An Introduction to **RcppEigen**

Douglas Bates

RcppEigen version 0.1.3 as of October 27, 2011

Abstract

The **RcppEigen** package provides access from R to the **Eigen** C++ template library for numerical linear algebra. **Rcpp** (Eddelbuettel and François, 2011b) classes and specializations of the C++ templated functions `as` and `wrap` from **Rcpp** provide the "glue" for passing objects from R to C++ and back.

1 Introduction

As stated in the **Rcpp** (Eddelbuettel and François, 2011a) vignette, "Extending **Rcpp**"

> **Rcpp** facilitates data interchange between R and C++ through the templated functions `Rcpp::as` (for conversion of objects from R to C++) and `Rcpp::wrap` (for conversion from C++ to R).

The **RcppEigen** package provides the header files composing the **Eigen** C++ template library and implementations of `Rcpp::as` and `Rcpp::wrap` for the C++ classes defined in **Eigen**.

The **Eigen** classes themselves provide high-performance, versatile and comprehensive representations of dense and sparse matrices and vectors, as well as decompositions and other functions to be applied to these objects. In the next section we introduce some of these classes and show how to interface to them from R.

2 Eigen classes

Eigen (`http://eigen.tuxfamily.org`) is a C++ template library providing classes for many forms of matrices, vectors, arrays and decompositions. These classes are flexible and comprehensive allowing for both high performance and well structured code representing high-level operations. C++ code based on Eigen is often more like R code, working on the "whole object", than compiled code in other languages where operations often must be coded in loops.

As in many C++ template libraries using template meta-programming (Abrahams and Gurtovoy, 2004), the templates themselves can be very complicated. However, **Eigen** provides typedef's for common classes that correspond to R matrices and vectors, as shown in Table 1. We will use these typedef's throughout this document.

The C++ classes shown in Table 1 are in the `Eigen` namespace, which means that they must be written as `Eigen::MatrixXd`. However, if we preface our use of these class names with a declaration like

```
using Eigen::MatrixXd;
```

we can use these names without the qualifier.

Table 1: Correspondence between R matrix and vector types and classes in the `Eigen` namespace.

R object type	**Eigen** class typedef
numeric matrix	`MatrixXd`
integer matrix	`MatrixXi`
complex matrix	`MatrixXcd`
numeric vector	`VectorXd`
integer vector	`VectorXi`
complex vector	`VectorXcd`
`Matrix::dgCMatrix`	`SparseMatrix<double>`

2.1 Mapped matrices in Eigen

Storage for the contents of matrices from the classes shown in Table 1 is allocated and controlled by the class constructors and destructors. Creating an instance of such a class from an R object involves copying its contents. An alternative is to have the contents of the R matrix or vector mapped to the contents of the object from the Eigen class. For dense matrices we use the Eigen templated class `Map`. For sparse matrices we use the Eigen templated class `MappedSparseMatrix`.

We must, of course, be careful not to modify the contents of the R object in the C++ code. A recommended practice is always to declare mapped objects as **const**.

2.2 Arrays in Eigen

For matrix and vector classes **Eigen** overloads the `'*'` operator to indicate matrix multiplication. Occasionally we want component-wise operations instead of matrix operations. The `Array` templated classes are used in **Eigen** for component-wise operations. Most often we use the `array()` method for Matrix or Vector objects to create the array. On those occasions when we wish to convert an array to a matrix or vector object we use the `matrix()` method.

2.3 Structured matrices in Eigen

There are **Eigen** classes for matrices with special structure such as symmetric matrices, triangular matrices and banded matrices. For dense matrices, these special structures are described as "views", meaning that the full dense matrix is stored but only part of the matrix is used in operations. For a symmetric matrix we need to specify whether the lower triangle or the upper triangle is to be used as the contents, with the other triangle defined by the implicit symmetry.

3 Some simple examples

C++ functions to perform simple operations on matrices or vectors can follow a pattern of:

1. Map the R objects passed as arguments into Eigen objects.
2. Create the result.
3. Return `Rcpp::wrap` applied to the result.

An idiom for the first step is

```
using Eigen::Map;
using Eigen::MatrixXd;
using Rcpp::as;

const Map<MatrixXd>  A(as<Map<MatrixXd> >(AA));
```

where `AA` is the name of the R object (called an `SEXP` in C and C++) passed to the C++ function.

The `cxxfunction` from the **inline** (Sklyar, Murdoch, Smith, Eddelbuettel, and François, 2010) package for R and its **RcppEigen** plugin provide a convenient way of developing and debugging the C++ code. For actual production code we generally incorporate the C++ source code files in a package and include the line `LinkingTo: Rcpp, RcppEigen` in the package's `DESCRIPTION` file. The `RcppEigen.package.skeleton` function provides a quick way of generating the skeleton of a package using **RcppEigen** facilities.

The `cxxfunction` with the `"Rcpp"` or `"RcppEigen"` plugins has the `as` and `wrap` functions already defined as `Rcpp::as` and `Rcpp::wrap`. In the examples below we will omit these declarations. Do remember that you will need them in C++ source code for a package.

The first few examples are simply for illustration as the operations shown could be more effectively performed directly in R. We do compare the results from **Eigen** to those from the direct R results.

