Biostatistics 615/815 Lecture 23: Using C++ code in R

Hyun Min Kang

December 6th, 2011

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 d
> getwd()  ## print the new working directory
/absolute/path/to/where/i/wanted/to/be/at
> x <- rnorm(1000,5,1);  ## generate 1000 random normal variables from N(5,1)
> y <- runif(1000,0,1);  ## generate 1000 uniform random variables from (0,1)
> Z <- matrix(rnorm(1000,0,1),50,20); ## create a 50 * 20 random matrix
> r <- as.integer(runif(1000,0,10)); ## 1000 uniform integer between 0 and 9
```

Interfacing C++ code with R

ex1.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
  }
}
```

ex1.R

```
## loadex1.so in UNIX/MacOS and load ex1.dll in Windows
dyn.load(paste("ex1", .Platform$dynlib.ext, sep="")) ##

## wrapper function to call the C/C++ function
hello <- function() {
  .C("hello")
}
```

Interfacing C++ code with R

Compile (output is dependent on the platform)

```
$ R CMD SHLIB ex1.cpp
g++-4.2 -arch x86_64 -I/Library/Frameworks/R.framework/Resources/include \
  -I/Library/Frameworks/R.framework/Resources/include/x86_64  \
  -I/usr/local/include -fPIC  -g -O2 -c ex1.cpp -o ex1.o
g++-4.2 -arch x86_64 -dynamiclib -Wl,-headerpad_max_install_names \
  -undefined dynamic_lookup -single_module \
  -multiply_defined suppress -L/usr/local/lib -o ex1.so ex1.o \
  -F/Library/Frameworks/R.framework/.. \
  -framework R -Wl,-framework -Wl,CoreFoundation
```

Run in your R console (use R64 if available)

```
> source('ex1.R')
> hello()
list() ## no return value is defined in the function
Hello, R
```

Argument passing

ex2.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

ex2.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)
}
```

Argument passing

Running Example (after compiling)

```
> source('ex2.R')
> square(10)  ## does the right thing
[1] 100
> square(c(10,20,30)) ## but only recognize the current value
[1] 100
```

Passing vector or matrix as argument

ex2b.cpp

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

ex2b.R

```
dyn.load(paste("ex2b", .Platform$dynlib.ext, sep=""))
square <- function(a) {
  n <- as.integer(length(a))
  r <- .C("square",as.double(a),n,out=double(n))$out
  if ( is.matrix(a) ) { return (matrix(r,nrow(a),ncol(a))); }
  else { return (r); }
}
```

Argument passing

Running Example (after compiling)

```
> source('ex2b.R')
> square(10)  ## takes a single input
[1] 100
> square(c(10,20,30)) ## takes a vector as input
[1] 100 400 900
> square(matrix(1:6,3,2)) ## takes a matrix as input
     [,1] [,2]
[1,]    1   16
[2,]    4   25
[3,]    9   36
```

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
  3.831   0.025   3.849
```

But built-in cumsum function is much faster

Running with built-in cumsum function

```
> system.time(cumsum(as.double(1:1000000)))
   user  system elapsed
  0.014   0.011   0.045
```

What's inside in the cumsum function?

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

- Uses internal implementation for the sake of efficiency

Making faster cumsum function

cumsum.cpp

```
void cumsumC(double* x, int* nx, double* y) {
  y[0] = x[0];
  int n = *nx;
  for(int i=1; i < n; ++i) {
    y[i] = y[i-1] + x[i];
  }
}
```

cumsum.R

```
cumsum.C <- function(a) {
  n <- length(a)
  .C("cumsumC",
     as.double(a),
     as.integer(length(a)),
     res = double(length(a)))$res
}
```

Running time is comparable with built-in cumsum function

```
> system.time(cumsum.R(as.double(1:1000000)))
   user  system elapsed
  3.831   0.025   3.849
> system.time(cumsum(as.double(1:1000000)))
   user  system elapsed
  0.014   0.011   0.045
> system.time(cumsum2(as.double(1:1000000)))
   user  system elapsed
  0.031   0.018   0.049
```

