Higher-Performance R via C++

Part 3: Key Rcpp Application Packages

Dirk Eddelbuettel

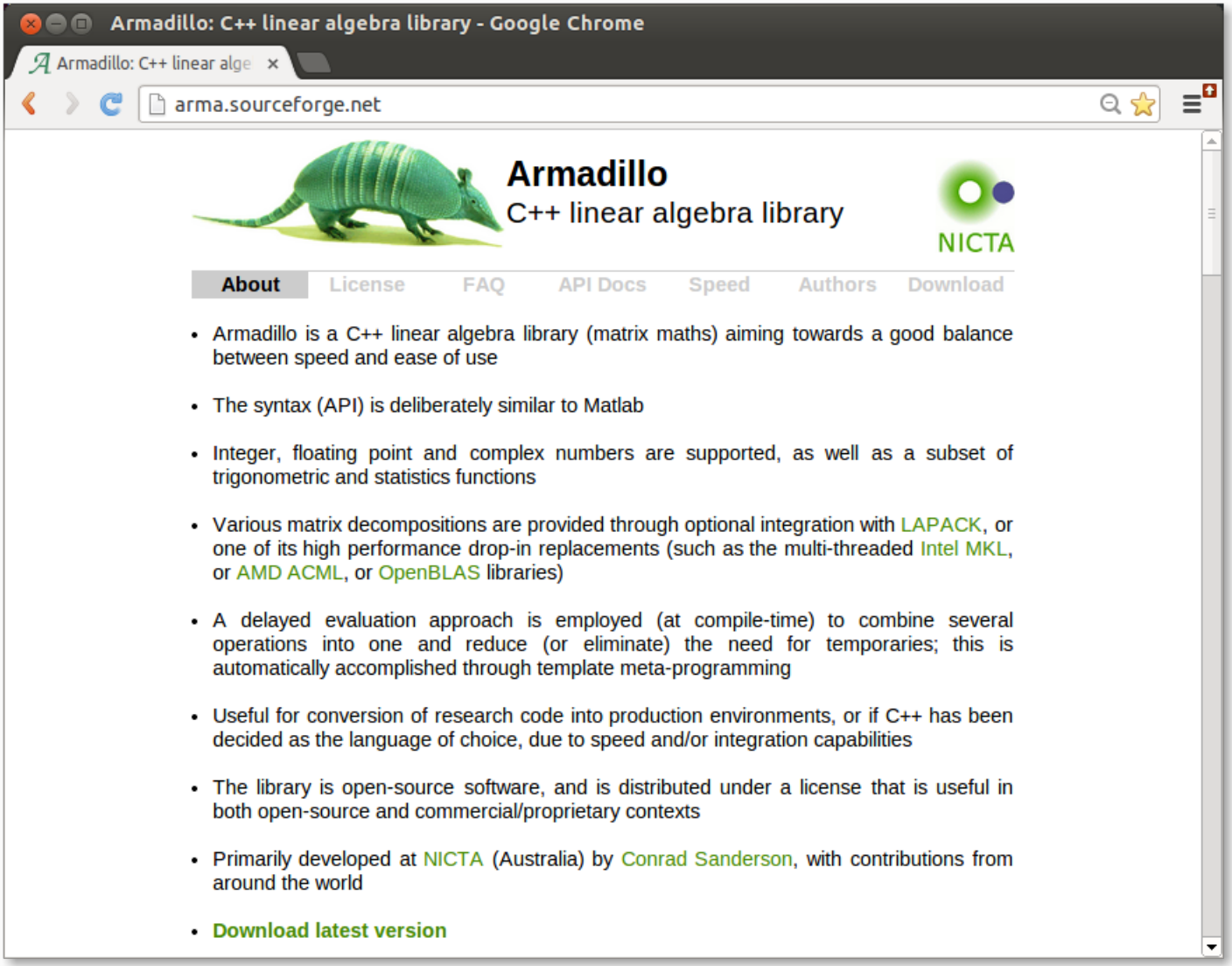
UZH/ETH Zürich R Courses

June 24-25, 2015

Overview

Armadillo

Armadillo


Armadillo: C++ linear algebra library - Google Chrome
Armadillo: C++ linear alge
arma.sourceforge.net
Armadillo
C++ linear algebra library
NICTA
About
License
FAQ
API Docs
Speed
Authors
Download
• Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use
• The syntax (API) 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 Intel MKL, or AMD ACML, or OpenBLAS libraries)
• A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries; this is automatically accomplished through 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
• Download latest version

What is Armadillo?

