C++ for statisticians, with a focus on interfacing from R and R packages

Chris Paciorek

February 6, 2014

These notes are the basis for a set of three 1.5 hour workshops on using C++ for statistical work. The first workshop focuses on the basics of C++ useful for statistical work. I don't expect to teach you C++ in that time, but to give you an overview so that you can go learn what you need more easily, or for those who know a bit of C or C++ already to help round out your knowledge. The second workshop focuses on calling C++ from R via a variety of methods. The third workshop focuses on creating R packages.

A few things to note in advance:

- My examples here will be silly toy examples for the purpose of keeping things simple and focused.
- I'll try to use *italics* to indicate names of things and `typewriter font` to indicate actual syntax. I'll likely slip up occasionally.
- A comment about the speed of R compared to C/C++. Oftentimes one will hear comparisons where R is orders of magnitude slower than C/C++ or some other software. I think these comparisons sometimes use R in naive ways (for example, avoidable for loops instead of vectorized calculations) and other times are not recognizing that a lot of the heavy numerical lifting in R, such as linear algebra, is done in compiled code and is likely to be comparable to doing it directly in compiled code. That said, there are lots of cases where you will want to use C/C++ to get substantial speedups. The motivation of this set of workshops is to enable you to use C/C++ from R for the slow parts and use R for the parts that don't involve serious computation, taking advantage of R's rapid coding, ease of use, input/output capabilities, and graphics.
- Also, most of this is focused on Linux as this is the environment in which most heavy-duty scientific computation gets done. Much of this should work on Macs since they run a variant

of UNIX under the hood. You'll need *xcode* installed on the Mac.

Resources

- Statistical Computing in C++ and R by Randall Eubank and Ana Kupresanin
- R extensions manual
- Dirk Eddelbuettel's Rcpp tutorial at useR! 2012 and Rcpp paper
- Hadley Wickham's guide to packages as part of his devtools package: https://github.com/hadley/devtools/wiki/Package-basics
- Papers on RcppArmadillo (http://www.sciencedirect.com/science/article/pii/S0167947313000492) and RcppEigen (http://www.jstatsoft.org/v52/i05/) [both also linked from Dirk's website]

1 C and C++ basics

1.1 C vs. C++

C++ builds on standard C in a number of ways. These include:

- additional functionality such as function overloading
- object-oriented programming
- the Standard Template Library that provides a variety of data structures and algorithms to operate on them
- templates, which allow you to more easily write functions that deal with multiple types

In the following, I'll describe the basics of coding in C and C++. I'll mix together standard C with features specific to C++ below, without being explicit about it and will generally refer to it as C++ even if I'm just using pure C functionality

1.2 Structure of a C/C++ program

A program consists of several pieces. Let's look though the example program below and see what the main pieces are.

This program does my favorite numerically intensive calculation, calculating the Cholesky decomposition of $X^\top X$ for random square X. *dsyrk* is Lapack's crossproduct function and *dpotrf* is Lapack's Cholesky decomposition.

```
// this is test.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <math.h>
#include <time.h>
#include <R.h>
#include <Rmath.h>
#define PI 3.14159

using namespace std;

// these declarations are needed as I don't think there is a lapack.h
extern "C" int dpotrf_(char* uplo, int* n, double* a, int* lda, int* info);
extern "C" int dsyrk_(char* uplo, char* trans, int* n, int* k,
  double* alpha, double* a, int* lda, double* beta, double* c, int* ldc);

// this is a one-line comment
/* This is
a multi-line
comment.
*/

// compilation:
// g++ -o test test.cpp -I/usr/share/R/include -llapack -lblas
//    -lRmath -lR -O3 -Wall

int main(){
  int size = 8000;
  int info = 0;
  char uplo = 'U';
  char trans = 'N';
  double alpha = 1.0;
  double beta = 0.0;
  double* x = new double[size*size];
  double* C = new double[size*size];
  for(int i = 0; i < size*size; i++){
    x[i] = rnorm(0.0, 1.0);
    C[i] = 0.0;
  }
  cout << "done with rnorm" << endl;
  dsyrk_(&uplo, &trans, &size, &size, &alpha, x, &size, &beta, C, &size);
   cout << "done crossprod" << endl;
  dpotrf_(&uplo,&size,C,&size,&info);
  cout << "done chol" << endl;
  return 0;
}
```

At the top are 'preprocessor' directives that provide information to the the preprocessor that runs before compilation. In particular, if you call functions from other libraries, you need to include the header files that contain the function prototypes (the definition of the function without the body) so that the compiler can check that you are calling the functions correctly (i.e., in terms of the arguments) in your code. You include standard system header files using <>, e.g.,

```
#include <iostream>
#include <Rmath.h>
```

For C++ system header files, you generally don't need the .h.

For non-standard header files (e.g., for external libraries obtained from other sources and your own user-written files), you put the file name in double quotes and you need the full filename with .h, e.g.,

```
# include "myheader.h"
```

You can define constants with *#define*. This allows one to avoid having "magic" numbers sprinkled around your code and then having to remember what they mean. However you may want to do this via *const* variables (more later).

You can do lots of other stuff with preprocessor directives, but we won't go into them here.

The "using namespace" tells the compiler that you'll make use of objects and functions from the standard template library (STL). This is akin to saying `library(pkgName)` in R, which loads objects and functions from the package into your workspace so they're accessible to you, and to Python's *import* statement.

The strangely named *dpotrf_* is the Lapack Cholesky routine and *dsyrk_* the crossproduct function. Calling Lapack routines involves a bunch of strange stuff, including the use of *extern*, which has to do with scoping issues that I don't fully understand. the _ occurs because a Fortran function is being called behind the scenes and typically an underscore is attached to the name of Fortran functions during their compilation.

If you're creating a full-fledged program that you call from the command line, you need a *main()* function that is where your execution begins and ends. From *main()*, you can of course call other functions. The return value of *main()* can just default to 0 as below or you could have it indicate whether the program executed correctly and finished without errors (0 is standard) or not (1 is standard in this case). If you're just going to create a library file that contains functions you'll call from R, you don't need *main()*. Just create your functions and compile as discussed below when we talk about calling C++ from R.

One line comments begin with "//". Multi-line comments can be enclose in the following syntax, e.g.: `/* THIS is a comment */`

1.3 Compiling and linking

Creating a program from C++ code involves compiling the program into a binary executable. The standard C++ compiler is *g++*. The standard C compiler is *gcc*. You can use *g++* for purely C code, so I'll just use g++ throughout. Also for code that includes MPI calls, there is *mpicxx* for C++ and *mpicc* for C.

When you compile code, a couple things happen. The code gets compiled into binary and code from different binaries gets linked together.

Often your program will use code from other libraries (e.g., BLAS and LAPACK). In this case you need to link the other libraries into your executable. There are two options for linking. You can link in a static version of the library, called an archive. Such files have names of the form: *lib{libName}.a*. The binary is included directly in your executable at linking time. Alternatively, you can link to dynamically to a *shared object* library (also called a *dynamic link library* - such files in Windows are *DLLs*). In this case the binary is not included directly in the executable at linking time, but is only referenced, and when your program is run, the code is obtained from the *lib{libName}.so* file. This reduces the size of your binary and allows one to use updated versions of the .so without changing the program, but it also means that the .so files that are linked to need to be available at run-time. In general this is not an issue, so using dynamic linking is very common.

There are a number of dependencies that need to be accounted for in compiling and linking. In the first step, code is compiled (creating a .o file). At this stage, you need to make sure that the compiler can find the necessary header files (.h) containing the signatures of any external library functions that you call in your code. Often these files are in standard places on your filesystem and the compiler can find them, but sometimes you need to add a flag pointing to the directory(ies) containing the header files that are in your #include statements in your code, e.g.:

```
g++ -c test.cpp -o test.o -I/usr/share/R/include
```

(Here I'm forcing compilation and linking to be in two steps using the -c flag, which says to compile but not to link.)

In the second step, the various binaries are linked together into a program that can run. At this stage you need to make sure the compiler (and for dynamic linking, the program at run-time) is able to find all of the necessary libraries. You need to indicate the libraries being linked in via -l flags. And, once again, the libraries are often in standard places on your file system and the compiler can find them, but sometimes you need to add a -L flag pointing to the directory containing the library. E.g.,

```
g++ -o test test.o -L/opt/acml5.2.0/gfortran64_mp/lib -lacml_mp -lR
```

where the -L points to the directory containing the .so file and -l to the actual .so file, *libacml_mp.so*. Note that the "lib" part of *libacml.so* is excluded. The compiler adds it automatically. Finally note that sometimes the order of the libraries in your linking invocation matter. Unsurprisingly, this can lead to headaches. Finally finally note that in the compilation above I link to a different BLAS/Lapack than my comment in the actual code file. So in certain cases I can link to different libraries providing the same underlying functions.

Oftentimes you'll do the compiling and linking all in one step, e.g.,

```
g++ test.cpp -o test -I/usr/share/R/include
-L/opt/acml5.2.0/gfortran64_mp/lib -lacml_mp -lR
```

For openMP, you need to include *-fopenmp* when compiling.

To run your program, you'll generally need to tell the operating system that it's an executable:

```
chmod guo+x test
```

Also, the directory your program is in will generally not be in your path, so you need to do:

```
./test
```

1.4 Variables, types, and pointers

First a bit of preparatory material.

An address in memory is stored as a 32-bit value on 32-bit processors and a 64-bit value on 64-bit processors. Usually addresses are written in base 16 (hexadecimal) so 32 bits can be represented as 8 digits, e.g., *0x13fe36a7*. Before 64-bit processors, the number of locations in memory that could be uniquely addressed limited the amount of physical memory that could be used to about 4 Gb.

Variables are stored in memory at these various addresses. So for example, a numeric value named *y* might be stored as a double precision floating point (a 'double') at address *0x13fe36a7*. When you tell the computer to do something with *y*, it goes to the memory location, grabs the value there and then manipulates it as you have requested. The compiler tries to optimize things such that it doesn't have to do a lot of fetching, so values are often stored in intermediate locations (e.g., registers) that can be accessed more quickly. Also, it's faster to access contiguous locations in memory than locations spread all around. So if you, as in C++, often have matrices stored as a vector, row-wise, it's faster to grab a row than to grab a column.

In R and other scripting languages, you do not have to define variables before using them and variables are not associated with a particular type (e.g., *double*, *integer*, *character*) until they are assigned a value. In compiled languages such as C++, before you use a variable, you need to declare the variable and give it a type. You have to declare the variable earlier in the code file than the variable is used. Here are some examples of declaring variables, as well as defining them (see also *types.cpp*). Defining a variable also involves allocating memory for it, which occurs when you assign a value to the variable here.

```
int i;
int i = 7;
double x, y;
double x=0.0;
```

To get the value of a variable in your code, we just use the name, e.g.:

```
y = 2.3 * x;
```

To switch between types, you need to *cast* the variable, e.g.,

```
x = (double) i;
```

To get the address of a variable, we can do the following using the address operator:

```
&x;
```

Suppose we did:

```
z = &x;
```

What type does *z* have? Well, first we have to declare *z*, of course. The type of *z* needs to be a *double pointer*. A pointer is a special variable whose value is an address.

```
double* z;
z = &x;
```

If we want to get the value stored in *x* via *z*, we can use the '*' operator, the dereferencing operator:

```
y = *z;
```

Or for assignment to x:

```
*z = y;
```

Next note that if we change what is stored in *x*, **z* refers to the new value, because *z* is just the address of *x* (which is unchanged) and we have changed the value of *x*:

```
x = 9.0;
*z == 9.0; // this should now be True
```

Similarly if I modify **z*, then *x* is modified.

To store the address of *i* we would need to declare an *int pointer*:

```
int* iptr = &i;
```

Note you can do either of the following:

```
double* x;
double *x;
```

I prefer `double* x` because I can think of *x* as being a pointer, instead of thinking of **x* as being a double.

