Why is R Slow?
Plus: Rcpp and Cython
Statistics 650/750
Week 14 Thursday

Alex Reinhart and Christopher Genovese

30 Nov 2017

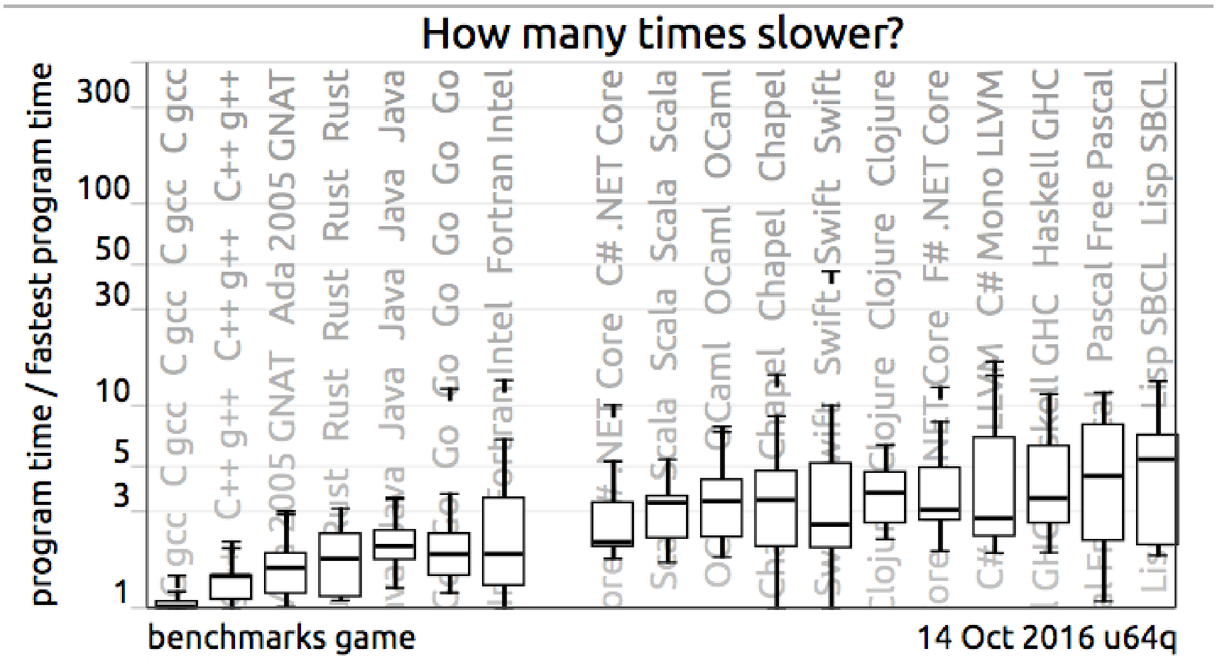
Announcements

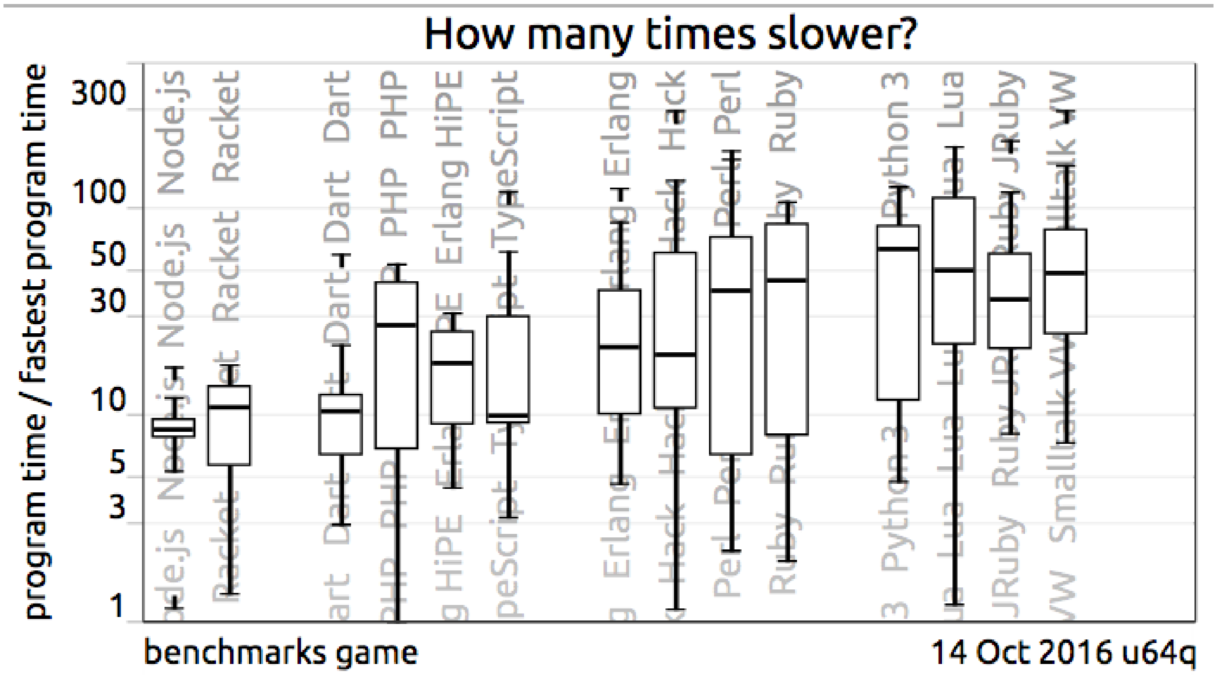
- Remember that your final version of Challenge 2 must be submitted by **December 14**
- We'll follow up with neural nets next Tuesday
- For those working on word-clouds: the `faster-bbox.js` script now accepts optional sizes on each line. So each line can be either `word` or `word @ size`, where the space before `@` is required and the size is a number with units like `24px`. The default `font-size` will be used if size is omitted.

Why so slow?

Standard R and Python advice – or any other dynamic language, like Ruby or PHP – is to write performance-critical code in C or C++. Use built-in vectorized functions, write hot loops in Rcpp or Cython, and rely on external libraries as much as possible.

But why should R and Python be so much slower than C?





Let's take a meandering tour through CPU architecture, programming language design, interpreters, and compilers, so we can see where these performance differences come from.

A bit of CPU architecture

CPUs execute *instructions*. Each instruction is very low-level: add these numbers, move this to memory, jump to these other instructions in memory, load a number from memory, etc.

Everything that runs on your computer is eventually turned into these instructions. They can be written in textual form as *assembly language*:

```
pushq   %rbp
movq    %rsp, %rbp
addq    %rsi, %rdi
movq    %rdi, %rax
popq    %rbp
ret
```

This is the code for a function adding two integers. We push the function pointer onto the stack, shuffle some values around, add two values, pop off the stack, and return.

