RcppArmadillo: Accelerating R with C++ Linear Algebra

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org

Presentation at R/Finance 2013
University of Illinois at Chicago
18 May 2013
Armadillo

C++ linear algebra library

About
License FAQ Docs Speed Contact Download

- Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use. The syntax is deliberately similar to Matlab.

- Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries).

- A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporary. This is accomplished through recursive templates and template meta-programming.

- Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities.

- The library is open-source software, and is distributed under a license that is useful in both open-source and commercial/proprietary contexts.

- Primarily developed at NICTA (Australia) by Conrad Sanderson, with contributions from around the world.
Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use. The syntax is deliberately similar to Matlab. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided [...] A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries. Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities. The library is open-source software, and is distributed under a license that is useful in both open-source and commercial/proprietary contexts.
Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use. The syntax is deliberately similar to Matlab.

Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided [...]

A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries. [...] 

Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities.

The library is open-source software, and is distributed under a license that is useful in both open-source and commercial/proprietary contexts.
Armadillo highlights

- Provides integer, floating point and complex vectors, matrices and fields (3d) with all the common operations.
- Very good documentation and examples at website [http://arma.sf.net](http://arma.sf.net), and a technical report (Sanderson, 2010).
- Modern code, building upon and extending from earlier matrix libraries.
- Responsive and active maintainer, frequent updates.
Template-only builds—no linking, and available wherever R and a compiler work (but Rcpp is needed)!

Easy to use, just add LinkingTo: RcppArmadillo, Rcpp to DESCRIPTION (i.e., no added cost beyond Rcpp)

Really easy from R via Rcpp

Frequently updated, easy to use
To name just a few:

- **Amelia** by Gary King et al: Multiple Imputation from cross-section, time-series or both;
- **forecast** by Rob Hyndman et al: Time-series forecasting including state space and automated ARIMA modeling;
- **rugarch** by Alexios Ghalanos: Sophisticated financial time series models;
- **gRbase** by Søren Højsgaard: Graphical modeling
Outline

1. Intro
2. Examples
3. Case Study: Kalman Filter
4. End
#include <RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
arma::vec getEigenValues(arma::mat M) {
    return arma::eig_sym(M);
}
set.seed(42)
X <- matrix(rnorm(4*4), 4, 4)
Z <- X %*% t(X)
getEigenValues(Z)

## [,1]
## [1,] 0.3319
## [2,] 1.6856
## [3,] 2.4099
## [4,] 14.2100

# R gets the same results (in reverse)
# and also returns the eigenvectors.

```cpp
#include <RcppArmadillo.h>

using namespace Rcpp;

arma::mat mvrnormArma(int n, arma::vec mu, arma::mat sigma) {
  int ncols = sigma.n_cols;
  arma::mat Y = arma::randn(n, ncols);
  return arma::repmat(mu, 1, n).t() +
         Y * arma::chol(sigma);
}
```
#include "RcppArmadillo.h"

extern "C" SEXP fastLm(SEXP Xs, SEXP ys) {

try {
    Rcpp::NumericVector yr(ys);
    Rcpp::NumericMatrix Xr(Xs);
    int n = Xr.nrow(), k = Xr.ncol();
    arma::mat X(Xr.begin(), n, k, false); // reuses memory and avoids extra copy
    arma::colvec y(yr.begin(), yr.size(), false);
    arma::colvec coef = arma::solve(X, y); // fit model y \sim X
    arma::colvec res = y - X*coef; // residuals
    double s2 = std::inner_product(res.begin(), res.end(), res.begin(), 0.0)/(n - k);
    arma::colvec std_err = arma::sqrt(s2*arma::diagvec(arma::pinv(arma::trans(X)*X))); // std.errors of coefficients
    return Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                               Rcpp::Named("stderr") = std_err,
                               Rcpp::Named("df.residual") = n - k);
}

