1) Overview
   a. C functions to be called from R include “R.h”, do not use main(), have all
      pointers for arguments, and return nothing (have void return types). C cannot
      determine the length of what the arguments point to, so “n” must also be an
      argument. C++ functions must be placed inside of extern "C" { ... }.
   b. These C functions should only use functions declared in R.h for I/O and memory
      management. E.g.: Rprintf("myint = %d and my d=%5.2f\n", myint, d);
   c. If you include “Rmath.h” you can call many standard random variable functions.
   d. From your shell, “R CMD SHLIB myfile.cpp” (or .c) constructs a “.so” shared
      object library. In R, dyn.load("mylib.so") loads the functions in your C
      code into your current R session..
   e. In R, a .C wrapper function is a safe and convenient way to call your C functions.
      Use as.double() and as.integer() in front of every input argument. Use
      as.integer(length(myvar)) to pass lengths (or nrow() or ncol()). Use
      as.double(rep(0,mylength)) or just double(mylength) to pass empty
      variables intended to hold return values. (Or use as.int() or integer().)
      The return value is a list, which has named elements corresponding to any named
      .C arguments. Often the wrapper function returns just one of the list elements.
   f. Must setup correctly for your computer/OS
      (http://www1.appstate.edu/~arnholta/Software/MakingPackagesUnderWindows.p
df) or (http://www.biostat.jhsph.edu/~rpeng/docs/interface.pdf)
   g. On unix.andrew.cmu.edu, e.g., the header files can be examined in /usr/include/R.

2) Super Simple Example:
   a. “x2.c” in C is converted to “x2.so” with OS command R CMD SHLIB x2.c
      /* C function to be called from R. */
      /* This function takes vector 'vec' and tries to return the
         second value. 'len' will be the length of 'vec'. The
         return value goes into 'two'. If len=1 the first value
         is returned. If len=0, 0.0 is returned. */
      void x2(double *vec, int *len, double *two) {
          int n = *len;
          if (n==0) {
              two[0] = 0.0;
          } else if (n==1) {
              two[0] = vec[0];
          } else {
              two[0] = vec[1];
          }
          return;
    }
b. In R:

```r
dyn.load("x2.so")
x2 = function(x) {
  rtn = .C("x2", as.double(x), as.integer(length(x)),
           v = as.double(0.0))
  return(rtn$v)
}
x2(sqrt(1:5))  # 1.414214
x2(5:10) # 6
x2(pi) # 3.141593
x2 = function(x) .C("x2", x, as.integer(length(x)), 0.0)
x2(c(1.2, 2.3, 3.4))
[[1]]
[1] 1.2 2.3 3.4

[[2]]
[1] 3

[[3]]
[1] 2.3

x2(c(1:3))
[[1]]
[1] 1 2 3

[[2]]
[1] 3

[[3]]
[1] 1.482197e-323
```

3) Matrices in C from R

a. In R:

```r
> matrix(1:12, nrow=3)
[1,]  1  4  7 10
[2,]  2  5  8 11
[3,]  3  6  9 12
> as.double(matrix(1:12, nrow=3))
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

b. To pass a matrix to .C, you still use `as.double()` which passes the matrix (stored in R in the Fortran-style “column-major” format) as a vector.

c. In C, compute the vector location of the matrix element.

d. To return a matrix, return the vector, and use `matrix(my.vec, nrow=nr, ncol=nc)` in your wrapper.
