R

Statistical Distributions

```
> pnorm(-2.57)
[1] 0.005084926
> pnorm(2.57)
[1] 0.994915
> pnorm(2.57,lower.tail=FALSE)
[1] 0.005084926
> pchisq(3.84,1,lower.tail=FALSE)
[1] 0.9499565
```

Using R

Row-wise or Column-wise statistics

```
> A <- matrix(1:10,2,5)
> rowMeans(A)
[1] 5 6
> colMeans(A)
[1] 1.5 3.5 5.5 7.5 9.5
> A <- matrix(1:10,2,5)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> rowMeans(A)
[1] 5 6
> colMeans(A)
[1] 1.5 3.5 5.5 7.5 9.5
> apply(A,1,mean)
[1] 5 6
> apply(A,2,mean)
[1] 1.5 3.5 5.5 7.5 9.5
> apply(A,1,sd)
[1] 3.162278 3.162278
```

Interfacing C++ code with R

hello.cpp

```
#include <iostream>  // May include C++ routines including STL
extern "C" {         // R interface part should be written in C-style
  void hello () {    // function name that R can load
    std::cout << "Hello, R" << std::endl; // print out message
  }
}
```

Compile (output is dependent on the platform)

```
$ R CMD SHLIB hello.cpp
R CMD SHLIB hello.cpp -o hello.so
g++ -I/usr/local/R-2.15/lib64/R/include -DNDEBUG  -I/usr/local/include
-fpic  -g -O2  -c hello.cpp -o hello.o
g++ -shared -L/usr/local/lib64 -o hello.so hello.o
```

Interfacing C++ code with R

hello.R

```
dyn.load(paste("hello", .Platform$dynlib.ext, sep=""))
## wrapper function to call the C/C++ function
hello <- function() {
  .C("hello")
}
hello()
```

Running hello.R

```
Hello, R
list()
```

Argument passing

square.cpp

```
extern "C" {
  void square (double* a, double* out) {
    *out = (*a) * (*a);
  }
}
```

Arguments must be passed as pointers, regardless whether it contains array values or not

square.R

```
dyn.load(paste("ex2", .Platform$dynlib.ext, sep=""))
square <- function(a) { ## a is input, out is output
  return(.C("square",as.double(a),out=double(1))$out)
}
square(1.414)
[1] 1.999396
```

Passing vector or matrix as argument

square2.cpp

```
extern "C" {
  void square2 (double* a, int* na, double* out) {
    for(int i=0; i < *na; ++i) {
      out[i] = a[i] * a[i];
    }
  }
}
```

square2.R

```
dyn.load(paste("square2", .Platform$dynlib.ext, sep=""))
square2 <- function(a) {
  n <- as.integer(length(a))
  r <- .C("square2",as.double(a),n,out=double(n))$out
  if ( is.matrix(a) ) { return (matrix(r,nrow(a),ncol(a))); }
  else { return (r); }
}
```

Argument passing

Running Example (after compiling)

```
> source('square2.R')
> square2(10)  ## takes a single input
[1] 100
> square2(c(10,20,30)) ## takes a vector as input
[1] 100 400 900
> square2(matrix(1:6,3,2)) ## takes a matrix as input
     [,1] [,2]
[1,]    1   16
[2,]    4   25
[3,]    9   36
```

Using SEXP - More flexible but complex approach

```
#include <R.h>
#include <Rinternals.h>
#include <Rdefines.h>

extern "C" {
  SEXP square3(SEXP in) { // Use SEXP data type for interfacing
    int nr = 0, nc = 0;
    SEXP out;  // output variable (matrix or vector) to return
    if ( isMatrix(in) ) {  // isMatrix can take SEXP as argument
      int *dimX = INTEGER(coerceVector(getAttrib(in,R_DimSymbol),INTSXP));
      nr = dimX[0]; nc = dimX[1];  // obtain matrix dimension
      PROTECT( out = allocMatrix(REALSXP, nr, nc) ); // allocate memory in R
    }
    else if ( isVector(in) ) {
      nr = length(in);
      nc = 1;
      PROTECT( out = allocVector(REALSXP, nr) );
    }
```