C++ provides string operations through the *string* class. You'll need `#include <string>`. This code is in the *string.cpp* file:

```
string s = "Hi there.\nWelcome.";
cout << "Length of s:  " << s.size() << endl; // s is an object of
class string, which has a size() method
cout << "First element:  " << s[0] << " and fourth element " << s[3]
<< endl;
s = s + " And goodbye."; // overloaded operator!
cout << s << endl;
```

C++ has lots of different types, but we'll stick with those for now; assuming you're primarily doing numeric computation in C++, you may not need to know that much more. If you're going to do a lot of work with text, I'd suggest you use Python. Doubles are stored as 8 bytes (64 bits), ints as 4 bytes, characters (char) as 1 byte. There are also long doubles, stored as 16 bytes for doing high-precision calculations. Doubles have about 16 digits of accuracy, while long doubles have twice as many digits of accuracy.

1.5 Scoping

Just like R, C++ has rules for variable *scoping*, which just means the locations in your code in which a particular variable or function name has meaning (i.e., can be understood by the compiler). Variables are generally local to a specific scope and can't be accessed outside that scope. Variables that are declared ouside a set of curly braces (a code block) will act as global variables. Here's an example:

```
#include<iostream>
using namespace std;
```

```
double z; // z is available throughout
void myfun(double* input, double* output) {
  double x = 3.0; // x is only available w/in the function and below here
  {  // new code block
    double y = 5.0; // y is only available within this set of braces
    // other code here
  }
  // y no longer available here
}
```

You don't have to include `using namespace std;`, but it does allow one to access the std functions/objects without the :: operator.

You could instead directly access, e.g., *cout* as *std::cout*;

```
std::cout << "hi" << std::endl;
```

Or you could just provide direct access to *cout* and *endl*, but not all of the *std* namespace:

```
using std::cout; using std::endl;
```

Note that in C++ variables can be declared anywhere in a block of code, provided they are declared before being used, while in C they need to be at the start of the block.

Global variables also need to be declared before they are used.

1.6 Input/output (I/O)

I won't go into this much because I'll assume that C++ is being called from R or another language. Dealing with I/O in C++ is a hassle, and if you can leave it to a higher-level language and just use C++ for the heavy-duty number crunching, that's usually a good strategy. However, you may need to insert print statements in your code, both to give informative messages and for your own debugging. We've basically already seen the syntax in C++, which is as follows.

Make sure to include `#include <iostream>` as a preprocessor directive. Then do things like the following:

```
cout << "The value of x is:  " << x << ", and it should be 7."  <<
endl;
```

Basically you can just glom together with << anything you want to print out. The *endl* means to print out a newline (i.e., a return ("\n")).

You can also set up output to a file by creating a 'file stream' and printing to the stream rather than to *cout*, which goes to *stdout*.

1.7 Arrays and memory allocation

What about vectors, matrices, etc? In C++, these are called *arrays*. A basic one-dimensional array can be created as follows:

```
double data[10];
```

You can refer to a value in the array as: `data[0]`, `data[1]`, ..., `data[9]`. C++, unlike R, is 0-indexed, so the first element is the 0th and the last is the (n-1)th.

It's useful to know that an array is the same as a pointer to the beginning of the array. So *data* is just the address of *data[0]*.

One construct that programmers use is something called pointer arithmetic, which is illustrated in *ptrArith.cpp*.

Suppose I do this:

```
*(data + 1)
```

The +1 says to add 1 to the address stored as *data*. This basically moves the pointer along to the next memory location (jumping 8 bytes since data refers to an array of doubles – because it's a double pointer). So `*(data+1)` is the same as `data[1]`. `*data` is the same as `data[0]`.

We can do so-called pointer arithmetic:

```
double* next = data + 1
```

next is now a pointer to `data[1]`. It's the same as if we did `&(data[1])`.

Variables declared as we have done so far, such as `int i=7; double x=0.0; double data[10];` are allocated from the *stack*. As soon as they go out of scope, the memory for these variables is freed up.

You can also dynamically allocate memory for arrays, e.g., when you are not sure at compile time how big the array needs to be. In C++, we do this as:

```
double* data = new double[n];
```

Or with some error checking

```
if (( double* data = new double[n]) == NULL)
cerr << "Error allocating memory for 'data'."  << endl;
```

The length of data will depend on the value of *n*. Such memory is allocated from the *heap* and is not freed unless you explicitly free it (or the program ends). If you don't free it, and if you use *new* within a loop, you will keep using up memory without freeing it, resulting in potentially massive memory use. This is called a *memory leak*. To avoid this; when you are done with the array, in particular before the function in which the array was allocated finishes, do

```
delete[] data;
```

A basic and important check of your code is to make sure there is a *delete* for every *new*. Sometimes this is a bit tricky because you may allocate memory in a function to hold the output and forget to free it later outside of the function.

You can create multi-dimensional arrays with of fixed sizes as follows:

```
double x[N][N];
```

Here's a program (*array.cpp*) that shows this and demonstrates that matrices are stored by row (row major), unlike in R. It also demonstrates that the two dimensional array is actually stored as a one-dimensional array of pointers to rows of the 2-d array.

```
#include <iostream>
#include <iomanip>

using namespace std;
// g++ -o array array.cpp
int main() {
  const int N=3;
  double x[N][N];
  x[0][0]=0.0; x[0][1]=1.0; x[0][2]=2.0;
  x[1][0]=3.0; x[1][1]=4.0; x[1][2]=5.0;
  x[2][0]=6.0; x[2][1]=7.0; x[2][2]=8.0;
  cout << x << ' ' << *x << ' ' << **x << endl;
  cout << x[0] << ' ' << x[1] << ' ' << x[2] << endl;
  cout << x[0][0] << ' ' << x[0][1] << ' ' << x[0][2] << ' '
     << x[1][0] << endl;
  cout << &(x[0][0]) << ' '  << &(x[0][1]) << ' ' << &(x[0][2])
     << ' ' << &(x[1][0]) << endl;
  cout << *(*x) << ' ' << *(*x+1) << ' ' << *(*x+2) << ' ' << *(*x+3)
     << endl;
  cout << **x << ' ' << **(x+1) << ' ' << **(x+2) << ' ' << **(x+3)
     << endl;
  return 0;
}
```

That code produces the following, which we can interpret based on a bit of base-16 arithmetic:

```
paciorek@smeagol:~/staff/workshops/CfromR> ./array
0x7fff6cb1b6f0 0x7fff6cb1b6f0 0
0x7fff6cb1b6f0 0x7fff6cb1b708 0x7fff6cb1b720
0 1 2 3
0x7fff6cb1b6f0 0x7fff6cb1b6f8 0x7fff6cb1b700 0x7fff6cb1b708
```

```
0 1 2 3
0 3 6 6.36599e-314
```

1.8 Functions

In general, all variables are local to the function in which they are defined (there are ways around this but it's often not good practice).

1.8.1 Pass by reference vs. pass by value

A key distinction between C++ and R is that in C++ you can pass variables into a function either *by value* (as with R – but notes that R often cleverly avoids or delays copying inputs and outputs from a function) or *by reference*. Passing *by reference* just means that you pass a pointer into the function. When you modify the value pointed to by that pointer, the value is changed globally in the sense that if you later access the value in that memory location outside the function, the value reflects the changes made inside the function.

Here is an example (*func.cpp*) of passing by reference vs. value and the implications of doing so:

```
#include <iostream>
#include <iomanip>
using namespace std;
// g++ -o func func.cpp

void myfun (double f_xval, double* f_xref, double* f_aval,
                                           const double* f_aref) {
  f_xval = 5.0;
  *f_xref = 5.0;
  f_aval[1] = 5.0;
  // f_aref[1] = 5.0;  // compilation error: "assignment of
                       // read-only location '*(f_aref + 8u)' "
}

int main() {
  const int N = 3;
  double x[N] = {0.0, 0.0, 0.0};
  double y[N] = {0.0, 0.0, 0.0};
  double xval = 0.0;
  double* xref = new double(0.0);
  // why does this fail?:
  // double* xref;
  // *xref = 0.0;
  myfun(xval, xref, x, y);
  cout << xval << ' ' << *xref << ' ' << x[1] << ' ' << y[1] <<  endl;
  *xref = 0.0; xval = 0.0; x[1] = 0.0; y[1] = 0.0;
  myfun(*xref, &xval, x, y);
  cout << *xref << ' ' << xval << ' ' << x[1] << ' ' << y[1] <<  endl;

  return 0;
}
```

Note that even the static array is treated as a pointer.

1.8.2 Default parameter values

You can use default parameter values, e.g.:
`void myfun (double a = 0.0, double b = 1.0, double c = 2.0)`
Unlike in R, if a user omits one of the arguments, they must omit **all** of the **following** arguments as well. Also note that C++ matches by position and not by name (unlike in R, which does both). In the code below (*funcArgs.cpp*), we haven't called `myfun()` with arguments 'a' and 'c', rather we've assigned to 'a', and 'c' and then called `myfun()` with the values stored in 'a' and 'c', which are taken to be the 'a' and 'b' arguments to `myfun()`.

```
#include <iostream>
#include <iomanip>
using namespace std;
// g++ -o funcArgs funcArgs.cpp
void myFun (double a = 0.0, double b = 1.0, double c = 2.0){
  cout << "a is" << a << ", b is " << b << ",c is " << c << endl;
}

int main() {
  double a, c;
  myFun();
  myFun(10.0);
  myFun(a = 10.0);
  cout << "In main(): a is " << a << ", c is " << c << endl;
  myFun(a = 10.0, c = 30.0);
  cout << "In main(): a is " << a << ", c is " << c << endl;
  return 0;
}
```

Here's the result:

```
a is 0, b is 1, c is 2
a is 10, b is 1, c is 2
a is 10, b is 1, c is 2
In main(): a is 10, c is 0
a is 10, b is 30, c is 2
In main(): a is 10, c is 30
```

1.8.3 Some other syntax stuff

If we want to pass by reference for efficiency, but protect the object passed in from being modified, here's good standard practice:

`int myFun(const double* x)`

This prevents modification of what 'x' points to (the scalar or array of doubles).

The *static* and *extern* keywords, e.g.,

```
static double x;
extern double x;
```

are used for messing around with scoping, such as creating global variables and variables declared in a function that do not die when a function finishes.

Functions need to be defined before they are used. You can have a function be defined below where it is used, but in this case you need to have the function declared (e.g. with the line, `int myFun(double x);`) above where it is used.

1.9 Basic syntax: math and flow control

Here's a basic for loop:

```
int i; int n=30; double x = new double[n];
for(i = 0; i < n; i++){
  x[i] = (double) i * 3.0;
}
```

and a basic while loop:

```
int i = 0; int n = 30; double x = new double[n];
while(i < n) {
  x[i] = (double) i*3.0;
  i++;
}
```

Here's a basic if-then-else statement. Note that the boolean operators &&, ||, ==, and != are as they are in R.

```
if((i < 7) && (i > 3)) {
  cout << medium << end;
}
else if((i > 12) || (i < 0)){
  cout << extreme << endl;
}
else{
  cout << something else << endl;
}
```

Ok, what about basic mathematical calculations? See http://en.wikipedia.org/wiki/C_mathematical_functions. Unlike in R, you need an explicit function (not the ^ operator) to take the power of a number: `pow(x, power)`.

For generating random numbers, I recommend using the C functions provided by R. More details in a bit.

1.10 Standard template library

The standard template library (STL) provides a lot of tools for easing your coding.

First, add the *stl* namespace (just below your preprocessor directives and before any function definitions) so that you can refer directly to the objects that you want (otherwise you'd need to refer to them as *std::object* (as with an R package that you haven't loaded)):

```
using namespace std;
```

The STL provides *container* types that can represent objects containing other objects. One of these is a *vector* class that is like an array but can also grow itself. Here's basic usage of the vector class (but I'm not showing how to grow the vector). Note you need `#include <vector>` as a preprocessor directive:

```
vector<double> myvec(N);
for(int i = 0; i < myvec.size(); i++) {
  myvec[i] = (double) i;
}
```

There are also related containers called *lists* and *deques* - which you use depends on whether you want to be able to insert/delete elements anywhere or just at the endpoints and whether you want access to any element anytime or always do this in sequential order.