3.1 Transpose of an integer matrix

We create a simple matrix of integers

```
> (A <- matrix(1:6, ncol=2))
```

Listing 1: transCpp: Transpose a matrix of integers

```
using Eigen::Map;
using Eigen::MatrixXi;
                // Map the integer matrix AA from R
const Map<MatrixXi>  A(as<Map<MatrixXi> >(AA));
                // evaluate and return the transpose of A
const MatrixXi      At(A.transpose());
return wrap(At);
```

Listing 2: prodCpp: Product and cross-product of two matrices

```
using Eigen::Map;
using Eigen::MatrixXi;
const Map<MatrixXi>    B(as<Map<MatrixXi> >(BB));
const Map<MatrixXi>    C(as<Map<MatrixXi> >(CC));
return List::create(_["B %*% C"]         = B * C,
                    _["crossprod(B, C)"] = B.adjoint() * C);
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
> str(A)
```

```
 int [1:3, 1:2] 1 2 3 4 5 6
```

and, in Listing 1, use the `transpose()` method for the `Eigen::MatrixXi` class to return its transpose. The R matrix in the `SEXP AA` is mapped to an `Eigen::MatrixXi` object then the matrix `At` is constructed from its transpose and returned to R. We check that it works as intended.

```
> ftrans <- cxxfunction(signature(AA="matrix"), transCpp, plugin="RcppEigen")
> (At <- ftrans(A))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
> stopifnot(all.equal(At, t(A)))
```

For numeric or integer matrices the `adjoint()` method is equivalent to the `transpose()` method. For complex matrices, the adjoint is the conjugate of the transpose. In keeping with the conventions in the **Eigen** documentation we will, in what follows, use the `adjoint()` method to create the transpose of numeric or integer matrices.

3.2 Products and cross-products

As mentioned in Sec. 2.2, the '`*`' operator performs matrix multiplication on `Eigen::Matrix` or `Eigen::Vector` objects. The C++ code in Listing 2 produces

```
> fprod <- cxxfunction(signature(BB = "matrix", CC = "matrix"), prodCpp, "RcppEigen")
> B <- matrix(1:4, ncol=2); C <- matrix(6:1, nrow=2)
> str(fp <- fprod(B, C))
```

```
List of 2
 $ B %*% C        : int [1:2, 1:3] 21 32 13 20 5 8
 $ crossprod(B, C): int [1:2, 1:3] 16 38 10 24 4 10
```

```
> stopifnot(all.equal(fp[[1]], B %*% C), all.equal(fp[[2]], crossprod(B, C)))
```

Notice that the `create` method for the **Rcpp** class `List` implicitly applies `Rcpp::wrap` to its arguments.

Listing 3: crossprodCpp: Cross-product and transposed cross-product of a single matrix

```
using Eigen::Map;
using Eigen::MatrixXi;
using Eigen::Lower;

const Map<MatrixXi> A(as<Map<MatrixXi> >(AA));
const int            m(A.rows()), n(A.cols());
MatrixXi            AtA(MatrixXi(n, n).setZero().
                        selfadjointView<Lower>().rankUpdate(A.adjoint()));
MatrixXi            AAt(MatrixXi(m, m).setZero().
                        selfadjointView<Lower>().rankUpdate(A));

return List::create(_["crossprod(A)"]  = AtA,
                    _["tcrossprod(A)"] = AAt);
```

3.3 Crossproduct of a single matrix

As shown in the last example, the R function `crossprod` calculates the product of the transpose of its first argument with its second argument. The single argument form, `crossprod(X)`, evaluates $\boldsymbol{X}'\boldsymbol{X}$. We could, of course, calculate this product as

```
> t(X) %*% X
```

but `crossprod(X)` is roughly twice as fast because the result is known to be symmetric and only one triangle needs to be calculated. The function `tcrossprod` evaluates `crossprod(t(X))` without actually forming the transpose.

To express these calculations in Eigen we create a `SelfAdjointView`, which is a dense matrix of which only one triangle is used, the other triangle being inferred from the symmetry. ("Self-adjoint" is equivalent to symmetric for non-complex matrices.)

The **Eigen** class name is `SelfAdjointView`. The method for general matrices that produces such a view is called `selfadjointView`. Both require specification of either the `Lower` or `Upper` triangle.

For triangular matrices the class is `TriangularView` and the method is `triangularView`. The triangle can be specified as `Lower`, `UnitLower`, `StrictlyLower`, `Upper`, `UnitUpper` or `StrictlyUpper`.

For self-adjoint views the `rankUpdate` method adds a scalar multiple of $\boldsymbol{A}\boldsymbol{A}'$ to the current symmetric matrix. The scalar multiple defaults to 1. The code in Listing 3 produces

```
> fcprd <- cxxfunction(signature(AA = "matrix"), crossprodCpp, "RcppEigen")
> str(crp <- fcprd(A))

List of 2
 $ crossprod(A) : int [1:2, 1:2] 14 32 32 77
 $ tcrossprod(A): int [1:3, 1:3] 17 22 27 22 29 36 27 36 45

> stopifnot(all.equal(crp[[1]], crossprod(A)), all.equal(crp[[2]], tcrossprod(A)))
```

To some, the expressions to construct `AtA` and `AAt` in that code fragment are compact and elegant. To others they are hopelessly confusing. If you find yourself in the latter group, you just need to read the expression left to right. So, for example, we construct `AAt` by creating a general integer matrix of size $m \times m$ (where $\boldsymbol{A}$ is $m \times n$), ensure that all its elements are zero, regard it as a self-adjoint (i.e. symmetric) matrix using the elements in the lower triangle, then add $\boldsymbol{A}\boldsymbol{A}'$ to it and convert back to a general matrix form (i.e. the strict lower triangle is copied into the strict upper triangle).

For these products we could use either the lower triangle or the upper triangle as the result will be symmetrized before it is returned.

3.4 Cholesky decomposition of the crossprod

The Cholesky decomposition of the positive-definite, symmetric matrix, $\boldsymbol{A}$, can be written in several forms. Numerical analysts define the "LLt" form as the lower triangular matrix, $\boldsymbol{L}$, such that $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}'$ and the "LDLt" form as a unit lower triangular matrix $\boldsymbol{L}$ and a diagonal matrix $\boldsymbol{D}$ with positive diagonal

Listing 4: cholCpp: Cholesky decomposition of a cross-product

```
using Eigen::Map;
using Eigen::MatrixXd;
using Eigen::LLT;
using Eigen::Lower;

const Map<MatrixXd>   A(as<Map<MatrixXd> >(AA));
const int             n(A.cols());
const LLT<MatrixXd> llt(MatrixXd(n, n).setZero().
                          selfadjointView<Lower>().rankUpdate(A.adjoint()));

return List::create(_["L"] = MatrixXd(llt.matrixL()),
                    _["R"] = MatrixXd(llt.matrixU()));
```

elements such that $\boldsymbol{A} = \boldsymbol{LDL}'$. Statisticians often write the decomposition as $\boldsymbol{A} = \boldsymbol{R}'\boldsymbol{R}$ where $\boldsymbol{R}$ is an upper triangular matrix. Of course, this $\boldsymbol{R}$ is simply the transpose of $\boldsymbol{L}$ from the "LLt" form.

