Linear Algebra with RcppArmadillo

Advanced Statistical Programming Camp
Jonathan Olmsted (Q-APS)

Day 4: May 30th, 2014
AM Session

Outline

Why Linear Algebra?

- Linear algebra is pervasive in statistical computations:
  - multiplication
  - inversion
  - decompositions

- Previous data structures (C++ types and Rcpp classes) do not provide us with the tools to avoid element-wise implementations of these algorithms.

- Programming these methods by hand would be tedious and error-prone.

Linear Algebra by “hand”

Inner product.

```
// [[Rcpp::export()]]
double inner1 (NumericVector x,
               NumericVector y
               ) {
  int K = x.length() ;
  double ip = 0 ;
  for (int k = 0 ; k < K ; k++) {
    ip += x(k) * y(k) ;
  }
  return(ip) ;
}
```

Vectors require 1 loop. Matrix operations would require 2 loops.

Linear Algebra by Linear Algebra Library

```
// [[Rcpp::export()]]
double inner2 (arma::vec x,
               arma::vec y
               ) {
  arma::mat ip = x.t() * y ;
  return(ip(0)) ;
}
```

Linear algebra structures allow code to resemble standard mathematical notation.

```
library("Rcpp")
sourceCpp("inner.cpp")
vec <- rnorm(1000)

inner1(vec, vec)

## [1] 983.4

inner2(vec, vec)

## [1] 983.4
```

Optimized Libraries

Consider a simulated dataset from the omitted variable bias example data generating process.

```
source("gendata.R")
sourceCpp("lmA.cpp")
dfFake <- genNormData(1000)

m1 <- lm(y ~  x1 + x2 + x3, data = dfFake)

X <- model.matrix(m1)
y <- matrix(dfFake$y)
```

Optimized Libraries

```
library(microbenchmark)

microbenchmark(lm = lm(y ~  x1 + x2 + x3, data = dfFake),
               R = solve(t(X) %*% X) %*% t(X) %*% y,
               cpp = lmA(X, y),
               times = 10
               )

## Unit: microseconds
##  expr      min       lq   median       uq      max neval
##    lm 2239.54 2338.80 2445.78 2500.3 3335.9    10
##     R  184.46  249.52  411.77  432.0  771.4    10
##   cpp   18.27   21.14   44.71   53.2  108.9    10
```

Outline

Armadillo

- Armadillo (`http://arma.sourceforge.net/`) is a popular, optimized linear algebra library for C++.
  - tuned algorithms
  - rich functionality
  - flexible
  - relatively easy to use
- **But**, hooking in to external C++ libraries isn't always easy.

- Primary classes of interest `arma::mat` and `arma::vec` (elements are `double`).
- Additional classes include sparse matrices and higher-dimensional arrays.

RcppArmadillo

- RcppArmadillo (`http://cran.r-project.org/web/packages/RcppArmadillo/index.html`) provides all the functionality of Armadillo
- Easy to install because it is distributed as an R package
- Tightly integrated with Rcpp:
  - Conversion from Rcpp classes to Armadillo classes (and back).
  - Conversion from Armadillo classes to R objects (and back).

Previously ...

```
# include <Rcpp.h>
```

Now ...

```
# include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
```

The `# include <Rcpp.h>` is implied.

Outline

Automatic Handling of Data Types

```
// [[Rcpp::export()]]
arma::mat a1(arma::mat x) {
  return(x);
}
```

Matrix in, matrix out.

Automatic Handling of Data Types

```
sourceCpp("arma_functions.cpp")
a1(diag(2))

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Automatic Handling of Data Types

```
// [[Rcpp::export()]]
arma::vec a2(arma::vec x) {
  return(x);
}
```

Vector in, matrix out.

Automatic Handling of Data Types

```
a2(1:2)

##      [,1]
## [1,]    1
## [2,]    2
```

Automatic Handling of Data Types

```
// [[Rcpp::export()]]
arma::mat a3(NumericMatrix x) {
  arma::mat y = as<arma::mat>(x);
  return(y);
}
```

From `Rcpp::NumericMatrix` to `arma::mat`.

Automatic Handling of Data Types

```
a3(diag(2))

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Automatic Handling of Data Types

```
// [[Rcpp::export()]]
NumericMatrix a4(arma::mat x) {
  NumericMatrix y = wrap(x);
  return(y);
}
```

From `arma::mat` to `Rcpp::NumericMatrix`.