The names (`%rbp`, `%rax`, etc.) refer to *registers*: cells of fast memory inside the processor. Registers are blazing fast, but each can only hold small values (usually 64 bits), and there are a limited number. Every instruction works on values in registers, so other instructions have to load registers from main memory, store their values back to memory, etc.

Compilers work hard to allocate registers, since you usually have way more variables than registers. Running out of registers and having to move stuff in and out of RAM ("spilling") is inefficient.

Cache

Besides registers, processors have caches. These are in a hierarchy of L1, L2, L3, etc. caches, in decreasing order of speed. Caches are usually a few megabytes of extremely fast RAM situated directly on the processor, so they can be accessed at high speed.

The processor automatically manages the cache. You can't directly manipulate it with your code. The cache contains copies of frequently used chunks of memory, automatically discarding the least recently used chunks to make space for new ones. When your program uses a certain memory location, an entire chunk of memory containing that location is copied into the cache.

If you're lucky, your data fits in the cache and operations will be extremely fast. If you're unlucky, or your program accesses a great deal of data spread widely over memory, the processor will have to wait ("stall") to retrieve data from RAM.

This is why it can be faster to iterate over a matrix in the right order. In R, matrices are stored in column-major order: the matrix

```
1 2 3
4 5 6
```

is actually stored in memory as `1 4 2 5 3 6`. If we iterate over the whole matrix, one column at a time, contiguous chunks of the matrix can be loaded into the cache. But if we iterate over rows, we keep skipping from one memory location to one far away, and new chunks have to be brought in from main memory, making the loop much slower.

(Numpy for Python stores arrays in row-major order by default, so the opposite is true there.)