- 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.
- 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.

Armadillo Highlights

- Provides integer, floating point and complex vectors, matrices and fields (3d) with all the common operations.
- Very good documentation and examples
  - website,
  - technical report (Sanderson, 2010)
  - CSDA paper (Sanderson and Eddelbuettel, 2014).
- Modern code, building upon and extending from earlier matrix libraries.
- Responsive and active maintainer, frequent updates.
- Used eg by MLPACK, see Curtin et al (JMLR, 2013)

RcppArmadillo Highlights

- Template-only builds—no linking, and available whereever 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 and automatic converters
- Frequently updated, widely used

Example: Eigen values

```
#include <RcppArmadillo.h>

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

// [[Rcpp::export]]
arma::vec getEigenValues(arma::mat M) {
    return arma::eig_sym(M);
}
```

Example: Eigen values

```
Rcpp::sourceCpp("code/arma_eigenvalues.cpp")
M <- cbind(c(1,-1), c(-1,1))
getEigenValues(M)
```

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

```
eigen(M)$values
```

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

Example: Vector products

```
#include <RcppArmadillo.h>

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

// another simple example: outer product of a vector,
// returning a matrix
//
// [[Rcpp::export]]
arma::mat rcpparma_outerproduct(const arma::colvec & x) {
    arma::mat m = x * x.t();
    return m;
}

// and the inner product returns a scalar
//
// [[Rcpp::export]]
double rcpparma_innerproduct(const arma::colvec & x) {
    double v = arma::as_scalar(x.t() * x);
    return v;
}
```

Armadillo: `FastLm` Case Study

Faster Linear Model with FastLm

Background

- Implementations of `fastLm()` have been a staple during development of Rcpp
- First version was in response to a question by Ivo Welch on r-help.
- Request was for a fast function to estimate parameters – and their standard errors – from a linear model,
- It used GSL functions to estimate $\hat{\beta}$ as well as its standard errors $\hat{\sigma}$ – as `lm.fit()` in R only returns the former.
- It has since been reimplemented for RcppArmadillo and RcppEigen

Initial FastLm

```
#include <RcppArmadillo.h>

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

  try {
    Rcpp::NumericVector yr(ys);                     // creates Rcpp vector from SEXP
    Rcpp::NumericMatrix Xr(Xs);                     // creates Rcpp matrix from SEXP
    int n = Xr.nrow(), k = Xr.ncol();
    arma::mat X(Xr.begin(), n, k, false);           // reuses memory, avoids extra copy
    arma::colvec y(yr.begin(), yr.size(), false);

    arma::colvec coef = arma::solve(X, y);          // fit model y ~ 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 =                          // std.errors of coefficients
        arma::sqrt(s2*arma::diagvec(arma::pinv(arma::trans(X)*X)));

    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
}
```

Newer Version

```
// [[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   );
}
```

Current Version

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

// [[Rcpp::export]]
List fastLm(const arma::mat& X, const arma::colvec& y) {
    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   );
}
```

Interface Changes

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

Convenient, yet incurs an additional copy. Next variant uses two steps, but only a pointer to objects is copied:

```
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);
```

Better if performance is a concern. But now RcppArmadillo has efficient `const references` too.

Benchmark

```
edd@don:~$ Rscript ~/git/rcpparmadillo/inst/examples/fastLm.r
                      test replications relative elapsed
3         fLmConstRef(X, y)          5000    1.000   0.245
2         fLmTwoCasts(X, y)          5000    1.045   0.256
4             fLmSEXP(X, y)          5000    1.094   0.268
1          fLmOneCast(X, y)          5000    1.098   0.269
6   fastLmPureDotCall(X, y)          5000    1.118   0.274
8              lm.fit(X, y)          5000    1.673   0.410
5          fastLmPure(X, y)          5000    1.763   0.432
7 fastLm(frm, data = trees)          5000   30.612   7.500
9     lm(frm, data = trees)          5000   30.796   7.545
## continued below
```

Benchmark

