RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra[1]

Dirk Eddelbuettel[a], Conrad Sanderson[b,c]

[a]*Debian Project, http://www.debian.org*
[b]*NICTA, PO Box 6020, St Lucia, QLD 4067, Australia*
[c]*Queensland University of Technology (QUT), Brisbane, QLD 4000, Australia*

Abstract

The R statistical environment and language has demonstrated particular strengths for interactive development of statistical algorithms, as well as data modelling and visualisation. Its current implementation has an interpreter at its core which may result in a performance penalty in comparison to directly executing user algorithms in the native machine code of the host CPU. In contrast, the C++ language has no built-in visualisation capabilities, handling of linear algebra or even basic statistical algorithms; however, user programs are converted to high-performance machine code, ahead of execution. A new method avoids possible speed penalties in R by using the Rcpp extension package in conjunction with the Armadillo C++ matrix library. In addition to the inherent performance advantages of compiled code, Armadillo provides an easy-to-use template-based meta-programming framework, allowing the automatic pooling of several linear algebra operations into one, which in turn can lead to further speedups. With the aid of Rcpp and Armadillo, conversion of linear algebra centered algorithms from R to C++ becomes straightforward. The algorithms retains the overall structure as well as readability, all while maintaining a bidirectional link with the host R environment. Empirical timing comparisons of R and C++ implementations of a Kalman filtering algorithm indicate a speedup of several orders of magnitude.

Keywords: Software, R, C++, linear algebra

1. Overview

Linear algebra is a cornerstone of statistical computing and statistical software systems. Various matrix decompositions, linear program solvers, and

---

[1]This vignette corresponds to a paper published in Computational Statistics and Data Analysis. Currently still identical to the paper, this vignette version may over time receive minor updates. For citations, please use **?** as provided by `citation("RcppArmadillo")`. This version corresponds to RcppArmadillo version 0.9.850.1.0 and was typeset on February 10, 2020.

eigenvalue / eigenvector computations are key to many estimation and analysis routines. As generally useful procedures, these are often abstracted and regrouped in specific libraries for linear algebra which statistical programmers have provided for various programming languages and environments.

One such environment and statistical programming language is R (**?**). It has become a tool of choice for data analysis and applied statistics (**?**). While R has particular strengths at fast prototyping and easy visualisation of data, its implementation has an interpreter at its core. In comparison to running user algorithms in the native machine code of the host CPU, the use of an interpreter often results in a performance penalty for non-trivial algorithms that perform elaborate data manipulation (**?**). With user algorithms becoming more complex and increasing in functionality, as well as with data sets which continue to increase in size, the issue of execution speed becomes more important.

The C++ language offers a complementary set of attributes: while it has no built-in visualisation capabilities nor handling of linear algebra or statistical methods, user programs are converted to high-performance machine code ahead of execution. It is also inherently flexible. One key feature is *operator overloading* which allows the programmer to define custom behaviour for mathematical operators such as $+$, $-$, $*$ (**?**). C++ also provides language constructs known as *templates*, originally intended to easily allow the reuse of algorithms for various object types, and later extended to a programming construct in its own right called *template meta-programming* (**??**)

Operator overloading allows mathematical operations to be extended to user-defined objects, such as matrices. This in turn allows linear algebra expressions to be written in a more natural manner (eg. $X = 0.1 * A + 0.2 * B$), rather than the far less readable traditional function call syntax, eg. $X = add(multiply(0.1, A), multiply(0.2, B))$.

Template meta-programming is the process of inducing the C++ compiler to execute, at compile time, Turing-complete programs written in a somewhat opaque subset of the C++ language (**??**). These meta-programs in effect generate further C++ code (often specialised for particular object types), which is finally converted into machine code.

An early and influential example of exploiting both meta-programming and overloading of mathematical operators was provided by the Blitz++ library (**?**), targeted for efficient processing of arrays. Blitz++ employed elaborate meta-programming to avoid the generation of temporary array objects during the evaluation of mathematical expressions. However, the library's capabilities and usage were held back at the time by the limited availability of compilers correctly implementing all the necessary features and nuances of the C++ language.

We present a new method of avoiding the speed penalty in R by using the Rcpp extension package (**???**) in conjunction with the Armadillo C++ linear algebra library (**?**). Similar to Blitz++, Armadillo uses operator overloading and various template meta-programming techniques to attain efficiency. However, it has been written to target modern C++ compilers as well as providing a much larger set of linear algebra operations than Blitz++. R programs augmented to use Armadillo retain the overall structure as well as readability, all while retaining a bidirectional link with the host R environment.