Interpreters, ASTs, JITs, VMs, and more

Interpreted languages like R and Python are not translated into machine code – there is no compiler that turns R into assembly code. Instead, they run with the help of an *interpreter*.

But why not compile?

Operations in a high-level language don't directly correspond to machine instructions. Consider:

```r
add <- function(x, y) { x + y }
```

An innocuous function. But:

- `x` and `y` might be numbers, which can be added by the processors.
- `x` and `y` might be vectors, which have to be added elementwise. One might be shorter than the other, which has to be checked and handled. We'll have to allocate a vector to store the result.
- `x` and `y` may be S4 classes with a special + method defined for them (like the hyperreal numbers I showed in class). We may need to allocate memory for the results, tracking this memory with the garbage collector.
- `x` and `y` may be objects for which addition is not defined.

None of this is known until the program runs. When R sees "+", it has to check which of these is true, and potentially do some very complex processing (like for S4 classes). Running + means loading `x` and `y`, checking their types, determining which operation is appropriate, and then invoking the relevant code.

We could turn this into machine code – very long, very tedious machine code – but there's no point. Instead, we write a program which reads the code and executes it. The program is, in effect, pretending to be a computer processor that understands R.

Simple interpreters

Interpreting starts by turning the source code into a *parse tree* or *abstract syntax tree* (AST), data structures representing the meaning of the code. Here's the AST for our `add` function, as printed by the `pryr` package:

```
> ast(function(x, y) { x + y } )
\- ()
  \- `function
  \- []
    \ x =`MISSING
    \ y =`MISSING
  \- ()
    \- `{
    \- ()
      \- `+
      \- `x
      \- `y
  \- <srcref>
```

This is just a textual representation. The built-in `quote` function returns this representation as an R list: you can process the list to retrieve the function calls, arguments, and so on:

```
> foo <- quote(function(x, y) { x + y } )
> foo[[1]]
`function`
> foo[[2]]
$x


$y


> foo[[3]][[1]]
`{`
```

The simplest possible interpreters simply read in the AST and operate on it. These are known as *AST walkers.*

AST walking is dead simple: read in the code piece by piece and do what it says. If it references a variable, look up the variable in a table and find its value; if it has a mathematical expression, fill out the values and calculate it. You could write R code that interprets R code by taking the output of `quote` and reading through it, element by element.

AST walking is also usually slow. Everything is referred to by name (variables, functions, objects, etc.), so everything has to be looked up in a set of tables (to determine what's in scope) every time it's accessed. There's a lot of overhead. The processor's cache is filled with AST data, variable scope tables, garbage collector data, and other stuff that's not your code or your data.

Aside: Functions that transform code

Hang on – if you can turn R code into an AST, and then read and even modify that AST, can you write functions that take *code* and return *new code*?

Yes.

This is a bit painful in R, since we have to work with deeply nested lists, but it's entirely possible. Imagine a function like this:

```
## Recurse deeply into an AST object, applying the provided function
## to elements that are numerics
replace_numeric <- function(ast, fn) {
    if (is.name(ast) || is.pairlist(ast) || inherits(ast, "srcref")) {
        return(ast)
    } else if (is.call(ast)) {
        replaced <- sapply(as.list(ast),
                           function(el) { replace_numeric(el, fn) })
        return(as.call(replaced))
    } else if (is.numeric(ast)) {
        return(fn(ast))
    } else {
        return(ast)
    }
}

randomize_constants <- function(const) {
    const + rnorm(1)
}

foo <- quote(function(x) { x + 4 })

bar <- replace_numeric(foo, randomize_constants)

bar
## function(x) {
##     x + 3.64477015719487
##}
```

Now, foo and bar are both AST objects, not functions, but we can evaluate these trees and turn them back into functions with eval:

```
foo_fn <- eval(foo)
bar_fn <- eval(bar)

foo_fn(4)  #=> 8
bar_fn(4)  #=> 7.64477
```

Why might it be useful to rewrite code like this? In R, it's not usually a good idea. Changing how the language works can be confusing. It's tough to write a good code-mangling function – you have to handle the AST properly.