```
## continued from above
                     test replications relative elapsed
2       fLmTwoCasts(X, y)        50000    1.000   2.327
3           fLmSEXP(X, y)        50000    1.049   2.442
4       fLmConstRef(X, y)        50000    1.050   2.444
1        fLmOneCast(X, y)        50000    1.150   2.677
6 fastLmPureDotCall(X, y)        50000    1.342   3.123
5        fastLmPure(X, y)        50000    1.988   4.627
7            lm.fit(X, y)        50000    2.141   4.982
edd@don:~$
```

Armadillo: VAR(1) Case Study

VAR(1) in R

Simulating a VAR(1) system of k variables:

$$X_t = X_{t-1}B + E_t$$

where X_t is a row vector of length k, B is a k by k matrix and E_t is a row of the error matrix of k columns.

We use $k = 2$ for this example.

VAR(1) in R

```
## parameter and error terms used throughout
a <- matrix(c(0.5,0.1,0.1,0.5),nrow=2)
e <- matrix(rnorm(10000),ncol=2)

## Let's start with the R version
rSim <- function(coeff, errors) {
    simdata <- matrix(0, nrow(errors), ncol(errors))
    for (row in 2:nrow(errors)) {
        simdata[row,] = coeff %*% simdata[(row-1),] +
            errors[row,]
    }
    return(simdata)
}
rData <- rSim(a, e)                          # generated by R
```

VAR(1) in C++

```
arma::mat rcppSim(const arma::mat& coeff,
                  const arma::mat& errors) {
    int m = errors.n_rows;
    int n = errors.n_cols;
    arma::mat simdata(m,n);
    simdata.row(0) = arma::zeros<arma::mat>(1,n);
    for (int row=1; row<m; row++) {
        simdata.row(row) = simdata.row(row-1) * coeff +
            errors.row(row);
    }
    return simdata;
}
```

Benchmark

```
library(rbenchmark)
Rcpp::sourceCpp("code/arma_var1.cpp")
benchmark(rSim(a,e), rcppSim(a, e))[,1:4]
```

```
##             test replications elapsed relative
## 2 rcppSim(a, e)          100   0.028    1.000
## 1    rSim(a, e)          100   2.987  106.679
```

Armadillo: Kalman Filter Case Study

Setup

The position of an object is estimated based on past values of 6×1 state vectors X and Y for position, V_X and V_Y for speed, and A_X and A_Y for acceleration.

Position updates as a function of the speed

$$X = X_0 + V_X dt \qquad \text{and} \qquad Y = Y_0 + V_Y dt,$$

which is updated as a function of the (unobserved) acceleration:

$$V_x = V_{X,0} + A_X dt \qquad \text{and} \qquad V_y = V_{Y,0} + A_Y dt.$$

Matlab Code: kalmanfilter.m

```
% Copyright 2010 The MathWorks, Inc.
function y = kalmanfilter(z)
  dt=1;
  % Initialize state transition matrix
  A=[ 1 0 dt 0 0 0; 0 1 0 dt 0 0;...   % [x ], [y ]
      0 0 1 0 dt 0; 0 0 0 1 0 dt;...   % [Vx], [Vy]
      0 0 0 0 1 0 ; 0 0 0 0 0 1 ];     % [Ax], [Ay]
  H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];    % Init. measuremnt mat
  Q = eye(6);
  R = 1000 * eye(2);
  persistent x_est p_est               % Init. state cond.
  if isempty(x_est)
    x_est = zeros(6, 1);               % x_est=[x,y,Vx,Vy,Ax,Ay]'
    p_est = zeros(6, 6);
  end

  x_prd = A * x_est;                   % Predicted state and covariance
  p_prd = A * p_est * A' + Q;

  S = H * p_prd' * H' + R;             % Estimation
  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;
  y = H * x_est;                       % Compute the estimated measurements
end                                    % of the function
```

Matlab Code: `kalmanM.m` with loop

```
function Y = kalmanM(pos)
  dt=1;
  %% Initialize state transition matrix
  A=[ 1 0 dt 0 0 0;...      % [x  ]
      0 1 0 dt 0 0;...      % [y  ]
      0 0 1 0 dt 0;...      % [Vx]
      0 0 0 1 0 dt;...      % [Vy]
      0 0 0 0 1 0 ;...      % [Ax]
      0 0 0 0 0 1 ];        % [Ay]
  H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];    % Initialize measurement matrix
  Q = eye(6);
  R = 1000 * eye(2);
  x_est = zeros(6, 1);                 % x_est=[x,y,Vx,Vy,Ax,Ay]'
  p_est = zeros(6, 6);
  numPts = size(pos,1);
  Y = zeros(numPts, 2);
  for idx = 1:numPts
    z = pos(idx, :)';
    x_prd = A * x_est;                 % Predicted state and covariance
    p_prd = A * p_est * A' + Q;
    S = H * p_prd' * H' + R;           % Estimation
    B = H * p_prd';
    klm_gain = (S \ B)';
    x_est = x_prd + klm_gain * (z - H * x_prd);  % Estimated state and covariance
    p_est = p_prd - klm_gain * H * p_prd;
    Y(idx, :) = H * x_est;             % Compute the estimated measurements
  end
end   % of the function
```

Now in R