4) Finding slow parts of R programs: Rprof()

```r
source("gaussMixEM.R")
Rprof("gmem.prof")
gaussMixEM("big.dat")
Rprof()
summaryRprof("gmem.prof")
```

$by.total

<table>
<thead>
<tr>
<th>Function</th>
<th>total.time</th>
<th>total.pct</th>
<th>self.time</th>
<th>self.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaussMixEM</td>
<td>2.10</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>table</td>
<td>0.92</td>
<td>43.81</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>factor</td>
<td>0.88</td>
<td>41.90</td>
<td>0.84</td>
<td>40.00</td>
</tr>
<tr>
<td>mstep</td>
<td>0.44</td>
<td>20.95</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>read.table</td>
<td>0.42</td>
<td>20.00</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>as.matrix.data.frame</td>
<td>0.32</td>
<td>15.24</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>var</td>
<td>0.32</td>
<td>15.24</td>
<td>0.02</td>
<td>0.95</td>
</tr>
<tr>
<td>as.matrix</td>
<td>0.32</td>
<td>15.24</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>estep</td>
<td>0.32</td>
<td>15.24</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>scan</td>
<td>0.30</td>
<td>14.29</td>
<td>0.30</td>
<td>14.29</td>
</tr>
<tr>
<td>mahalanobis</td>
<td>0.30</td>
<td>14.29</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>doTryCatch</td>
<td>0.30</td>
<td>14.29</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>try</td>
<td>0.30</td>
<td>14.29</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>tryCatch</td>
<td>0.30</td>
<td>14.29</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>tryCatchList</td>
<td>0.30</td>
<td>14.29</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>tryCatchOne</td>
<td>0.30</td>
<td>14.29</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>row.names.data.frame</td>
<td>0.26</td>
<td>12.38</td>
<td>0.26</td>
<td>12.38</td>
</tr>
<tr>
<td>row.names</td>
<td>0.26</td>
<td>12.38</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>sweep</td>
<td>0.14</td>
<td>6.67</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>[.data.frame</td>
<td>0.12</td>
<td>5.71</td>
<td>0.10</td>
<td>4.76</td>
</tr>
<tr>
<td>]</td>
<td>0.12</td>
<td>5.71</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>rowSums</td>
<td>0.10</td>
<td>4.76</td>
<td>0.02</td>
<td>0.95</td>
</tr>
<tr>
<td>aperm</td>
<td>0.10</td>
<td>4.76</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>is.data.frame</td>
<td>0.08</td>
<td>3.81</td>
<td>0.08</td>
<td>3.81</td>
</tr>
<tr>
<td>type.convert</td>
<td>0.08</td>
<td>3.81</td>
<td>0.08</td>
<td>3.81</td>
</tr>
<tr>
<td>aperm.default</td>
<td>0.06</td>
<td>2.86</td>
<td>0.06</td>
<td>2.86</td>
</tr>
<tr>
<td>array</td>
<td>0.04</td>
<td>1.90</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>unique.default</td>
<td>0.04</td>
<td>1.90</td>
<td>0.04</td>
<td>1.90</td>
</tr>
<tr>
<td>unique</td>
<td>0.04</td>
<td>1.90</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>==</td>
<td>0.02</td>
<td>0.95</td>
<td>0.02</td>
<td>0.95</td>
</tr>
<tr>
<td>&gt;</td>
<td>0.02</td>
<td>0.95</td>
<td>0.02</td>
<td>0.95</td>
</tr>
<tr>
<td>unlist</td>
<td>0.02</td>
<td>0.95</td>
<td>0.02</td>
<td>0.95</td>
</tr>
</tbody>
</table>
5) Silly example 1: matrix usage (and preprocessor macros)

a. Good quick macro reference:
   http://www.cprogramming.com/tutorial/cpreprocessor.html

b. Remember preprocessor “defines” just substitute text:
   #define MAX_SIZE 1024
   #define MY_QUOTED_NAME “Howard Seltman”
   They just plug in text. The all caps convention helps a reader to know to look in
   the #include files for the definition of the preprocessor symbol, not in a standard
   C/C++ definition or declaration. These directives do not end with a semicolon.