Automatic Handling of Data Types

```
a4(diag(2))

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Dimensions in Armadillo

```
// [[Rcpp::export()]]
arma::mat a5(arma::mat x) {
  int R = x.n_rows;
  int C = x.n_cols;
  Rcout << "Rows: " << R << std::endl;
  Rcout << "Cols: " << C << std::endl;
  return(x);
}
```

Note the absence of parenthesis (unlike Rcpp classes).

Dimensions in Armadillo

```
a5(matrix(0, nrow = 2, ncol = 3))

## Rows: 2
## Cols: 3
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
```

Dimensions in Armadillo

```
// [[Rcpp::export()]]
arma::vec a6(arma::vec x) {
  int R = x.n_rows;
  int C = x.n_cols;
  Rcout << "Rows: " << R << std::endl;
  Rcout << "Cols: " << C << std::endl;
  return(x);
}
```

`arma::vec` objects are just matrices.

Dimensions in Armadillo

```
a6(1:4)

## Rows: 4
## Cols: 1
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

Inspection of Armadillo Objects

```
// [[Rcpp::export()]]
int a7(arma::mat x) {
  x.print();
  x.print("Note");
  return(0);
}
```

More useful than `Rcout`. No need to loop through elements.

Inspection of Armadillo Objects

```
a7(diag(2))

##    1.0000         0
##         0    1.0000
## Note
##    1.0000         0
##         0    1.0000
## [1] 0
```

Manipulating Armadillo Objects

```
// [[Rcpp::export()]]
List a8 (int n, int r, double v) {
  arma::mat x1 ;
  x1.print() ;
  x1.reshape(n, r) ;
  x1.fill(v) ;

  arma::mat x2(n, r) ;
  x2.fill(v) ;

  arma::mat x3 = x2 ;
  x3.reshape(r, n) ;

  List ret ;
  ret["x1"] = x1 ;
  ret["x2"] = x2 ;
  ret["x3"] = x3 ;
  return(ret) ;
}
```

Inspection of Armadillo Objects

```
a8(2, 3, 3/4)

## [matrix size: 0x0]
## $x1
##      [,1] [,2] [,3]
## [1,] 0.75 0.75 0.75
## [2,] 0.75 0.75 0.75
##
## $x2
##      [,1] [,2] [,3]
## [1,] 0.75 0.75 0.75
## [2,] 0.75 0.75 0.75
##
## $x3
##      [,1] [,2]
## [1,] 0.75 0.75
## [2,] 0.75 0.75
## [3,] 0.75 0.75
```

Manipulating Armadillo Objects

```
// [[Rcpp::export()]]
arma::mat a9(int n, int r) {
  arma::mat x(n, r);
  x.print("");
  x.ones();
  return(x);
}
```

Uninitialized values are unreliable.
Populate with ones or zeros (`.zeros`).

Manipulating Armadillo Objects

```
a9(2, 3)

##   6.9442e-310  6.9442e-310  6.9442e-310
##   2.2225e-314  2.2217e-314  2.2217e-314
##       [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
```

Indexing Armadillo Objects

```
// [[Rcpp::export()]]
double a10(arma::mat x, int i, int j) {
  return(x(i, j)) ;
}
```

Indexing Armadillo Objects

```
Z <- matrix(rnorm(6), 2, 3)
Z

##          [,1]     [,2]     [,3]
## [1,] -0.6580  1.98935 -0.02781
## [2,] -0.1626 -0.06105  0.51153

a10(Z, 1, 2)

## [1] 0.5115
```

Indexing Armadillo Objects

```
// [[Rcpp::export()]]
arma::mat a11(arma::mat x, int i) {
  return(x.row(i));
}
```

Index entire rows.

Indexing Armadillo Objects

```
a11(Z, 1)

##          [,1]     [,2]   [,3]
## [1,] -0.1626 -0.06105 0.5115
```

Indexing Armadillo Objects

```
// [[Rcpp::export()]]
arma::mat a12(arma::mat x, int j) {
  return(x.col(j));
}
```

Index entire columns.