```
FirstKalmanR <- function(pos) {
    kalmanfilter <- function(z) {
        dt <- 1
        A <- matrix(c( 1, 0, dt, 0, 0, 0, 0, 1, 0, dt, 0, 0,  # x,  y
                       0, 0, 1, 0, dt, 0, 0, 0, 0, 1, 0, dt,  # Vx, Vy
                       0, 0, 0, 0, 1,  0, 0, 0, 0, 0, 0,  1), # Ax, Ay
                    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)
        xprd <- A %*% xest          # predicted state and covriance
        pprd <- A %*% pest %*% t(A) + Q
        S <- H %*% t(pprd) %*% t(H) + R     # estimation
        B <- H %*% t(pprd)
        kalmangain <- t(solve(S, B))
        ## estimated state and covariance, assign to vars in parent env
        xest <<- xprd + kalmangain %*% (z - H %*% xprd)
        pest <<- pprd - kalmangain %*% H %*% pprd
        y <- H %*% xest                     # compute the estimated measurements
    }
    xest <- matrix(0, 6, 1)
    pest <- matrix(0, 6, 6)
    N <- nrow(pos)
    y <- matrix(NA, N, 2)
    for (i in 1:N) y[i,] <- kalmanfilter(t(pos[i,,drop=FALSE]))
    invisible(y)
}
```

Improved in R

```
KalmanR <- function(pos) {
    kalmanfilter <- function(z) {
        xprd <- A %*% xest                          # predicted state and covariance
        pprd <- A %*% pest %*% t(A) + Q
        S <- H %*% t(pprd) %*% t(H) + R             # estimation
        B <- H %*% t(pprd)
        kalmangain <- t(solve(S, B))
        xest <<- xprd + kalmangain %*% (z - H %*% xprd)    # est. state and covariance
        pest <<- pprd - kalmangain %*% H %*% pprd          # ass. to vars in parent env
        y <- H %*% xest                             # compute the estimated measurements
    }
    dt <- 1
    A <- matrix( c( 1, 0, dt, 0, 0, 0,   # x
                    0, 1, 0, dt, 0, 0,   # y
                    0, 0, 1, 0, dt, 0,   # Vx
                    0, 0, 0, 1, 0, dt,   # Vy
                    0, 0, 0, 0, 1,  0,   # Ax
                    0, 0, 0, 0, 0,  1),  # Ay
                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)
    xest <- matrix(0, 6, 1)
    pest <- matrix(0, 6, 6)
    for (i in 1:N) Y[i,] <- kalmanfilter(t(pos[i,,drop=FALSE]))
    invisible(Y)
}
```

And now in C++

```
// [[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) = 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);
    }

