Seamless R and C++ Integration with Rcpp: Part 2 – RcppArmadillo Examples

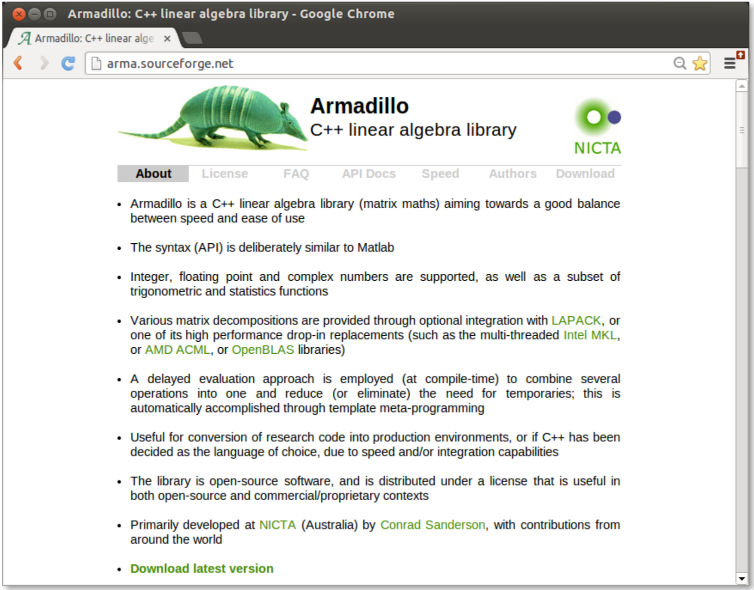
Dirk Eddelbuettel
dirk.eddelbuettel@R-Project.org

Statistical Computing Seminar
Booth School of Business, University of Chicago
October 4, 2013

Outline

1. Intro
   - Armadillo
   - Users

Armadillo



- 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?

From `arma.sf.net` and slightly edited

- 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.

What is Armadillo?

From `arma.sf.net` and slightly edited

- 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 at website `http://arma.sf.net`, a technical report (Sanderson, 2010)
- Modern code, building upon and extending from earlier matrix libraries.
- Responsive and active maintainer, frequent updates.
- Used by MLPACK; cf 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 with R packages: just add `LinkingTo: RcppArmadillo, Rcpp` to DESCRIPTION (*i.e.*, no added cost beyond **Rcpp**)
- Data exchange really seamless from R via **Rcpp**
- Frequently updated; documentation includes Eddelbuettel and Sanderson (CSDA, 2013/in press).

Well-know packages using RcppArmadillo

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

2 Simple Examples
- Eigenvalues
- Multivariate Nornal RNGs

Armadillo Eigenvalues

http://gallery.rcpp.org/articles/armadillo-eigenvalues/

```
#include <RcppArmadillo.h>

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

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

Armadillo Eigenvalues

http://gallery.rcpp.org/articles/armadillo-eigenvalues/

```
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.
```

Multivariate Normal RNG Draw

http://gallery.rcpp.org/articles/simulate-multivariate-norn

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

// [[Rcpp::export]]
arma::mat mvrnormArma(int n, arma::vec mu,
                      arma::mat sigma) {
   arma::mat Y = arma::randn(n, sigma.n_cols);
   return arma::repmat(mu, 1, n).t() +
                 Y * arma::chol(sigma);
}
```

Outline

3 Case Study: FastLM

Faster Linear Model with FastLm
Background

- Implementations of ‘fastLm()‘ have been a staple all along the development of **Rcpp**
- The very first version was in response to a question by Ivo Welch on r-help.
- The 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 had since been reimplemented for **RcppArmadillo** and **RcppEigen**.

Faster Linear Model with FastLm

Initial RcppArmadillo `src/fastLm.cpp`

```
#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 and 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

}
```

Faster Linear Model with FastLm

Edited version of RcppArmadillo's `src/fastLm.cpp`

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

Faster Linear Model with FastLm

Newer version of RcppArmadillo's `src/fastLm.cpp`

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

// [[Rcpp::export]]
List fastLm2(const colvec& y, const 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    );

}
```

Faster Linear Model with FastLm

Note on as<>() casting with Armadillo

```
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);
```

Preferable if performance is a concern. Newest **RcppArmadillo** has efficient `const references` too.

Faster Linear Model with FastLm

Performance comparison

Running the script included in the **RcppArmadillo** package:

```
edd@max:~/svn/rcpp/pkg/RcppArmadillo/inst/examples$ r fastLm.r
Loading required package: Rcpp
                       test replications relative elapsed
2          fLmTwoCasts(X, y)         5000    1.000   0.188
3          fLmConstRef(X, y)         5000    1.000   0.188
1           fLmOneCast(X, y)         5000    1.005   0.189
5   fastLmPureDotCall(X, y)         5000    1.064   0.200
4           fastLmPure(X, y)         5000    2.000   0.376
7               lm.fit(X, y)         5000    2.691   0.506
6 fastLm(frm, data = trees)         5000   35.596   6.692
8     lm(frm, data = trees)         5000   44.883   8.438
edd@max:~/svn/rcpp/pkg/RcppArmadillo/inst/examples$
```

Outline

4 Case Study: Kalman Filter
- Setup
- Matlab
- R
- C++
- Performance

Kalman Filter

Setup at Mathworks site

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:

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

Kalman Filter

Basic Matlab Function