Using SEXP - More flexible but complex approach

```
    else error("Could not parse the input");
    PROTECT(in = AS_NUMERIC(in)); // Use PROTECT to bind R/C++ memory space
    double* p_in = NUMERIC_POINTER(in);
    for(int i=0; i < nr*nc; ++i) {
      REAL(out)[i] = p_in[i]*p_in[i];  // accessing memory
    }
    UNPROTECT(2);  // Release PROTECT before finishing
    return (out);
  }
}
```

Running Examples

```
> dyn.load(paste("lib/square3", .Platform$dynlib.ext, sep=""))

> .Call("square3",matrix(1:10,5,2))
     [,1] [,2]
[1,]    1   36
[2,]    4   49
[3,]    9   64
[4,]   16   81
[5,]   25  100

> .Call("square3",1:10)
 [1]   1   4   9  16  25  36  49  64  81 100

> .Call("square3",1)
[1] 1
```

Calculating cumulative sum of an array

cumsum.R

```
cumsum.R <- function(a) {
  res <- a   ## copy the original matrix
  n <- length(a)
  for (i in 2:n) {
    res[i] = res[i-1] + res[i]  ## get cumulative sum
  }
  return (res)
}
```

Running Example

```
> system.time(cumsum.R(as.double(1:1000000)))
   user  system elapsed
  3.548   0.016   3.563
```

But built-in cumsum function is much faster

Running with built-in cumsum function

```
> system.time(cumsum(as.double(1:1000000)))
  user  system elapsed
  0.017   0.007   0.024
```

What's inside in the cumsum function?

```
> cumsum
function (x)  .Primitive("cumsum")
```

- `.Primitive` indicates that the function is defined in R library
- Uses internal implementation for the sake of efficiency

Making faster cumsum function

cumsumC.cpp

```
#include <R.h>
#include <Rinternals.h>
#include <Rdefines.h>
extern "C" {
  SEXP cumsumC(SEXP in) {
    int n = length(in);
    int sum = 0, csum = 0;
    SEXP out;
    PROTECT(in = AS_NUMERIC(in));
    PROTECT(out = allocVector(REALSXP, n));
    double* p_in = NUMERIC_POINTER(in);
    REAL(out)[0] = p_in[0];
    for(int i=1; i < n; ++i)
      REAL(out)[i] = REAL(out)[i-1] + p_in[i];
    UNPROTECT(2);
    return (out);
  }
}
```

Running cumsumC

```
> dyn.load(paste("lib/cumsumC", .Platform$dynlib.ext, sep=""))

> system.time(cumsum.R(as.double(1:1000000)))
  user  system elapsed
  3.548   0.016   3.563

> system.time(cumsum(as.double(1:1000000)))
   user  system elapsed
  0.017   0.007   0.024

> system.time(.Call("cumsumC",as.double(1:1000000)))
   user  system elapsed
  0.016   0.010   0.026
```

Many built-in routines use C implementation inside

```
> fisher.test
function (x, y = NULL, workspace = 2e+05, hybrid = FALSE, control = list(),
    or = 1, alternative = "two.sided", conf.int = TRUE, conf.level = 0.95,
    simulate.p.value = FALSE, B = 2000)
{
    DNAME <- deparse(substitute(x))
    METHOD <- "Fisher's Exact Test for Count Data"
    ## skipping some lines...
            STATISTIC <- -sum(lfactorial(x))
            tmp <- .C(C_fisher_sim, as.integer(nr), as.integer(nc),
                as.integer(sr), as.integer(sc), as.integer(n),
                as.integer(B), integer(nr * nc), double(n + 1),
                integer(nc), results = double(B), PACKAGE = "stats")$results
            almost.1 <- 1 + 64 * .Machine$double.eps
            PVAL <- (1 + sum(tmp <= STATISTIC/almost.1))/(B +
                1)
    ## skipping the rest of them
```