   // cont. below
```

And now in C++

```
// continued
    // sole member function: 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, kalmangain;
       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();
           kalmangain = (solve(S, B)).t();
           // estimated state and covariance
           xest = xprd + kalmangain * (z - H * xprd);
           pest = pprd - kalmangain * H * pprd;
           // compute the estimated measurements
           y = H * xest;
           Y.row(i) = y.t();
       }
       return Y;
    }
};
```

And now in C++

And the call:

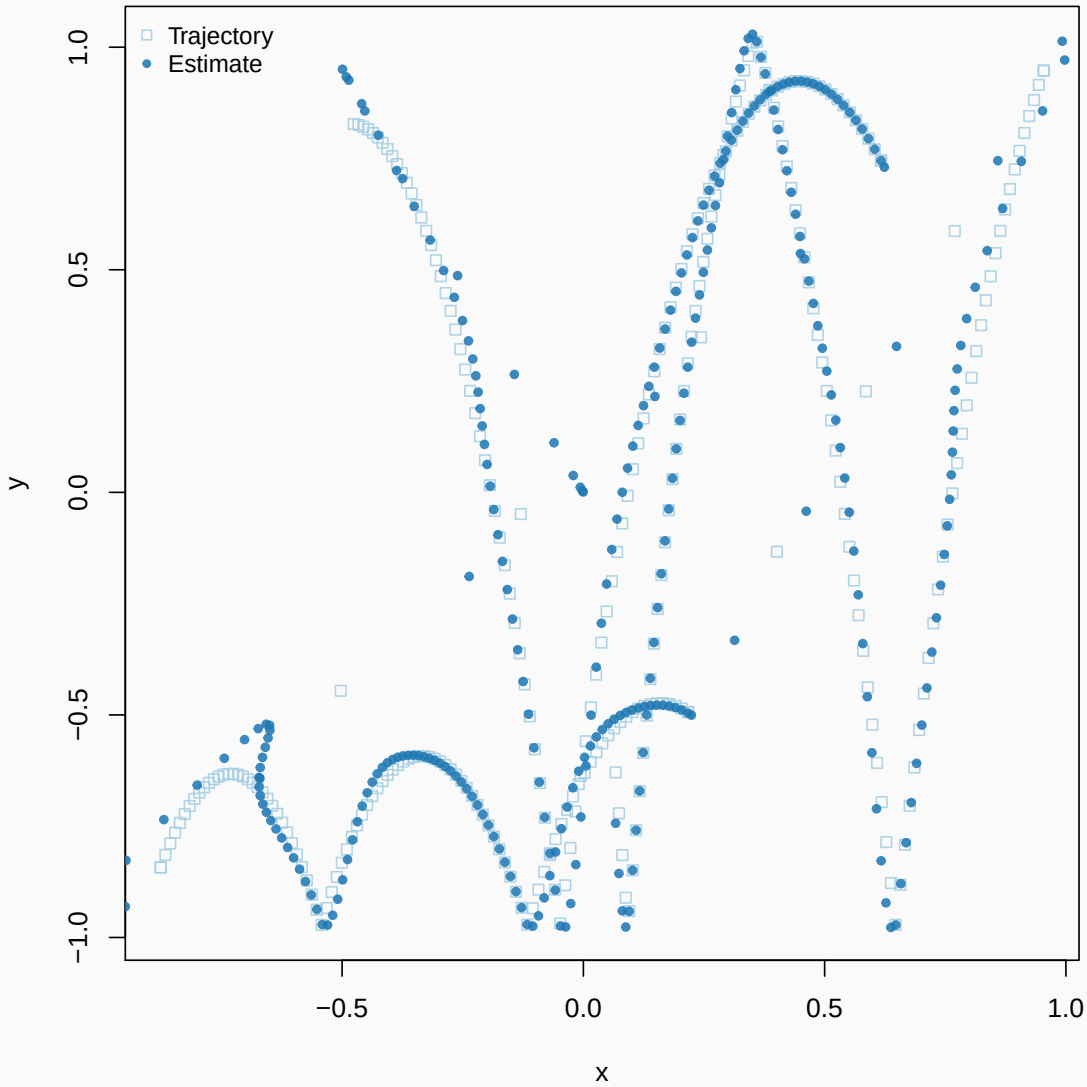
```
// [[Rcpp::export]]
mat KalmanCpp(mat Z) {
  Kalman K;
  mat Y = K.estimate(Z);
  return Y;
}
```

Benchmark

```
library(rbenchmark)
Rcpp::sourceCpp("code/kalman.cpp")
source("code/kalman.R")
p <- as.matrix(read.table("code/pos.txt",
                          header=FALSE,
                          col.names=c("x","y")))
benchmark(KalmanR(p), FirstKalmanR(p), KalmanCpp(p),
          order="relative", replications=500)[,1:4]
```

```
##              test replications elapsed relative
## 3    KalmanCpp(p)          500   1.581    1.000
## 1      KalmanR(p)          500  35.880   22.694
## 2 FirstKalmanR(p)          500  45.067   28.505
```

Reproduced Figure



Armadillo: Sparse Matrix Case Study

Sparse Matrix

A nice example for work on R objects.