   Preprocessor numbers do not have a data type and do not allocate storage. They
   are just text:
   int mySize = MAX_SIZE; /* integer */
   double myD = MAX_SIZE; /* converted to double by compiler via context */

c. Macros functions are text-based function-like preprocessor directives. They can
   have any number of arguments, which are treated like text, substituted into the
   macro, then substituted into your C/C++ code:
   #define DUMB_DOUBLE(x) x+x
   #define DOUBLE(x) (x+x)
   int y = 5;
   val = 3 * DUMB_DOUBLE(y);  // 3 * y+y = 20
   val = 3 * DOUBLE(y);       // 3 * (y+y) = 30

d. C from R code to replace a matrix diagonal beyond [1,1] with a constant
   times the matrix value to the left:
   /* compile: R CMD SHELL matShift.cpp */
   #include <R.h>
   #define M(row,col) mat[(col)*nr+(row)]

   extern "C" {
      void matShift(double *mat, int *nrow, int *ncol,
                    double *value) {
         int nr = *nrow;
         for (int i=1; i < *nrow; i++) {
            if (i < *ncol) M(i, i) = *value * M(i, i-1);
         }
         return;
      }  // end matDemo
   }  // end extern "C"
R code to test matShift():

dyn.load("matShift.so")
ms = function(mat, v) {
  if (!is.matrix(mat)) stop("'mat' must be a matrix")
  if (!is.finite(v) && length(v)==1) {
    stop("'v' must be a single numeric value")
  }
  nr = nrow(mat)
  rtn = .C("matShift", mat=as.double(mat), as.integer(nr),
          as.integer(ncol(mat)), as.double(v))
  mat = rtn$mat
  return(matrix(mat, nrow=nr))
}

m = matrix(sample(1:20,12), nrow=3, ncol=4)
print(m)
print(ms(m, 1))
print(ms(m, 3))

6) BLAS/Lapack


b. Linear Algebra Package: http://www.netlib.org/lapack/explore-html/d8/d70/group__lapack.html

c. These are Fortran routines, so all arguments are pointers. Only single strings can
   be passed not vectors of strings (char[ ] not *char[ ]). For pure C/C++ code,
   BLAS and Lapack work nicely with gsl, so use #include<gsl/blas.h> and
   #include<gsl/lapack.h>.

d. For C within R, gsl is tricky. But you can use BLAS/Lapack directly with
   #include<R_ext/BLAS.h> and #include<R_ext/Lapack.h>. Special
   requirement for R CMD SHLIB when BLAS/Lapack are used: create a file called
   Makevars in the current directory with the contents:
   "PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS)".

e. In general you may need some system info on how to adjust the names of the
   BLAS/Lapack (Fortran) routines when using them in C/C++. Typically you
   precede them with an underscore and use all lower case for pure C/C++ programs.
   In C, you use a macro function (from R.h) to do the work for you, e.g.:
   F77_CALL(dcopy)(&size, sourceVec, &sincr, destVec, &dincr);
   usually becomes:
   _dcopy(&size, sourceVec, &sincr, destVec, &dincr);
7) Abbreviated details of BLAS ddot():

    DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
C***BEGIN PROLOGUE  DDOT
C***DATE WRITTEN   791001   (YYMMDD)
C***REVISION DATE  820801   (YYMMDD)
C***CATEGORY NO.  D1A4
C***KEYWORDS  BLAS,DOUBLE PRECISION,INNER PRODUCT,LINEAR ALGEBRA,VECTOR
C***AUTHOR  LAWSON, C. L., (JPL)
C           HANSON, R. J., (SNLA)
C           KINCAID, D. R., (U. OF TEXAS)
C           KROGH, F. T., (JPL)
C***PURPOSE  D.P. inner product of d.p. vectors
C***DESCRIPTION
C
B L A S  Subprogram
Description of Parameters

--Input--
N  number of elements in input vector(s)
DX  double precision vector with N elements
INCX  storage spacing between elements of DX
DY  double precision vector with N elements
INCY  storage spacing between elements of DY

--Output--
DDOT  double precision dot product (zero if N .LE. 0)

Returns the dot product of double precision DX and DY.