The templated **Eigen** classes for the LLt and LDLt forms are called `LLT` and `LDLT`. In general we would preserve the objects from these classes so that we could use them for solutions of linear systems. For illustration we simply return the matrix $\boldsymbol{L}$ from the "LLt" form.

Because the Cholesky decomposition involves taking square roots we switch to numeric matrices

```
> storage.mode(A) <- "double"
```

before applying the code in Listing 4.

```
> fchol <- cxxfunction(signature(AA = "matrix"), cholCpp, "RcppEigen")
> (ll <- fchol(A))
```

```
$L
         [,1]     [,2]
[1,] 3.741657 0.000000
[2,] 8.552360 1.963961

$R
         [,1]     [,2]
[1,] 3.741657 8.552360
[2,] 0.000000 1.963961
```

```
> stopifnot(all.equal(ll[[2]], chol(crossprod(A))))
```

3.5 Determinant of the cross-product matrix

The "D-optimal" criterion for experimental design chooses the design that maximizes the determinant, $|\boldsymbol{X}'\boldsymbol{X}|$, for the $n \times p$ model matrix (or Jacobian matrix), $\boldsymbol{X}$. The determinant, $|\boldsymbol{L}|$, of the $p \times p$ lower Cholesky factor $\boldsymbol{L}$, defined so that $\boldsymbol{LL}' = \boldsymbol{X}'\boldsymbol{X}$, is the product of its diagonal elements, as is the case for any triangular matrix. By the properties of determinants,

$$|\boldsymbol{X}'\boldsymbol{X}| = |\boldsymbol{LL}'| = |\boldsymbol{L}|\,|\boldsymbol{L}'| = |\boldsymbol{L}|^2$$

Alternatively, if we use the "LDLt" decomposition, $\boldsymbol{LDL}' = \boldsymbol{X}'\boldsymbol{X}$ where $\boldsymbol{L}$ is unit lower triangular and $\boldsymbol{D}$ is diagonal then $|\boldsymbol{X}'\boldsymbol{X}|$ is the product of the diagonal elements of $\boldsymbol{D}$. Because we know that the diagonals of $\boldsymbol{D}$ must be non-negative, we often evaluate the logarithm of the determinant as the sum of the logarithms of the diagonal elements of $\boldsymbol{D}$. Several options are shown in Listing 5.

```
> fdet <- cxxfunction(signature(AA = "matrix"), cholDetCpp, "RcppEigen")
> unlist(ll <- fdet(A))
```

```
       d1        d2        ld
54.000000 54.000000  3.988984
```

Note the use of the `array()` method in the calculation of the log-determinant. Because the `log()` method applies to arrays, not to vectors or matrices, we must create an array from `Dvec` before applying the `log()` method.

Listing 5: cholDetCpp: Determinant of a cross-product using the Cholesky decomposition

```
using Eigen::Lower;
using Eigen::Map;
using Eigen::MatrixXd;
using Eigen::VectorXd;

const Map<MatrixXd>    A(as<Map<MatrixXd> >(AA));
const int              n(A.cols());
const MatrixXd       AtA(MatrixXd(n, n).setZero().
                         selfadjointView<Lower>().rankUpdate(A.adjoint()));
const MatrixXd      Lmat(AtA.llt().matrixL());
const double        detL(Lmat.diagonal().prod());
const VectorXd      Dvec(AtA.ldlt().vectorD());

return List::create(_["d1"] = detL * detL,
                    _["d2"] = Dvec.prod(),
                    _["ld"] = Dvec.array().log().sum());
```

4 Least squares solutions

A common operation in statistical computing is calculating a least squares solution, $\widehat{\boldsymbol{\beta}}$, defined as

$$\widehat{\boldsymbol{\beta}} = \arg\min_{\beta} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2$$

where the model matrix, $\boldsymbol{X}$, is $n \times p$ ($n \geq p$) and $\boldsymbol{y}$ is an n-dimensional response vector. There are several ways based on matrix decompositions, to determine such a solution. We have already seen two forms of the Cholesky decomposition: “LLt” and “LDLt”, that can be used to solve for $\widehat{\boldsymbol{\beta}}$. Other decompositions that can be used are the QR decomposition, with or without column pivoting, the singular value decomposition and the eigendecomposition of a symmetric matrix.

Determining a least squares solution is relatively straightforward. However, in statistical computing we often require additional information, such as the standard errors of the coefficient estimates. Calculating these involves evaluating the diagonal elements of $(\boldsymbol{X}'\boldsymbol{X})^{-1}$ and the residual sum of squares, $\|\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}\|^2$.