```
library(Matrix)
i <- c(1,3:8)
j <- c(2,9,6:10)
x <- 7 * (1:7)
A <- sparseMatrix(i, j, x = x)
A
```

```
## 8 x 10 sparse Matrix of class "dgCMatrix"
##
## [1,] . 7 . . .  .  .  .  .  .
## [2,] . . . . .  .  .  .  .  .
## [3,] . . . . .  .  .  . 14  .
## [4,] . . . . . 21  .  .  .  .
## [5,] . . . . .  . 28  .  .  .
## [6,] . . . . .  .  . 35  .  .
## [7,] . . . . .  .  .  . 42  .
## [8,] . . . . .  .  .  .  . 49
```

Sparse Matrix

```
str(A)
```

```
## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:7] 0 3 4 5 2 6 7
##   ..@ p       : int [1:11] 0 0 1 1 1 1 2 3 4 6 ...
##   ..@ Dim     : int [1:2] 8 10
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:7] 7 21 28 35 14 42 49
##   ..@ factors : list()
```

Note how the construction was in terms of $< i, j, x >$, yet the representation in in terms of $< i, p, x >$ – CSC format.

Sparse Matrix

```
#include <RcppArmadillo.h>

using namespace Rcpp;
using namespace arma;

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

// [[Rcpp::export]]
sp_mat armaEx(S4 mat, bool show) {
    IntegerVector dims = mat.slot("Dim");
    arma::urowvec i = Rcpp::as<arma::urowvec>(mat.slot("i"));
    arma::urowvec p = Rcpp::as<arma::urowvec>(mat.slot("p"));
    arma::vec x     = Rcpp::as<arma::vec>(mat.slot("x"));

    int nrow = dims[0], ncol = dims[1];
    arma::sp_mat res(i, p, x, nrow, ncol);
    if (show) Rcpp::Rcout << res << std::endl;
    return res;
}
```

Sparse Matrix

```
Rcpp::sourceCpp('code/arma_sparse.cpp')
B <- armaEx(A, TRUE)
```

```
## [matrix size: 8x10; n_nonzero: 7; density: 8.75%]
##
##      (0, 1)          7.0000
##      (3, 5)         21.0000
##      (4, 6)         28.0000
##      (5, 7)         35.0000
##      (2, 8)         14.0000
##      (6, 8)         42.0000
##      (7, 9)         49.0000
```

RcppEigen

RcppEigen

- RcppEigen wraps the Eigen library for linear algebra.
- Eigen is similar to Armadillo, and very highly optimised—by internal routines replacing even the BLAS for performance.
- Eigen offers a more complete API than Armadillo (but I prefer to work with the simpler Armadillo, most of the time).
- RcppEigen was started by Doug Bates who needed sparse matrix support for his C++ rewrite of lme4.
- Eigen can be faster than Armadillo, eg CRAN package robustHD (using Armadillo) with a drop-in replacement sparseLTSEigen sees gain of 1/4 to 1/3.
- Documented in Bates and Eddelbuettel (JSS, 2013) paper

RcppEigen fastLm

```
// part of larger example showing ability to compute
// model fitting projection via different decompositions

static inline lm do_lm(const Map<MatrixXd> &X, const Map<VectorXd> &y, int type) {
switch(type) {
case ColPivQR_t:
    return ColPivQR(X, y);
case QR_t:
    return QR(X, y);
case LLT_t:
    return Llt(X, y);
case LDLT_t:
    return Ldlt(X, y);
case SVD_t:
    return SVD(X, y);
case SymmEigen_t:
    return SymmEigen(X, y);
case GESDD_t:
    return GESDD(X, y);
}
throw invalid_argument("invalid type");
return ColPivQR(X, y);  // -Wall
}
```

RcppEigen fastLm

```
const Map<MatrixXd>  X(as<Map<MatrixXd> >(Xs));
const Map<VectorXd>  y(as<Map<VectorXd> >(ys));
Index                n = X.rows();

                     // Select and apply the least squares method
lm                  ans(do_lm(X, y, ::Rf_asInteger(type)));

                     // Copy coefficients and install names, if any
NumericVector     coef(wrap(ans.coef()));
List          dimnames(NumericMatrix(Xs).attr("dimnames"));
VectorXd         resid = y - ans.fitted();
int               rank = ans.rank();
int                 df = (rank == ::NA_INTEGER) ? n - X.cols() : n - rank;
double               s = resid.norm() / std::sqrt(double(df));

                     // Create the standard errors
VectorXd             se = s * ans.se();

return List::create(Named("coefficients")  = coef,
                    Named("se")            = se,
                    Named("rank")          = rank,
                    Named("df.residual")   = df,
                    Named("residuals")     = resid,
                    Named("s")             = s,
                    Named("fitted.values") = ans.fitted());
```

RcppEigen fastLm

Doug defines a base class `lm` from which the following classes derive:

- LLt (standard Cholesky decomposition)
- LDLt (robust Cholesky decompostion with pivoting)
- SymmEigen (standard Eigen-decomposition)
- QR (standard QR decomposition)
- ColPivQR (Householder rank-revealing QR decomposition with column-pivoting)
- SVD (standard SVD decomposition)

The example file `lmBenchmark.R` in the package runs through these.

RcppEigen fastLm