```
% Copyright 2010 The MathWorks, Inc.
function y = kalmanfilter(z)
% #codegen
    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 ];
    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
```

Plus a simple wrapper function calling this 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,   # 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)

    ## 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 covriance
    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, 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,] <- kf(t(pos[i,,drop=FALSE]))
  }
  invisible(y)
}
```

Kalman Filter: In C++

Using a simple class

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

Kalman Filter in C++

Trivial to use from R

Given the code from the previous slide, we just add

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

Kalman Filter: Performance

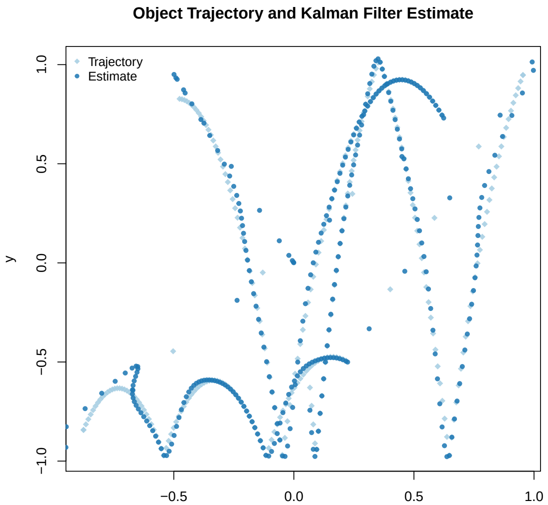
Quite satisfactory relative to R

Even byte-compiled 'better' R version is 66 times slower:

```
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
```

Kalman Filter: Figure

Last but not least we can redo the plot as well



Outline

5 Case Study: Sparse Matrices
- R
- C++
- Example

Sparse Matrices

Growing (but incomplete) support in Armadillo

A nice example for work on R objects.

```
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 Matrices
Representation in R

```
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 Matrices

C++ access

```
#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 Matrices

C++ access

```
sourceCpp('code/sparseEx.cpp')
i <- c(1,3:8)
j <- c(2,9,6:10)
x <- 7 * (1:7)
A <- sparseMatrix(i, j, x = x)
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
```

Outline

6 XPtr

Function Pointers

http://gallery.rcpp.org/articles/passing-cpp-function-

Consider two simple functions modifying a given Armadillo vector:

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

using namespace arma;
using namespace Rcpp;

vec fun1_cpp(const vec& x) {  // a first function
    vec y = x + x;
    return (y);
}

vec fun2_cpp(const vec& x) {  // and a second function
    vec y = 10*x;
    return (y);

}
```

Function Pointers

http://gallery.rcpp.org/articles/passing-cpp-function-

Using a `typedef` to declare an interface to a function taking and returning a vector — and a function returning a function pointer given a string argument

```
typedef vec (*funcPtr)(const vec& x);

// [[Rcpp::export]]
XPtr<funcPtr> putFunPtrInXPtr(std::string fstr) {
    if (fstr == "fun1")
        return(XPtr<funcPtr>(new funcPtr(&fun1_cpp)));
    else if (fstr == "fun2")
        return(XPtr<funcPtr>(new funcPtr(&fun2_cpp)));
    else
        return XPtr<funcPtr>(R_NilValue); // runtime err.: NULL no XPtr

}
```

Function Pointers

http://gallery.rcpp.org/articles/passing-cpp-function-

We then creation function calling the supplied function on a given vector by 'unpacking' the function pointer:

```
// [[Rcpp::export]]
vec callViaXPtr(const vec x, SEXP xpsexp) {
    XPtr<funcPtr> xpfun(xpsexp);
    funcPtr fun = *xpfun;
    vec y = fun(x);
    return (y);
}
```

Function Pointers

http://gallery.rcpp.org/articles/passing-cpp-function-

```r
## get us a function
fun <- putFunPtrInXPtr("fun1")
## and pass it down to C++ to
## have it applied on given vector
callViaXPtr(1:4, fun)

##      [,1]
## [1,]    2
## [2,]    4
## [3,]    6
## [4,]    8
```

Could use same mechanism for user-supplied functions, gradients, or samplers, ...

Outline

7 Resources

Rcpp Resources

Book `http://www.rcpp.org/book`

Gallery `http://gallery.rcpp.org`

Blog `http://dirk.eddelbuettel.com/blog/code/rcpp`

List `http://news.gmane.org/gmane.comp.lang.r.rcpp` and `http://lists.r-forge.r-project.org/mailman/listinfo/rcpp-devel`