4.1 Least squares using the “LLt” Cholesky

Listing 6 shows a calculation of the least squares coefficient estimates (`betahat`) and the standard errors (`se`) through an “LLt” Cholesky decomposition of the crossproduct of the model matrix, $\boldsymbol{X}$. We check that the results from this calculation do correspond to those from the `lm.fit` function in R (`lm.fit` is the workhorse function called by `lm` once the model matrix and response have been evaluated).

```
> lltLS <- cxxfunction(signature(XX = "matrix", yy = "numeric"), lltLSCpp, "RcppEigen")
> data(trees, package="datasets")
> str(lltFit <- with(trees, lltLS(cbind(1, log(Girth)), log(Volume))))

List of 7
 $ coefficients : num [1:2] -2.35 2.2
 $ fitted.values: num [1:31] 2.3 2.38 2.43 2.82 2.86 ...
 $ residuals    : num [1:31] 0.0298 -0.0483 -0.1087 -0.0223 0.0727 ...
 $ s            : num 0.115
 $ df.residual  : int 29
 $ rank         : int 2
 $ Std. Error   : num [1:2] 0.2307 0.0898

> str(lmFit <- with(trees, lm.fit(cbind(1, log(Girth)), log(Volume))))

List of 8
 $ coefficients : Named num [1:2] -2.35 2.2
```

Listing 6: lltLSCpp: Least squares using the Cholesky decomposition

```
using Eigen::LLT;
using Eigen::Lower;
using Eigen::Map;
using Eigen::MatrixXd;
using Eigen::VectorXd;

const Map<MatrixXd>   X(as<Map<MatrixXd> >(XX));
const Map<VectorXd>   y(as<Map<VectorXd> >(yy));
const int             n(X.rows()), p(X.cols());
const LLT<MatrixXd> llt(MatrixXd(p, p).setZero().
                         selfadjointView<Lower>().rankUpdate(X.adjoint()));
const VectorXd  betahat(llt.solve(X.adjoint() * y));
const VectorXd   fitted(X * betahat);
const VectorXd    resid(y - fitted);
const int            df(n - p);
const double          s(resid.norm() / std::sqrt(double(df)));
const VectorXd        se(s * llt.matrixL().solve(MatrixXd::Identity(p, p)).
                         colwise().norm());
return     List::create(_["coefficients"]   = betahat,
                         _["fitted.values"]  = fitted,
                         _["residuals"]      = resid,
                         _["s"]              = s,
                         _["df.residual"]    = df,
                         _["rank"]           = p,
                         _["Std.␣Error"]     = se);
```

```
 ..- attr(*, "names")= chr [1:2] "x1" "x2"
$ residuals    : num [1:31] 0.0298 -0.0483 -0.1087 -0.0223 0.0727 ...
$ effects      : Named num [1:31] -18.2218 2.8152 -0.1029 -0.0223 0.0721 ...
 ..- attr(*, "names")= chr [1:31] "x1" "x2" "" "" ...
$ rank         : int 2
$ fitted.values: num [1:31] 2.3 2.38 2.43 2.82 2.86 ...
$ assign       : NULL
$ qr           :List of 5
 ..$ qr   : num [1:31, 1:2] -5.57 0.18 0.18 0.18 0.18 ...
 ..$ qraux: num [1:2] 1.18 1.26
 ..$ pivot: int [1:2] 1 2
 ..$ tol  : num 1e-07
 ..$ rank : int 2
 ..- attr(*, "class")= chr "qr"
$ df.residual  : int 29
```

```
> for (nm in c("coefficients", "residuals", "fitted.values", "rank"))
+     stopifnot(all.equal(lltFit[[nm]], unname(lmFit[[nm]])))
> stopifnot(all.equal(lltFit[["Std. Error"]],
+                     unname(coef(summary(lm(log(Volume) ~ log(Girth), trees)))[,2])))
```

There are several aspects of the C++ code in Listing 6 worth mentioning. The `solve` method for the `LLT` object evaluates, in this case, $(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$ but without actually evaluating the inverse. The calculation of the residuals, $\boldsymbol{y} - \widehat{\boldsymbol{y}}$, can be written, as in R, as `y - fitted`. (But note that **Eigen** classes do not have a "recycling rule as in R. That is, the two vector operands must have the same length.) The `norm()` method evaluates the square root of the sum of squares of the elements of a vector. Although we don't explicitly evaluate $(\boldsymbol{X}'\boldsymbol{X})^{-1}$ we do evaluate $\boldsymbol{L}^{-1}$ to obtain the standard errors. Note also the use of the `colwise()` method in the evaluation of the standard errors. It applies a method to the columns of a matrix, returning a vector. The **Eigen** `colwise()` and `rowwise()` methods are similar in effect to the `apply` function in R.

Listing 7: QRLSCpp: Least squares using the unpivoted QR decomposition

```
using Eigen::HouseholderQR;

const HouseholderQR<MatrixXd> QR(X);
const VectorXd           betahat(QR.solve(y));
const VectorXd            fitted(X * betahat);
const int                     df(n - p);
const VectorXd                se(QR.matrixQR().topRows(p).triangularView<Upper>().
                                 solve(MatrixXd::Identity(p,p)).rowwise().norm());
```

In the descriptions of other methods for solving least squares problems, much of the code parallels that shown in Listing 6. We will omit the redundant parts and show only the evaluation of the coefficients, the rank and the standard errors. Actually, we only calculate the standard errors up to the scalar multiple of s, the residual standard error, in these code fragments. The calculation of the residuals and s and the scaling of the coefficient standard errors is the same for all methods. (See the files `fastLm.h` and `fastLm.cpp` in the **RcppEigen** source package for details.)

4.2 Least squares using the unpivoted QR decomposition

A QR decomposition has the form

$$\boldsymbol{X} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{Q}_1\boldsymbol{R}_1$$

where $\boldsymbol{Q}$ is an $n \times n$ orthogonal matrix, which means that $\boldsymbol{Q}'\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}' = \boldsymbol{I}_n$, and the $n \times p$ matrix $\boldsymbol{R}$ is zero below the main diagonal. The $n \times p$ matrix $\boldsymbol{Q}_1$ is the first p columns of $\boldsymbol{Q}$ and the $p \times p$ upper triangular matrix $\boldsymbol{R}_1$ is the top p rows of $\boldsymbol{R}$. There are three **Eigen** classes for the QR decomposition: `HouseholderQR` provides the basic QR decomposition using Householder transformations, `ColPivHouse-holderQR` incorporates column pivots and `FullPivHouseholderQR` incorporates both row and column pivots.

Listing 7 shows a least squares solution using the unpivoted QR decomposition. The calculations in Listing 7 are quite similar to those in Listing 6. In fact, if we had extracted the upper triangular factor (the `matrixU()` method) from the `LLT` object in Listing 6, the rest of the code would be nearly identical.