But in other languages, functions that modify code are common – even part of the core language. Consider Lisp and its derivatives (Scheme, Clojure, Racket, etc.). You've seen some examples where code is written in a weird notation with lots of parentheses:

```
(/ (+ (- b) (sqrt (- (expt b 2) (* 4 a c))))
   (* 2 a))
```

But this notation reveals an elegant advantage. The notation for a list – a linked list of elements – is just

```
'(1 2 3 4 5 6)
```

The ' at the front is the `quote` operator – sound familiar? `quote` tells Lisp that this is a bare list. If there is no quote, as in

```
(* 2 a)
```

Lisp takes the list, assumes the first element is a function, and applies it to with the remaining elements as arguments. So we can write

```
'(/ (+ (- b) (sqrt (- (expt b 2) (* 4 a c))))
    (* 2 a))
```

with the quote, and this returns a *list* representing the code. Just like code can operate on lists, it can operate on code, returning new lists that are also code.

Users of Scheme and Lisp-like languages often write *macros*, which take their arguments as lists of code and return new code, to do useful things, letting them essentially build their own programming language. When could this be useful? Imagine doing some operation on every row of results from an SQL query:

```
(doquery (:select 'x 'y :from 'some-imaginary-table) (x y)
  (format t "On this row, x = ~A and y = ~A.~%" x y))
```

Here `doquery` is a macro which takes a query, names the resulting columns, and executes a piece of code once for every row, using the values from each column. When the code is *read* – not when it runs – the `doquery` macro runs and transforms this code into the full code needed to convert this to an SQL query, send it to the database, and do the loop over the results.

(This example is from Postmodern, a PostgreSQL package for Common Lisp.)

The key lesson: *code is data*. Interpreters and compilers are just programs that work on code as their data.

Bytecode and virtual machines

Before compiling, the next-best option is to produce *bytecode*, which is almost, but not quite, entirely unlike assembly language. Bytecode is a set of instructions for a *virtual machine* – a hypothetical CPU. Instead of having the typical operations your CPU provides, this hypothetical CPU has instructions that do the types of things your programming language needs. For example, here's some Python bytecode for a function called `min(x, y)`:

```
2           0 LOAD_FAST                0 (x)
            3 LOAD_FAST                1 (y)
            6 COMPARE_OP               0 (<)
            9 POP_JUMP_IF_FALSE       16
```

```
3           12 LOAD_FAST                0 (x)
            15 RETURN_VALUE

5     >>    16 LOAD_FAST                1 (y)
            19 RETURN_VALUE
            20 LOAD_CONST               0 (None)
            23 RETURN_VALUE
```

Python's hypothetical processor is a *stack machine*: each instruction takes arguments off the stack and pushes results onto the stack. The two LOAD_FAST instructions push the arguments onto the stack, and COMPARE_OP compares them and pushes True or False onto the stack, and so on.

Instead of parsing the code into an AST and stopping, the AST has to be converted into bytecode. Notice the bytecode doesn't reference variables by name, so variable accesses and lookups are faster. (This is why global variables are slow in languages like Python: function arguments are known when the function is parsed, so they can be pushed on the stack easily, but globals are only know when the function runs, so the interpreter has to look them up in a table every time.)

Stack machines are easy to write but require shuffling data around on the stack, which may require extra instructions and overhead. Consider a simple Scheme function in the Guile interpreter:

```
(lambda (x y)
  (let ((z (+ x y)))
    (* z z)))
```

In bytecode, it is:

```
> ,disassemble (lambda (x y)
                 (let ((z (+ x y)))
                   (* z z)))

   0    (assert-nargs-ee/locals 10)     ;; 2 args, 1 local
   2    (local-ref 0)                   ;; `x'
   4    (local-ref 1)                   ;; `y'
   6    (add)
   7    (local-set 2)                   ;; `z'
   9    (local-ref 2)                   ;; `z'
  11    (local-ref 2)                   ;; `z'
  13    (mul)
  14    (return)
```

We push the two arguments onto the stack, add them, name the result, push it onto the stack twice, multiply, and then return the result. This is inefficient – only two of the instructions are actual math.

Other languages, like Lua (and more recent Guile versions), use a register-based VM with named locations for storing data, more like actual processors use.

Lots of languages run on bytecode: Python, Java, PHP, Lua, C#, and many others.

R gained a bytecode compiler several years ago, and base R functions are bytecode-compiled. This gives a modest speed benefit over the default AST walker.

Optimizers

Because bytecode is intended to be a simple set of core instructions, it's easier to optimize. The interpreter can pattern-match certain sets of bytecode and replace them with more efficient constructions. This is known as *peephole optimization*, because the optimizer only looks at a few instructions at a time.