STL provides something called *iterators* that allow you to index through your container. Iterators behave like pointers:

```
int index = 0;
for(vector<double>::iterator iter = myvec.begin();
                  iter != myvec.end(); iter++) {
  *iter = *iter + 3.0;
  cout << index++ << ": "  << *iter << endl;
}
```

STL provides algorithms for searching, sorting, applying an operation to each element (like an R *apply*), and insertion/deletion, among other things.

As you could guess, there are lots more details here; I just wanted to make you aware of the STL.

1.11 Structs and classes, plus function/operator overloading

In C, the variable type that allows you to aggregate a variety of disparate objects (like an R *list*) is a *struct*. In C++, one generally uses *objects* since an emphasis in C++ is on OOP. An object is a data structure with methods (i.e., functions) that operate on that structure. Unlike R, C++ emphasizes that some members and functions are private and can't be accessed from outside the object, while others are public.

One nice feature in C++ is the ability to overload functions and operators. That is you can define functions that work with different types for one's inputs. You can do the same sort of thing with operators. Note that operators might be member functions of a class. Here's an example of function/operator overloading:

```
void foo (int i) { cout << "You sent me an int:  " << i << endl; }
void foo (double d) { cout << "You sent me a double:  " << d << endl;}
```

Here's an example (*class.cpp*) of defining a class, with a constructor and overloading the (operator to access private member objects.

```
#include <iostream>
#include <iomanip>
using namespace std;
// g++ -o class class.cpp
class badMatrix {
// 'bad' because you wouldn't want reinvent the wheel
//    with your own matrix class!
 private:
  int n_rows, n_cols;
  double* vals;
 public:
  // declarations w/o actually defining the member functions
  badMatrix(int n_rows, int n_cols, double* vals); //constructor
  ~badMatrix(void); // destructor
  // overload the ( operator to do subsetting
  double operator() (int row, int col);
};

// here are the method definitions

badMatrix::badMatrix(int n_rows, int n_cols, double* vals) :
  n_rows(n_rows), n_cols(n_cols) //succinct initialization
{
  badMatrix::vals = new double[n_rows * n_cols];
  for(int i = 0; i < n_rows*n_cols; i++)
    badMatrix::vals[i] = vals[i];
}

badMatrix::~badMatrix(void) {
  delete [] vals;
  cout << "Deallocating memory" << endl;
}
```

```
double badMatrix::operator() (int row, int col) {
  return vals[row*n_cols + col];
  // note you could even write this to mimic 1-based indexing
}

int main() {
  double vals[6] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0};
  badMatrix mymat(2, 3, vals);

  cout << "mymat(0,1): " << mymat(0,1) << endl;
  cout << "mymat(1,0): " << mymat(1,0) << endl;

  return 0;
}
```

We can access members of a class with the . operator:
`myObject.myMethod()` or `myObject.myMember`
Or if we have a pointer to the object:
`badMatrix* myObjectPtr = new badMatrix(2, 3, vals);`
`myObjectPtr->myMethod()` or `myObjectPtr->myMember`
That's just shorthand for `(*myObjectPtr).myMethod()` or `(*myObjectPtr).myMember`.

1.12 Linear algebra and multi-dimensional arrays

There are a variety of ways for dealing with matrices and arrays of more than one dimension. One approach is just to have the matrix strung out as a 1-d array. This is the approach needed if you are using Lapack and BLAS calls. You'll have to be careful about row-major vs. column-major ordering (from R, this generally works without much thought as R and Fortran store matrices column-major and the underlying Lapack routines are Fortran, but from C++ directly you may need to do more).

In addition to Lapack, other options for doing linear algebra are the Eigen and Armadillo packages, which have connections to Rcpp via *RcppEigen* and *RcppArmadillo* (more later). These provide C++ class structures with linear algebra methods.

1.12.1 BLAS and Lapack calls

In the initial example, we've already seen the basics of making BLAS/Lapack calls.

BLAS is the Basic Linear Algebra Subroutines, which carry out things like matrix-vector multiplication. *Lapack* is the Linear Algebra PACKage, which carries out more complicated linear algebra operations such as Cholesky and SVD factorization. So long as you have a threaded BLAS on your system that you are linking against, your linear algebra calls will be able to use multiple cores.

The basic steps in a C++ program are:

1. declare the BLAS/Lapack function you want to use; noting the use of extern and inclusion of the _, which has to do with the functions being actually written in Fortran:
   ```
   extern "C" int dpotrf_(char* uplo, int* n, double* a, int* lda,
   int* info);
   ```
2. Call the function appropriately, including passing in the required arguments as the right types (e.g., pointers in many cases – here I just pass in the address of anything not already a pointer):
   ```
   dpotrf_(&uplo, &size, C, &size, &info);
   ```
3. Make sure to link appropriately when compiling your code:
   ```
   g++ -o prog prog.cpp -llapack -lblas -O3 -Wall
   ```
   Alternatively to link against non-standard BLAS or Lapack (here against ACML, which is a good choice on the SCF Linux cluster):
   ```
   g++ -o prog prog.cpp -L/opt/acml5.2.0/gfortran64_ mp/lib -lacml_mp
   -lgfortran
   ```

But, you might comment, this all seems a bit antiquated. We're talking about C++, but are calling obscurely-named functions with matrices as one-dimensional arrays. There are a number of ways to work with linear algebra in a more modern fashion that have better APIs (the interface you use to interact with the functionality).

1.12.2 Eigen and Armadillo

One nice package is *Eigen*. It does not use the BLAS or Lapack, but is advertised as being fast. It can use threading for some of its algorithms, via openMP, so just use the *-fopenmp* flag when compiling with g++/gcc. It has some functionality for sparse matrices. Note that Eigen may not be available on Macs.

The file *EigenExample.cpp* has an example of doing a Cholesky decomposition using Eigen in two different ways. Note the use of OOP. Also note the use of templates (see the <> syntax), which allow programmers to easily write a method that can be used on multiple different types of input.

Armadillo is another option. Armadillo has pretty straightforward syntax that looks like R and Matlab (we'll see this in the section on Rcpp). Behind the scenes it does rely on BLAS/Lapack, unlike Eigen. From various mailing lists, the rough consensus is that Eigen may be a bit faster, but this is heavily dependent on what BLAS/Lapack you have on your system (since Armadillo uses those and Eigen does not), but Armadillo is probably easier to use. Armadillo is probably safer since BLAS/Lapack have been used for decades and are very reliable. Armadillo does not have much functionality for sparse matrices. We'll see a bit of Armadillo in the context of calling it via Rcpp from R.

Both Eigen and Armadillo make use of something called template meta-programming in which the compiler optimizes the linear algebra steps at a high level before even getting to optimizing the compilation of the explicit code. What do I mean by this? An example would be calculating A-B+C without first doing A-B, storing it temporarily and then doing the addition.

1.13 Calling R's C functions (such as rnorm, dnorm, etc.) and R's BLAS/Lapack

One of R's best features is the ease with which we can manipulate distributions and generate random numbers. We can use this functionality in C++, remembering that R's functions are just C code, so all we need are the right header file (*Rmath.h*) and the right shared object library (*libRmath.so*).

Here's an example where I use R's underlying C code distributional functions directly in C++. Compilation is done with `g++ -I/usr/share/R/include -lRmath`. You might need `-lR` in addition (or instead).

```
#include <Rmath.h>
#include <R.h>
void genRandN (int* size, double* x, double* dens) {
// this assumes space for 'x' and 'dens' are allocated outside
  getRNGstate();
  for(int i = 0; i <size; i++){
    x[i] = rnorm(1.0, 3.0);
    dens[i] = dnorm(x[i], 1.0, 3.0);
  }
  putRNGstate();
}
```

```
// would need a main() or would call this from R
// if call from R, allocate memory for 'x' and 'dens' in R
```

Note that R's C distribution functions do not operate in a vectorized fashion - you need to explicitly iterate as above.

The getting and putting of the RNG state involves reading in or creating *.Random.seed* and writing it out. According to the R extensions manual there is no way to select the kind of RNG or set the seed from C++. The default RNG when done from C++ should be the default in R, the Mersenne twister. Note that if you are calling C++ from R, you can set the seed in R (with *set.seed()*) and set the RNG type with *RNGkind()* and that propagates to the RNG in C++. This seems like magic, and I don't see it documented anywhere, but one can see that it works empirically.

R also provides C functions for a lot of useful mathematical functions: *gammafn*, *beta*, *bessel_j*, *choose*, etc. And it provides a variety of mathematical constants in *Rmath.h*. For details see Section 6.7 of the *R extensions manual*. You can call the C code underlying R's *optim()* function from C too.

Finally, you can call R's C interfaces to BLAS and Lapack based on the header files: *R_ext/BLAS.h*, *R_ext/Lapack.h*. In particular, if you are calling C++ from R and then using BLAS/Lapack, this may be easier than calling the system's BLAS/Lapack directly, as (I believe) this will avoid linking issues since R already links in the BLAS and Lapack. However, you'll generally need a *Makevars* file in the current directory (or in */src* in an R package directory) with the contents:

```
PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS)
```

so that R CMD SHLIB can link properly.

Finally, if you are using Rmath functions in a stand-alone C++ program rather than calling from R, you may need to have the line #define MATHLIB_STANDALONE at the top of your file, before the #include preprocessing directives.

If calling C++ from R you can allocate memory either in R or in C++, but it's generally safer to do in R as you don't need to worry about memory leaks.

1.14 Debugging

1.14.1 Debugging tools

Using debugging tools is much more important with a compiled language than a scripting language because we can't just step through the code line by line. The standard debuggers are *gdb* (a command line debugger) and *ddd* (a GUI). Both are available on the SCF Linux machines.

First, you need to compile your program in a way that allows for debugging (this slows down the execution, so don't do this for production code), by using the *-g* flag, e.g.

```
g++ -g -o myProg myProg.cpp
```

To invoke *gdb* for a particular program:

```
gdb myProg
```

We'll do a demo in the workshop but below I list the basic syntax used. The basic idea is to set breakpoints inside your code at places where you think it will help to examine the state of the code and then walk through your code and check the values of variables and see the flow of execution. If you're familiar with the R debugger invoked by *browser()*, many of the ideas/syntax are similar.

Here's some example syntax for gdb.

- Set breakpoints at a particular line of code and/or a particular function call:
  ```
  b myProg.cpp:123
  b myFun
  ```
- Get info on the breakpoints that are set
  ```
  info breakpoints
  ```
- Run your program (it will then stop at the breakpoints)
  ```
  r
  ```
- Continue to the next break point:
  ```
  c
  ```
- Continue to the next line:
  ```
  n
  ```
- To step inside the function called on the next line:
  ```
  s
  ```
- Printing values of variables
  ```
  p non_pointer
  p *pointer
  p *myvec@20
  ```
- Setting variables' values
  ```
  set it=7
  ```

1.14.2 Common bugs

One standard bug is memory leaks. Another standard bug is to try to write to a location for which memory has not been allocated. Here's an example (*bug.cpp*):

```
const int N = 5;
double x[N] = {0.0, 0.0, 0.0, 0.0, 0.0};
for(int i = 0; i <= N; i++) {
  cout << x[i] << endl;
  x[i] = double(i);
  cout << x[i] << endl;
}
```

1.14.3 Assertions and try

One way to insert checks in your code is with *assert()*. E.g., to check that a value is bigger than zero:

```
assert(n > 0);
```

If *n* does not satisfy this, an error is thrown, with indication of where the error occurred.

C++ also has try/catch functionality, similar to *try()* in R, that can deal with errors gracefully.