Section 2 provides an overview of Armadillo, followed by its integration with the Rcpp extension package. Section 4 shows an example of an R program and its conversion to C++ via Rcpp and Armadillo. Section 5 discusses an empirical timing comparison between the R and C++ versions before Section 6 concludes.

2. Armadillo

The Armadillo C++ library provides vector, matrix and cube types (supporting integer, floating point and complex numbers) as well as a subset of trigonometric and statistics functions (?). In addition to elementary operations such as addition and matrix multiplication, various matrix factorisations and submatrix manipulation operations are provided. The corresponding application programming interface (syntax) enables the programmer to write code which is both concise and easy-to-read to those familiar with scripting languages such as Matlab and R. Table 1 lists a few common Armadillo functions.

Matrix multiplication and factorisations are accomplished through integration with the underlying operations stemming from standard numerical libraries such as BLAS and LAPACK (?). Similar to how environments such as R are implemented, these underlying libraries can be replaced in a transparent manner with variants that are optimised to the specific hardware platform and/or multi-threaded to automatically take advantage of the now-common multi-core platforms (?).

Armadillo uses a delayed evaluation approach to combine several operations into one and reduce (or eliminate) the need for temporary objects. In contrast

Armadillo function	Description
`X(1,2) = 3`	Assign value 3 to element at location $(1,2)$ of matrix X
`X = A + B`	Add matrices A and B
`X( span(1,2), span(3,4) )`	Provide read/write access to submatrix of X
`zeros(rows [, cols [, slices]]))`	Generate vector (or matrix or cube) of zeros
`ones(rows [, cols [, slices]]))`	Generate vector (or matrix or cube) of ones
`eye(rows, cols)`	Matrix diagonal set to 1, off-diagonal elements set to 0
`repmat(X, row_copies, col_copies)`	Replicate matrix X in block-like manner
`det(X)`	Returns the determinant of matrix X
`norm(X, p)`	Compute the p-norm of matrix or vector X
`rank(X)`	Compute the rank of matrix X
`min(X, dim=0); max(X, dim=0)`	Extremum value of each column of X (row if `dim=1`)
`trans(X)` or `X.t()`	Return transpose of X
`R = chol(X)`	Cholesky decomposition of X such that $R^T R = X$
`inv(X)` or `X.i()`	Returns the inverse of square matrix X
`pinv(X)`	Returns the pseudo-inverse of matrix X
`lu(L, U, P, X)`	LU decomp. with partial pivoting; also `lu(L, U, X)`
`qr(Q, R, X)`	QR decomp. into orthogonal Q and right-triangular R
`X = solve(A, B)`	Solve system $AX = B$ for X
`s = svd(X); svd(U, s, V, X)`	Singular-value decomposition of X

Table 1: Selected Armadillo functions with brief descriptions; see `http://arma.sf.net/docs.html` for more complete documentation. Several optional additional arguments have been omitted here for brevity.

to brute-force evaluations, delayed evaluation can provide considerable performance improvements as well as reduced memory usage. The delayed evaluation machinery is accomplished through template meta-programming (**??**), where the C++ compiler is induced to reason about mathematical expressions at *compile time*. Where possible, the C++ compiler can generate machine code that is tailored for each expression.

As an example of the possible efficiency gains, let us consider the expression $X = A - B + C$, where A, B and C are matrices. A brute-force implementation would evaluate $A - B$ first and store the result in a temporary matrix T. The next operation would be $T + C$, with the result finally stored in X. The creation of the temporary matrix, and using two separate loops for the subtraction and addition of matrix elements is suboptimal from an efficiency point of view.

Through the overloading of mathematical operators, Armadillo avoids the generation of the temporary matrix by first converting the expression into a set of lightweight `Glue` objects, which only store references to the matrices and Armadillo's representations of mathematical expressions (eg. other `Glue` objects). To indicate that an operation comprised of subtraction and addition is required, the exact type of the `Glue` objects is automatically inferred from the given expression through template meta-programming. More specifically, given the expression $X = A - B + C$, Armadillo automatically induces the compiler to generate an instance of the lightweight `Glue` storage object with the following C++ *type*:

```
Glue< Glue<Mat, Mat, glue_minus>, Mat, glue_plus>
```

where `Glue<...>` indicates that `Glue` is a C++ template class, with the items between '<' and '>' specifying template parameters; the outer `Glue<..., Mat, glue_plus>` is the `Glue` object indicating an addition operation, storing a reference to a matrix as well as a reference to another `Glue` object; the inner `Glue<Mat, Mat, glue_minus>` stores references to two matrices and indicates a subtraction operation. In both the inner and outer `Glue`, the type `Mat` specifies that a reference to a matrix object is to be held.

