Working with compiled code

Robert B. Gramacy
Virginia Tech Department of Statistics

bobby.gramacy.com

Why Compiled Code?

Faster:

- Compiled code is usually faster than interpreted code.
- Better control over memory management, use of pointers.

External Libraries:

- The best (most reliable and optimized) libraries, especially for scientific computing, are in C and Fortran.

Production:

- R is great for prototyping and data analysis, but its not ideal for plugging into existing workflows.

Loops

Loops are very slow in R.

- E.g., `mutlinks.R()` from last week's lecture.

```
mutlinks.R <- function(A)
  {
    n <- ncol(A)
    if(nrow(A) != n) stop("A must be square")
    S <- 0
    for(i in 1:(n-1))
      for(j in (i+1):n)
        for(k in 1:n) S <- S + A[i,k]*A[j,k]
    S/choose(n,2)
  }
```

Here is a C implementation.

```
double mutlinks(int **A, int n)
  {
    unsigned int i, j, k, S;

    S = 0;
    for(i=0; i<n; i++)
      for(j=i+1; j<n; j++)
        for(k=0; k<n; k++) S += A[i][k]*A[j][k];

    return(((double) S)/(n*(n-1)/2));
  }
```

How can we use the C version in R?

Four ingredients are required.

1. A way to compile the C code — build a library — an R-friendly way.
2. Load that compiled code into R.
3. Pass data (R objects) to the C-side, and prepare data structures to receive the results.
4. Receive the data on the C side and pass the results back to R.

Compiling

I usually keep C code for a specific R project in a directory called `src`, adjacent to an `R` directory, containing the R code.

Inside the `src` directory, I use `R CMD SHLIB` to build a shared object containing the compiled code I want to load into R.

- Here is how I built the shared object for code supporting this lecture.

```
% R CMD SHLIB -o clect.so mutlinks.c bootreg.c
```

- building the shared object `clect.so`.

Loading shared objects

From inside the `R` directory, the shared object can be loaded with `dyn.load()`.

In R:

```
> dyn.load("../src/clect.so")
```

Then, all C functions in the shared object are callable from R

- e.g., with `.C()`.

However, most C functions are not "ready" to receive data from R

- e.g., via `.C()`.

.C interface

`.C()` can only "send" (and "recieve") pointers to `double`s, `int`egers, and `char`acter strings.

- and it must pre-allocate memory pointed to by results it will recieve.
- The lengths of data passed (i.e., pointed to) can be arbitrary, but must be unstructured (i.e., a flat array).
- Any flattening of arrays will be in col-major order.
- Dimensions of objects must be passed separately, also as pointers.

`.C()` is less sophisticated than MATLAB `mex`.

Here is an appropriate `.C()` call for `mutlinks()`.

```
mutlinks <- function(A)
  {
    n <- ncol(A)
    if(nrow(A) != n) stop("A must be square")

    ret <- .C("mutlinks_R",
              A = as.integer(t(A)),  ## col-major!
              n = as.integer(n),
              aS = double(1),
              DUP = FALSE)

    return(ret$aS)
  }
```

Receiving .C()

Now, we need a C-side function that can accept the pointers,

- and call our `mutlinks` C function.

```
void mutlinks_R(int *A_in, int *n_in, double *as_out)
  {
    unsigned int i;
    int **A;
    A = (int **) malloc(sizeof(int*) * (*n_in));
    A[0] = A_in;
    for(i=1; i<*n_in; i++) A[i] = A[i-1] + *n_in;
    *as_out = mutlinks(A, *n_in);
    free(A);
  }
```

Is all this effort worth it?

```
> A <- matrix(sample(0:1, (16^2)^2, replace=TRUE),
+             nrow=16^2)
> system.time(aS <- mutlinks(A))
   user  system elapsed
   0.04    0.00    0.04
> aS
[1] 64.53793
> system.time(aS.R <- mutlinks.R(A))
   user  system elapsed
 11.085   0.022  11.118
> aS.R
[1] 64.53793
```

Best practice/Style

There are lots of ways to cut corners and come out ok.