4.3 Handling the rank-deficient case

One important consideration when determining least squares solutions is whether rank($\boldsymbol{X}$) is p, a situation we describe by saying that $\boldsymbol{X}$ has "full column rank". When $\boldsymbol{X}$ does not have full column rank we say it is "rank deficient".

Although the theoretical rank of a matrix is well-defined, its evaluation in practice is not. At best we can compute an effective rank according to some tolerance. We refer to decompositions that allow us to estimate the rank of the matrix in this way as "rank-revealing".

Because the `model.matrix` function in R does a considerable amount of symbolic analysis behind the scenes, we usually end up with full-rank model matrices. The common cases of rank-deficiency, such as incorporating both a constant term and a full set of indicators columns for the levels of a factor, are eliminated. Other, more subtle, situations will not be detected at this stage, however. A simple example occurs when there is a "missing cell" in a two-way layout and the interaction of the two factors is included in the model.

```
> dd <- data.frame(f1 = gl(4, 6, labels = LETTERS[1:4]),
+                  f2 = gl(3, 2, labels = letters[1:3]))[-(7:8), ]
> xtabs(~ f2 + f1, dd)                    # one missing cell

   f1
f2  A B C D
  a 2 0 2 2
  b 2 2 2 2
  c 2 2 2 2

> mm <- model.matrix(~ f1 * f2, dd)
> kappa(mm)         # large condition number, indicating rank deficiency
```

```
[1] 4.309225e+16

> rcond(mm)         # alternative evaluation, the reciprocal condition number

[1] 2.320603e-17

> (c(rank=qr(mm)$rank, p=ncol(mm))) # rank as computed in R's qr function

rank    p
  11   12

> set.seed(1)
> dd$y <- mm %*% seq_len(ncol(mm)) + rnorm(nrow(mm), sd = 0.1)
>                          # lm detects the rank deficiency
> fm1 <- lm(y ~ f1 * f2, dd)
> writeLines(capture.output(print(summary(fm1), signif.stars=FALSE))[9:22])

Coefficients: (1 not defined because of singularities)
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.97786    0.05816   16.81 3.41e-09
f1B         12.03807    0.08226  146.35  < 2e-16
f1C          3.11722    0.08226   37.90 5.22e-13
f1D          4.06852    0.08226   49.46 2.83e-14
f2b          5.06012    0.08226   61.52 2.59e-15
f2c          5.99759    0.08226   72.91 4.01e-16
f1B:f2b     -3.01476    0.11633  -25.92 3.27e-11
f1C:f2b      7.70300    0.11633   66.22 1.16e-15
f1D:f2b      8.96425    0.11633   77.06  < 2e-16
f1B:f2c           NA         NA      NA       NA
f1C:f2c     10.96133    0.11633   94.23  < 2e-16
f1D:f2c     12.04108    0.11633  103.51  < 2e-16
```

The `lm` function for fitting linear models in R uses a rank-revealing form of the QR decomposition. When the model matrix is determined to be rank deficient, according to the threshold used in R's QR decomposition, the model matrix is reduced to rank $(\boldsymbol{X})$ columns by pivoting selected columns (those that are apparently linearly dependent on columns to their left) to the right hand side of the matrix. A solution for this reduced model matrix is determined and the coefficients and standard errors for the redundant columns are flagged as missing.

An alternative approach is to evaluate the "pseudo-inverse" of $\boldsymbol{X}$ from the singular value decomposition (SVD) of $\boldsymbol{X}$ or the eigendecomposition of $\boldsymbol{X}'\boldsymbol{X}$. The SVD is of the form

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}' = \boldsymbol{U}_1\boldsymbol{D}_1\boldsymbol{V}'$$

where $\boldsymbol{U}$ is an orthogonal $n \times n$ matrix and $\boldsymbol{U}_1$ is its leftmost p columns, $\boldsymbol{D}$ is $n \times p$ and zero off the main diagonal so that $\boldsymbol{D}_1$ is a $p \times p$ diagonal matrix with non-decreasing non-negative diagonal elements, and $\boldsymbol{V}$ is a $p \times p$ orthogonal matrix. The pseudo-inverse of $\boldsymbol{D}_1$, written $\boldsymbol{D}_1^+$ is a $p \times p$ diagonal matrix whose first $r = \text{rank}(\boldsymbol{X})$ diagonal elements are the inverses of the corresponding diagonal elements of $\boldsymbol{D}_1$ and whose last $p - r$ diagonal elements are zero.

The tolerance for determining if an element of the diagonal of $\boldsymbol{D}$ is considered to be (effectively) zero is a multiple of the largest singular value (i.e. the $(1, 1)$ element of $\boldsymbol{D}$).

In Listing 8 we define a utility function, `Dplus`, to return the pseudo-inverse as a diagonal matrix, given the singular values (the diagonal of $\boldsymbol{D}$) and the apparent rank. To be able to use this function with the eigendecomposition where the eigenvalues are in increasing order we include a Boolean argument `rev` indicating whether the order is reversed.

4.4 Least squares using the SVD

With these definitions the code for least squares using the singular value decomposition can be written as in Listing 9. In the rank-deficient case this code will produce a complete set of coefficients and their standard errors. It is up to the user to note that the rank is less than p, the number of columns in $\boldsymbol{X}$, and hence that the estimated coefficients are just one of an infinite number of coefficient vectors that produce the same fitted values. It happens that this solution is the minimum norm solution.