The expression evaluator in Armadillo is then automatically invoked through the "=" operation, which interprets (at compile time) the template parameters of the compound `Glue` object and generates C++ code equivalent to:

```
for(int i=0; i<N; i++) { X[i] = (A[i] - B[i]) + C[i]; }
```

where N is the number of elements in A, B and C, with `A[i]` indicating the i-th element in A. As such, apart from the lightweight `Glue` objects (for which memory is pre-allocated at compile time), no other temporary object is generated, and only one loop is required instead of two. Given a sufficiently advanced C++ compiler, the lightweight `Glue` objects can be optimised away, as they are automatically generated by the compiler and only contain compile-time generated references; the resultant machine code can appear as if the `Glue` objects never existed in the first place.

4

Note that due to the ability of the `Glue` object to hold references to other `Glue` objects, far longer and more complicated operations can be easily accommodated. Further discussion of template meta-programming is beyond the scope of this paper; for more details, the interested reader is referred to **?** as well as **?**. **?** provide a recent application of Armadillo in computer vision and pattern recognition.

3. RcppArmadillo

The RcppArmadillo package (**?**) employs the Rcpp package (**???**) to provide a bidirectional interface between R and C++ at the object level. Using templates, R objects such as vectors and matrices can be mapped directly to the corresponding Armadillo objects.

```
R> library(inline)
R>
R> g <- cxxfunction(signature(vs="numeric"),
+                    plugin="RcppArmadillo", body='
+       arma::vec v = Rcpp::as<arma::vec>(vs);
+       arma::mat op = v * v.t();
+       double ip = arma::as_scalar(v.t() * v);
+       return Rcpp::List::create(Rcpp::Named("outer")=op,
+                                  Rcpp::Named("inner")=ip);
+')
R> g(7:11)
$outer
     [,1] [,2] [,3] [,4] [,5]
[1,]   49   56   63   70   77
[2,]   56   64   72   80   88
[3,]   63   72   81   90   99
[4,]   70   80   90  100  110
[5,]   77   88   99  110  121

$inner
[1] 415
```

Listing 1: Integrating Armadillo-based C++ code via the RcppArmadillo package.

Consider the simple example in Listing 1. Given a vector, the `g()` function returns both the outer and inner products. We load the inline package (**?**), which provides `cxxfunction()` that we use to compile, link and load the C++ code which is passed as the `body` argument. We declare the function signature to contain a single argument named 'vs'. On line five, this argument is used to instantiate an Armadillo column vector object named 'v' (using the templated conversion function `as()` from Rcpp). In lines six and seven, the outer and inner product of the column vector are calculated by appropriately multiplying the vector with its transpose. This shows how the `*` operator for multiplication has been overloaded to provide the appropriate operation for the types implemented by Armadillo. The inner product creates a scalar variable, and in contrast to R

where each object is a vector type (even if of length one), we have to explicitly convert using `as_scalar()` to assign the value to a variable of type `double`.

Finally, the last line creates an R named list type containing both results. As a result of calling `cxxfunction()`, a new function is created. It contains a reference to the native code, compiled on the fly based on the C++ code provided to `cxxfunction()` and makes it available directly from R under a user-assigned function name, here `g()`. The listing also shows how the `Rcpp` and `arma` namespaces are used to disambiguate symbols from the two libraries; the `::` operator is already familiar to R programmers who use the NAMESPACE directive in R in a similar fashion.

The listing also demonstrates how the new function `g()` can be called with a suitable argument. Here we create a vector of five elements, containing values ranging from 7 to 11. The function's output, here the list containing both outer and inner product, is then displayed as it is not assigned to a variable.

This simple example illustrates how R objects can be transferred directly into corresponding Armadillo objects using the interface code provided by Rcpp. It also shows how deployment of RcppArmadillo is straightforward, even for interactive work where functions can be compiled on the fly. Similarly, usage in packages is also uncomplicated and follows the documentation provided with Rcpp (**??**).