- We don't really need separate `mutlinks` and `mutlinks_R` functions.
- We could skip converting `A` to a 2-d array (and avoid the transpose).
- `DUP=FALSE` isn't necessary (in fact, its risky).

But I consider this structure to be good practice.

- Prototype an R-only version first (with a `.R` extension),
- and then write a C-version (with matching output).

Lets revisit our bootstrap regression example from last time.

Here is a simple R version.

```
bootols.R <- function(X, y, B=199, icept=TRUE, uselm=FALSE)
{
  if(icept) X <- cbind(1, X)
  if(nrow(X) != length(y)) stop("dimension mismatch")
  beta <- matrix(NA, nrow=B, ncol=ncol(X))
  for(b in 1:B) {
    i <- sample(1:n, n, replace=TRUE)
    Xb <- X[i,]; yb <- Y[i]
    beta[b,] <- ols.R(Xb, yb, uselm=TRUE, icept=FALSE)
  }
  return(beta)
}
```

Here is the OLS subroutine applied to each bootstrap sample.

```
ols.R <- function(X, y, icept=TRUE, uselm=FALSE)
{
  if(icept) X <- cbind(1, X)
  if(nrow(X) != length(y)) stop("dimension mismatch")
  if(uselm) beta <- drop(coef(lm(y~X-1)))
  else beta <- drop(solve(t(X) %*% X) %*% t(X) %*% y)
  return(beta)
}
```

- It offers the option of calling `lm()`, which does extra work we don't need for our bootstrap (but it does it very quickly in C);
- Or, it can do the simple matrix algebra alone (but in R).

Can we write a better/faster version in C

- without reinventing the wheel on aspects of linear algebra?
- without writing a random number generator (for `sample()`)?

Yes, because R's compiled Fortran and C libraries are available to us on the C-side.

- But that doesn't mean it'll be trivial.

We'll start by writing a simple OLS calculation in C and calling it from R.

See `bootreg.c` for a C version of our R prototype `ols.R()`, called `ols()`

- and all of the other C code for this example as well.

Notice that this C function

- calls some Fortran libraries for optmized BLAS/Lapack linear algebra: (`dgemv`, `dgemm`, `dposv`, `dsymv`) — the same ones R and MATLAB use.
- It uses temporary space allocated elsewhere (`XtY`, `XtX`, `XtXi`). Why?
- The comments indicate what R commands the C/Fortran calls correspond to.

It also uses a function that writes the identity to a pre-allocated matrix:

```
void zero(double **M, unsigned int n1, unsigned int n2)
{
  unsigned int i, j;
  for(i=0; i<n1; i++) for(j=0; j<n2; j++) M[i][j] = 0;
}

void id(double **M, unsigned int n)
{
  unsigned int i;
  zero(M, n, n);
  for(i=0; i<n; i++) M[i][i] = 1.0;
}
```

That's required by `dposv` which solves a linear system.

Use of the BLAS/Lapack library requires special linking arguments in the building of the shared object.

- Basically, we have to tell the linker where to look for those libraries.

`R CMD SHLIB` will read a `Makevars` file in the CWD.

- For BLAS/Lapack that file should contain, at minimum

```
PKG_LIBS = ${LAPACK_LIBS} ${BLAS_LIBS} ${FLIBS}
```

- We'll add some other things to it later.
- These definitions extend the ones in `$RINSTALL/etc/Makevars`.
- You can hard-code paths to whatever libraries you want.

The BLAS/Lapack bundled with R binaries may not be optimized for your machine.

If you have access to custom linear algebra libraries

- Intel's MKL library
- Apple's `vecLib` Framework
- ATLAS's automatically tuned libraries

then R, and your add-on C-routines, will get great benefit by linking to those libraries.

Re-compiling R to link to those libraries isn't that difficult.

- Competent IT folks are familiar with this sort of thing.
- And its well worth your/their time.