Indexing Armadillo Objects

```
a12(Z, 2)

##           [,1]
## [1,] -0.02781
## [2,]  0.51153
```

Indexing Armadillo Objects

```
// [[Rcpp::export()]]
arma::mat a13(arma::mat x) {
  return(x.cols(0, 1));
}
```

Index multiple rows or columns at once.

Indexing Armadillo Objects

```
a13(Z)

##          [,1]     [,2]
## [1,] -0.6580  1.98935
## [2,] -0.1626 -0.06105
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a14(arma::mat x) {
  return(x + x);
}
```

Element-wise addition.

Operators and Functions

```
a14(Z)

##          [,1]    [,2]     [,3]
## [1,] -1.3160  3.9787 -0.05562
## [2,] -0.3252 -0.1221  1.02305
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a15(arma::mat x) {
  return(x - x);
}
```

Element-wise subtraction.

Operators and Functions

```
a15(Z)

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a16(arma::mat x) {
  return(x % x) ;
}
```

Element-wise multiplication.

Operators and Functions

```
a16(Z)

##          [,1]     [,2]      [,3]
## [1,] 0.43299 3.957510 0.0007734
## [2,] 0.02644 0.003727 0.2616587
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a17(arma::mat x) {
  return( exp(x) ) ;
}
```

Element-wise applications of functions.

Operators and Functions

```
a17(Z)

##         [,1]   [,2]   [,3]
## [1,] 0.5179 7.3108 0.9726
## [2,] 0.8499 0.9408 1.6678
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a18(arma::mat x) {
  return(x.t()) ;
}
```

Transpose.

Operators and Functions

```
a18(Z)

##           [,1]     [,2]
## [1,] -0.65802 -0.16262
## [2,]  1.98935 -0.06105
## [3,] -0.02781  0.51153
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a19(arma::mat x) {
  return( x.t() * x ) ;
}
```

Matrix multiplication.

Operators and Functions

```
a19(Z)

##           [,1]     [,2]     [,3]
## [1,]  0.45943 -1.29910 -0.06488
## [2,] -1.29910  3.96124 -0.08655
## [3,] -0.06488 -0.08655  0.26243
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a20(arma::mat x) {
  return( inv(x * x.t()) ) ;
}
```

Matrix inversion.

Operators and Functions

```
a20(Z)

##          [,1]    [,2]
## [1,] 0.22787 0.02239
## [2,] 0.02239 3.42885
```

Operators and Functions

```
// [[Rcpp::export()]]
arma::mat a21(arma::mat x) {
  return( chol(x * x.t()) ) ;
}
```

Decompositions.

Operators and Functions

```
a21(Z)

##        [,1]     [,2]
## [1,] 2.096 -0.01368
## [2,] 0.000  0.54004
```

Operators and Functions

```
// [[Rcpp::export()]]
List a22(arma::mat x) {
  arma::mat xtx = x.t() * x ;

  arma::mat U ;
  arma::vec s ;
  arma::mat V ;

  svd(U, s, V, xtx) ;

  List ret ;
  ret["U"] = U ;
  ret["s"] = s ;
  ret["V"] = V ;

  return(ret) ;
}
```

Operators and Functions

```
a22(Z)

## $U
##          [,1]     [,2]   [,3]
## [1,] -0.31345 -0.30964 0.8977
## [2,]  0.94949 -0.08729 0.3014
## [3,] -0.01498  0.94684 0.3214
##
## $s
##           [,1]
## [1,] 4.391e+00
## [2,] 2.916e-01
## [3,] 1.082e-16
##
## $V
##          [,1]     [,2]   [,3]
## [1,] -0.31345 -0.30964 0.8977
## [2,]  0.94949 -0.08729 0.3014
## [3,] -0.01498  0.94684 0.3214
```

Informative Runtime Error Messages

```
// [[Rcpp::export()]]
double a23(int i) {
  arma::mat A(2, 3);
  A.randu();
  return( A(i, i) );
}
```

This indexing is valid for some *i*.