DDOT = sum for I = 0 to N-1 of  DX(LX+I*INCX) * DY(LY+I*INCY)
where LX = 1 if INCX .GE. 0, else LX = (-INCX)*N, and LY is
defined in a similar way using INCY.
C***REFERENCES  LAWSON C.L., HANSON R.J., KINCAID D.R., KROGH F.T.,
   *BASIC LINEAR ALGEBRA SUBPROGRAMS FOR FORTRAN USAGE*,
   ALGORITHM NO. 539, TRANSACTIONS ON MATHEMATICAL
   SOFTWARE, VOLUME 5, NUMBER 3, SEPTEMBER 1979, 308-323
C***ROUTINES CALLED  (NONE)
C***END PROLOGUE  DDOT
8) Abbreviated details of Lapack dgels():

```
subroutine DGELS ( CHARACTER TRANS,                  
                   INTEGER M,                          
                   INTEGER N,                          
                   INTEGER NRHS,                       
                   DOUBLE PRECISION, dimension( lda, * ) A, 
                   INTEGER LDA,                        
                   DOUBLE PRECISION, dimension( ldb, * ) B, 
                   INTEGER LDB,                        
                   DOUBLE PRECISION, dimension( * ) WORK, 
                   INTEGER LWORK,                      
                   INTEGER INFO                       
               )
```

DGELS solves overdetermined or underdetermined systems for GE matrices

Purpose:

DGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank.

The following options are provided:

1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

Parameters:

- **[in]** TRANS
  - TRANS is CHARACTER*1
  - 'N': the linear system involves A;
  - 'T': the linear system involves A**T.

- **[in]** M
  - M is INTEGER
  - The number of rows of the matrix A. M >= 0.

- **[in]** N
  - N is INTEGER
  - The number of columns of the matrix A. N >= 0.

- **[in]** NRHS
  - NRHS is INTEGER
  - The number of right hand sides, i.e., the number of columns of the matrices B and X. NRHS >=0.

- **[in,out]** A
  - A is DOUBLE PRECISION array, dimension (LDA,N)
  - On entry, the M-by-N matrix A.
  - On exit, if M >= N, A is overwritten by details of its QR factorization as returned by DGEQRF;

- **[in]** LDA
  - LDA is INTEGER
  - The leading dimension of the array A. LDA >= max(1,M).
[in, out] B

B is DOUBLE PRECISION array, dimension (LDB,NRHS)
On entry, the matrix B of right hand side vectors, stored
columnwise; B is M-by-NRHS if TRANS = 'N', or N-by-NRHS
if TRANS = 'T'.
On exit, if INFO = 0, B is overwritten by the solution
vectors, stored columnwise:
if TRANS = 'N' and m >= n, rows 1 to n of B contain the least
squares solution vectors; the residual sum of squares for the
solution in each column is given by the sum of squares of
elements N+1 to M in that column;

[in] LDB
LDB is INTEGER
The leading dimension of the array B. LDB >= MAX(1,M,N).

[out] WORK
WORK is DOUBLE PRECISION array, dimension (MAX(1,LWORK))
On exit, if INFO = 0, WORK(1) returns the optimal LWORK.

[in] LWORK
LWORK is INTEGER
The dimension of the array WORK.
LWORK >= max( 1, MN + max( MN, NRHS ) ).
For optimal performance,
LWORK >= max( 1, MN + max( MN, NRHS )*NB ).
where MN = min(M,N) and NB is the optimum block size.

If LWORK = -1, then a workspace query is assumed; the routine
only calculates the optimal size of the WORK array, returns
this value as the first entry of the WORK array, and no error
message related to LWORK is issued by XERBLA.

[out] INFO
INFO is INTEGER
= 0: successful exit
< 0: if INFO = -i, the i-th argument had an illegal value
> 0: if INFO = i, the i-th diagonal element of the
triangular factor of A is zero, so that A does not have
full rank; the least squares solution could not be
computed.