Listing 8: DplusCpp: Create the diagonal matrix $\boldsymbol{D}^+$ from the array of singular values $\boldsymbol{d}$

```
using Eigen::DiagonalMatrix;
using Eigen::Dynamic;

inline DiagonalMatrix<double, Dynamic> Dplus(const ArrayXd& D,
                                             int r, bool rev=false) {
    VectorXd   Di(VectorXd::Constant(D.size(), 0.));
    if (rev) Di.tail(r)  = D.tail(r).inverse();
    else Di.head(r)      = D.head(r).inverse();
    return DiagonalMatrix<double, Dynamic>(Di);
}
```

Listing 9: SVDLSCpp: Least squares using the SVD

```
using Eigen::JacobiSVD;

const JacobiSVD<MatrixXd> UDV(X.jacobiSvd(Eigen::ComputeThinU|Eigen::ComputeThinV));
const ArrayXd                D(UDV.singularValues());
const int                    r((D > D[0] * threshold()).count());
const MatrixXd             VDp(UDV.matrixV() * Dplus(D, r));
const VectorXd         betahat(VDp * UDV.matrixU().adjoint() * y);
const int                   df(n - r);
const VectorXd              se(s * VDp.rowwise().norm());
```

The standard errors of the coefficient estimates in the rank-deficient case must be interpreted carefully. The solution with one or more missing coefficients, as returned by the `lm.fit` function in R and the column-pivoted QR decomposition described in Section 4.6 does not provide standard errors for the missing coefficients. That is, both the coefficient and its standard error are returned as `NA` because the least squares solution is performed on a reduced model matrix. It is also true that the solution returned by the SVD method is with respect to a reduced model matrix but the p coefficient estimates and their p standard errors don't show this. They are, in fact, linear combinations of a set of r coefficient estimates and their standard errors.

4.5 Least squares using the eigendecomposition

The eigendecomposition of $\boldsymbol{X}'\boldsymbol{X}$ is defined as

$$\boldsymbol{X}'\boldsymbol{X} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}'$$

where $\boldsymbol{V}$, the matrix of eigenvectors, is a $p \times p$ orthogonal matrix and $\boldsymbol{\Lambda}$ is a $p \times p$ diagonal matrix with non-increasing, non-negative diagonal elements, called the eigenvalues of $\boldsymbol{X}'\boldsymbol{X}$. When the eigenvalues are distinct this $\boldsymbol{V}$ is the same as that in the SVD. Also the eigenvalues of $\boldsymbol{X}'\boldsymbol{X}$ are the squares of the singular values of $\boldsymbol{X}$.

With these definitions we can adapt much of the code from the SVD method for the eigendecomposition, as shown in Listing 10.

4.6 Least squares using the column-pivoted QR decomposition

The column-pivoted QR decomposition provides results similar to those from R in both the full-rank and the rank-deficient cases. The decomposition is of the form

$$\boldsymbol{X}\boldsymbol{P} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{Q}_1\boldsymbol{R}_1$$

where, as before, $\boldsymbol{Q}$ is $n \times n$ and orthogonal and $\boldsymbol{R}$ is $n \times p$ and upper triangular. The $p \times p$ matrix $\boldsymbol{P}$ is a permutation matrix. That is, its columns are a permutation of the columns of $\boldsymbol{I}_p$. It serves to reorder the columns of $\boldsymbol{X}$ so that the diagonal elements of $\boldsymbol{R}$ are non-increasing in magnitude.

Listing 10: SymmEigLSCpp: Least squares using the eigendecomposition

```
using Eigen::SelfAdjointEigenSolver;

const SelfAdjointEigenSolver<MatrixXd>
    VLV(MatrixXd(p, p).setZero().selfadjointView<Lower>.rankUpdate(X.adjoint()));
const ArrayXd              D(eig.eigenvalues());
const int                  r((D > D[p - 1] * threshold()).count());
const MatrixXd           VDp(VLV.eigenvectors() * Dplus(D.sqrt(), r, true));
const VectorXd       betahat(VDp * VDp.adjoint() * X.adjoint() * y);
const VectorXd            se(s * VDp.rowwise().norm());
```

An instance of the class `Eigen::ColPivHouseholderQR` has a `rank()` method returning the computational rank of the matrix. When $\boldsymbol{X}$ is of full rank we can use essentially the same code as in the unpivoted decomposition except that we must reorder the standard errors. When $\boldsymbol{X}$ is rank-deficient we evaluate the coefficients and standard errors for the leading r columns of $\boldsymbol{XP}$ only.

In the rank-deficient case the straightforward calculation of the fitted values, as $\boldsymbol{X}\widehat{\boldsymbol{\beta}}$, cannot be used. We could do some complicated rearrangement of the columns of X and the coefficient estimates but it is conceptually (and computationally) easier to employ the relationship

$$\widehat{\boldsymbol{y}} = \boldsymbol{Q}_1\boldsymbol{Q}_1'\boldsymbol{y} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{I}_r & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0}\end{bmatrix}\boldsymbol{Q}'\boldsymbol{y}$$

The vector $\boldsymbol{Q}'\boldsymbol{y}$ is called the "effects" vector in R.

Just to check that the code in Listing 11 does indeed provide the desired answer

```
> print(summary(fmPQR <- fastLm(y ~ f1 * f2, dd)), signif.stars=FALSE)

Call:
fastLm.formula(formula = y ~ f1 * f2, data = dd)

             Estimate Std. Error t value  Pr(>|t|)
(Intercept)  0.977859   0.058165  16.812 3.413e-09
f1B         12.038068   0.082258 146.346 < 2.2e-16
f1C          3.117222   0.082258  37.896 5.221e-13
f1D          4.068523   0.082258  49.461 2.833e-14
f2b          5.060123   0.082258  61.516 2.593e-15
f2c          5.997592   0.082258  72.912 4.015e-16
f1B:f2b     -3.014763   0.116330 -25.916 3.266e-11
f1C:f2b      7.702999   0.116330  66.217 1.156e-15
f1D:f2b      8.964251   0.116330  77.059 < 2.2e-16
f1B:f2c            NA         NA      NA        NA
f1C:f2c     10.961326   0.116330  94.226 < 2.2e-16
f1D:f2c     12.041081   0.116330 103.508 < 2.2e-16

Residual standard error: 0.2868 on 11 degrees of freedom
Multiple R-squared: 0.9999,       Adjusted R-squared: 0.9999

> all.equal(coef(fm1), coef(fmPQR))

[1] TRUE

> all.equal(unname(fitted(fm1)), fitted(fmPQR))

[1] TRUE

> all.equal(unname(residuals(fm1)), residuals(fmPQR))

[1] TRUE
```