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
```

R/C++ interface for Gibbs Sampler

- Gibbs Sampler is more efficient in C++ than R
  - Because a large number of iteration is involved
- Output from Gibbs sampler can be analyzed in various ways with R
  - Approximate the joint distribution of the parameters
  - Plot the distribution of parameters with respect to iteration
- R/C++ interface for efficient Gibbs sampling + flexible downstream analysis

`mixGS.h` : Very similar to the original Gibbs sampler

```
#include <vector>
#include <cmath>
#include <ctime>

#define ZEPS 1e-10
#define MIN_COUNTS 20

#include "NormMix615.h"

class normMixGibbs {
public:
  int k;                         // # of components
  int n;                         // # of data
  std::vector<double> data;    // observed data
  std::vector<double> pis;     // pis
  std::vector<double> means;   // means
  std::vector<double> sigmas; // sds
  std::vector<double> labels; // label assignment
  std::vector<int>    counts;
  std::vector<double> sums;
  std::vector<double> sumsqs;
  // ...
```

mixGS.h (cont'd)

```
  // ...
  normMixGibbs(std::vector<double>& _data, int _k);

  void initParams();
  void updateParams(int numObs);
  void remove(int i);
  void add(int i, int label);
  int sampleLabel(double x);
  // only the function below has been changed from previous code
  double runGibbs(int iter, int burnin, int thin, double* tllks,
                  double* tpis, double* tmeans, double* tsigmas);
  static double randu(double min, double max);
  static int randn(int min, int max);
};
```

mixGS.h (cont'd)

```
double normMixGibbs::runGibbs(int iter, int burnin, int thin,
     double* tllks, double* tpis, double* tmeans, double* tsigmas) {
  initParams();
  for(int i=0, is = 0; i < iter; ++i) {
    int id = randn(0,n);
    if ( counts[labels[id]] < MIN_COUNTS ) continue;
    remove(id); updateParams(n-1);
    int label = sampleLabel(data[id]);
    add(id, label);
    if ( ( i >= burnin ) && ( i % thin == 0 ) ) {
      double llk = NormMix615::mixLLK(data,pis,means,sigmas);
      tllks[is] = llk;
      for(int j=0; j < k; ++j) {
        tpis[is*k+j] = pis[j];
        tmeans[is*k+j] = means[j];
        tsigmas[is*k+j] = sigmas[j];
      }
      ++is;
    }
  }
  return minllk;
}
```

gibbs.cpp

```
#include "mixGS.h"
extern "C" {
  void gibbs(double* a, int* na, int* nk, int* iter, int* burnin, int* thin,
             double* llks, double *pis, double* means, double* sigmas) {
    std::vector<double> data;
    int n = *na;
    for(int i=0; i < n; ++i) {
      data.push_back(a[i]);
    }

    srand(std::time(0));
    int k = *nk;

    normMixGibbs gs(data,k);
    double llk = gs.runGibbs(*iter, *burnin, *thin, llks, pis, means, sigmas);
  }
}
```

gibbs.R

```
dyn.load(paste("ex4", .Platform$dynlib.ext, sep=""))

gibbs <- function(x, k, iter, burnin, thin) {
  r <- as.integer(ceiling(iter-burnin)/thin)
  res <- .C("gibbs",
            as.double(x),
            as.integer(length(x)),
            as.integer(k),
            as.integer(iter),
            as.integer(burnin),
            as.integer(thin),
            llks = double(r),   ## return thinned parameters
            pis = double(k*r),
            means = double(k*r),
            sigmas = double(k*r))
  return (list(llks=res$llks,pis=matrix(res$pis,r,k,byrow=T),
               means=matrix(res$means,r,k,byrow=T),
               sigmas=matrix(res$sigmas,r,k,byrow=T)))
}
```

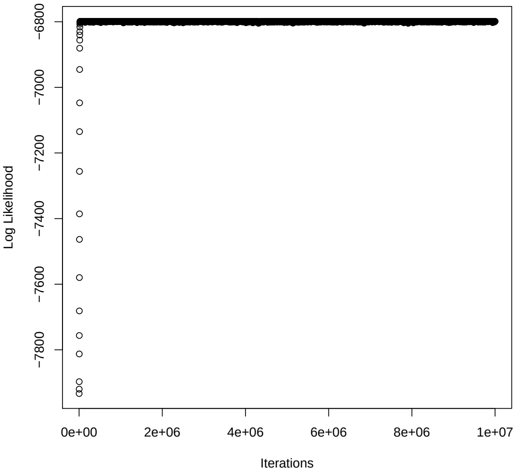
gibbsTest.R