} catch (std::exception &ex) {
    forward_exception_to_r(ex);
} catch (...) {
    ::Rf_error("c++ exception (unknown reason)" );
}

return R_NilValue; // -Wall
}
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
List fastLm(NumericVector yr, NumericMatrix Xr) {
    int n = Xr.nrow(), k = Xr.ncol();
    mat X(Xr.begin(), n, k, false);
    colvec y(yr.begin(), yr.size(), false);

    colvec coef = solve(X, y);
    colvec resid = y - X*coef;

    double sig2 = as_scalar(trans(resid)*resid/(n-k));
    colvec stderrest = sqrt(sig2 * diagvec( inv(trans(X)*X)) );

    return List::create(Named("coefficients") = coef,
                        Named("stderr") = stderrest,
                        Named("df.residual") = n - k );
}
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
List fastLm2(colvec y, mat X) {
  int n = X.n_rows, k = X.n_cols;

  colvec coef = solve(X, y);
  colvec resid = y - X*coef;

  double sig2 = as_scalar(trans(resid)*resid/(n-k));
  colvec stderrest = sqrt(sig2 * diagvec( inv(trans(X)*X)) );

  return List::create(Named("coefficients") = coef,
                       Named("stderr") = stderrest,
                       Named("df.residual") = n - k
                      );
}
One note on direct casting with Armadillo

The code as just shown:

```cpp
arma::colvec y = Rcpp::as<arma::colvec>(ys);
arma::mat X = Rcpp::as<arma::mat>(Xs);
```

is very convenient, but does incur an additional copy of each object. A lighter variant uses two steps in which only a pointer to the object is copied:

```cpp
Rcpp::NumericVector yr(ys);
Rcpp::NumericMatrix Xr(Xs);
int n = Xr.nrow(), k = Xr.ncol();
arma::mat X(Xr.begin(), n, k, false);
arma::colvec y(yr.begin(), yr.size(), false);
```

If performance is a concern, the latter approach may be preferable.
Outline

1. Intro
2. Examples
3. Case Study: Kalman Filter
4. End
The Mathworks site has a nice and short example of a classic 'object tracking' problem.

% Copyright 2010 The MathWorks, Inc.
function y = kalmanfilter(z)
% #codegen
dt=1;
% Initialize state transition matrix
A=[1 0 dt 0 0 0; 0 1 0 dt 0 0; 0 0 1 0 dt 0; 0 0 0 1 0; 0 0 0 0 1 ];
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
Q = eye(6);
R = 1000 * eye(2);
persistent x_est p_est
if isempty(x_est)
    x_est = zeros(6, 1);
p_est = zeros(6, 6);
end
% Predicted state and covariance
x_prd = A * x_est;
p_prd = A * p_est * A' + Q;
% Estimation
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';
% Estimated state and covariance
x_est = x_prd+klm_gain*(z-H*x_prd);
p_est = p_prd-klm_gain*H*p_prd;
% Compute the estimated measurements
y = H * x_est;
end
% of the function
Kalman Filter: In R
Easy enough – first naive solution

FirstKalmanR <- function(pos) {

    kf <- function(z) {
        dt <- 1

        A <- matrix(c(1, 0, dt, 0, 0, 0, 
                      0, 1, 0, dt, 0, 0, 
                      0, 0, 1, 0, dt, 0, 
                      0, 0, 0, 1, 0, 0, 
                      0, 0, 0, 0, 1, 0), 
                   6, 6, byrow=TRUE)

        H <- matrix(c(1, 0, 0, 0, 0, 0, 
                      0, 1, 0, 0, 0, 0), 
                   2, 6, byrow=TRUE)

        Q <- diag(6)
        R <- 1000 * diag(2)

        N <- nrow(pos)
        y <- matrix(NA, N, 2)

        ## predicted state and covriance
        xprd <- A %*% xest
        pprd <- A %*% pest %*% t(A) + Q

        ## estimation
        S <- H %*% t(pprd) %*% t(H) + R
        B <- H %*% t(pprd)
        ## kalmangain <- (S \ B)'
        kg <- t(solve(S, B))

        ## est. state and cov, assign to vars in parent env
        xest <<- xprd + kg %*% (z-H%*%xprd)
        pest <<- pprd - kg %*% H %*% pprd

        ## compute the estimated measurements
        y <<- H %*% xest

    }

    xest <- matrix(0, 6, 1)
    pest <- matrix(0, 6, 6)