The R interface:

```
ols <- function(X, y, icept=TRUE)
{
  if(icept) X <- cbind(1, X)
  m <- ncol(X)
  n <- nrow(X)
  if(n != length(y)) stop("dimension mismatch")
  ret <- .C("ols_R",
            X = as.double(t(X)),
            y = as.double(y),
            n = as.integer(n),
            m = as.integer(m),
            beta.hat = double(m),
            DUP = FALSE)
  return(ret$beta.hat)
}
```

So the C function `ols_R` must

- convert inputs
- allocate the temporary space,
- call `ols()`
- clean up.

It requires some new matrix creation/destruction functions, so that the temporary space is set up correctly.

- `new_matrix()` and `delete_matrix()`

[see `bootreg.c`]

How does the C-version compare?

```
> source("bootreg.R")
> n <- 2000; d <- 100
> X <- 1/matrix(rt(n*d, df=1), ncol=d)
> beta <- c(1:d, 0)
> Y <- beta[1] + X %*% beta[-1] + rnorm(100, sd=3)
> system.time(fit.R <- ols.R(X, Y))
   user  system elapsed
  0.046   0.001   0.046
> system.time(fit.lm <- ols.R(X, Y, uselm=TRUE))
   user  system elapsed
  0.068   0.002   0.070
> system.time(fit <- ols(X, Y))
   user  system elapsed
  0.024   0.001   0.024
```

Maybe not super impressive ...

- But how about inside the bootstrap?
- This is where the re-use of space will come in handy — something R can't do.

```
bootols.R <- function(X, y, B=199, icept=TRUE, uselm=FALSE)
{
  if(icept) X <- cbind(1, X)
  beta <- matrix(NA, nrow=B, ncol=ncol(X))
  for(b in 1:B) {
    i <- sample(1:n, n, replace=TRUE)
    Xb <- X[i,]; yb <- Y[i]
    beta[b,] <- ols.R(Xb, yb, uselm=TRUE, icept=FALSE)
  }
  return(beta)
}
```

The C version, `bootols()` looks rather similar,

- except it pre-allocates space for `ols`

[see `bootreg.R`]

It uses R's uniform random number generator (RNG), `unif_rand()`,

- which defaults to the the Mersenne Twister algorithm.
- This is state of the art (I'd trust R on RNGs).
- Our usage of the RNG assumes we've already got the seed/state form R.
- We'll save that for `bootols_R()` since it is good practice to deal with the RNG state as part of the C/R interface.

```
bootols <- function(X, y, B=199, icept=TRUE)
{
  if(icept) X <- cbind(1, X)
  m <- ncol(X)
  n <- nrow(X)
  if(n != length(y)) stop("dimension mismatch")
  ret <- .C("bootols_R",
            X = as.double(t(X)),
            y = as.double(y),
            n = as.integer(n),
            m = as.integer(m),
            B = as.integer(B),
            beta.hat = double(m*B),
            DUP = FALSE)
  return(matrix(ret$beta.hat, nrow=B, byrow=TRUE))
}
```

Notice how we allocate a `double()`-vector of size `m*B` for `beta.hat`, when we really want the output to be a $B \times m$ matrix.

- Again, we'll need to remember that R uses column-major ordering when we convert it back to a matrix upon return.

`bootols_R()` must ...

- convert `X` back into a matrix
- and do the same with `beta.hat`.
- It also has to get the RNG state, and
- give it back to R when its done.

[see `bootreg.c`]

Now for a comparison

- using the same data as before.

```
> system.time(beta.hat.R <-
+     bootols.R(X, Y, B=1000, uselm=TRUE))
   user  system elapsed
 90.619   3.916 105.457
> system.time(beta.hat.R <- bootols.R(X, Y, B=1000))
   user  system elapsed
 91.451   3.226  96.882
> system.time(beta.hat <- bootols(X, Y, B=1000))
   user  system elapsed
 32.467   0.041  32.809
```

- $>$ 3$\times$ better with a slim compiled version.

R's C-library routines.

We used `unif_rand()` and R's linear algebra.

You are free to use any C/Fortran function used to build R.