The rank-revealing SVD method produces the same fitted values but not the same coefficients.

```
> print(summary(fmSVD <- fastLm(y ~ f1 * f2, dd, method=4L)), signif.stars=FALSE)
```

Listing 11: ColPivQRLSCpp: Least squares using the pivoted QR decomposition

```
using Eigen::ColPivHouseholderQR;
typedef ColPivHouseholderQR<MatrixXd>::PermutationType  Permutation;

const ColPivHouseholderQR<MatrixXd> PQR(X);
const Permutation                   Pmat(PQR.colsPermutation());
const int                              r(PQR.rank());
VectorXd                        betahat, fitted, se;
if (r == X.cols()) {      // full rank case
    betahat  = PQR.solve(y);
    fitted   = X * betahat;
    se       = Pmat * PQR.matrixQR().topRows(p).triangularView<Upper>().
                 solve(MatrixXd::Identity(p, p)).rowwise().norm();
} else {
    MatrixXd                       Rinv(PQR.matrixQR().topLeftCorner(r, r).
                                        triangularView<Upper>().
                                        solve(MatrixXd::Identity(r, r)));
    VectorXd                    effects(PQR.householderQ().adjoint() * y);
    betahat.head(r)                   = Rinv * effects.head(r);
    betahat                           = Pmat * betahat;
                          // create fitted values from effects
                          // (cannot use X * betahat when X is rank-deficient)
    effects.tail(X.rows() - r).setZero();
    fitted                            = PQR.householderQ() * effects;
    se.head(r)                        = Rinv.rowwise().norm();
    se                                = Pmat * se;
}
```

```
Call:
fastLm.formula(formula = y ~ f1 * f2, data = dd, method = 4L)

             Estimate Std. Error t value  Pr(>|t|)
(Intercept)  0.977859   0.058165  16.812 3.413e-09
f1B          7.020458   0.038777 181.049 < 2.2e-16
f1C          3.117222   0.082258  37.896 5.221e-13
f1D          4.068523   0.082258  49.461 2.833e-14
f2b          5.060123   0.082258  61.516 2.593e-15
f2c          5.997592   0.082258  72.912 4.015e-16
f1B:f2b      2.002847   0.061311  32.667 2.638e-12
f1C:f2b      7.702999   0.116330  66.217 1.156e-15
f1D:f2b      8.964251   0.116330  77.059 < 2.2e-16
f1B:f2c      5.017610   0.061311  81.838 < 2.2e-16
f1C:f2c     10.961326   0.116330  94.226 < 2.2e-16
f1D:f2c     12.041081   0.116330 103.508 < 2.2e-16

Residual standard error: 0.2868 on 11 degrees of freedom
Multiple R-squared: 0.9999,       Adjusted R-squared: 0.9999

> all.equal(coef(fm1), coef(fmSVD))

[1] "'is.NA' value mismatch: 0 in current 1 in target"

> all.equal(unname(fitted(fm1)), fitted(fmSVD))

[1] TRUE

> all.equal(unname(residuals(fm1)), residuals(fmSVD))

[1] TRUE
```

The coefficients from the symmetric eigendecomposition method are the same as those from the SVD

```
> print(summary(fmVLV <- fastLm(y ~ f1 * f2, dd, method=5L)), signif.stars=FALSE)

Call:
fastLm.formula(formula = y ~ f1 * f2, data = dd, method = 5L)

             Estimate Std. Error t value  Pr(>|t|)
(Intercept)  0.977859   0.058165  16.812 3.413e-09
f1B          7.020458   0.038777 181.049 < 2.2e-16
f1C          3.117222   0.082258  37.896 5.221e-13
f1D          4.068523   0.082258  49.461 2.833e-14
f2b          5.060123   0.082258  61.516 2.593e-15
f2c          5.997592   0.082258  72.912 4.015e-16
f1B:f2b      2.002847   0.061311  32.667 2.638e-12
f1C:f2b      7.702999   0.116330  66.217 1.156e-15
f1D:f2b      8.964251   0.116330  77.059 < 2.2e-16
f1B:f2c      5.017610   0.061311  81.838 < 2.2e-16
f1C:f2c     10.961326   0.116330  94.226 < 2.2e-16
f1D:f2c     12.041081   0.116330 103.508 < 2.2e-16

Residual standard error: 0.2868 on 11 degrees of freedom
Multiple R-squared: 0.9999,       Adjusted R-squared: 0.9999

> all.equal(coef(fmSVD), coef(fmVLV))

[1] TRUE

> all.equal(unname(fitted(fm1)), fitted(fmSVD))

[1] TRUE

> all.equal(unname(residuals(fm1)), residuals(fmSVD))

[1] TRUE
```

4.7 Comparative speed

In the **RcppEigen** package the R function to fit linear models using the methods described above is called `fastLm`. The natural question to ask is, "Is it indeed fast to use these methods based on **Eigen**?". We have provided benchmarking code for these methods, R's `lm.fit` function and the `fastLm` implementations in the **RcppArmadillo** (François, Eddelbuettel, and Bates, 2011) and **RcppGSL** (François and Eddelbuettel, 2011) packages, if they are installed. The benchmark code, which uses the **rbenchmark** (Kusnierczyk, 2010) package, is in a file named `lmBenchmark.R` in the `examples` subdirectory of the installed **RcppEigen** package.

It can be run as

```
> source(system.file("examples", "lmBenchmark.R", package="RcppEigen"))
```

Results will vary according to the speed of the processor, the number of cores and the implementation of the BLAS (Basic Linear Algebra Subroutines) used. (**Eigen** methods do not use the BLAS but the other methods do.)