4. Kalman Filtering Example

The Kalman filter is ubiquitous in many engineering disciplines as well as in statistics and econometrics (**?**). A recent example of an application is volatility extraction in a diffusion option pricing model (**?**). Even in its simplest linear form, the Kalman filter can provide simple estimates by recursively applying linear updates which are robust to noise and can cope with missing data. Moreover, the estimation process is lightweight and fast, and consumes only minimal amounts of memory as few state variables are required. We discuss a standard example below. The (two-dimensional) position of an object is estimated based on past values. A 6×1 state vector includes X and Y coordinates determining the position, two variables for speed (or velocity) V_X and V_Y relative to the two coordinates, as well as two acceleration variables A_X and A_Y.

We have the positions being updated as a function of the velocity

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

and the velocity being updated as a function of the (unobserved) acceleration:

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

With covariance matrices Q and R for (Gaussian) error terms, the standard Kalman filter estimation involves a linear prediction step resulting in a new predicted state vector, and a new covariance estimate. This leads to a residuals vector and a covariance matrix for residuals which are used to determine the

(optimal) Kalman gain, which is then used to update the state estimate and covariance matrix.

All of these steps involve only matrix multiplication and inversions, making the algorithm very suitable for an implementation in any language which can use matrix expressions. An example for Matlab is provided on the Mathworks website[2] and shown in Listing 2.

```matlab
% 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 + covar.
  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;                        % Comp. estim. measurements
end                                     % of the function
```

Listing 2: Basic Kalman filter in Matlab.

```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]See http://www.mathworks.com/products/matlab-coder/demos.html?file=/products/demos/shipping/coder/coderdemo_kalman_filter.html.

```
10                     2, 6, byrow=TRUE)
          Q <- diag(6)
12        R <- 1000 * diag(2)

14        xprd <- A %*% xest           # predicted state and
              covriance
          pprd <- A %*% pest %*% t(A) + Q
16
          S <- H %*% t(pprd) %*% t(H) + R     # estimation
18        B <- H %*% t(pprd)
          kalmangain <- t(solve(S, B))
20
          ## estimated state and covariance, assign to vars in
               parent env
22        xest <<- xprd + kalmangain %*% (z - H %*% xprd)
          pest <<- pprd - kalmangain %*% H %*% pprd
24
          ## compute the estimated measurements
26        y <- H %*% xest
      }
28    xest <- matrix(0, 6, 1)
      pest <- matrix(0, 6, 6)
30
      N <- nrow(pos)
32    y <- matrix(NA, N, 2)
      for (i in 1:N) {
34        y[i,] <- kalmanfilter(t(pos[i,,drop=FALSE]))
      }
36    invisible(y)
}
```

Listing 3: Basic Kalman filter in R (referred to as *FirstKalmanR*).

A straightforward R implementation can be written as a close transcription of the Matlab version; we refer to this version as `FirstKalmanR`. It is shown in Listing 3. A slightly improved version (where several invariant statements are moved out of the repeatedly-called function) is provided in Listing 4 on page 9 showing the function `KalmanR`. The estimates of the state vector and its covariance matrix are updated iteratively. The Matlab implementation uses two variables declared 'persistent' for this. In R, which does not have such an attribute for variables, we store them in the enclosing environment of the outer function `KalmanR`, which contains an inner function `kalmanfilter` that is called for each observation.

Armadillo provides efficient vector and matrix classes to implement the Kalman filter. In Listing 5 on page 10, we show a simple C++ class containing a basic constructor as well as one additional member function. The constructor can be used to initialise all variables as we are guaranteed that the code in the class constructor will be executed exactly once when this class is instantiated. A class also makes it easy to add 'persistent' local variables, which is a feature

we need here. Given such a class, the estimation can be accessed from R via a short and simple routine such as the one shown in Listing 6.

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

    kalmanfilter <- 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)

        kalmangain <- t(solve(S, B))

        ## estimated state and covariance
        ## assigned to vars in parent env
        xest <<- xprd + kalmangain %*% (z - H %*% xprd)
        pest <<- pprd - kalmangain %*% 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,] <- kalmanfilter(t(pos[i,,drop=FALSE]))
    }
    invisible(Y)
}
```