- The header files are in `$RINSTALL/include`, and can be included with the usual `#include` pre-compiler macros.
- on `OSX` `$RINSTALL` is

  `/Library/Frameworks/R.framework/Resources/`
- E.g., `#include<R.h>` and `#include<Rmath.h>`.
- Unfortunately, the headers aren't very revealing about what the functions do (and what their arguments are).
- Instead, I suggest consulting the full source.

Debugging

Debugging C code attached to R involves the same tools as debugging standalone C code,

- e.g., using an interactive debugger like `gdb`.
- or `valgrind` for memory checking, etc.

To start the debugger, e.g., `gdb` or `valgrind`, do

```
% R -d gdb
% R -d valgrind
```

and interact with `gdb` or `valgrind` as usual.

For both, it is helpful to compile the shared object without any optimization flags.

- `-O2` is the default.
- The default already includes `-g` on most systems.

Otherwise, it may not be possible to inspect the values of all objects, e.g., in `gdb`.

Add the following line to `~/.R/Makevars`:

```
CFLAGS = -g -Wall -pedantic
```

- The others help catch problems at compile time.
- Don't forget to comment this line out when you're debugging: your code will run slower without `-O2`.

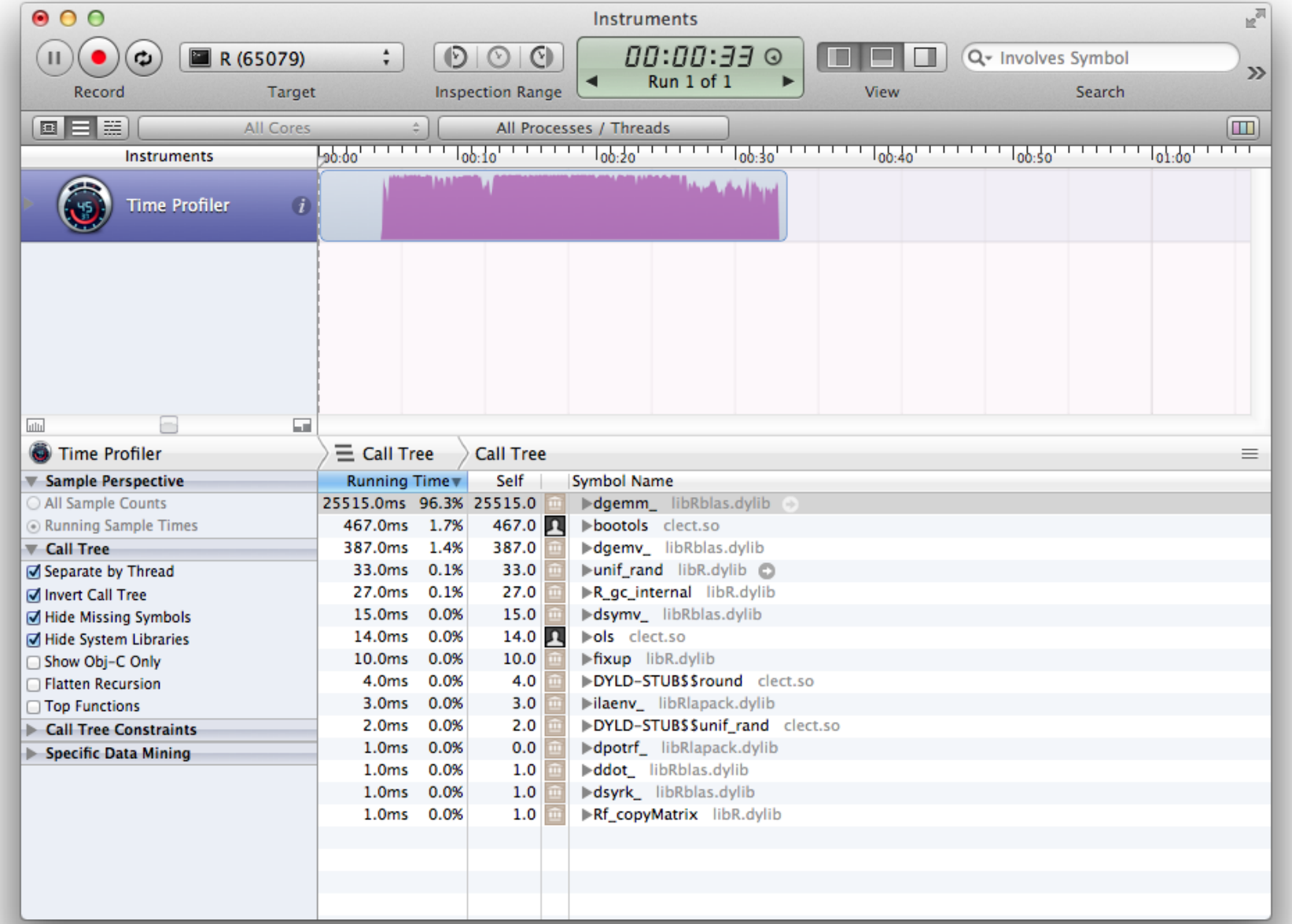
Profiling C

On Linux you can use `sprof` to profile a shared object, the same way you would any other shared object.

- `sprof` works like `gprof` which is for entire executables.