Results obtained on a desktop computer, circa 2010, are shown in Table 2

These results indicate that methods based on forming and decomposing $\boldsymbol{X}'\boldsymbol{X}$, (i.e. LDLt, LLt and SymmEig) are considerably faster than the others. The SymmEig method, using a rank-revealing decomposition, would be preferred, although the LDLt method could probably be modified to be rank-revealing. Do bear in mind that the dimensions of the problem will influence the comparative results. Because there are 100,000 rows in $\boldsymbol{X}$, methods that decompose the whole $\boldsymbol{X}$ matrix (all the methods except those named above) will be at a disadvantage.

The pivoted QR method is 1.6 times faster than R's `lm.fit` on this test and provides nearly the same information as `lm.fit`. Methods based on the singular value decomposition (SVD and GSL) are much slower but, as mentioned above, this is caused in part by $\boldsymbol{X}$ having many more rows than columns. The GSL method from the GNU Scientific Library uses an older algorithm for the SVD and is clearly out of contention.

Table 2: `lmBenchmark` results on a desktop computer for the default size, $100,000 \times 40$, full-rank model matrix running 20 repetitions for each method. Times (Elapsed, User and Sys) are in seconds. The BLAS in use is a single-threaded version of Atlas (Automatically Tuned Linear Algebra System).

Method	Relative	Elapsed	User	Sys
LLt	1.000000	1.227	1.228	0.000
LDLt	1.037490	1.273	1.272	0.000
SymmEig	2.895681	3.553	2.972	0.572
QR	7.828036	9.605	8.968	0.620
PivQR	7.953545	9.759	9.120	0.624
arma	8.383048	10.286	10.277	0.000
lm.fit	13.782396	16.911	15.521	1.368
SVD	54.829666	67.276	66.321	0.912
GSL	157.531377	193.291	192.568	0.640

Listing 12: badtransCpp: Transpose producing incorrect results

```
using Eigen::Map;
using Eigen::MatrixXi;
const Map<MatrixXi>  A(as<Map<MatrixXi> >(AA));
return wrap(A.transpose());
```

An SVD method using the Lapack SVD subroutine, `dgesv`, may be faster than the native **Eigen** implementation of the SVD, which is not a particularly fast method.

5 Delayed evaluation

A form of delayed evaluation is used in **Eigen**. That is, many operators and methods do not force the evaluation of the object but instead return an "expression object" that is evaluated when needed. As an example, even though we write the $\boldsymbol{X}'\boldsymbol{X}$ evaluation using `.rankUpdate(X.adjoint())` the `X.adjoint()` part is not evaluated immediately. The `rankUpdate` method detects that it has been passed a matrix that is to be used in its transposed form and evaluates the update by taking inner products of columns of $\boldsymbol{X}$ instead of rows of $\boldsymbol{X}'$.

Occasionally the method for `Rcpp::wrap` will not force an evaluation when it should. This is at least what Bill Venables calls an "infelicity" in the code, if not an outright bug. In the code for the transpose of an integer matrix shown in Listing 1 we assigned the transpose as a `MatrixXi` before returning it with `wrap`. The assignment forces the evaluation. If we skip this step, as in Listing 12 we get an answer with the correct shape but incorrect contents.

```
> Ai <- matrix(1:6, ncol=2L)
> ftrans2 <- cxxfunction(signature(AA = "matrix"), badtransCpp, "RcppEigen")
> (At <- ftrans2(Ai))

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> all.equal(At, t(Ai))

[1] "Mean relative difference: 0.4285714"
```

Another recommended practice is to assign objects before wrapping them for return to R.

6 Sparse matrices

Eigen provides sparse matrix classes. An R object of class `dgCMatrix` (from the **Matrix** (Bates and Maechler, 2011) package) can be mapped as in Listing 13.

Listing 13: sparseProdCpp: Transpose and product with sparse matrices

```
using Eigen::Map;
using Eigen::MappedSparseMatrix;
using Eigen::SparseMatrix;
using Eigen::VectorXd;

const MappedSparseMatrix<double>  A(as<MappedSparseMatrix<double> >(AA));
const Map<VectorXd>               y(as<Map<VectorXd> >(yy));
const SparseMatrix<double>        At(A.adjoint());
return List::create(_["At"]  = At,
                    _["Aty"] = At * y);
```

```
> sparse1 <- cxxfunction(signature(AA = "dgCMatrix", yy = "numeric"),
+                        sparseProdCpp, "RcppEigen")
> data(KNex, package="Matrix")
> rr <- sparse1(KNex$mm, KNex$y)
> stopifnot(all.equal(rr$At, t(KNex$mm)),
+           all.equal(rr$Aty, as.vector(crossprod(KNex$mm, KNex$y))))
```

A sparse Cholesky decomposition is provided in **Eigen** as the `SimplicialCholesky` class. There are also linkages to the **CHOLMOD** code from the **Matrix** package. At present, both of these are regarded as experimental.

References

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

Douglas Bates and Martin Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*, 2011. URL `http://CRAN.R-Project.org/package=Matrix`. R package version 1.0-2.

Dirk Eddelbuettel and Romain François. *Rcpp: Seamless R and C++ Integration*, 2011a. URL `http://CRAN.R-Project.org/package=Rcpp`. R package version 0.9.4.

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

Romain François and Dirk Eddelbuettel. *RcppGSL: Rcpp integration for GNU GSL vectors and matrices*, 2011. URL `http://CRAN.R-Project.org/package=RcppGSL`. R package version 0.1.1.

Romain François, Dirk Eddelbuettel, and Douglas Bates. *RcppArmadillo: Rcpp integration for Armadillo templated linear algebra library*, 2011. URL `http://CRAN.R-Project.org/package=RcppArmadillo`. R package version 0.2.18.

Wacek Kusnierczyk. *rbenchmark: Benchmarking routine for R*, 2010. URL `http://CRAN.R-Project.org/package=rbenchmark`. R package version 0.3.

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.