# RcppArmadillo: Easily Extending R with High-Performance C++ Code

Dirk Eddelbuettel Conrad Sanderson

RcppArmadillo version 0.3.2.3 as of July 1, 2012

**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. We present a new method of avoiding 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. We demonstrate that with the aid of Rcpp and Armadillo, conversion of linear algebra centered algorithms from R to C++ becomes straightforward, with the algorithms retaining 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.

## 1 Overview

Linear algebra is a cornerstone of statistical computing and statistical software systems. Various matrix decompositions, linear program solvers, and eigenvalue / eigenvector computations are key to many estimation and analysis routines. As generally useful procedure, 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 (R Development Core Team, 2012). It has become a tool of choice for data analysis and applied work in statistics (Morandat, Hill, Osvald, and Vitek, 2012). While R has particular strengths at fast prototyping and easy visualisation of data, its current 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 (Morandat et al., 2012). 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 $+$, $-$, $*$ (Meyers, 2005). 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* (Vandevoorde and Josuttis, 2002; Abrahams and Gurtovoy, 2004)

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 (Vandevoorde and Josuttis, 2002; Abrahams and Gurtovoy, 2004). 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 (Veldhuizen, 1998), 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.

In this paper we present a new method of avoiding the speed penalty in R: using the Rcpp extension package (Eddelbuettel and François, 2011, 2012) in conjunction with the Armadillo C++ linear algebra library (Sanderson, 2010). In a similar manner 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.

We continue the paper as follows. In Section 2 we provide an overview of the Armadillo C++ library, followed by its integration with the Rcpp extension package in Section 3. In Section 4 we provide an example of an R program and its conversion to C++ via the use of Rcpp and Armadillo. Section 5 provides an empirical timing comparison between the R and C++ versions. We conclude the paper in Section 6.

## 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 (Sanderson, 2010). In addition to elementary operations such as addition and matrix multiplication, various matrix factorisations and other commonly-used functions are provided. The corresponding application programming interface (syntax) enables the programmer to write code which is both concise yet 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 (Demmel, 1997). 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 (Kurzak, Bader, and Dongarra, 2010).

Armadillo uses a delayed evaluation approach to combine several operations into one and reduce (or eliminate) the need for temporary objects. In contrast to brute-force evaluations, delayed evaluation can provide considerable performance improvements as well as reduced memory usage. The delayed evaluation machinery accomplished through template meta-programming (Vandevoorde and Josuttis, 2002; Abrahams and Gurtovoy, 2004), 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

| **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.

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 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.

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 is of template meta-programming is beyond the scope of this paper; for more details, the interested reader is referred to Vandevoorde and Josuttis (2002) as well as Abrahams and Gurtovoy (2004).

## 3 RcppArmadillo

The RcppArmadillo package (François, Eddelbuettel, and Bates, 2012) employs the Rcpp package (Eddelbuettel and François, 2011, 2012) 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.

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 (Sklyar, Murdoch, Smith, Eddelbuettel, and François, 2010), 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 six, this argument is used to instantiate an Armadillo column vector object named '`v`' (using the templated conversion function `as()` from Rcpp). In lines seven and eight, 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`.

```
 1 R>
   R> library(inline)
 3 R>
   R> g <- cxxfunction(signature(vs="numeric"),
 5 +                  plugin="RcppArmadillo", body='
   +     arma::vec v = Rcpp::as<arma::vec>(vs);
 7 +     arma::mat op = v * v.t();
   +     double ip = arma::as_scalar(v.t() * v);
 9 +     return Rcpp::List::create(Rcpp::Named("outer")=op,
   +                               Rcpp::Named("inner")=ip);
11 +')
   R>
13 R> g(7:11)
   $outer
15       [,1] [,2] [,3] [,4] [,5]
   [1,]   49   56   63   70   77
17 [2,]   56   64   72   80   88
   [3,]   63   72   81   90   99
19 [4,]   70   80   90  100  110
   [5,]   77   88   99  110  121
21
   $inner
23 [1] 415

25 R>
```

Listing 1: Example of integrating Armadillo based C++ code within R via the RcppArmadillo package.

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 different 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 (Eddelbuettel and François, 2012).

As of mid-2012, there are nineteen R packages on CRAN which deploy RcppArmadillo[1], showing both the usefulness of RcppArmadillo and its acceptance by the R community.

## 4 Kalman Filtering Example

The Kalman filter is ubiquitous in many engineering disciplines as well as in statistics and econometrics (Tusell, 2011). Even 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 $6 \times 1$ 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.