```
edd@don:~$ r git/rcppeigen/inst/examples/lmBenchmark.R
lm benchmark for n = 100000 and p = 40: nrep = 20
      test relative elapsed user.self sys.self
8      LLt    1.000   1.749     1.712    0.040
3     LDLt    1.042   1.823     1.776    0.048
6  SymmEig    2.751   4.812     3.964    0.844
7       QR    4.012   7.017     6.044    0.964
1   lm.fit    4.359   7.624    12.308   16.764
2    PivQR    4.523   7.911     6.932    1.128
9     arma    5.719  10.003    15.768   22.080
4    GESDD   10.447  18.272    25.112   31.372
5      SVD   35.584  62.236    59.304    2.952
10     GSL   69.646 121.811   122.168    8.952
edd@don:~$
```

RcppEigen Final Remark

Doug often reminds us about the occassional fine differences between *statistical* numerical analysis and standard numerical analysis.

Pivoting schemes are a good example. R uses a custom decomposition (with pivoting) inside of `lm()` which makes it both robust and precise, particularly for rank-deficient matrices.

The example for `fastLm()` in both RcppArmadillo and RcppEigen provides an illustration.

If you are *really* sure your data is well-behaved, then using a faster (non-pivoting) scheme as in RcppArmadillo is ok.

GSL

RcppGSL

- RcppGSL is a convenience wrapper for accessing the GNU GSL, particularly for vector and matrix functions.
- Given that the GSL is a C library, we need to
  - do memory management and free objects
  - arrange for the GSL libraries to be found
- RcppGSL may still be a convenient tool for programmers more familiar with C than C++ wanting to deploy GSL algorithms.

GSL Vector Norm Example

```
#include <RcppGSL.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

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

// [[Rcpp::export]]
Rcpp::NumericVector colNorm(Rcpp::NumericMatrix NM) {
    // this conversion involves an allocation
    RcppGSL::matrix<double> M = Rcpp::as< RcppGSL::matrix<double> >(NM);
    int k = M.ncol();
    Rcpp::NumericVector n(k);           // to store results
    for (int j = 0; j < k; j++) {
        RcppGSL::vector_view<double> colview = gsl_matrix_column (M, j);
        n[j] = gsl_blas_dnrm2(colview);
    }
    M.free();
    return n;                           // return vector
}
```

GSL Vector Norm Example

```
Rcpp::sourceCpp("code/gslNorm.cpp")
set.seed(42)
M <- matrix(rnorm(25), 5, 5)
colNorm(M)                                  # via GSL
```

```
## [1] 1.701241 2.526438 2.992635 3.903917 2.892030
```

```
apply(M, 2, function(x) sqrt(sum(x^2)))    # via R
```

```
## [1] 1.701241 2.526438 2.992635 3.903917 2.892030
```

GSL bSpline Example

- The example comes from Section 39.7 of the GSL Reference manual, and constructs a data set from the curve $y(x) = \cos(x)\exp(-x/10)$ on the interval $[0, 15]$ with added Gaussian noise — which is then fit via linear least squares using a cubic B-spline basis functions with uniform breakpoints.
- Obviously all this could be done in R too as R can both generate data, and fit models including (B-)splines. But the point to be made here is that we can very easily translate a given GSL program (thanks to RcppGSL), and get it into R with ease thanks to Rcpp and Rcpp attributes.

GSL bSpline Example: C++ (1/6)

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

#include <gsl/gsl_bspline.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

const int N = 200;                      // number of data points to fit
const int NCOEFFS = 12;                 // number of fit coefficients */
const int NBREAK = (NCOEFFS - 2);       // nbreak = ncoeffs + 2 - k = ncoeffs - 2 since k = 4 */

// [[Rcpp::export]]
Rcpp::List genData() {
    const size_t n = N;
    size_t i;
    double dy;
    gsl_rng *r;
    RcppGSL::vector<double> w(n), x(n), y(n);
    gsl_rng_env_setup();
    r = gsl_rng_alloc(gsl_rng_default);

    // ...
```