1.15 Using *make*

It can be tedious to deal with the steps involved in compiling and linking a program, particularly when you have your code across many files.

The *make* utility allows you to automate this. Here's a basic example. The syntax is to define in each block the target of that block (i.e., what will be produced) and the dependencies needed to make that target, and then the commands that generate the target. Here's an example. The file should be called *makefile*, as that is where *make* looks for the file:

```
myProg : myProg.o auxil.o
   g++ myProg.o auxil.o -o myProg
myProg.o : myProg.cpp auxil.h
   g++ -c myProg.cpp -o myProg.o
auxil.o : auxil.cpp
   g++ -c auxil.cpp -o auxil.o
```

Note that the code lines must be preceded by a tab. One nice feature is *make* checks if files have been changed and only executes a block if any of the dependencies have changed. The -c flag tells the compiler not to do any linking for those steps.

If you now invoke *make* it will carry out the operations using whatever file named *makefile* is in the current directory.

In fact, you could even use *make* more generally to do reproducible research by defining each task and its component operations as shell commands. Here are some ideas along those lines: http://kbroman.github.io/minimal_make/.

2 Calling C/C++ from R

In general, the R extensions manual on CRAN is your guidance for all things related to calling external code.

As a result of R being written in C, R provides a number of C types and functions/macros for manipulating R objects within C. Most of these will come up in the context of *.Call()*.

2.1 Compiling C++ code into a shared object library usable by R

Basically, you write one or more C++ functions that you will call from R. Note that your C++ code should not include a *main()* function, just the functions you want to call from R. It does need to include the usual preprocessing directives, and in general should also have `#include <R.h>`. For R to work with C++ code (or even C code compiled with g++), you need to wrap your functions inside an an *extern* statement: `extern "C" { yourC++_code_here.... }`. However, I believe that any C++ class and class method definitions should be done outside of the *extern* statement.

Once you have the C++ code, you need to compile it. A standard way to do this using an R wrapper to the compiler is as follows:
`R CMD SHLIB fancyPlus.cpp`
This will automatically invoke something like this:

```
g++ -I/usr/share/R/include -DNDEBUG -fpic -O3 -pipe -g
   -c fancyPlus.cpp -o fancyPlus.o
g++ -shared -o fancyPlus.so fancyPlus.o -L/usr/lib/R/lib -lR
```

If you need to link against additional libraries or use additional include flags, you can do the compilation manually with g++ (mimicing the above but with the added flags), or you can just pass the additional arguments directly to R CMD SHLIB, e.g., `R CMD SHLIB fancyPlus.cpp -I/usr/share/include -L/usr/lib`. Note that you may need to ensure `-fopenmp` is part of the first g++ call (the compilation step), and not just the linking step. You can also set flags via a *Makevars* file (see Section 5.5 of the R extensions manual) if you're compiling the code within the context of an R package.

2.2 The *.C* interface

This is the most basic interface. I've heard/read some people disparage it as being the 'old' way of doing things, but for basic functionality, it's straightforward and may be quite useful. I'll demonstrate by calling a C++ function to add together two vectors, element-wise, with the C++ code in *fancyPlus.cpp* and the R code that calls C++ in *fancyPlus.R*. The .C functionality is really only intended for passing standard vectors (not including lists) and dealing with character vectors is a bit tricky. For basic numerical vectors as arguments, the steps involved are:

1. Write a function (or functions) in C++ that returns void. All of your arguments for the C++ function should be pointers.

2. Compile that function into binary in a UNIX shell, creating a .so file (shared object library). The compiled function will be an object in the .so file that can be accessed from R. R CMD SHLIB does this for us, and can generally deal with the include files and library linking automatically.

   ```
   R CMD SHLIB fancyPlus.cpp
   ```

   Note that you can check that the function you expect is in the .so file as follows:

   ```
   nm fancyPlus.so | grep ' T '
   ```

3. Load the .so in R:

   ```
   dyn.load("fancy.so")
   ```

4. Create a wrapper function that allows you to call the C++ code with minimal hassle (you can just directly use *.C()*, but it's often nice to have the wrapper)

```
fancyPlus <- function(in1, in2) {

# R wrapper to the C function

  if(length(in1) != length(in2))

    stop("Lengths of vectors must match")

  if(!is.numeric(in1) || !is.numeric(in2))

    stop("Must provide numeric vector inputs")
```

```
  return(.C('fancyPlus', length = as.integer(length(in1)),

    input1andOutput = as.double(in1),

    input2 = as.double(in2))$input1andOutput)

}
```

Note that you may not need the *as.double()*, *as.integer()*, etc. (and can avoid the extra computation and memory use) if you are careful about setting up the R variable types in advance.

5. Now you can use the C++ function without even being aware that C++ is being called: `z <- fancyPlus(x, y)`

Note that the .C call passes pointers into the C++ function (note that the C++ function arguments are pointers). However, there actually are copies made of all the arguments. You can see this in the demo code in *fancyPlus.R*, which uses R's internal inspect function (`.Internal(inspect(obj))`) to examine memory addresses of R objects.

The main things to remember in using *.C()* are:

1. Make sure the R types match what the C++ function is expecting, with R's *numeric* type corresponding to C++'s *double*. Often this will involve using functions such as *as.double()* and *as.integer()*.

2. Figure out whether the output from C++ can just overwrite one of the inputs. Alternatively, a good approach is to create a new output object in R and pass it as one of the arguments to the C++ function. Returning output using pass by reference is the only way to get output back from C++.

3. If you allocate memory in C++, make sure to free it so you don't create a memory leak. This is particularly important if you call the C++ function over and over again.

4. In your C++ code you may need to add particular header files that contain the function definitions for external library functions that you use.

5. As noted previously, you may need to include some linking arguments or header arguments to R CMD SHLIB

2.3 The *.Call* interface

This is a more modern approach than .C and allows you to handle R data structures inside of C++. It's a bit complicated because it involves understanding how R objects are stored in C. We'll look at the basics, but the general plan is to focus on Rcpp (see later sections of this document) which allows you to ignore a lot of the messiness associated with .Call.

On the R side, things look like .C, but you can pass more general R objects:

```
.Call('myFun', a, b)
```

On the C++ side things look something like this:

```
#include <R.h>
#include <Rinternals.h>
SEXP myFun(SEXP a, SEXP b) {
...
}
```

Yikes, what's a SEXP?

It stands for "S Expression". It's a pointer to a C struct, with the struct able to handle all types of R objects. So at the C level, all R objects are SEXPs and all .Call's look like:

```
SEXP myfun(SEXP in1, SEXP in2, ..., SEXP inp) // in C
.Call('myfun', var1, var2, ..., varp) # in R
```

Some of the types of SEXP's are: REALSXP (numeric); INTSXP (integer); LGLSXP (logical); VECSXP (list); CLOSXP (function/closure); ENVSXP (environment), among others (see Section 5.9.3 of R extensions manual).

If you create an R object in C, you need to tell R about the object by wrapping the creation in PROTECT() in order to prevent R from garbage collecting it. Note that in simple cases R may never be invoked, but the *R extensions manual* argues that in other cases, R calls might be made while the C code is running, so it's best to always use PROTECT(). Here's a basic example of creating an R object:

```
PROTECT(ab = allocVector(REALSXP, 2));
REAL(ab)[0] = 123.45;
REAL(ab)[1] = 67.89;
// now ab is a 2-vector of reals stored in the form of an R vector
UNPROTECT(1); // UNPROTECT unprotects the last n objects
              //protected (so in this case just the last one)
```

UNPROTECT removes the protection and should be done at the point in the code at which you're done with the object. Note that we've referred to the values stored within the REALSXP using

REAL(ab). I believe this basically provides a pointer that accesses the storage of the vector within the R object. If this code were used in a function called from R via .Call, you could pass *ab* directly back to R and it would be a regular R vector.

Here's a more extended example that shows some more type conversions as well as manipulating attributes of R objects.

```
#include <R.h>
#include <Rinternals.h>
SEXP out(SEXP x, SEXP y)
{
  int i, j, nx = length(x), ny = length(y);
  double tmp;
  double* rx = REAL(x);
  double* ry = REAL(y);
  double* rans;
  SEXP ans, dim, dimnames;
  PROTECT(ans = allocVector(REALSXP, nx*ny));
  rans = REAL(ans);
  for(i = 0; i < nx; i++) {
    tmp = rx[i];
    for(j = 0; j < ny; j++)
      rans[i + nx*j] = tmp * ry[j];
  }
  PROTECT(dim = allocVector(INTSXP, 2));
  INTEGER(dim)[0] = nx; INTEGER(dim)[1] = ny;
  setAttrib(ans, R_DimSymbol, dim);
  PROTECT(dimnames = allocVector(VECSXP, 2));
  SET_VECTOR_ELT(dimnames, 0, getAttrib(x, R_NamesSymbol));
  SET_VECTOR_ELT(dimnames, 1, getAttrib(y, R_NamesSymbol));
  setAttrib(ans, R_DimNamesSymbol, dimnames);
  UNPROTECT(3);
  return(ans);
}
```

Note that with *.Call()*, what is returned is an R object constructed in C++ and returned via *return()* in your C++ function.

You can extract elements from list SEXPs (i.e. VECSXP's) as follows. Suppose you have an R list, `a <- list(f = 1, g = 2, h = 3)`, that is passed into your C++ function as an argument named 'a' via *.Call()*. Then in C++ you can do the following (remembering to switch to 0-based indexing):

```
double g;
g = REAL(VECTOR_ELT(a, 1))[0]; // where the 1 is the 2nd element of
the list and the 0 is the first element of that.
```

The R extensions manual (p. 112) provides a wrapper function that can be called more easily as

```
g = REAL(getListElement(a, "g"))[0];
```

I suspect you'll agree with me that this all seems pretty complicated and has us needing to understand how R works at the C level more than we would like. It's probably more than you need for doing basic computation in C++, for which you might just use .C, but gives you an idea of some of what is possible, and, indeed, of how R's internal functions are written (e.g., things that are called via *.Internal()*):

```
matrix

## function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
## {
##     if (is.object(data) || !is.atomic(data))
##         data <- as.vector(data)
##     .Internal(matrix(data, nrow, ncol, byrow, dimnames, missing(nrow),
##         missing(ncol)))
## }
## <bytecode: 0x17d47e0>
## <environment: namespace:base>
```

2.4 Allocating memory in C++

We've seen allocation of R objects (e.g., the *allocVector()* call in the code above). To allocate standard C/C++ objects there are a couple options.

You can allocate memory with *malloc* (in C) and *new* (in C++), but of course being careful to free it.

Alternatively, you can use the *R_alloc()* function, which causes memory to be freed when .C/.Call finishes. Here's the function signature (note it looks like *malloc()* and the like)

```
char *R_alloc(size_t n, int size );
```

and here's a typical use (for 100 ints), noting the casting as an integer pointer:

```
x = (int *) R_alloc(100, sizeof(int));
```

2.5 Print statements

The *R extensions manual* suggests not using C++ *iostreams* (e.g., *cout*). In particular it's suggested to use the C function *Rprintf()* or *REprintf()* to write to stdout or stderr, respectively. You (may) need the *R_ext/Print.h* header file. Here's an example call:

```
REprintf("This is an error message.\n")
```

2.6 Evaluating R expressions in C++

It's possible to evaluate R expressions in C++, which can be handy if we want to do something that is easily expressed in R code.

For this we use a C function provided by R, *expr()*, which has the following prototype:

```
SEXP eval(SEXP expr, SEXP rho);
```

which is like `eval(expr, envir = rho)` in R. It evaluates R code (in the form of an expression) in the context of an environment in which to look for the objects involved in the expression.