Listing 4: An improved Kalman filter implemented in R (referred to as *KalmanR*).

```
  using namespace arma;
2
  class Kalman {
4 private:
      mat A, H, Q, R, xest, pest;
6     double dt;

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

20
      // sole member function: estimate model
22    mat estimate(const mat & Z) {
          unsigned int n = Z.n_rows, k = Z.n_cols;
24        mat Y = zeros(n, k);
          mat xprd, pprd, S, B, kalmangain;
26        colvec z, y;

28        for (unsigned int i = 0; i<n; i++) {
              z = Z.row(i).t();
30            // predicted state and covariance
              xprd = A * xest;
32            pprd = A * pest * A.t() + Q;
              // estimation
34            S = H * pprd.t() * H.t() + R;
              B = H * pprd.t();
36            kalmangain = (solve(S, B)).t();
              // estimated state and covariance
38            xest = xprd + kalmangain * (z - H * xprd);
              pest = pprd - kalmangain * H * pprd;
40            // compute the estimated measurements
              y = H * xest;
42            Y.row(i) = y.t();
          }
44        return Y;
      }
46 };
```

Listing 5: A Kalman filter class in C++, using Armadillo classes.

```
R> kalmanSrc <- '
2 +  mat Z = as<mat>(ZS);       // passed from R
  +  Kalman K;
4 +  mat Y = K.estimate(Z);
  +  return wrap(Y);'
6 R> KalmanCpp <- cxxfunction(signature(ZS="numeric"),
  +                           body=kalmanSrc, include=
     kalmanClass,
8 +                           plugin="RcppArmadillo")
```

Listing 6: A Kalman filter function implemented in a mixture of R and C++ code, using the RcppArmadillo package to embed Armadillo based C++ code (using the *Kalman* class from Listing 5) within R code. The resulting program is referred to as *KalmanCpp*.

The content of Listing 5 is assigned to a variable `kalmanClass` which (on line seven) is passed to the `include=` argument. This provides the required class declaration and definition. The four lines of code in lines two to five, assigned to `kalmanSrc`, provide the function body required by `cxxfunction()`. From both these elements and the function signature argument, `cxxfunction()` creates a very simple yet efficient C++ implementation of the Kalman filter which we can access from R. Given a vector of observations Z, it estimates a vector of position estimates Y.

This is illustrated in Figure 1 which displays the original object trajectory (using light-coloured square symbols) as well as the position estimates provided by the Kalman filter (using dark-coloured circles). This uses the same dataset provided by the Mathworks for their example; the data is believed to be simulated.

We note that this example is meant to be illustrative and does not attempt to provide a reference implementation of a Kalman filter. R contains several packages providing various implementations, as discussed in the survey provided by **?**.

5. Empirical Speed Comparison

```
R> require(rbenchmark)
2 R> require(compiler)
  R>
4 R> FirstKalmanRC <- cmpfun(FirstKalmanR)
  R> KalmanRC <- cmpfun(KalmanR)
6 R>
  R> ## Read data, ensure identical results
8 R> pos <- as.matrix(read.table("pos.txt", header=FALSE,
  +                               col.names=c("x","y")))
10 R> stopifnot(all.equal(KalmanR(pos), KalmanRC(pos)),
  +            all.equal(KalmanR(pos), KalmanCpp(pos)),
```
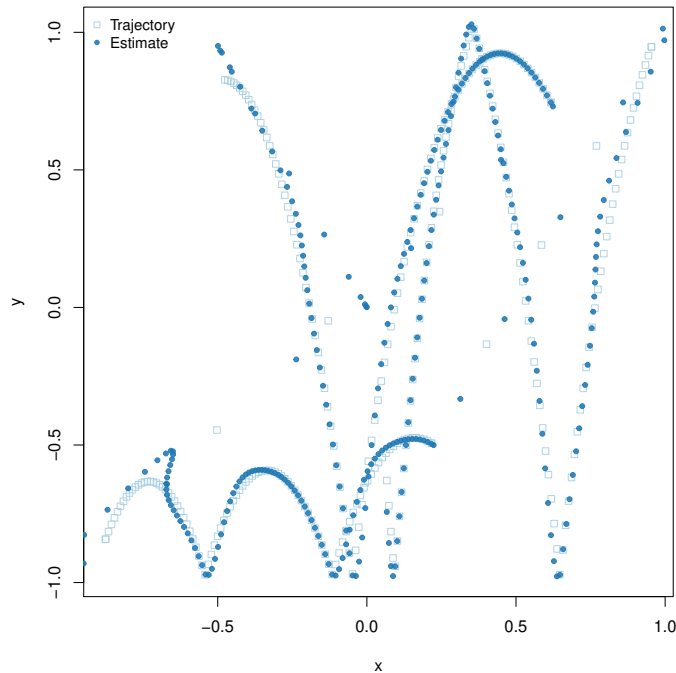
Figure 1: An example of object trajectory and the corresponding Kalman filter estimate.

```
12 +                 all.equal(FirstKalmanR(pos), FirstKalmanRC(pos)
      ),
   +                 all.equal(KalmanR(pos), FirstKalmanR(pos)))
14 R>
   R> res <- benchmark(KalmanR(pos), KalmanRC(pos),
16 +                   FirstKalmanR(pos), FirstKalmanRC(pos),
   +                   KalmanCpp(pos),
18 +                   columns = c("test", "replications",
   +                               "elapsed", "relative"),
20 +                   order="relative",
   +                   replications=100)
```