Bytecode optimization can be combined with other types of optimization which use knowledge of the AST and the control flow in the program:

Constant folding Constant expressions (like `1/sqrt(2 * pi)`) can be recognized and evaluated in advance, instead of evaluated every time the code runs.

Loop invariant code motion Expressions inside a loop which do not change from one iteration to the next are pulled out, so they are only calculated once.

Constant subexpression elimination If the same expression appears multiple times, it can be calculated once and stored to a temporary variable.

Dead code elimination Calculations whose results are not used can be skipped entirely.

There are many others. Sophisticated compilers do dozens of separate optimization passes; bytecode interpreters like Python are usually much less sophisticated, since fancy optimization delays execution. LLVM, a framework for building compilers, has an industrial-strength optimization system, as does GCC.

Just-in-time compilation

It's hard to produce efficient machine code for an interpreted language because any variable could have any type – a number, a list, an object with overloaded operators, whatever. Many types of optimization aren't feasible.

But sometimes the interpreter can deduce the possible types. It might observe the program running and see what types are common, or use *type inference* using the code it can see. What then?

In *just-in-time compilation*, the interpreter recognizes when the types of variables are known and generates specialized machine code for them. JITed languages include Java, C#, JavaScript, Julia, and even Python with the PyPy system.

This compilation adds overhead: the interpreter does extra work recognizing when code can be JIT compiled, but saves time interpreting that code.

Compiling to machine code

C, C++, Common Lisp, Go, Haskell, OCaml and many others can be compiled directly to machine code instead of run by an interpreter.

Ahead-of-time (AOT) compilation changes the tradeoffs. An AOT compiler can spend massive amounts of time optimizing code, since the optimization only happens once. A JIT compiler needs to work as fast as possible so the program isn't slowed down by compilation. An AOT compiler can analyze the entire program at once, inferring data types and properties to make better optimization decisions. AOT compilers can even use *profile-guided optimization* (PGO), which involves running the program and observing its behavior to make better optimization decisions.

Resources

- (How to Write a (Lisp) Interpreter (in Python)), Peter Norvig's tutorial on writing a simple parser and interpreter in Python.
- Write Yourself a Scheme in 48 Hours, a more intense introduction to using Haskell to interpret Scheme.
- Why Python is Slow
- Andy Wingo's blog post A Register VM for Guile, explaining the internal details of one kind of VM.
- How L1 and L2 CPU caches work, and why they're an essential part of modern chips

Rcpp

Rcpp Basics

Wrapping

`evalCpp()` evaluate short C++ code snippets, given as string

`cppFunction()` defines an R function from a C++ function given as a string

`sourceCpp()` compiles and links a C++ source file and exports tagged functions into R

Example:

```
f <- cppFunction('double weightedMean(NumericVector x, NumericVector w) {
  int n = x.size();
  double numerator = 0.0;
  double denominator = 0.0;
  for ( int i = 0 ; i < n ; ++i ) {
      numerator += x[i] * w[i];
      denominator += w[i];
  }
  // No error checking or assertions in this example, see below
  return numerator/denominator;
}')
f(1:4, rep(1,4))  # => 2.5
```

Note the structure of the `for` loop:

```
for ( initializers; condition; updater ) { BODY }
```

where the initializer can contain declaration of variables that are then **only visible** inside the loop. For error checking we might include:

```
try {
    if ( is_true(any(w < 0.0)) || denominator <= 0.0 ) {
        throw std::domain_error("Invalid weights");
    }
    return numerator/denominator;
} catch(std::exception &ex) {
    forward_exception_to_r(ex);
} catch(...) {
    ::Rf_error("c++ exception (unknown reason)");
}
```

`sourceCpp()` reads its input from a file and creates a shared, dynamically linked library. (It can really also take a string with the `code` argument, which is how the other two work.)

```cpp
// File slow-fib.cpp
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci(const int x) {
    if (x == 0) return(0);
    if (x == 1) return(1);
    return fibonacci(x - 1) + fibonacci(x - 2);
}
```

Note the export comment tag (the space matters), which marks the function for export to R.

```r
sourceCpp("slow-fib.cpp")
fibonacci(10) # => 55
```

C++ Features

Unlike R, C++ is a compiled, statically typed language.

Each variable must be given a specific type, and each function must be declared with the types of its arguments and of its return value.