On OSX you can get a similar, online, summary via the `Time Profiler` in the `Instrument` panel.

- This may be the best free profiling tool for compiled code.
- There are similar ones for Windows/Linux but I am not familiar with them.
- The memory/leaks instrument is pretty good too.


Instruments
Record
R (65079)
Target
Inspection Range
00:00:33
Run 1 of 1
View
Involves Symbol
Search
All Cores
All Processes / Threads
Instruments
00:00
00:10
00:20
00:30
00:40
00:50
01:00
Time Profiler
Time Profiler
Call Tree
Call Tree
Sample Perspective
All Sample Counts
Running Sample Times
Call Tree
Separate by Thread
Invert Call Tree
Hide Missing Symbols
Hide System Libraries
Show Obj-C Only
Flatten Recursion
Top Functions
Call Tree Constraints
Specific Data Mining
Running Time
Self
Symbol Name
25515.0ms 96.3% 25515.0 dgemm_ libRblas.dylib
467.0ms 1.7% 467.0 bootols clect.so
387.0ms 1.4% 387.0 dgemv_ libRblas.dylib
33.0ms 0.1% 33.0 unif_rand libR.dylib
27.0ms 0.1% 27.0 R_gc_internal libR.dylib
15.0ms 0.0% 15.0 dsymv_ libRblas.dylib
14.0ms 0.0% 14.0 ols clect.so
10.0ms 0.0% 10.0 fixup libR.dylib
4.0ms 0.0% 4.0 DYLD-STUB$$round clect.so
3.0ms 0.0% 3.0 ilaenv_ libRlapack.dylib
2.0ms 0.0% 2.0 DYLD-STUB$$unif_rand clect.so
1.0ms 0.0% 0.0 dpotrf_ libRlapack.dylib
1.0ms 0.0% 1.0 ddot_ libRblas.dylib
1.0ms 0.0% 1.0 dsyrk_ libRblas.dylib
1.0ms 0.0% 1.0 Rf_copyMatrix libR.dylib

Windows Rtools

Most of what I have described has been for a Unix (Linux + OSX) environment.

You can do all this on Windows too:

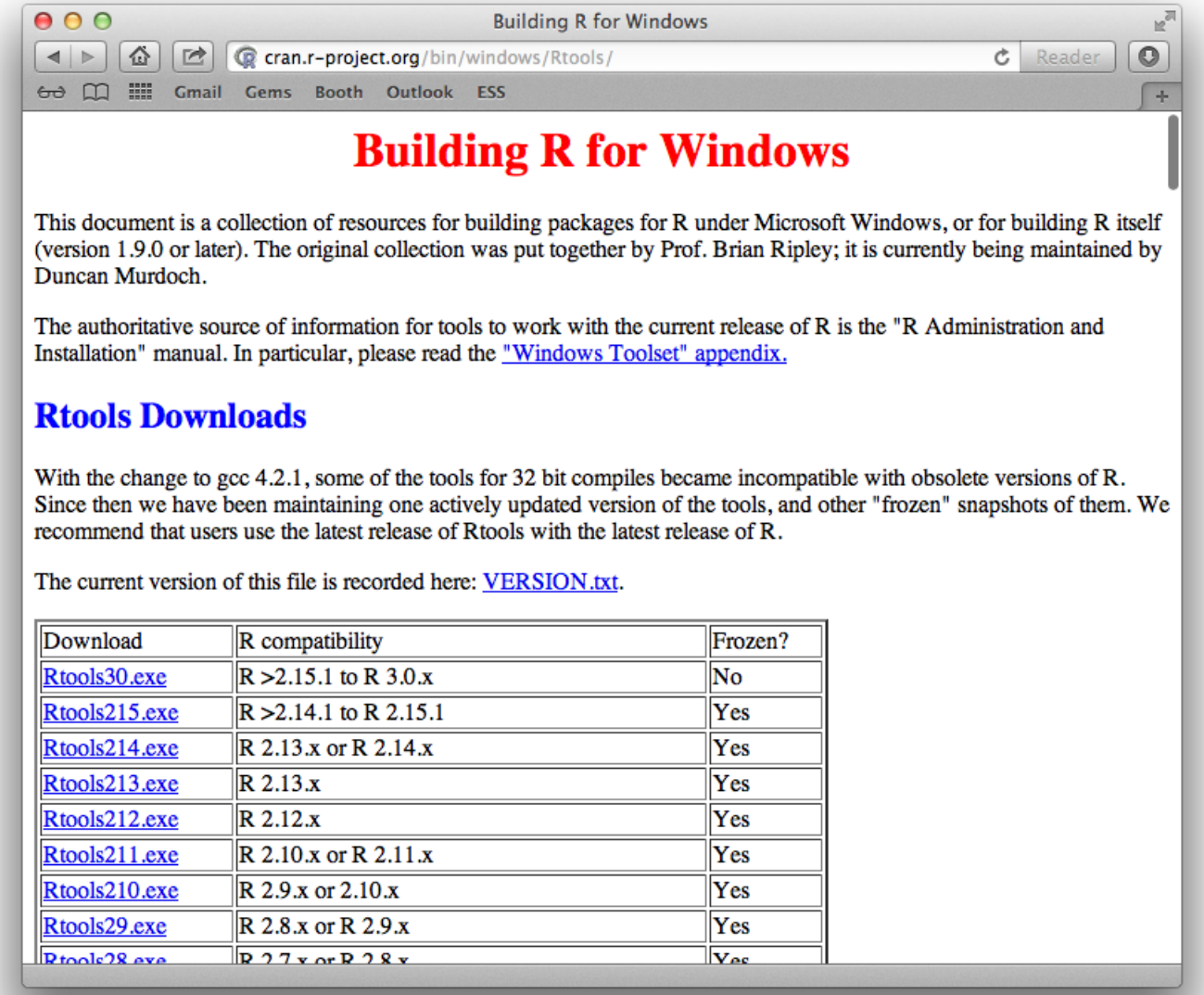
- compile R from scratch (possibly linking to a fast BLAS/Lapack)
- write your own external C libraries
- load shared objects (DLLs)

There are many options depending on your C compiler.

The simplest is to use `gcc`, just like in Unix

- but that requires installing Rtools.

http://cran.r-project.org/bin/windows/Rtools/



Building R for Windows

This document is a collection of resources for building packages for R under Microsoft Windows, or for building R itself (version 1.9.0 or later). The original collection was put together by Prof. Brian Ripley; it is currently being maintained by Duncan Murdoch.

The authoritative source of information for tools to work with the current release of R is the "R Administration and Installation" manual. In particular, please read the "Windows Toolset" appendix.

Rtools Downloads

With the change to gcc 4.2.1, some of the tools for 32 bit compiles became incompatible with obsolete versions of R. Since then we have been maintaining one actively updated version of the tools, and other "frozen" snapshots of them. We recommend that users use the latest release of Rtools with the latest release of R.