[1]See `http://cran.r-project.org/web/packages/RcppArmadillo/`

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

A straightforward R implementation can be written as a close transcription of the Matlab version; we refer to this version as `FirstKalmanR` but have omitted it here for brevity. A slightly improved version (where several invariant statements are moved out of the repeatedly-called function) is provided in Listing 2 on page 8 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 3 on page 9, 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 4 below.

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

Listing 4: A Kalman filter function implemented in a mixture of R and C++ code, using the Rcpp package to embed Armadillo based C++ code (using the *Kalman* class from Listing 3) within R code.

The content of Listing 3 is assigned to a variable `kalmanClass` which (on line ten) 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

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

```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 2: A Kalman filter implemented in R.

```cpp
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);
    }

    // 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;
    }
};
```

Listing 3: A Kalman filter class in C++, using matrix and vector classes from Armadillo.

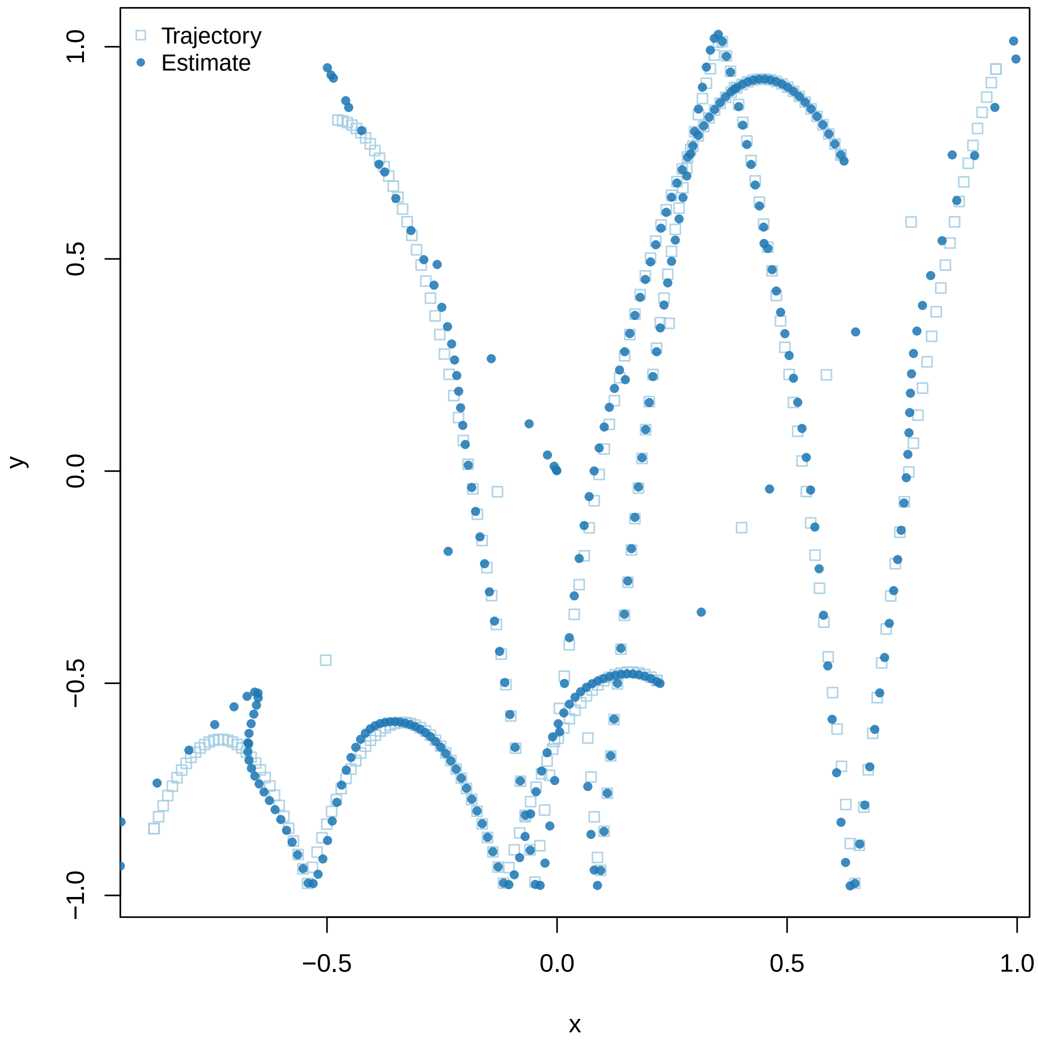

Figure 1: An example of object trajectory and the corresponding Kalman filter estimate.

symbols) as well as the position estimates provided by the Kalman filter (using dark-coloured circles).

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 Tusell (2011).

## 5 Empirical Speed Comparison

Listing 5 contains a simple benchmarking exercise. It compares five unique functions for implementing the Kalman filter, all executed within the R environment. Specifically, we examine both the initial R version `FirstKalmanR` (mentioned in Section 4) as well the refactored version `KalmanR` shown in Listing 2. Using the byte-code compiler introduced with R version 2.13.0 (Tierney, 2012), we also (byte-)compile these function creating the variants designated with a trailing 'C'. Finally, the C++ version shown in Listings 3 and 4 is used.