Static typing lets the compiler optimize effectively, but it puts more constraints on the developer.

C++ is a large, complex language with many features. A few things are worth remembering:

- Standard C is also legal C++.
- Syntax has similarities with R, but (non-compound) statements must all be terminated with a `;`.
- C++ (like C) is zero-indexed, not one-indexed like R. Beware.
- There is no `<-` operator: use = for assignment.
- Scalars and vectors (or other aggregate types) are not interchangeable (though a spoonful of Sugar helps).
- Functions must explicitly `return` their value.
- You can use C libraries and functions directly (note: externs).
- The Standard Template Library (or STL) exposes a wide variety of rich and well-tested data structures and algorithms.
- The Boost library is a powerful third-party library that goes above and beyond the STL.
- C++ has evolved, modern versions: C++11 and C++14 offer many nice new features. You may have to configure specially to use those features with Rcpp.

Scalar Types

The common "scalar" types are bool, int, double, and String. (All but the last of these are C++ primitive types.)

```
double trim(double x, double threshold) {
    if ( x > threshold ) {
        return threshold;
    } else if ( x < -threshold ) {
        return -threshold;
    } else {
        return x;
    }
}
```

Exercise: Write a function `signum()` that takes an integer and returns -1, 0, or 1 if that integer is negative, zero, or positive.

```
int signum(int x) {
  if (x > 0) {
    return 1;
  }
  if (x == 0) {
    return 0;
  }
  return -1;
}
```

Vector Types

Rcpp defines several classes to handle R vectors. These have a nice range of methods and work well with "sugar" as we'll see below.

NumericVector, IntegerVector, CharacterVector, LogicalVector

For instance, you use the `.size()` method to get the length of the vector, as illustrated above.

Several ways to create vectors:

```
SEXP x;
std::vector<double> y(10);

NumericVector xx(x);       // create from a SEXP
NumericVector xx(10);      // of a given size (filled with 0)
NumericVector xx(10, 2.0); // ... with a default for all values
NumericVector xx( y.begin(), y.end() ); // range constructor

// using create
NumericVector xx = NumericVector::create(
    1.0, 2.0, 3.0, 4.0 );
// with names attribute
```

```
NumericVector yy = NumericVector::create(
    Named["foo"] = 1.0,
    _["bar"] = 2.0 ); // short for Named
```

Extracting and assigning values:

```
double u = xx[0];
double v = xx(1);
double z = yy["foo"] + yy["bar"];

xx[0] = 1.618;
xx(1) = -1.0;
yy["foo"] = 10.0;

yy["foobar"] = 1;  // grow the vector
```

These vectors support some nice R-like operations:

```
// [[Rcpp::export]]
NumericVector positives(NumericVector x) {
    return x[x > 0];
}

// [[Rcpp::export]]
NumericVector in_range(NumericVector x, double low, double high) {
    return x[x > low & x < high];
}

// [[Rcpp::export]]
NumericVector no_na(NumericVector x) {
    return x[ !is_na(x) ];
}

// [[Rcpp::export]]
List first_three(List x) {
    IntegerVector idx = IntegerVector::create(0, 1, 2);
    return x[idx];
}

// [[Rcpp::export]]
List with_names(List x, CharacterVector y) {
    return x[y];
}
```

Returning new vectors

```
pdistR <- function(x, ys) {
  sqrt((x - ys)^ 2)
}
```

```
NumericVector pdistCpp(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);  // <- note constructor

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}
```

Matrix Types

Rcpp supplies various matrix types as well: NumericMatrix, IntegerMatrix, CharacterMatrix, LogicalMatrix

- Use `.nrow()` and `.ncol()` methods to get dimensions
- Use `()` not `[]` for indexing

```
NumericVector rowSumsCpp(NumericMatrix x) {
  int nrow = x.nrow();
  int ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out[i] = total;
  }
  return out;
}
```

Functions

You can pass, use, and return R functions from within C++. Note the `_[]` construction for named arguments.

```
Function rnorm("rnorm");

rnorm(100, _["mean"]=10.2, _["sd"]=3.2);
```

Other Useful Classes

List, DataFrame, Environment are often directly useful, analogously to how we use them in R.

(Note: DataFrames are not easy to use as input because of static typing.)

There are other specialized classes in the library that are less commonly used but are valuable when you need them: SEXP, DottedPair, ...