Informative Runtime Error Messages

```
a23(1)

## [1] 0.8533

a23(4)

##
## error: Mat::operator(): index out of bounds

## Error:  Mat::operator():  index out of bounds
```

Informative Runtime Error Messages

```
// [[Rcpp::export()]]
arma::mat a24 (int i) {
  arma::mat A(2, 3) ;
  arma::mat B(i, 1) ;
  A.randu() ;
  B.randu() ;
  return( A * B ) ;
}
```

This multiplication is valid for only $i = 3$.

Informative Runtime Error Messages

```
a24(3)

##         [,1]
## [1,] 0.4896
## [2,] 1.1367

a24(2)

##
## error: matrix multiplication: incompatible matrix dimensions

## Error:  matrix multiplication:  incompatible matrix
dimensions:  2x3 and 2x1
```

Outline

Big Linear Regressions

```
dfTrade <- read.csv("trade.csv", nrows = 4e5) # just first 500k
dfTrade2 <- subset(dfTrade, source1 != -9 & source2 != -9) # drop missing
nrow(dfTrade2)

## [1] 311763

dfTrade2$total <- with(dfTrade2, flow1 + flow2)
dfTrade2$dem <- with(dfTrade2, dem1 + dem2)
dfTrade2$maj <- with(dfTrade2, majpow1 + majpow2)
f <- formula(total ~ dem + poly(year, 3) + factor(abbrev1) + factor(abbrev2))
```

Total trade flows between country pairs from 1950 through 2008. Over 300 fixed-effects.

Big Linear Regression

```
// [[Rcpp::export()]]
arma::mat lmA(arma::mat X,
              arma::mat y
              ) {
  arma::mat betahat;
  betahat = (X.t() * X ).i() * X.t() * y ;
  return(betahat) ;
}
```

Big Linear Regressions

```
system.time({
    fit <- lm(f, data = dfTrade2)
})

##    user  system elapsed
##   36.79    0.77   37.57

X <- model.matrix(fit)
system.time({
    fit2 <- lmA(model.matrix(fit), matrix(dfTrade2$total))
})

##    user  system elapsed
##   6.786   0.605   2.439
```

Don't try this at home (or on Adroit)

EM Probit Regression

- EM algorithm augments censored data with expectation of the latent (censored) variable.
- Conditional on latent varible: linear regression.

```
sourceCpp("em_probit.cpp")
library(Zelig)
data(turnout)
```

EM Probit Regression

```
arma::mat em_probit (arma::mat y,
                     arma::mat X,
                     int maxit = 10
                     ) {
  int N = y.n_rows ; int K = X.n_cols ;
  arma::mat beta(K, 1) ;
  beta.fill(0.0) ; // initialize betas to 0
  arma::mat eystar(N, 1) ;
  eystar.fill(0) ;
  for (int it = 0 ; it < maxit ; it++) {
    arma::mat mu = X * beta ;
    for (int n = 0 ; n < N ; n++) {
      if (y(n, 0) == 1) { // y = 1
        eystar(n, 0) = mu(n, 0) + f(mu(n, 0)) ; // defined elsewhere
      }
      if (y(n, 0) == 0) { // y = 0
        eystar(n, 0) = mu(n, 0) - g(mu(n, 0)) ; // defined elsewhere
      }
    }
    beta = (X.t() * X).i() * X.t() * eystar ; // linear regression
  }
  return(beta) ;
}
```

EM Probit Regression

```
system.time({
fit3 <- glm(vote ~ income + educate + age,
            data = turnout,
            family = binomial(link = "probit")
            )})

##    user  system elapsed
##   0.019   0.003   0.021

system.time({
fit4 <- em_probit(y = matrix(turnout$vote),
                  X = model.matrix(fit3),
                  maxit = 25
                  )})

##    user  system elapsed
##   0.005   0.000   0.005
```

EM Probit Regression

```
coef(fit3)

## (Intercept)      income     educate         age
##    -1.68241     0.09936     0.10667     0.01692

fit4

##           [,1]
## [1,] -1.68214
## [2,]  0.09933
## [3,]  0.10666
## [4,]  0.01691
```