GSL bSpline Example: C++ (2/6)

```
        for (i = 0; i < n; ++i) {
        double sigma;
        double xi = (15.0 / (N - 1)) * i;
        double yi = cos(xi) * exp(-0.1 * xi);
        sigma = 0.1 * yi;
        dy = gsl_ran_gaussian(r, sigma);
        yi += dy;
        gsl_vector_set(x, i, xi);
        gsl_vector_set(y, i, yi);
        gsl_vector_set(w, i, 1.0 / (sigma * sigma));
    }
    Rcpp::DataFrame res = Rcpp::DataFrame::create(Rcpp::Named("x") = x,
                                                  Rcpp::Named("y") = y,
                                                  Rcpp::Named("w") = w);
    x.free();
    y.free();
    w.free();
    gsl_rng_free(r);
    return(res);
}
```

GSL bSpline Example: C++ (3/6)

```
// [[Rcpp::export]]
Rcpp::List fitData(Rcpp::DataFrame ds) {
    const size_t ncoeffs = NCOEFFS;
    const size_t nbreak = NBREAK;
    const size_t n = N;
    size_t i, j;

    Rcpp::DataFrame D(ds);          // construct the data.frame object
    RcppGSL::vector<double> y = D["y"]; // access columns by name,
    RcppGSL::vector<double> x = D["x"]; // assigning to GSL vectors
    RcppGSL::vector<double> w = D["w"];

    gsl_bspline_workspace *bw;
    gsl_vector *B;
    gsl_vector *c;
    gsl_matrix *X, *cov;
    gsl_multifit_linear_workspace *mw;
    double chisq, Rsq, dof, tss;
```

GSL bSpline Example: C++ (4/6)

```
bw = gsl_bspline_alloc(4, nbreak);      // allocate a cubic bspline workspace (k = 4)
B = gsl_vector_alloc(ncoeffs);
X = gsl_matrix_alloc(n, ncoeffs);
c = gsl_vector_alloc(ncoeffs);
cov = gsl_matrix_alloc(ncoeffs, ncoeffs);
mw = gsl_multifit_linear_alloc(n, ncoeffs);
gsl_bspline_knots_uniform(0.0, 15.0, bw);   // use uniform breakpoints on [0, 15]

for (i = 0; i < n; ++i) {           // construct the fit matrix X
    double xi = gsl_vector_get(x, i);
    gsl_bspline_eval(xi, B, bw);        // compute B_j(xi) for all j
    for (j = 0; j < ncoeffs; ++j) {     // fill in row i of X
        double Bj = gsl_vector_get(B, j);
        gsl_matrix_set(X, i, j, Bj);
    }
}
```

GSL bSpline Example: C++ (5/6)

```
gsl_multifit_wlinear(X, w, y, c, cov, &chisq, mw);  // do the fit
dof = n - ncoeffs;
tss = gsl_stats_wtss(w->data, 1, y->data, 1, y->size);
Rsq = 1.0 - chisq / tss;
Rcpp::NumericVector FX(151), FY(151);   // output the smoothed curve
double xi, yi, yerr;

for (xi = 0.0, i=0; xi < 15.0; xi += 0.1, i++) {
    gsl_bspline_eval(xi, B, bw);
    gsl_multifit_linear_est(B, c, cov, &yi, &yerr);
    FX[i] = xi;
    FY[i] = yi;
}
```

GSL bSpline Example: C++ (6/6)

```
    Rcpp::List res =
      Rcpp::List::create(Rcpp::Named("X")=FX,
                         Rcpp::Named("Y")=FY,
              Rcpp::Named("chisqdof")=Rcpp::wrap(chisq/dof),
              Rcpp::Named("rsq")=Rcpp::wrap(Rsq));
    gsl_bspline_free(bw);
    gsl_vector_free(B);
    gsl_matrix_free(X);
    gsl_vector_free(c);
    gsl_matrix_free(cov);
    gsl_multifit_linear_free(mw);
    y.free();
    x.free();
    w.free();
    return(res);
}
```

GSL bSpline Example

```
Rcpp::sourceCpp("bSpline.cpp")

dat <- genData()                    # generate the data
fit <- fitData(dat)                 # fit the model

X <- fit[["X"]]                     # extract vectors
Y <- fit[["Y"]]

par(mar=c(3,3,1,1))
plot(dat[,"x"], dat[,"y"], pch=19, col="#00000044")
lines(X, Y, col="orange", lwd=2)
```

GSL bSpline Example