STL Interface

One of the big advantages of C++ is a fantastic and well-tuned run-time library. The STL is the center of this. Rcpp plays nicely with the STL.

An important type in the STL is the *iterator* over some collection.

```
double iteratorSum(NumericVector x) {
    double total = 0;
    NumericVector::iterator it;
    for(it = x.begin(); it != x.end(); ++it) {
      total += *it;
    }
    return total;
}
```

Note operations

`.begin()` iterator pointing to beginning of collection

`.end()` iterator pointing just past the end

`=` or `!` equality checks (cf. distance)

`++` advance (also `--` for bidirectional iterators)

`*` dereferencing.

Algorithms:

```
// sum a vector from beginning to end
double s = std::accumulate(x.begin(), x.end(), 0.0);

// prod of elements from beginning to end
int p = std::accumulate(vec.begin(),
                        vec.end(), 1, std::multiplies<int>());

// inner_product to compute sum of squares
double s2 = std::inner_product(res.begin(), res.end(),
                               res.begin(), 0.0);
```

Another example:

```
IntegerVector findInterval(NumericVector x, NumericVector breaks) {
    IntegerVector out(x.size());

    NumericVector::iterator it, pos;
    IntegerVector::iterator out_it;

    for(it = x.begin(), out_it = out.begin(); it != x.end();
        ++it, ++out_it) {
```

```
      pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
      *out_it = std::distance(breaks.begin(), pos);
    }

    return out;
}
```

Rcpp Syntactic Sugar

Rcpp provides R-like "syntactic sugar" for operating on vectors in a concise way.

Commonly types of functions:

- Math functions: abs(), ceil(), sin(), cos(), ...
- Scalar summaries: min(), max(), sum(), ...
- Vector summaries: cumsum(), diff(), pmin(), pmax()
- Search: match(), which$_{\text{max}}$(), duplicates(), unique(), ...
- Distribution functions (d, q, p, and r versions)
- Vector views: head(), tail(), rev(), seq$_{\text{along}}$(), seq$_{\text{len}}$(), rep$_{\text{each}}$(), rep$_{\text{len}}$()

And More

There are many additional deep features in Rcpp that are useful in practice. Check the resources.

There are also many plugins and packages that are easy to include and use:

- Fast matrix computations (Armadillo)
- Eigenvalue Problems (Eigen)
- Optimization
- Monte Carlo Simulation
- Numpy interface
- Boost interfaces

See `http://rcpp.org/` for links to these packages.

Cython

Cython is an optimizing compiler for Python. It turns Python code into C code which can be compiled into highly efficient native code, provided you do a tiny bit of extra work to annotate variable types.

Cython also makes it easy to call C or C++ libraries, so if you need Python to call an external package, Cython may be the way to go. (cffi is a simpler way, if you just need to call a few C functions.)

Cython is useful when you've done extensive profiling to find bottlenecks in your code. Intensive loops and calculations can be factored out into Cython and easily made fast.

Examples

Here's some real Python code from my project:

```
def intensity_grid(xs, ys, xy, Ms, ts, alpha, theta, omega, sigma2, eta2,
                   T, t, min_dist2, min_t):
    tents = np.empty((xs.shape[0], ys.shape[0]))

    for ii in range(xs.shape[0]):
        for jj in range(ys.shape[0]):
            tents[ii, jj] = intensity_at(xs[ii], ys[jj], xy, Ms, ts, alpha,
                                         theta, omega, sigma2, eta2, T, t,
                                         min_dist2, min_t)

    return tents
```

The grid is often very big, and every `intensity_at` call requires a sum over every crime – so this is a very slow and very expensive function. I'd like to speed it up, and parallelize it if possible. We can move it to a separate file, `intensity.pyx`, and start by annotating the variable types:

```
import numpy as np

def intensity_grid(double[:] xs, double[:] ys, double[:,:] xy, long[:] Ms,
                   double[:] ts, double[:] alpha, double[:] theta, double omega,
                   double sigma2, double eta2, double T, double t,
                   double min_dist2, double min_t):
    cdef int ii, jj
    cdef double[:,:] tents = np.empty((xs.shape[0], ys.shape[0]))

    for ii in range(xs.shape[0]):
        for jj in range(ys.shape[0]):
            tents[ii, jj] = intensity_at(xs[ii], ys[jj], xy, Ms, ts, alpha,
                                         theta, omega, sigma2, eta2, T, t,
                                         min_dist2, min_t)

    return np.asarray(tents)
```

(I'll assume `intensity_at` has been moved to the same file and is getting the same sort of treatment.)