    for (i in 1:N) {
        y[i,] <- kf(t(pos[i,"drop=FALSE"]))
    }

    invisible(y)
}
Kalman Filter: In R
Easy enough – with some minor refactoring

```r
KalmanR <- function(pos) {

kf <- function(z) {
  ## predicted state and covariance
  xprd <- A %*% xest
  pprd <- A %*% pest %*% t(A) + Q

  ## estimation
  S <- H %*% t(pprd) %*% t(H) + R
  B <- H %*% t(pprd)
  ## kg <- (S \ B)'
  kg <- t(solve(S, B))

  ## estimated state and covariance
  ## assigned to vars in parent env
  xest <<- xprd + kg %*% (z-H%*%xprd)
  pest <<- pprd - kg %*% H %*% pprd

  ## compute the estimated measurements
  y <- H %*% xest
}

dt <- 1

A <- matrix(c(1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1), 6, 6, byrow=TRUE)
H <- matrix(c(1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 2, 6, byrow=TRUE)
Q <- diag(6)
R <- 1000 * diag(2)
N <- nrow(pos)
y <- matrix(NA, N, 2)
xest <- matrix(0, 6, 1)
pest <- matrix(0, 6, 6)

for (i in 1:N) {
  y[i,] <- kf(t(pos[i,,drop=FALSE]))
} invisible(y)
```
Kalman Filter: In C++
Using a simple class

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

using namespace arma;

class Kalman {
private:
  mat A, H, Q, R, xest, pest;
  double dt;

public:
  // constructor, sets up data structures
  Kalman() : dt(1.0) {
    A.eye(6,6);
    A(0,2) = A(1,3) = dt;
    A(2,4) = A(3,5) = dt;
    H.zeros(2,6);
    H(0,0) = H(1,1) = 1.0;
    Q.eye(6,6);
    R = 1000 * eye(2,2);
    xest.zeros(6,1);
    pest.zeros(6,6);
  }

  // sole member func.: estimate model
  mat estimate(const mat & Z) {
    unsigned int n = Z.n_rows,
                k = Z.n_cols;
    mat Y = zeros(n, k);
    mat xprd, pprd, S, B, kg;
    colvec z, y;

    for (unsigned int i = 0; i < n; i++) {
      z = Z.row(i).t();
      // predicted state and covariance
      xprd = A * xest;
      pprd = A * pest * A.t() + Q;
      // estimation
      S = H * pprd.t() * H.t() + R;
      B = H * pprd.t();
      kg = (solve(S, B)).t();
      // estimated state and covariance
      xest = xprd + kg * (z - H * xprd);
      pest = pprd - kg * H * pprd;
      // compute estimated measurements
      y = H * xest;
      Y.row(i) = y.t();
    }
    return Y;
  }
};
```

Dirk Eddelbuettel
RcppArmadillo
Given the code from the previous slide, we just add

```cpp
// [[Rcpp::export]]
mat KalmanCpp(mat Z) {
    Kalman K;
    mat Y = K.estimate(Z);
    return Y;
}
```
Even byte-compiled 'better' R version is 66 times slower:

```r
R> FirstKalmanRC <- cmpfun(FirstKalmanR)
R> KalmanRC <- cmpfun(KalmanR)
R>
R> stopifnot(identical(KalmanR(pos), KalmanRC(pos)),
+ all.equal(KalmanR(pos), KalmanCpp(pos)),
+ identical(FirstKalmanR(pos), FirstKalmanRC(pos)),
+ all.equal(KalmanR(pos), FirstKalmanR(pos)))
R>
R> res <- benchmark(KalmanR(pos), KalmanRC(pos),
+ FirstKalmanR(pos), FirstKalmanRC(pos),
+ KalmanCpp(pos),
+ columns = c("test", "replications", "elapsed", "relative"),
+ order="relative",
+ replications=100)
R>
R> print(res)

           test  replications elapsed relative
5  KalmanCpp(pos)         100   0.087  1.0000
2   KalmanRC(pos)         100   5.774  66.3678
1    KalmanR(pos)         100   6.448  74.1149
4 FirstKalmanRC(pos)      100  8.153  93.7126
3 FirstKalmanR(pos)       100  8.901 102.3103
```
Last but not least we can redo the plot as well
Outline

1. Intro
2. Examples
3. Case Study: Kalman Filter
4. End
RcppArmadillo: Accelerating R with high-performance C++ linear algebra

Dirk Eddelbuettel\textsuperscript{a}, 1, 2, Conrad Sanderson\textsuperscript{b}, c

\textsuperscript{a} 711 Monroe Avenue, River Forest, IL 60305, USA
\textsuperscript{b} NICTA, PO Box 6020, St Lucia, QLD 4067, Australia
\textsuperscript{c} Queensland University of Technology (QUT), Brisbane, QLD 4000, Australia

http://dx.doi.org/10.1016/j.csda.2013.02.005, How to Cite or Link Using DOI

View full text

Purchase $31.50
Intially expected in May 2013.

*Real Soon Now.*
The C in Rcpp stands for ...