Listing 7: R code for timing comparison of Kalman filter implementations.

Listing 7 contains the code for creating a simple benchmarking exercise. It compares several functions for implementing the Kalman filter, all executed within the R environment. Specifically, we examine the initial R version `FirstKalmanR` shown in Listing 3, a refactored version `KalmanR` shown in Listing 4, an improved version[3] due to an anonymous referee (not shown, but included in the package),

---

[3]The improved version replaces explicit transpose and multiplication with the `crossprod` function.

Implementation	Time in seconds	Relative to best solution
KalmanCpp	0.73	1.0
KalmanRimpC	21.10	29.0
KalmanRimp	22.01	30.2
KalmanRC	28.64	39.3
KalmanR	31.30	43.0
FirstKalmanRC	49.40	67.9
FirstKalmanR	64.00	88.0

Table 2: Performance comparison of various implementations of a Kalman filter. KalmanCpp is the RcppArmadillo based implementation in C++ shown in Listings 5 and 6. KalmanRimp is an improved version supplied by an anonymous referee which uses the `crossprod` function instead of explicit transpose and multiplication. KalmanR is the R implementation shown in Listing 4; FirstKalmanR is a direct translation of the original Matlab implementation shown in Listing 3. In all cases, the trailing 'C' denotes a byte-compiled variant of the corresponding R code. Timings are averaged over 500 replications. The comparison was made using R version 2.15.2, Rcpp version 0.10.2 and RcppArmadillo version 0.3.6.1 on Ubuntu 12.10 running in 64-bit mode on a 2.67 GHz Intel i7 processor.

as well as byte-compiled versions (designated with a trailing 'C') created by using the byte-code compiler introduced with R version 2.13.0 (**?**). Finally, the C++ version shown in Listings 5 and 6 is used. Also shown are the R statements for creating the byte-compiled variants via calls to `cmpfun()`. This is followed by a test to ensure that all variants provide the same results. Next, the actual benchmark is executed before the result is displayed.

The results are shown in Table 2. Optimising and improving the R code has merits: we notice a steady improvement from the slowest R version to the fastest R version. Byte-compiling R code provides an additional performance gain which is more pronounced for the slower variant than the fastest implementation in R. However, the `KalmanCpp` function created using RcppArmadillo clearly outperforms all other variants, underscoring the principal point of this paper.

These results are consistent with the empirical observations made by **?**, who also discuss several reasons for the slow speed of R compared to the C language, a close relative of C++.

6. Conclusion

This paper introduced the RcppArmadillo package for use within the R statistical environment. By using the Rcpp interface package, RcppArmadillo brings the speed of C++ along with the highly expressive Armadillo linear algebra library to the R language. A small example implementing a Kalman filter illustrated two key aspects. First, orders of magnitude of performance gains can be obtained by deploying C++ code along with R. Second, the ease of use and readability of the corresponding C++ code is similar to the R code from which it was derived.

This combination makes RcppArmadillo a compelling tool in the arsenal of applied researchers deploying computational methods in statistical computing and data analysis. As of early-2013, about 30 R packages on CRAN deploy RcppArmadillo[4], showing both the usefulness of Armadillo and its acceptance by the R community.

Acknowledgements

---

[4]See http://cran.r-project.org/package=RcppArmadillo for more details.