```
R> require(rbenchmark)
R> require(compiler)
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>
```

Listing 5: R code for timing comparison of various implementations of the Kalman filter.

Listing 5 shows the R statements for creating the byte-compiling variants via simple calls to `cmdfun()`. This is followed by a test to ensure that all variants provide equivalent results. Next, the actual benchmark is executed with one-hundred replications of each variant before the result is displayed.

The results are shown in Table 2. Byte-compiling R code provides a modest but noticeable performance gain. However, the `KalmanCpp` function created using RcppArmadillo clearly outperforms all other variants; the KalmanRC implementation in pure R is slower by several orders of magnitude. These results are consistent with the empirical observations made by Morandat et al. (2012), who also discuss several reasons for the slow speed of R compared to the C language, a close relative of C++.

| **Implementation** | **Time in seconds** | **Relative to best solution** |
|---|---:|---:|
| KalmanCpp | 0.098 | 1.00 |
| KalmanRC | 5.579 | 56.93 |
| KalmanR | 5.807 | 59.26 |
| FirstKalmanRC | 8.196 | 83.63 |
| FirstKalmanR | 8.686 | 88.63 |

Table 2: Performance comparison of various implementations of a Kalman filter. KalmanCpp is the RcppArmadillo based implementation in C++ shown in Listings 3 and 4. KalmanR is the R implementation shown in Listing 2; KalmanRC is the byte-compiled version of KalmanR. FirstKalmanR is a direct translation of the original Matlab implementation mentioned in Section 4; FirstKalmanRC is the corresponding byte-compiled version. Timings are averaged over 100 replications. The comparison was made using R version 2.15.1, Rcpp version 0.9.12 and RcppArmadillo version 0.3.2.0 on Ubuntu 12.04 running in 64-bit mode on a 2.67 GHz Intel i7 processor.

## 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.

## Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy, as well as the Australian Research Council through the ICT Centre of Excellence program.

Adam M. Johansen provided helpful comments on an earlier draft.

## References

David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison-Wesley Professional, 2004.

James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997. ISBN 978-0898713893.

Dirk Eddelbuettel and Romain François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL `http://www.jstatsoft.org/v40/i08/`.

Dirk Eddelbuettel and Romain François. *Rcpp: Seamless R and C++ Integration*, 2012. URL `http://CRAN.R-Project.org/package=Rcpp`. R package version 0.9.10.

Romain François, Dirk Eddelbuettel, and Douglas Bates. *RcppArmadillo: Rcpp integration for Armadillo templated linear algebra library*, 2012. URL `http://CRAN.R-Project.org/package=RcppArmadillo`. R package version 0.3.0.1.

Jakub Kurzak, David A. Bader, and Jack Dongarra, editors. *Scientific Computing with Multicore and Accelerators*. CRC Press, 2010. ISBN 978-1439825365.

Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, third edition, 2005. ISBN 978-0321334879.

Floral Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language. In *ECOOP 2012: Proceedings of European Conference on Object-Oriented Programming*, 2012.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL `http://www.R-project.org/`. ISBN 3-900051-07-0.

Conrad Sanderson. Armadillo: An open source C++ algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010. URL `http://arma.sourceforge.net`.

Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel, and Romain François. *inline: Inline C, C++, Fortran function calls from R*, 2010. URL `http://CRAN.R-Project.org/package=inline`. R package version 0.3.8.

Luke Tierney. A byte-code compiler for R. Manuscript, Department of Statistics and Actuarial Science, University of Iowa, 2012. URL `www.stat.uiowa.edu/~luke/R/compiler/compiler.pdf`.

Fernando Tusell. Kalman filtering in R. *Journal of Statistical Software*, 39(2):1–27, 2011. URL `http://www.jstatsoft.org/v39/i02`.

David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

Todd L. Veldhuizen. Arrays in Blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag. ISBN 3-540-65387-2.