```
source('gibbs.R')
test.gibbs <- function() {
  x <- c(rnorm(2000,0,1),rnorm(1000,5,4))
  ni <- 1e7; nb <- 0; nt <- 1000;
  nr <- as.integer(ceiling((ni-nb)/nt))
  r <- gibbs(x,2,ni,nb,nt)
  lx <- (1:nr)*nt
  pdf("test-gibbs-llks.pdf")
  plot(lx,r$llks,xlab='Iterations',ylab='Log Likelihood')
  dev.off()
  pdf("test-gibbs-pis.pdf")
  plot(lx,r$pis[,1],xlab='Iterations',ylab='Pis',xlim=c(0,ni),ylim=c(0,1),col="blue")
  points(lx,r$pis[,2],col="red")
  dev.off()
  pdf("test-gibbs-means.pdf")
  plot(lx,r$means[,1],xlab='Iterations',ylab='Means',xlim=c(0,ni),ylim=c(-1,6),col="blue")
  points(lx,r$means[,2],col="red")
  dev.off()
  pdf("test-gibbs-sigmas.pdf")
  plot(lx,r$sigmas[,1],xlab='Iterations',ylab='Means',xlim=c(0,ni),ylim=c(0,7),col="blue")
  points(lx,r$sigmas[,2],col="red")
  dev.off()
}
```

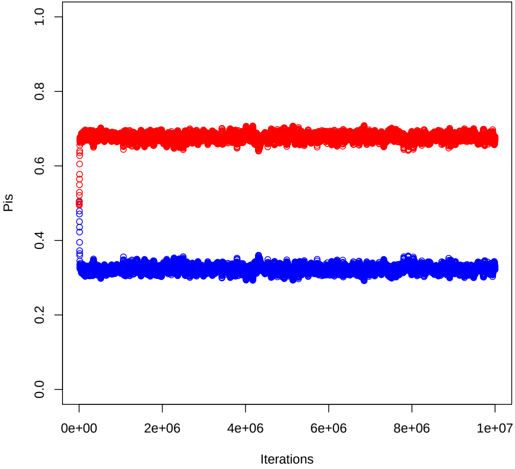
gibbsTest.R

```
> source('gibbsTest.R')
> system.time(test.gibbs())
   user  system elapsed
  6.180   0.038   6.490
```

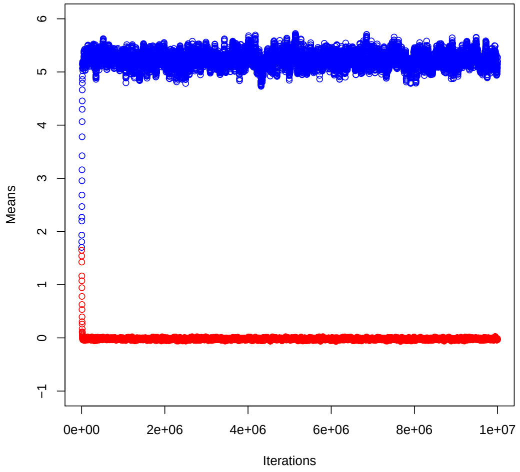
Log-likelihoods



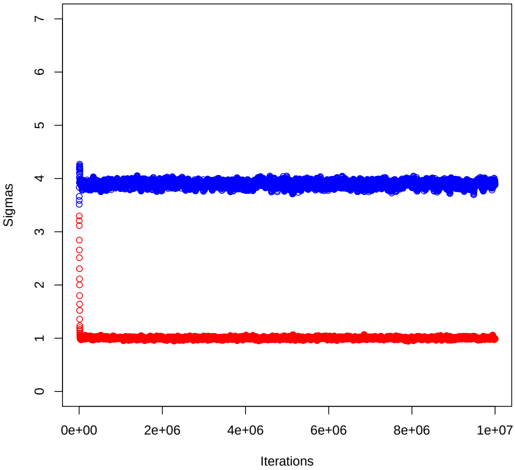
Priors



Means



Standard deviations



Today

- 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.