Reading matrix from a file

```
#include "Matrix615.h"  // same Matrix615.h used for the HW3

#include <R.h>
#include <Rinternals.h>
#include <Rdefines.h>

extern "C" {
  char* strAllocCopy(SEXP s, int idx = 0) {
    PROTECT(s = AS_CHARACTER(s));
    char* p = R_alloc(strlen(CHAR(STRING_ELT(s,idx))), sizeof(char));
    strcpy(p, CHAR(STRING_ELT(s, idx)));
    UNPROTECT(1);
    return(p);
  }
```

Reading matrix from a file

```
  SEXP readMatrix(SEXP fname) {
    const char* sfname = strAllocCopy(fname);
    SEXP out;
    Matrix615<double> m(sfname);
    int nr = m.rowNums();
    int nc = m.colNums();

    PROTECT( out = allocMatrix(REALSXP, nr, nc) );
    for(int i=0,k=0; i < nc; ++i) {
      for(int j=0; j < nr; ++j, ++k) {
        REAL(out)[k] = m.data[j][i];
      }
    }

    UNPROTECT(1);
    return (out);
  }
}
```

Running and compiling `readMatrix.cpp`

Compiling is a bit tricky

```
$ setenv PKG_CPPFLAGS "-I. -I ~hmkang/Public/include"  ## to include boost library
$ R CMD SHLIB readMatrix.cpp
  g++ -I/usr/local/R-2.15/lib64/R/include -DNDEBUG -I.
     -I ~hmkang/Public/include -I/usr/local/include  -fpic  -g -O2
     -c readMatrix.cpp -o readMatrix.o
  g++ -shared -L/usr/local/lib64 -o readMatrix.so readMatrix.o
```

Running Examples

```
> dyn.load(paste("lib/readMatrix", .Platform$dynlib.ext, sep=""))
> fn <- "m1000x1000.txt"  ## a 1000 by 1000 matrix
> print(system.time(M <- .Call("readMatrix",fn)))
   user  system elapsed
  1.487   0.075   1.562
> print(system.time(N <- as.matrix(read.table(fn))))
   user  system elapsed
  9.256   0.067   9.323
```

Programming with Matrix

Why Matrix matters?

- Many statistical models can be well represented as matrix operations
  - Linear regression
  - Logistic regression
  - Mixed models
- Efficient matrix computation can make difference in the practicality of a statistical method
- Understanding C++ implementation of matrix operation can expedite the efficiency by orders of magnitude

Ways for Matrix programming in C++

- Implementing Matrix libraries on your own
  - Implementation can well fit to specific need
  - Need to pay for implementation overhead
  - Computational efficiency may not be excellent for large matrices
- Using BLAS/LAPACK library
  - Low-level Fortran/C API
  - ATLAS implementation for gcc, MKL library for intel compiler (with multithread support)
  - Used in many statistical packages including R
  - Not user-friendly interface use.
  - `boost` supports C++ interface for BLAS
- Using a third-party library, `Eigen` package
  - A convenient C++ interface
  - Reasonably fast performance
  - Supports most functions BLAS/LAPACK provides

Using a third party library

Downloading and installing `Eigen` package

- Download at *http://eigen.tuxfamily.org/*
- To install - just uncompress it, no need to build

Using `Eigen` package

- Add `-I ~hmkang/Public/include` option (or include directory containing `Eigen/`) when compile
- No need to install separate library. Including header files is sufficient