Here's a basic example of evaluating an R expression that we pass into C++, "calling back" to R to evaluate the function. In R (see *Ceval.R*), we have the usual sort of stuff for calling C++, in this case passing an expression (not a function) and an empty environment into C++.

```
system("R CMD SHLIB Ceval.cpp")
dyn.load("Ceval.so")

crazy <- function(x) {
  (x^2 + 1) * (x - 1.5) + besselK(x, 1)
}

wrapper <- function(f) {
  .Call('myfun', body(f), new.env())
}

wrapper(crazy)

## [1] 275

crazy(7)

## [1] 275
```

In C++ (see *Ceval.cpp*), I have my primary function, which calls a wrapper function that does the evaluation of the R expression, as well as an auxiliary function for creating R vectors.

```
#include <R.h>
#include <Rinternals.h>
extern "C" {

SEXP mkans(double x) {
  SEXP ans;
  PROTECT(ans = allocVector(REALSXP, 1));
  REAL(ans)[0] = x;
  UNPROTECT(1);
  return ans;
}

double feval(double x, SEXP f, SEXP rho) {
  defineVar(install("x"), mkans(x), rho);
  return(REAL(eval(f, rho))[0]);
}

SEXP myfun(SEXP f, SEXP rho) {
  double x = 7.0;
  double result = feval(x, f, rho);
  return(mkans(result));
}

}
```

Note that there's a bunch of overhead involved – starting with a C++ variable that we want to use in the R expression, we need to make it into an R variable (a REALSXP) and we need to place that R variable, named correctly (as ’x’ here) in an environment (*rho* here) in which the expression will be evaluated. The code illustrates taking the result of the evaluation, which is an R variable and pulling out the value as a C++ variable, which could then be used subsequently in the C++ code. [Note that for the purpose of checking that things are working, I convert back to an R variable and pass back to R as the result of .Call, so the conversion to a C++ variable and then to an R variable is redundant.]

2.7 Using C++ code directly in R via inline

You actually don't need to go to the trouble of creating a separate file of C++ code, compiling to a shared object library, and loading the .so. The *inline* package can handle all that behind the scenes, dealing with header files, compiling, linking, and loading the .so. *inline* allows you to use .C, .Call, .Fortran, or Rcpp as the back-end interface to the compiled language. We'll talk about Rcpp in the next section.

The key functions are *cfunction()* (for C code, using either .C, .Call, or .Fortran) or *cxxfunction* (for C++ code with .Call)

Here's an example, using .C (because I don't want to deal with .Call, which would be needed for C++ code), from *inline.R*:

```
library(inline)
src <- '
  for (int i = 0; i < *n; i++) {
    x[i] = exp(x[i]);
  }
'

sillyExp <- cfunction(signature(n = "integer", x = "numeric"),
   src, convention = ".C")

n <- as.integer(100)
x <- rnorm(n)
out1 <- sillyExp(n, x = x)$x
out2 <- exp(x)
identical(out1, out2)

## [1] TRUE
```

The timing seen by running the full code in *inline.R* is pretty encouraging both from the perspective that calling C is easy and quick and that R's vectorized calculations are very fast.

You can take a look at `help(cfunction)` for an example with .Call.

Sometimes, you may need to specify specific *#include* statements (via the *includes* argument to *cfunction()* or *cxxfunction()*), or -I flags (via *cppargs*) or library info (-l or -L flags via *libargs*).

2.8 Rcpp

Rcpp provides a bunch of tools that ease our interaction with C++ code. In particular we saw that with .Call there was a lot of coding overhead involved with object types and memory management. Rcpp provides a set of useful wrapper functions that allow us to work with R objects in C++ very easily, thereby avoiding a lot of that overhead.

2.8.1 Basic Rcpp via the inline package

Both *cfunction()* and *cxxfunction()* allow you to work with Rcpp. Here's a simple example (*basicRcpp.R*):

```r
library(inline)
src <- '
return wrap( as<int>(x) * as<double>(y) ) ;
'

fx <- cxxfunction( signature(x = "integer", y = "numeric" ),
   body = src, plugin = "Rcpp" )
fx(2L, 5)   # fx(2, 5) works fine too

## [1] 10
```

The *as<int>* / *as<double>* are dealing with the fact that x and y are passed as R objects, so *as* in this case converts to standard C++ int and double types that can be easily manipulated with standard C++ code. *wrap()* deals with passing back the result in the form of an R object, which is what .Call (and therefore *cxxfunction()*) is equipped to handle.

Note that *wrap* and *as* are part of the Rcpp namespace, so in some cases you'd need *Rcpp::wrap* and *Rcpp::as* or you'd need `using namespace Rcpp` in your C++ code.

2.8.2 Rcpp overview

Rcpp provides the *Robject* class, which is a wrapper around SEXPs that manages a lot of the ugliness involved in working with them. There are a bunch of derived classes (C++ classes can *inherit* from other classes) that build on the *Robject* class:
IntegerVector, *IntegerMatrix*, *Numeric(Vector|Matrix)*, *Logical(Vector|Matrix)*, *Character(Vector|Matrix)*, *List*, *Expression(Vector|Matrix)*, *Environment*, and *Function*. The classes support NAs.

Rcpp provides a bunch of standard functions for operating on *Robject* variables:

- () operator – for multi-dimensional indexing
- [] operator – for single-dimensional indexing
- *length()*
- *begin()*, *end()* – gives a pointer to the beginning/end of vector
- *push_back()*, *push_front()*, *insert()*, *erase()* – these add or remove elements from vectors.

2.8.3 An example

Let's see a simple implementation of our silly *exp()* function (see *sillyExpRcpp.R*). Surprisingly, it will turn out to not be so silly after all.

```
library(inline)
src <- '
  NumericVector vec(xin);
  for(int i=0; i<vec.size(); i++) {
    vec[i] = exp(vec[i]);
  }
  return(vec);
'
notSoSillyExp <- cxxfunction(signature(xin="numeric"), src,
  plugin = "Rcpp")
x <- rnorm(1e7)
library(rbenchmark)
benchmark(
  out <- exp(x),
  out <- notSoSillyExp(x),
  replications = 5,
  columns = c("test", "elapsed", "replications"))

##                       test elapsed replications
## 1            out <- exp(x)   1.425            5
## 2 out <- notSoSillyExp(x)   0.839            5
```

In repeated tests, the timing for the Rcpp code gave about a 25% reduction relative to simply using *exp()*, which is somewhat less of a speedup than seen above. Regardless, I find this hard to explain since R's *exp()* just directly goes to C.

2.8.4 as and wrap

We've already seen *as* and *wrap*. *as* takes a SEXP and convert to a C++ type, including the various *Robject* C++ classes provided by Rcpp, such as *NumericVector* and *List*. *wrap* takes a C++ type and converts it to a SEXP that R can then interpret. Sometimes *as* and *wrap* are being used behind the scenes, which is what happens in the example above. For example, the following does an implicit *as*:

`NumericVector x(vectorFromR);`

which is equivalent to

`NumericVector x = as<NumericVector>(vectorFromR);`

And here's an example of an implicit *wrap*:

`return(List::create(Named("x", 3), Named("y", 5)));`

equivalent to:

`return(wrap(List::create(Named("x", 3), Named("y", 5))));`

In contrast, in the example in the previous section, we didn't need *as* and *wrap* explicitly because we were using basic R vector types.

2.8.5 Some more details

An Robject is a pointer, as are SEXPs. We can make a complete (i.e., a deep) copy of Robjects or SEXPs using *clone()*. Here's an example (*RcppCopying.R*) that illustrates shallow vs. deep copies. It also illustrates how we can create data frames via Rcpp.

```r
library(inline)
src <- '
NumericVector x1(xs);
NumericVector x2(xs);
NumericVector x3(clone(xs));
x1[0] = 22;
x3[1] = 44;
return(DataFrame::create(Named("orig", xs),
Named("x1", x1),
Named("x2", x2),
Named("x3", x3)));
'
fun <- cxxfunction(signature(xs="numeric"), src, plugin = 'Rcpp')
fun(c(1,2,3))
```

```
##   orig x1 x2 x3
## 1   22 22 22  1
## 2    2  2  2 44
## 3    3  3  3  3
```

Here's a test of your understanding (and of types in R). Why is this different?

```
fun(1:3)

##   orig x1 x2 x3
## 1    1 22  1  1
## 2    2  2  2 44
## 3    3  3  3  3
```

Here's an example of manipulating and creating lists (*RcppList.R*). Note also the use of the STL string class.

```
library(inline)

src <- '
List inputs(inp);
std::string method = as<std::string>(inputs["method"]);
double tol = as<double>(inputs["tol"]);
int nIts = as<int>(inputs["nIts"]);
return(List::create(Named("methodX", method),
Named("tolX", tol),
Named("nItsX", nIts)));
'

fun <- cxxfunction(signature(inp = "list"), src, plugin="Rcpp")
myList = list(method = "BFGS", tol = 1e-6, nIts = 100)
fun(myList)

## $methodX
## [1] "BFGS"
##
## $tolX
## [1] 1e-06
```

```
##
## $nItsX
## [1] 100
```

2.8.6 Calling R functions

Rcpp can run any functions defined in your R session from within C++. This can be handy if there is functionality already written that you want to call or to use R code from a user without requiring them to know C++. Here's an example (*RcppWithRFunctions.R*).

```
library(inline)
# using a built-in R function
src <- '
Function rt("rt"); // define the Rcpp Function via its constructor,
                   // based on the R rt() function
return(rt(as<int>(n), 1));
'
fun <- cxxfunction(signature(n="integer"), src, plugin = "Rcpp")
set.seed(0)
rt(5, 1)

## [1]  0.722873 12.299773 -1.443844 -0.002051  2.149760

set.seed(0)
fun(5)

## [1]  0.722873 12.299773 -1.443844 -0.002051  2.149760

# using a user-defined function
myprodR <- function(a, b) return(a*b)
src <- ' Function myprodC("myprodR");
return(myprodC(3,5)); '
fun <- cxxfunction(signature(), src, plugin = "Rcpp")
fun()

## [1] 15
```

For some reason that last bit didn't work when called via the *knitr* package in creating this document, but it does indeed work in general.

Note that Rcpp will make use of the RNG state and type as it currently exists in R.

Of course this is a bit convoluted; we're in R, calling C++ to then run some R code. But there are cases where being able to do this is useful.

Note that doing this has overhead; we'll see an alternative in the Rcpp sugar section.

2.8.7 Stand-alone Rcpp

So far we've seen Rcpp via *inline*. Here's the basic structure of a stand-alone Rcpp C++ file that has two arguments:

```
#include <Rcpp.h>
RcppExport SEXP myfunction(SEXP input1, SEXP input2) {
...
}
```

The RcppExport business is an alias to `extern "C"` that handles the fact that we are calling C++ code via .Call.

We can compile the .so as

```
R CMD SHLIB myfile.cpp
```

And then in R:

```
dyn.load('myfile.so')
```

2.8.8 Rcpp sugar

Rcpp provides a lot of C++ functions/operators that allow you to write code in C++ that looks more like R code. This is called *syntactic sugar*.

Some of the functionality includes:

1. Using vectorized calculations (+, -, *, /, <, >, ==, !=, !, abs, exp, log, pow, sqrt among others) on Rcpp vectors (NumericVector, IntegerVector, LogicalVector).