Now the generated C code doesn't need all sorts of expensive type-checking operations – it checks the variable types at the beginning of the function and then generates highly efficient code for the rest.

We're doing a lot of array accesses. Python checks the bounds on every access to make sure we don't access out of bounds. But we know we're not going out of bounds, so we can annotate:

```
@cython.boundscheck(False)
@cython.initializedcheck(False)
@cython.wraparound(False)
def intensity_grid(double[:] xs, double[:] ys, double[:,:] xy, long[:] Ms,
                   ...):
```

This tells Cython not to check array bounds, not to check if the array is initialized before we use it, and not to do wraparound indexing (i.e. `A[-1]` gets the last element of the array). This generates yet more efficient code.

One last bonus – Cython supports OpenMP, a framework for parallelization of code. This function is a prime candidate for parallelization, since each pass through the inner loop is separate from the other passes. It's embarrassingly parallel. Let's write the full parallel version:

```
@cython.boundscheck(False)
@cython.initializedcheck(False)
@cython.wraparound(False)
def intensity_grid(double[:] xs, double[:] ys, double[:,:] xy, long[:] Ms,
                   double[:] ts, double[:] alpha, double[:] theta, double omega,
                   double sigma2, double eta2, double T, double t,
                   double min_dist2, double min_t):
    cdef int ii, jj
    cdef double[:,:] tents = np.empty((xs.shape[0], ys.shape[0]))

    for ii in prange(xs.shape[0], nogil=True, schedule='static'):
        for jj in range(ys.shape[0]):
            tents[ii, jj] = intensity_at(xs[ii], ys[jj], xy, Ms, ts, alpha,
                                         theta, omega, sigma2, eta2, T, t,
                                         min_dist2, min_t)

    return np.asarray(tents)
```

`prange` is a Cython build-in function which acts like `range`, but evaluates in parallel. The `nogil` option tells Cython that I'm not going to use Python's Global Interpreter Lock, which prevents multiple threads from accessing the Python interpreter simultaneously – here I'm promising to only call Cython code, so it can be called in parallel.

The `schedule` option chooses how work will be assigned to threads (on different CPU cores). I chose to have the work just evenly divided, since each iteration should take about the same amount of time; other schemes split dynamically based on how long each iteration is taking and which threads are free.

Now my code executes in parallel. A task which would have taken over an hour, largely inside `intensity_at`, now takes fifteen or twenty minutes (on a quad-core machine).

Notice I didn't have to tell Cython that `tents` should be shared between threads; it deduced this automatically. It can also handle "reduction variables":

```
def sum(double[:] big_array):
    cdef double s = 0.0

    for ii in prange(big_array.shape[0], nogil=True, schedule='static'):
        s += big_array[ii]

    return s
```

Here each thread adds up its portion of `big_array` and the results are automatically summed together at the end, producing a nice parallel sum. This only works for simple operations, like `+`, which Cython can automatically figure out how to reduce.

Parallelization won't work for variables that have to be shared for reading and writing by multiple threads simultaneously – that's a much more difficult task, and one we'll talk about more later.

Analyzing Cython performance

The only way Cython code can be fast is if it understands your data types well enough to generate efficient C. Cython provides tools for inspecting the generated C code with `cython -a example.pyx`.

Building Cython code

Cython code has to be compiled before you can use it. You can manually convert it to C and compile it with your favorite C compiler, like GCC:

```
cython example.pyx

gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \
    -I/usr/include/python2.7 -o example.so example.c
```

This produces an `example.so` file, a shared library you can import into Python code with the normal `import example` statement.

Alternately, you can use distutils, Python's tool for building packages. You first create a file named `setup.py` that specifies what has to be compiled:

```python
from distutils.core import setup
from Cython.Build import cythonize

setup(
    name = "An example module",
    ext_modules = cythonize('example.pyx')
)
```

Then you can just run `python setup.py build_ext --inplace` and your module will be Cythonized and compiled automatically for you. There are various options you can add if you need to call other libraries, like OpenMP; see the manual for details.

You can also build this step into a Makefile to automatically compile your Cython whenever necessary.