The current version of this file is recorded here: VERSION.txt.

Download	R compatibility	Frozen?
Rtools30.exe	R >2.15.1 to R 3.0.x	No
Rtools215.exe	R >2.14.1 to R 2.15.1	Yes
Rtools214.exe	R 2.13.x or R 2.14.x	Yes
Rtools213.exe	R 2.13.x	Yes
Rtools212.exe	R 2.12.x	Yes
Rtools211.exe	R 2.10.x or R 2.11.x	Yes
Rtools210.exe	R 2.9.x or 2.10.x	Yes
Rtools29.exe	R 2.8.x or R 2.9.x	Yes
Rtools28.exe	R 2.7.x or R 2.8.x	Yes

Fortran

Linking compiled Fortran code into R follows the same rubric as C, except

- use `.Fortran()` instead of `.C()`
- no need to transpose matrices: Fortran also uses column-major order.

As an example, see the files below which build up the MLE of a MVN mean μ and covariance Σ through a series of `sweep()`ing regressions provided in two Fortran subroutines.

- `sweep.f`
- `sweep.R` and `rubin.R`.

Other C interfaces in R

There are two other ways to interface between R and C.

- `.Call()`
- `.External()`

They are very similar (and share a documentation file).

They are more sophisticated than `.C()` — more like `mex`

- allowing any R object to be passed to/from the C-side.
- But cumbersome "work" is needed on the C-side to manipulate such objects.
- As a result, the C code is not portable.

`.C()` has never been insufficient for my own work.

One potential advantage of this framework is that it allows you to call-back to R and execute R code.

This may be the only (clean) way to execute an R function from inside of a C program.

Of course, if that R function is written in C under-the-hood,

- then it is better (faster/easier) to call the C version directly.

If not (its entirely in R),

- then perhaps calling slow R code defeats the purpose of writing a C-subroutine in the first place.

C++

`.C()` can be used for interfacing to C++ codes.

- Mixtures are allowed too.

If the C-side function that receives the R data

- e.g., `bootreg_R()` or `mutlinks_R()`

is in a C++ source (`.cc` or `.cpp`) file compiled by `g++`, say,

- then it needs to be wrapped in "`#extern "C" { ... }`"

This tells the compiler not to "dress" the symbol built for the function with extra C++ markup.

[See `blasso.cc` in the `monomvn` package for example.]

Another option is the `Rcpp` package.

- It is the C++ version of `.Call()`/`.External()`,
- allowing R objects to be passed to a C++ program.

The added value is that `Rcpp` provides dedicated C++ classes for all R objects. E.g.,

- `Rcpp::NumericVector`,
- `Rcpp::Function`,
- `Rcpp::Environment`
- `Rcpp::List`
- `Rcpp::Date`, ...

Neat — but I've yet to find it useful in my own work.

OpenMP

Anything you can do in C/Fortran/C++ is fair game when building shared objects for use with R.

E.g.,

1. `MPI` or `PVM` for cluster computing
2. `Pthreads` for parallelization
3. STLs for C++

As a testament to that, I've decided to illustrate the use of `OpenMP` inside of a C-library for R.

- `OpenMP` is the easiest way to get a C program to exploit multiple computing (SMP) cores.

`for` loops are the most easily parallelized with `OpenMP`.

Simply precede the `for` statement with the following pragma.

```
#pragma omp parallel for private(i)
for(i=0; i<n; i++) {
```

This causes each iteration of the `for` loop, for each `i`, to (potentially) execute in parallel.

- It is up to you to ensure that each iteration is independent of others.
- Sometimes that means changing the `for`-loop innards compared to a non-parallelized version.

In the `mutlinks()` example, I re-coded the `for` loop to write components of the grand sum into an array, slotted for each iteration `i` of the loop.

```
Sa = (int *) malloc(sizeof(int) * n);
#pragma omp parallel for private(i)
for(i=0; i<n; i++) {
   Sa[i] = 0;
   int j,k; /* must be private to i-loop */
   for(j=i+1; j<n; j++)
     for(k=0; k<n; k++) Sa[i] += A[i][k]*A[j][k];
}
S = 0;
for(i=0; i<n; i++) S += Sa[i];
```

The compiler and linker need a few extra flags,

- or they won't know how to interpret the `OpenMP` pragmas.