2. Functions such as *any()*, *all()*, *seq_along()*, *ifelse()*, *sum()*, and others.

3. Direct use of *{d,p,q,r}{norm,exp,gamma,unif,...}*. . Here's an example. Note two things: first, unlike the standard functions that R provides through its C API that we've already seen (*rnorm.cpp*), these are vectorized; and second, we are not calling an R function. Here's an example (*RcppSugar.R*):

```
library(inline)

src <- '
return(rnorm(as<int>(n), as<double>(mean), as<double>(sd)));
'
rnormRcpp <- cxxfunction(signature(n="integer", mean="numeric",
   sd="numeric"), src, plugin = "Rcpp")
rnormRcpp(5, 0, 1)

## [1] -1.1477 -0.2895 -0.2992 -0.4115  0.2522

src2 <- '
Function rnorm("rnorm");
return(rnorm(as<int>(n), as<double>(mean), as<double>(sd)));
'

rnormRcppFunc <- cxxfunction(signature(n="integer", mean="numeric",
  sd="numeric"), src2, plugin = "Rcpp")

n <- 1000000
library(rbenchmark)
benchmark(
rnorm(n),
rnormRcpp(n, 0, 1),
rnormRcppFunc(n, 0, 1),
replications = 5,
columns = c("test", "elapsed", "replications") )

##                      test elapsed replications
## 1                rnorm(n)   0.390            5
## 3 rnormRcppFunc(n, 0, 1)   0.405            5
## 2     rnormRcpp(n, 0, 1)   0.289            5
```

Running the benchmarking code in *RcppSugar.R*, we see that as with the case of our 'silly' *exp()* example, it looks like we can beat R's internal call to C, which is surprising. Also, we see that calling an R function via Rcpp involves some overhead (actually that's not obvious from the output above when R is run through *knitr* as for this document, but it did seem to be the case when just running directly in R).

2.8.9 Rcpp Modules

The basic mode of using C++ from R with Rcpp is

1. Convert inputs (from SEXP) to Rcpp or C++ types via *as*
2. Do calculations in C++
3. Convert the result to a SEXP via *wrap* and return it

That's a lot easier than using .Call, but still a bit of a hassle. *Rcpp Modules* allow you to do this in a single step. Here's an example from *RcppModuleExample.R*:

```
library(inline)
library(Rcpp)
fun <- cxxfunction(, "", includes = '
  double norm(NumericVector x, NumericVector y) {
    double sum = 0.0;
    for(int i = 0; i < x.size(); i++) {
      sum += x[i]*x[i] + y[i]*y[i];
    }
    return(sum);
  }
  RCPP_MODULE(mymodule){
    function("norm", &norm);
  }
', plugin = "Rcpp")

mymodule <- Module("mymodule", getDynLib(fun))
norm <- mymodule$norm
norm

## internal C++ function <0x4072b90>
##      signature : double norm(Rcpp::NumericVector, Rcpp::NumericVector)

n <- 100
set.seed(0); u <- rnorm(n); v <- rnorm(n)
norm(u, v)
```

```
## [1] 169.7

sum(u^2+v^2)

## [1] 169.7
```

The *sourceCpp()* function in the Rcpp package allows you to do this easily with an external C++ code file, simply as `sourceCpp(file.cpp)`. In the case below it would automatically produce a function *convolveCpp()* that you can call in R. Here's an example of the C++ file:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector convolveCpp(NumericVector a, NumericVector b) {
  int na = a.size(), nb = b.size();
  int nab = na + nb - 1;
  NumericVector xab(nab);
  for (int i = 0; i < na; i++)
    for (int j = 0; j < nb; j++)
      xab[i + j] += a[i] * b[j];
  return xab;
}
```

The `// [[Rcpp::export]]` is part of the Rcpp *attributes* functionality. Then in R, all we do is

```
sourceCpp("convolve.cpp")
convolveCpp(x, y)
```

You can use module functionality within an R package by writing the same C++ code and then loading the module when the package loads, so the R interface to the C++ function is automatically available (details omitted).

Another use of Rcpp Modules is to easily access C++ classes. You can write your C++ object with fields and methods and then expose the fields and methods to R. So you can basically create a C++ object from R and manipulate it from R. This could serve as an alternative to using ReferenceClasses. See Dirk Eddelbuettel's Rcpp tutorial material (e.g., the useR! 2012 workshop).

2.8.10 Linear algebra: RcppArmadillo and RcppEigen

Let's try an example with *Armadillo* via the *RcppArmadillo* interface (*RcppArmadilloExample.R*). Note the use of object-oriented stuff in the C++ code, operator overloading, and the *arma* namespace.

```
library(inline)
library(RcppArmadillo)
src <- '
arma::mat m1 = as<arma::mat>(mx);
arma::mat matMult = m1 * m1;
arma::mat crProd = m1.t() * m1;
arma::mat matScalMult = m1 * 2.0;
return(List::create(matMult, crProd, matScalMult));
'

fun <- cxxfunction(signature(mx = "numeric"), src,
  plugin="RcppArmadillo")
mat <- matrix(as.double(1:9), 3)
identical(fun(mat), list(mat %*% mat, crossprod(mat), mat * 2))

## [1] TRUE

src <- '
arma::mat m1 = as<arma::mat>(mx);
return(wrap(m1.t() * m1));
'

crossprodA = cxxfunction(signature(mx = "numeric"), src,
  plugin="RcppArmadillo")
n = 3000
x = matrix(rnorm(n^2), n)
library(rbenchmark)
benchmark(
out <- crossprod(x),
out <- crossprodA(x),
out <- t(x) %*% x,
replications = 5,
columns = c("test", "elapsed", "replications") )

##                      test elapsed replications
## 2 out <- crossprodA(x)   4.095           5
## 1  out <- crossprod(x)   3.901           5
## 3    out <- t(x) %*% x   7.998           5
```

Note that this test is (in some sense) unfair to Armadillo as Armadillo doesn't have a matrix crossproduct function, while R's *crossprod()* exploits simplifications relative to `t(X)%*%X`.

Now let's see a simple *Eigen* example via the *RcppEigen* interface (*RcppEigenExample.R*).

```
library(inline)
library(RcppEigen)
incl <- c("using namespace Eigen;")
src <- '
const Map<MatrixXd> mat(as<Map<MatrixXd> >(x));
LLT<MatrixXd> cholFactor(mat);
return(wrap(MatrixXd(cholFactor.matrixU())));
'

cholEigen = cxxfunction(signature(x = "matrix"), src,
plugin="RcppEigen", includes = incl, libargs="-fopenmp")
n = 4000
x = crossprod(matrix(rnorm(n^2), n))
library(rbenchmark)
benchmark(
out <- chol(x),
out <- cholEigen(x),
replications = 5,
columns = c("test", "elapsed", "replications")
)

##                  test elapsed replications
## 2 out <- cholEigen(x)   9.185            5
## 1      out <- chol(x)   3.560            5
```

Note that the "R" version (i.e., R calling Lapack's Cholesky function via R's C interface to Lapack) is faster because it is using a threaded BLAS, which Eigen does not do. With a single thread, I got the following:

```
# 2 out <- cholEigen(x) 15.953 5
# 1 out <- chol(x) 17.496 5
```

2.9 Debugging C++ code called from R

Suppose we want to debug C++ code that is called from R. We need to know how to use a C/C++ debugger from within R. Here's how one would do it with the *rnorm.cpp* code called from *rnorm.R*.

```
R -d gdb --vanilla
(gdb) b rnorm.cpp:7
(gdb) b genRandN
```

Note that it will say the symbol table is not loaded; answer 'y' to have the breakpoint come into effect when R loads your .so file

```
(gdb) r
> source('rnorm.R') # this runs the R code; the debugger will break
and put you in gdb at the breakpoint, at which point you can use gdb
as discussed previously when talking just about C++
```

As you're working through things, when you're in R, you can do `Ctrl-C` to return to gdb. When you're in gdb, you can type `signal 0` and `<enter>` to return to R (you may need to hit a second return to see the R prompt)

Comments on compilation You may also want to remove optimization from when your .so is created (such a the -O3 flag) as the compiler may do strange things to your code such that variables you expect do not exist, so you may want to use g++ manually rather than R CMD SHLIB. If you just use the debugger on the example above, and do "`p it`", it will say "optimized out", which has something to do with the compiler optimizing the execution of the loop. You would also see that the lines of code seem to be executed in an odd order.

Note that if you do compile manually make sure to leave the -g flag so that debugging symbols needed by gdb are left in the compiled code.

3 R packages

In general, the R extensions manual on CRAN is your guidance for all things related to R packages.

To develop R packages on a Mac, you need *XCode*, and on Windows you need *Rtools* (http://cran.r-project.org/bin/windows/Rtools/).

There are three versions of R packages.

- *package bundle/package source*: compressed version of the package that contain the raw R code, any raw C/C++/Fortran code, help files, etc. This is produced by *R CMD build*. Provided as "Package source" on the CRAN page for a package.

- *installed package*: the form of the package in the directory on your machine after it has been installed. The contents are somewhat different than package bundles, with the R code stored in an efficient manner, compiled C/C++/Fortran code, and the help files stored differently as well.

- *binary package*: a single file that can be installed on a machine by simply unpacking/unzipping and is operating system specific. This is for Mac OS X and Windows and binary packages are on the CRAN page for a package.

3.1 Exploring (and modifying) R packages

When you install an R *package*, you end up with a directory for each *package* in a *library*, which is a directory structure in your filesystem. On the SCF system, the packages for Linux that are installed systemwide are at */usr/local/linux/lib/R/Current/x86_64/site-library* and */usr/local/mac/lib/10.8/R/Current/x86_64/site/library* (this path will be relocated to */usr/local/mac/lib/R/Current/x86_64/site/library* soon). If you install a package locally in your home directory, it will be at something like *~/R/x86_64-pc-linux-gnu-library/R-version-num*. The core R packages (those that are installed with R) on the SCF Linux machines are local to each machine and are at */usr/lib/R/library*.

You can look in this directory for some info on the package but all the raw R code will be packaged up into a database file (.rdb/.rdx) and any C/Fortran code will already be compiled (in the *libs* directory). To see (and possibly modify) the raw R and external code, download the .tar.gz file (the *package bundle*) from CRAN. Here's how we'd do it with the *Matrix* package.

```
wget http://www.cran.r-project.org/src/contrib/Matrix_1.0-12.tar.gz
tar -xzvf Matrix_1.0-12.tar.gz
```

Then in *Matrix/R*, you'll see the R code files and in *Matrix/src*, you'll see the raw C/Fortran code.

If you want, you can modify the code and rebuild the package.

Suppose you are in */tmp* and have untarred to */tmp/pkg* (e.g., with *pkg=Matrix* and *version=1.0-12*). Then from */tmp*, do:

```
R CMD build pkg
R CMD INSTALL pkg_version.tar.gz -l /tmp
```

and in R load the (new) local version of the package:

```
library(pkg, lib.loc = '/tmp')
```

Note you don't have to use the `-l /tmp` when installing – if you don't it will install it in your default library. But during testing you may want to do this in a place like */tmp*.

As an example, Wayne and I found a memory leak in an isotonic regression package he was using. I inserted the necessary commands in the C code in */src* to delete memory allocated but never freed, followed the steps above, and things worked fine. I then emailed the package maintainer to suggest to that person that they fix their code; reporting bugs such as this is a good way to help the general community.

In general you can install an R package in a directory of your choosing, either using *R CMD INSTALL* with the -l flag as above or directly in R as `install.packages('pkg', lib = '/path/to/installation')`. You can load a package from a specific location with the *lib.loc* argument to *library()*, as shown above.

3.2 R package structure

The next few sections walk through the basic pieces of an R package, referring to a few example packages on occasion, such as *Matrix*, *plyr*, *lme4*, and *bigGP*. *bigGP* is a package I've just created recently, so the details are fairly fresh in my mind and (hopefully) correspond to the current R standards for packages.

The following describes the components of a package **before** it is installed. In the installed version, many of the pieces will be in a form not readily readable/accessible. E.g., the R code will be bundled up in a database format and the C/C++/Fortran code will be compiled into the .so file (in the *libs* directory).

The basic components are:

- *DESCRIPTION*: a file with metadata about the package
- *NAMESPACE*: a file indicating what objects should be publicly available, and which also specifies the shared object library (if the package contains compiled code) to load
- *R*: one or more files of R code that make up the package
- *man*: help files for the R functions, objects, classes and datasets in the package

The additional optional stuff includes:

- *data*: data objects included in the package
- *src*: raw C/C++/Fortran code

- *TODO*: self-reminders to the maintainer/developer of stuff that still needs to be done
- *NEWS*: updates on changes to the package
- *CITATION* or *inst/CITATION*: how to cite your package
- *README* or *INSTALL*: info to help users install the package (generally when there is C/C++/Fortran code and issues with linking) or alternatively (as suggested by Hadley Wickham for *README*), an overview of your package
- *tests* or *inst/tests*: R code to test the package is operating correctly. These will get run during package checking (if in *tests*) or with *test()* in *devtools* if in *inst/tests*.
- *demo*: R code that demonstrates the use of the package, potentially code for examples in published papers about the package
  You can run a demo as: `demo(topic, package)`; e.g., with bigGP: `demo('article-example', 'bigGP')`.
- *vignettes*: tutorial material on using the package:
  You can view a vignette as: `vignette(topic, package)`.

The R function *stopifnot()* can be useful for inserting checks in code in tests and the examples section of the man files.

3.3 Creating R packages

Let's go through the basics of creating a package. In the workshop I'll do this for a basic example using the *devtools* package (see below for details). Here I describe how to do things in a basic way without using *devtools*.

Note that looking at the package bundle for other packages is often very helpful. But remember that lots of packages are created by people who don't have much experience, so if you're looking for a good template package, it's a good idea to figure out which package creators are experts - one strategy is to look at packages created or maintained by an R core member. That's why we'll look at *Matrix* and *lme4*.

3.3.1 Creating an initial package

The *package.skeleton()* function in the *utils* package will create an initial package using R code that you specify. Here's a basic usage. Suppose we have an R file that contains the R code we want in the package:

```
package.skeleton('mypkg', code_files = 'method.R')
```

The most basic package One good way to share code with someone, and even with yourself as part of a project, is just to make a bare-bones package with R code in /R, a DESCRIPTION file, and a basic NAMESPACE file. The easiest thing to do with the NAMESPACE is just export everything:

```
exportPattern("^[^\\.]")
```

Then all you have to do is the steps above to build and install.

The basic build/install steps Once you have the guts of the package, you can build and install it. You'll probably do this a lot while you develop the package. The basic steps are as follows under Linux. Suppose the directory that contains your package is */accounts/grads/user/Rpkgs/mypkg*:

```
cd ~/Rpkgs
R CMD build mypkg
R CMD INSTALL mypkg_version.tar.gz
```

If you are installing locally on the SCF system you may need to do something like this:

```
R CMD INSTALL mypkg_version.tar.gz -l ~/R/x86_64/2.15
```

Or for a temporary install, you might use `-l /tmp`. In these cases you load the package in the following manner (here with */tmp*): `library(mypkg, lib.loc = '/tmp')`.

3.3.2 R, src, and data directories

Put your R file or files in */R* (this can occur after you use *package.skeleton()*). It's best to use the .R extension for these. Put your C++ code in a file or files in */src*. In both cases, it's a nice idea to break up your code into separate files organized in a logical way.

You can create *.RData/.rda* files that you put in */data*. These will become part of your package as well. Remember to include help information for them. If you have data in your package it's a good idea to have the line

```
LazyData: yes
```

in your DESCRIPTION file. With this setting, R loads the data into memory only when the data object is used. The alternative involves the user using *data()* to load the data in 'manually'.

3.3.3 Namespaces and the DESCRIPTION and NAMESPACE files

The DESCRIPTION file contains a number of required fields, described in the R extensions manual. I give some details below on some of these.

Background There is a distinction between *loading* a package and *attaching* it. A package can be loaded but not be attached. In this case you need to use :: to access the objects as they are not on the search path. By analogy to C++ namespaces, attaching is like inserting *using namespace std;* in your code, while simply loading requires you to explicitly specify the namespace of the package. Here's an example with the *fields* package:

```
tim.colors(32)

## Error: could not find function "tim.colors"

fields::tim.colors(32)  # this causes fields to be loaded but not attached,

##  [1] "#00008F" "#0000B0" "#0000D0" "#0000F1" "#0012FF"
##  [6] "#0033FF" "#0053FF" "#0074FF" "#0093FF" "#00B4FF"
## [11] "#00D4FF" "#00F6FF" "#17FFE8" "#37FFC8" "#57FFA8"
## [16] "#78FF88" "#97FF68" "#B8FF47" "#D8FF27" "#FAFF05"
## [21] "#FFE400" "#FFC400" "#FFA400" "#FF8400" "#FF6400"
## [26] "#FF4300" "#FF2300" "#FF0100" "#E00000" "#C00000"
## [31] "#A00000" "#800000"

search()

##  [1] ".GlobalEnv"            "package:Rcpp"
##  [3] "package:lattice"       "package:Matrix"
##  [5] "package:RcppEigen"     "package:inline"
##  [7] "package:methods"       "package:RcppArmadillo"
##  [9] "package:rbenchmark"    "package:knitr"
## [11] "package:stats"         "package:graphics"
## [13] "package:grDevices"     "package:utils"
## [15] "package:datasets"      "package:SCF"
## [17] "Autoloads"             "package:base"

tim.colors(32)

## Error: could not find function "tim.colors"

fields::rdist  # note this is accessed quickly, indicating fields is alread

## function (x1, x2)
## {
##     if (!is.matrix(x1))
##         x1 <- as.matrix(x1)
##     if (missing(x2))
##         x2 <- x1
##     if (!is.matrix(x2))
##         x2 <- as.matrix(x2)
##     d <- ncol(x1)
##     n1 <- nrow(x1)
##     n2 <- nrow(x2)
##     par <- c(1/2, 0)
##     temp <- .Fortran("radbas", nd = as.integer(d), x1 = as.double(x1),
##         n1 = as.integer(n1), x2 = as.double(x2), n2 = as.integer(n2),
##         par = as.double(par), k = as.double(rep(0, n1 * n2)))$k
##     return(matrix(temp, ncol = n2, nrow = n1))
## }
## <environment: namespace:fields>

# fields::image.plot(1:2, 1:2, matrix(1:4, 2)) # this works
# but don't plot in the tutorial...
```

Note that if the entire package is not attached, there can be issues with finding relevant objects in function calls, as you can see by running the following code (I don't embed the output as I'm having some trouble with knitr using this code when I try to create the tutorial pdf.

```
detach(package:stats)
lm(dist ~ speed, data = cars)  # not found
stats::lm(dist ~ speed, data = cars)  # fails to find model.frame()
```

You can access objects that are not exported from a package (i.e., private objects) by the ::: operator, e.g., *stats:::secretMagicLM*.

You can see what is in the search path (packages, attached data frames and lists, etc.) with *search()* and where on the filesystem a package is stored using *searchpaths()*:

```
search()

##  [1] ".GlobalEnv"            "package:codetools"
##  [3] "package:Rcpp"          "package:lattice"
##  [5] "package:Matrix"        "package:RcppEigen"
##  [7] "package:inline"        "package:methods"
##  [9] "package:RcppArmadillo" "package:rbenchmark"
## [11] "package:knitr"         "package:stats"
## [13] "package:graphics"      "package:grDevices"
## [15] "package:utils"         "package:datasets"
## [17] "package:SCF"           "Autoloads"
## [19] "package:base"
```

```
searchpaths()
```

```
##  [1] ".GlobalEnv"
##  [2] "/usr/lib/R/library/codetools"
##  [3] "/server/linux/lib/R/3.0/x86_64/site-library/Rcpp"
##  [4] "/usr/lib/R/library/lattice"
##  [5] "/server/linux/lib/R/3.0/x86_64/site-library/Matrix"
##  [6] "/server/linux/lib/R/3.0/x86_64/site-library/RcppEigen"
##  [7] "/server/linux/lib/R/3.0/x86_64/site-library/inline"
##  [8] "/usr/lib/R/library/methods"
##  [9] "/server/linux/lib/R/3.0/x86_64/site-library/RcppArmadillo"
## [10] "/server/linux/lib/R/3.0/x86_64/site-library/rbenchmark"
## [11] "/server/linux/lib/R/3.0/x86_64/site-library/knitr"
## [12] "/usr/lib/R/library/stats"
## [13] "/usr/lib/R/library/graphics"
## [14] "/usr/lib/R/library/grDevices"
## [15] "/usr/lib/R/library/utils"
## [16] "/usr/lib/R/library/datasets"
## [17] "/server/linux/lib/R/3.0/x86_64/site-library/SCF"
## [18] "Autoloads"
## [19] "/usr/lib/R/library/base"
```

This can be useful if there is a version installed by the administrative user and a version you have installed locally from your home directory.

NAMESPACE The lines of the NAMESPACE file indicate what objects should be accessible to a user of your package. Anything else is objects you don't expect a user to use and only accessible via the ::: operator.

You can export all objects that don't start with a period as

```
exportPattern("^[^\\.]")
```

However, it's recommended that you only export objects that you expect/want users to use, e.g.,

```
export(x, y)
```

For S3 generics and S3 methods, here's what you do:

```
export(myGeneric)
S3method(print, myClass)
S3method(myMethod, myClass)
```

The R extensions manual (and by example the Matrix package) indicate that one should even export generics (suggesting the use of *exportMethods()* for S4) that are not local to your package (such as *print()*), if you use them.

For S4 classes and methods:

```
exportClasses(myClass)
exportMethods(myMethod)
```

For ReferenceClasses:

```
exportClasses(myClass)
```

In NAMESPACE, you would also list any imported functions from other packages or entire packages:

```
importFrom(fields, tim.colors)
```

You can also import the entire package:

```
import(fields)
```

This will cause the package to be loaded but not attached, so it doesn't 'pollute' the user's workspace. As an example, if your package used *tim.colors()* from fields, but that was all it used from fields, you might do `importFrom(fields, tim.colors)`. You can then use *tim.colors()* directly without the :: operator.

If you're using *roxygen*, you can add the *@export* tag in your *roxygen* comments and this will automatically deal with the exporting and the necessary inclusion in the NAMESPACE file. You can also include *@import* tags (either for an entire package, or *@importFrom* for specific functions from a package) and this will set up the NAMESPACE file correctly. You still need to modify the DESCRIPTION file manually in terms of specifying packages that are imported (see next section).

How to rely on other packages: depend, imports, suggests In your DESCRIPTION file, you can choose to rely on other packages in several ways:

- *Depends*: packages that should be loaded and attached for your package to work; in this case you do not need to use *imports* for these in NAMESPACE
- *Imports*: for packages for which variables are *imported* from (given in the NAMESPACE file) but which do not need to be on the search path (i.e., don't need to be *attached*). You

do not need to use :: within your package functions, but the imported functions will not be attached for general use by the user.

- *Suggests*: for non-critical packages, such as those used in examples, tests, or vignettes. For packages that are suggested, their use should be used conditionally via `if(require(pkg))` such that the code does not fail if the package is not available.

You can indicate a dependence specific versions of packages (or a specific version of R itself). So if you need version 2.14.1 of R then you would have `R (>= 2.14.1)` on the *Depends* line of DESCRIPTION. Note that the dependence is checked on loading the package for 'depends' but not for 'imports'.

The R extensions guide says that *Depends* should be used for packages that need to be attached for the package to be loaded, while *Imports* is for packages whose namespace is needed in order to load the package. This is hard to understand (at least for me). Here's my understanding of the situation: If you want to be able to use functions from another package in the functions in your package, you can use *Imports* and then in NAMESPACE import specific functions or the entire package. If you want the user to have access to functions in the other package, e.g., in their own coding, without having to load the package (either explicitly or implicitly via use of ::), then use *Depends*. *Depends* also causes the package to be loaded and attached when your package is loaded. In the *devtools* package documentation, Hadley Wickham strongly recommends using *Imports* rather than *Depends*.

For base packages (e.g., *stats*), an advantage of importing rather than just relying on them being in the search path by default is that R looks for the variables that are imported before looking in the search path. However, any modest speedup from this is probably not particularly important...

Version numbering A convention is to number as X.Y-Z. X is the major version number. You may want to start with X=0. Y indicates minor version numbers that don't mark changes as major as X. Z indicates patch numbers for fixes to problems. So as you create a package, you might start with 0.1-0 and then as you develop it, use 0.2-0, 0.3-0, etc. Then you might release it on CRAN as 0.3-0 or the like and if you fix issues, use 0.3-1, 0.3-2, etc. If you have a version that is fully tested and has been used by others, you might release that as 1.0-0.

Any modifications to a package that you have released on CRAN or to anyone else should result in a new version number.

More info is available at http://semver.org/.

3.3.4 Hooks

You can have code that executes when a package is loaded (or when it is attached, but this is less likely). This code could provide the user with necessary information or do some initialization steps. Some info can be found via `help('.onLoad')`.

Here's an example with the bigGP package I'm writing. My goal is to warn the user that an initialization function must be called before anything else is done:

