An Introduction to **RcppEigen**

Douglas Bates

RcppEigen version 0.1.3 as of October 21, 2011

Abstract

The **RcppEigen** package provides access to the **Eigen** C++ template library from R. **Rcpp** classes and especially the packages C++ templated functions `as` and `wrap` provide the "glue" for passing objects from R to C++ and back.

1 Introduction

As stated in the **Rcpp** 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 a high-performance, versatile and comprehensive representation 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 provides templated classes for matrices, vectors and arrays. As in many C++ template libraries using template meta-programming (Abrahams and Gurtovoy, 2004), the templates themselves are very complicated. However, **Eigen** provides typedef's for the classes that correspond to R matrices and vectors, as shown in Table˜1, and 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. I prefer this approach.

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

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>`

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. It is a good idea to declare such 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.

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** 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 `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))

     [,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 use the `transpose` method for the **Eigen::MatrixXi** class to return its transpose.

```
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);
```

```
> ftrans <- cxxfunction(signature(AA = "matrix"),
+                       paste(readLines( "code.cpp" ), collapse = "\n"),
+                       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 prefer the name `adjoint` with numeric or integer matrices.

3.2 Products and cross-products

As mentioned in Sec.~2.2, the `'*'` operator performs matrix multiplication on Matrix or Vector objects.

```
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);
```

```
> fprod <- cxxfunction(signature(BB = "matrix", CC = "matrix"),
+                      paste(readLines( "code.cpp" ), collapse = "\n"), "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.

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 half the result 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 when applied to 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` and the equivalent specifications for `Upper`.

For self-adjoint views the `rankUpdate` method adds a scalar multiple (which defaults to 1) of $\boldsymbol{AA}'$ to the current symmetric 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);
```

```
> fcprd <- cxxfunction(signature(AA = "matrix"),
+                  paste(readLines( "code.cpp" ), collapse = "\n"), "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 $\boldsymbol{AA}'$ 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 because the result is going to be symmetrized before being 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 $\boldsymbol{A} = \boldsymbol{LL}'$ and the “LDLt” form as a unit lower triangular matrix $\boldsymbol{L}$ and a diagonal matrix $\boldsymbol{D}$ with positive diagonal 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"
```

```
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()));
```

```
> fchol <- cxxfunction(signature(AA = "matrix"),
+                      paste(readLines( "code.cpp" ), collapse = "\n"), "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 the $p \times p$ lower Cholesky factor $\boldsymbol{L}$, defined so that $\boldsymbol{L}\boldsymbol{L}' = \boldsymbol{X}'\boldsymbol{X}$, is the product of its diagonal elements. (This is true for any triangular matrix.) By the properties of determinants,

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

Alternatively, if we use the "LDLt" decomposition, $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}' = \boldsymbol{X}'\boldsymbol{X}$ 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 $\boldsymbol{D}$.

```
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());
```

```
> fdet <- cxxfunction(signature(AA = "matrix"),
+                     paste(readLines( "code.cpp" ), collapse = "\n"), "RcppEigen")
> unlist(ll <- fdet(A))
```

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

4 Least squares solutions

A common operation in statistical computing is calculating the least squares solution

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

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 as the QR decomposition, with or without column pivoting, the singular value decomposition and the eigendecomposition of a symmetric matrix.

References

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