Example usages of `Eigen` library

```cpp
#include <iostream>
#include <Eigen/Dense> // For non-sparse matrix
using namespace Eigen; // avoid using Eigen::
int main()
{
  Matrix2d a;          // 2x2 matrix type is defined for convenience
  a << 1, 2,
       3, 4;
  MatrixXd b(2,2);     // but you can define the type from arbitrary-size matrix
  b << 2, 3,
       1, 4;
  std::cout << "a + b =\n" << a + b << std::endl;  // matrix addition
  std::cout << "a - b =\n" << a - b << std::endl;  // matrix subtraction
  std::cout << "Doing a += b;" << std::endl;
  a += b;
  std::cout << "Now a =\n" << a << std::endl;
  Vector3d v(1,2,3);                               // vector operations
  Vector3d w(1,0,0);
  std::cout << "-v + w - v =\n" << -v + w - v << std::endl;
}
```

More examples

```cpp
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;
int main()
{
  Matrix2d mat;              // 2*2 matrix
  mat << 1, 2,
         3, 4;
  Vector2d u(-1,1), v(2,0);  // 2D vector
  std::cout << "Here is mat*mat:\n" << mat*mat << std::endl;
  std::cout << "Here is mat*u:\n" << mat*u << std::endl;
  std::cout << "Here is u^T*mat:\n" << u.transpose()*mat << std::endl;
  std::cout << "Here is u^T*v:\n" << u.transpose()*v << std::endl;
  std::cout << "Here is u*v^T:\n" << u*v.transpose() << std::endl;
  std::cout << "Let's multiply mat by itself" << std::endl;
  mat = mat*mat;
  std::cout << "Now mat is mat:\n" << mat << std::endl;
  return 0;
}
```

More examples

```
#include <Eigen/Dense>
#include <iostream>
using namespace Eigen;
int main()
{
  MatrixXd m(2,2), n(2,2);
  MatrixXd result(2,2);
  m << 1,2,
       3,4;
  n << 5,6,7,8;
  result = m * n;
  std::cout << "-- Matrix m*n: --" << std::endl << result << std::endl << std::endl;
  result = m.array() * n.array();
  std::cout << "-- Array m*n: --" << std::endl << result << std::endl << std::endl;
  result = m.cwiseProduct(n);
  std::cout << "-- With cwiseProduct: --" << std::endl << result << std::endl << std::endl;
  result = (m.array() + 4).matrix() * m;
  std::cout << "-- (m+4)*m: --" << std::endl << result << std::endl << std::endl;
  return 0;
}
```

Time complexity of matrix computation

Square matrix multiplication / inversion

- Naive algorithm : $O(n^3)$
- Strassen algorithm : $O(n^{2.807})$
- Coppersmith-Winograd algorithm : $O(n^{2.376})$ (with very large constant factor)

Determinant

- Laplace expansion : $O(n!)$
- LU decomposition : $O(n^3)$
- Bareiss algorithm : $O(n^3)$
- Fast matrix multiplication algorithm : $O(n^{2.376})$

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$
- If the order is $(((\mathbf{u}'(AB))\mathbf{v})$
  - $O(n^3) + O(n^2) + O(n)$ operations
  - $O(n^2)$ overall
- If the order is $(((\mathbf{u}'A)B)\mathbf{v})$
  - Two $O(n^2)$ operations and one $O(n)$ operation
  - $O(n^2)$ overall

Quadratic multiplication

Same time complexity, but one is slightly more efficient

- Computing $\mathbf{x}'A\mathbf{y}$.
- $O(n^2) + O(n)$ if ordered as $(\mathbf{x}'A)\mathbf{y}$.
- Can be simplified as $\sum_i \sum_j x_i A_{ij} y_j$

A symmetric case

- Computing $\mathbf{x}'A\mathbf{x}$ where $A = LL'$
- $\mathbf{u} = L'\mathbf{x}$ can be computed more efficiently than $A\mathbf{x}$.
- $\mathbf{x}'A\mathbf{x} = \mathbf{u}'\mathbf{u}$

Today

R/C++ Interface

- Combining C++ code base with R extension
- C++ implementation more efficiently handles loops and complex algorithms than R
- R is efficient in matrix operation and convenient in data visualization and statistical tools
- R/C++ interface increases your flexibility and efficiency at the same time.

Matrix Library

- `Eigen` library for convenient use and robust performance
- Time complexity of matrix operations