Append

```
PKG_CFLAGS = -fopenmp
PKG_LIBS = -lgomp
```

to any existing specifications in your Makevars file.

Then compile as usual.

- I find it helpful to use pre-compiler macros to separate `_OPENMP` implementations from non-parallel ones.
- Shortcutting the need for new C/R-interfaces.

A comparison on my 8-core iMac, where four cores were dedicated to another task.

```
> A <- matrix(sample(0:1, (40^2)^2, replace=TRUE),
+             nrow=40^2)
> system.time(aS <- mutlinks(A))
   user  system elapsed
  9.785   0.012   9.797
> dyn.load("../src/clect.so")
> system.time(aS <- mutlinks(A))
   user  system elapsed
 28.093   0.023   6.091
```

A 30% improvement.

- Not impressive!
- What is going on?

The inner `for`-loops, for `j` and `k`, are very "tight".

- They don't take long to execute,
- parallel execution or each iteration finishes quickly, passing control back to the scheduler.
- Plus, the code is burdened by memory management, and an extra final `for` loop.

The extra `for` loop can be eliminated with an `atomic` pragma,

- eliminating an race condition, causing the parallel threads to check that they are "alone" in executing the following line,

```
// #pragma omp atomic
// S += Sa[i];
```

- so no accumulations are missed.

But that doesn't help much, and could defeat the purpose:

- On a many-core machine, preventing race conditions in this way could drastically *slow* execution.

A better approach involves manually breaking the computation into parallel chunks.

Having fewer chunks

- ideally one per computing node
- with roughly equal-sized work for each chucks

should minimize time spent coordinating the parallelization

- maximizing use of multiple cores.

It takes a bit more planning to pull this off,

- but not necessarily much more code.

First, lets create a subroutine that tallies the sum for each `i`,

- encapsulating the inner two, `j` and `k`, `for`-loops.

```
int procpairs(int i, int **A, int n)
{
  int j, k, S;
  S=0;
  for(j=i+1; j<n; j++)
    for (k = 0; k < n; k++) S += A[i][k]*A[j][k];
  return S;
}
```

- Then we can call `procpairs()` in batches.

```
double mutlinks_omp2(int **A, int n)
  {
    int tot = 0;
    #pragma omp parallel
    {
      int i, mysum, me, nth;
      me = omp_get_thread_num();
      nth = omp_get_num_threads();
      mysum = 0;
      for(i=me; i<n; i+=nth) {
        mysum += procpairs(i, A, n);
      }
      #pragma omp atomic
      tot += mysum;
    }
    return((double) tot)/(n*(n-1)/2);
}
```

We probably could have used pre-compiler macros to make it so that a new function name, `mutlinks_omp2()`, was not needed.

- But I created a new one so we didn't need to re-compile to make a timing comparison.

I wrote new interface functions too

- and they look a lot like the old ones.

[see `mutlinks.c` and `mutlinks.R`]

Now for a final comparison.

- From before:

```
> system.time(aS <- mutlinks(A))
   user  system elapsed
 28.093   0.023   6.091
> aS
[1] 400.4916
```

- New chunky version:

```
> system.time(aS.omp2 <- mutlinks.omp2(A))
   user  system elapsed
 18.113   0.021   2.669
 > aS.omp2
[1] 400.4916
```

- more than $2\times$ faster than our first, simple, `OpenMP` version,
- totaling $\sim 4\times$ faster than our non-parallel version.

That's pretty good for a machine with ~ 4 free computing nodes.

- You can't hope for better than a linear speedup.

`OpenMP` has been around for a while, but its my new favorite thing.

- If you're a C-programmer, you need to get familiar with it.
- Multi-core computing is the modern *de facto*
- and all trends point to ever more cores on a chip.