```
   \begin{verbatim}
   .onLoad <- function(libname, pkgname){
   packageStartupMessage(" =================================================

   Loading bigGP.\n
   Warning:  before using bigGP, you must initialize the slave processes
using bigGP.init().\n
   If R was started through mpirun/orterun/mpiexec, please quit by
using bigGP.quit().  ======================================================
   \n ")
   }
   \end{verbatim}
```

Using *packageStartupMessage()* allows for users to suppress such messages when loading packages and is the recommended approach, as opposed to *cat()* or *print()*.

3.3.5 Help pages

Help pages are given as .Rd files in the */man* directory and are written in a Latex-like markup language. One can create them by hand. An alternative is to include the relevant info in structured comments accompanying your function and use the *roxygen2* package (in particular using *roxygen2* via *devtools*), as described later in this document.

For help pages for functions, the standard sections are:

- *name*: name of the function being documented
- *alias*: this allows you to specify related functions that will also use this help page, so you don't need help pages for every single function
- *docType*: not included for all help files, but specified as 'data' for help on data objects, 'package' for the package help file, 'class' for classes, and 'methods' for help files on class methods.

- *title*: one line title for the function
- *description*: one paragraph description
- *usage*: should contain the exact call signature, including argument names and default values
- *arguments*: list of arguments, including information about the allowable classes or types of the arguments, using \item.
- *value*: full description of what the function returns, including all the components of any list that is returned (in which case, use \item)
- *details*: more info
- *section*: for specifying sections that you choose the name of, such as slots and methods for S4 class help files (i.e., you'd have `\section{slots}{...}`
- *author*: optional, since this info is generally part of DESCRIPTION, but useful if the code for a function was not written by the package author.
- *references*: journal articles, etc., that explain the use of or methodology behind the function
- *seealso*: other related functions that a user might find helpful (using `\code{\link{...}}`)
- *examples*: these are important. Include examples that illustrate how the function is used. These must run correctly or the package will not install, so they can be used as tests of the package as well. You can avoid having some of the code run by wrapping the code in `\dontrun{code}` – for example code that would take a long time to run so you wouldn't want it run when R installs the package and runs the examples.
- *source* and *format*: the original source and the format of the data set, for help files for datasets.

Most of these are self-explanatory by looking at some examples in R.

The Latex-like markup language for the help pages has a number of keywords for various types of special text:

- `\code{text}` and `\pkg{package_name}`: for referring to R code, including function names, and R packages, respectively
- `\link{text}` and `\linkS4class{className}`: in the pdf versions of help pages, make hyperlinks to other topics – this is particularly useful in the seealso section, e.g., `\code{\link{relatedFunctionName}}`.

- `\item{name}{description}`: for lists of items, in particular the arguments of functions
- `\eqn{text}`: for inserting mathematical syntax using Latex math mode. This will be translated into textual form for the plain text help and compiled for the pdf versions of the help pages. You can use `\deqn{text}` for display equations.
- `\emph{text}` and `\bold{text}`: for making text italicized and bold, respectively
- `\sQuote{text}` and `\dQuote{text}`: for putting text in explicit single or double quotes, but without hardwiring the type of quoting symbol used.
- `\S3method{methodName}{className}` and `\S4method{methodName}{className}`: this is used to refer to the method, e.g. in the *usage* section

It's a good idea to have an overall help page for the package, e.g., *mypkg.Rd*. You might need a different name if you have function with the same name as the package. This help page could have an example of overall usage of the package.

It's also a good idea to have help pages for any datasets you include in the package.

For S3, S4, and ReferenceClass classes, it's common to have a help page for the class, where the name of the Rd file is *yourClassName-class.Rd*. You would probably then include *\alias{yourClassName}*. For S4 classes, the help file will have a *Slots* section and a *Methods* section, where each slot or method is listed using *\item*. The *Matrix* package has examples. For ReferenceClasses, this will have a *Fields* and a *Methods* section. One generally also includes a line like the following for methods from the class
`\alias{myMethod, myClass-method}`
so that R will allow the user to select help on your method (as opposed to other function with that same name) when doing `help(myMethod)`.

Note that for ReferenceClasses, you'll have methods bound up in the class. To provide help info on these, the best thing is to include a doc-string (similar to Python) in the method definition itself. E.g.:

```
methods = list(
  doCalc = function(x = 0) {
    'Description: this method does blah.
     Arguments:
       x: numeric vector. The input value.
     Value:
       a numeric vector. The blah of the input.
     Details:
       more info
     '
  # R code for doCalc() here
})
```

This can then be accessed with `className$help(methodName)`.

3.3.6 Tests, demos, examples, and vignettes

You can put R code that serves as demo code in the form of one or more R files in the */demo* folder. These can then be run by the user as `demo('topic')`, where topic is the file name, leaving off the .R. It's best to use the .R extension for test and demo R code files. The demo directory must have a *00Index* file as discussed in the R extensions manual.

It's a good idea to put examples at the end of the help file for each of the non-trivial functions or objects that you intend users to use. These examples get run during *R CMD check* to make sure they run without error. Users can also run them via `example('topic')`, where topic is the name of the object of interest.

You can also set up tests that make sure the code runs correctly in */tests*. *Matrix* and *spam* both provide examples of this.

Finally you can include vignettes, which are expository documents intended to demonstrate the use and usefulness of a package. You'll likely want to write the vignette via *Sweave* (I'm not sure if *knitr* will work) to enable production of a pdf document that embeds example R code and output within the text. It appears that the vignette directory should contain the Rnw source code (and any needed Latex style files or Bibtex files) that is then compiled to pdf during package installation.

3.3.7 Details for compiled code

For basic code that you write, you'll generally just be able to include the files in */src* and build and install the package without any modifications.

You'll need to load the necessary .so files with *useDynLib()* in the NAMESPACE file:

```
useDynLib(mypkg, .registration = TRUE)
```

where *mypkg.so* would be the name of the shared object file if the package name is *mypkg*.

If your code needs to link to other code, then you may need to do more to make sure the package installs properly. The next level of complication involves using a file called *Makevars* placed in */src*. *Makevars* is a make file. Some of the common things to do in *Makevars* are to specify PKG_CPPFLAGS for additional include/header paths and PKG_LIBS for additional linking options. An example is if you have code that is written for openMP, your *src/Makevars* file would have

```
PKG_CPPFLAGS = $(SHLIB_OPENMP_CXXFLAGS)
PKG_LIBS = $(SHLIB_OPENMP_CXXFLAGS)
```

Yet more complicated is to include a *configure* script that sets things up to allow the package to link to other libraries on the system. There is a tool called *Autoconf* that generates *configure* scripts automatically that can be useful in such cases.

This all gets complicated quickly and is beyond the scope of what we'll go into here. One path forward is to try to find a package that uses compiled code and add-on libraries in similar fashion to what you are trying to do and see how they set up the configuration stuff.

Using Rcpp in a package Given the above, we're lucky to have the *Rcpp.package.skeleton()* function provided by Rcpp that allows us to easily create a package that uses C++ via Rcpp. *Rcpp.package.skeleton()* deals with the details needed for linking, in particular setting up *Makevars* and *Makevars.win* (for Windows version of the package) files, as well as giving an example of the .h and .cpp files in */src*. You can see an example of the skeleton package that gets created in the Rcpp folder installed on your machine, */path/to/Rpackages/on/your/system/Rcpp/skeleton*. Note that that skeleton leaves out the line

```
PKG_CPPFLAGS=`Rscript -e 'Rcpp:::CxxFlags()'`
```

needed in *Makevars* (and the analogous line in *Makevars.win*).

For more details see this Rcpp package vignette:
http://www.cran.r-project.org/web/packages/Rcpp/vignettes/Rcpp-package.pdf.

3.3.8 Submitting to CRAN

You need to make sure your package passes all the checks. Often it will fail with messages about your DESCRIPTION or about some syntax you have used, or about the structure of your help pages. Before you submit, you should make sure that R CMD check returns no errors as well as trying to avoid any warnings.

```
R CMD check mypkg_version.tgz
```

Then follow the instructions at http://cran.r-project.org/banner.shtml#submitting. Upon submission, CRAN will take care of building Linux, Mac and Windows versions, though if you have compiled code, dealing with Windows is likely to be tricky.

3.4 Using *devtools* to create a package

The *devtools* package (https://github.com/hadley/devtools/wiki/Package-basics) provides a lot of tools for creating and working with packages from within R, avoiding the operating system command line. In the workshop, we'll work through an example, the *cjpUtils* package, using devtools, touching on various topics from the previous section as we do so.

Here are some of the basic things you can do with *devtools* functions:

- *create()*: create an initial package (analogous to *package.skeleton()*)
- *build()*: analogous to *R CMD build* for creating the package bundle. You can create a binary package with `build(binary = TRUE).`
- *check()*: analogous to *R CMD check*
- *install()* and *install_github()*: you can install a package from a file on your machine via devtools with *install()*. And you can install a package from a github repository using *install_github()*.
- *release()*: releases a package to CRAN!

As you develop your package, you'll be iterating between changing your code and trying it out. The *load_all()* function will load from a source package rather than having to build and install. If you want to try out the help files or examples from a source package rather than an installed package, you can use *dev_help()* and *dev_example()*. Here's how your development cycle might look:

```
create('mypkg')
load_all('mypkg')
document('mypkg')
test('mypkg')
dev_help('myfun.Rd')
dev_help('mypkg.Rd')
dev_example('myfun.Rd')
dev_example('mypkg.Rd')
```

This would generally involve a bunch of iteration amongst all the steps (except for *create()*).

Using devtools to create the *cjpUtils* example package

In the demo code accompanying this document, you will find the file *makePkg.R* that provides an example of the development cycle for a package. If you use the commands in that file, and with the files in *pkgExample/files*, you should be able to recreate the *cjpUtils* package. What you produce should look like what is in the *pkgExample/cjpUtilsExample* folder in the demo code. I.e., I created *cjpUtilsExample* by following the instructions in *makePkg.R* and then renaming the resulting *cjpUtils* folder to be *cjpUtilsExample*.

3.5 Other tools for working with packages

roxygen2 This provides a way to generate the .Rd help files without ever having to touch them directly. You put the necessary text as comments in your R code and then 'roxygenize' the code and the .Rd files are created. Some advantages are that the documentation is right there with the code, that *roxygen2* can fill in a bunch of stuff in the .Rd automatically, and it can deal with issues in document S3 and S4 stuff. In the *devtools* approach above, it was using *roxygen* behind the scenes of document().

You can see examples of how the comments are structured to work with *roxygen* at https://github.com/hadley/de
functions.

Here's an example of making an R package from a single file by employing the *roxyPackage* package: http://lamages.blogspot.com/2013/03/create-r-package-from-single-r-file.html#more.

Rstudio Rstudio provides tools for package development. See http://www.rstudio.com/ide/docs/. One can create a package from within Rstudio, as well as rebuild/reinstall/reload it in one step: http://www.rstudio.com/ide/docs/packages/overview. This allows you to easily modify your package and then try it out. Rstudio also includes the ability to generate the outline of a .Rd file and to preview the appearance of a help page from a .Rd file.