R and C/C++

Jean Feng

I hear C/C++ is fast...

From Darren Wilkinson's "Gibbs sampler in various languages":

Language	Relative Speed compared to R
R	1.00x
Python	1.86x
C	53.70x

But the relative speedup depends on what you are trying to do and how you originally coded it in R.

Why and why not C/C++

Why not C/C++:

- Many operations in R are already backed by C/C++ and Fortran implementations.
- One line of code in R is typically many lines of code in C/C++.
- In C/C++, there will be more bugs and they are harder to fix.
- R is much better for prototyping.

Why and why not C/C++

Why not C/C++:

- Many operations in R are already backed by C/C++ and Fortran implementations.
- One line of code in R is typically many lines of code in C/C++.
- In C/C++, there will be more bugs and they are harder to fix.
- R is much better for prototyping.

Why C/C++:

- C/C++ allows for efficient implementation of exactly the kinds of algorithms that R is not good at: e.g. tasks with loops that can't be vectorized away.

Easy to "vectorize"

```
% R code: Dot product
total = 0
x = rnorm(10)
y = rnorm(10)
for (i in 1:10) {
  total = total + x[i] * y[i]
}
```

Easy to "vectorize"

```
% R code: Dot product
total = 0
x = rnorm(10)
y = rnorm(10)
for (i in 1:10) {
  total = total + x[i] * y[i]
}
```

```
% R code: Dot product - vectorized
x %*% y
```

Hard to "vectorize"

Iterative procedures:

- Optimization algorithms, such as
  - Gradient descent
  - Expectation maximization
- Markov chain monte carlo

Hard to "vectorize"

Iterative procedures:

- Optimization algorithms, such as
  - Gradient descent
  - Expectation maximization
- Markov chain monte carlo
- Lots of things in phylogenetics

C/C++ vs. R

C/C++ are low-level *compiled* languages

- You are in complete control of memory management.

C/C++ vs. R

C/C++ are low-level *compiled* languages

- You are in complete control of memory management.

R is a high-level *interpreted* language

- Handles memory allocation, different data types, and all types of R magic.

Calling R from C: the old way

1. Write your function in C, ex: `hello_world.c`
2. Compile the file into an executable program. Use 'R' from command line to compile C program into a dynamic library, ex: `R CMD SHLIB hello_world.c` [1]
3. Load the executable program into R using the `dyn.load` function.
4. Write an R wrapper function using the `.C` or `.Call` function.

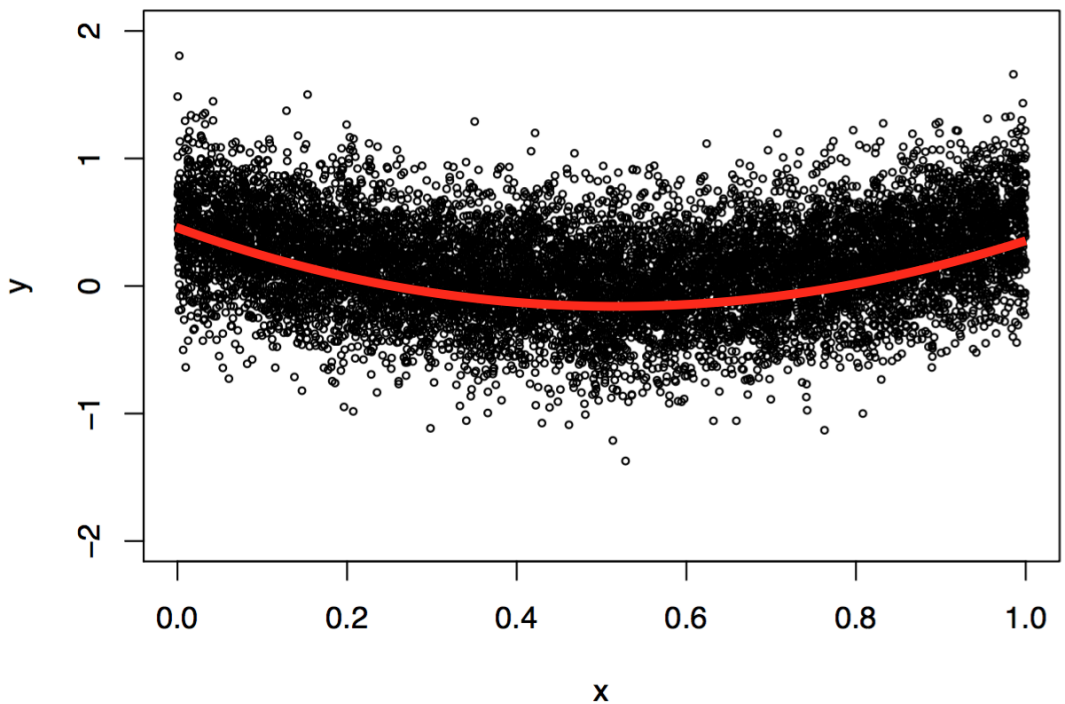
See `http://cran.r-project.org/doc/manuals/R-exts.pdf` for detailed steps.

[1] Details differ based on Unix vs Windows.

Calling R from C: example

For example, we want to smooth a curve, given points (x_i, y_i) so that the new curve is:

$$y.smooth(x) = \frac{\sum_{z:|z-x|<r} y(z)}{\sum_{z:|z-x|<r} 1}.$$



Calling R from C: example

```c
#include "stdlib.h"
#include "math.h"

void smooth(double *x, double *y, double *y_hat,
            int *np, double *radiusp){
  int n = *np;
  double radius = *radiusp;
  for (int i = 0; i < n; i++) {
    int count = 0;
    for (int j = 0; j < n; j++) {
      if (fabs(x[i] - x[j]) < radius) {
        count++;
        y_hat[i] += y[i];
      }
    }
    y_hat[i] /= count;
  }
}
```

Calling R from C: example

```
# Wrapper function in R
smooth <- function(x, y, r){
  y_hat <- rep(0, length(x))
  # Call out to C
  .C(
    "smooth",
    x=as.double(x),
    y=as.double(y),
    y_hat=as.double(y_hat),
    n=length(x),
    radius=r)
}

# Run our new function
smooth(seq(10), seq(10))
```

Calling R from C

Please don't bother with this old way! Use Rcpp...

Rcpp

`Rcpp` was created by Dirk Eddelbuettel and Romain Francois in 2011 because the old way of adding C to R was too clunky.

- Permits direct interchange of rich R objects between R and C++.
- Provides syntactic sugar

Setup

- R (>= 3.1)
- The R package Rcpp.
- You'll also need a working C++ compiler:
  - For Windows users, install `Rtools`.
  - For Mac users, install Xcode from the app store.

Example: T-statistic in R

An R function

```
t.test.me <- function(x1, x2) {
  n1 <- length(x1)
  n2 <- length(x2)
  # Generate numerator and denominator
  nume <- mean(x1) - mean(x2)
  denom <- sqrt(var(x1)/n1 + var(x2)/n2)

  return(nume/denom)
}
```

Example: T-statistic in C++

File: `t_test_cpp.cpp`

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double t_test_cpp(
    NumericVector x1,
    NumericVector x2) {
  int n1 = x1.size();
  int n2 = x2.size();

  // Generate numerator and denominator
  double nume = mean(x1) - mean(x2);
  double denom = sqrt(var(x1)/n1 + var(x2)/n2);
  return nume/denom;
}
```

Example: Compare T-statistic calculations

Now run your C++ code

```
// Compile the C++ code
Rcpp::sourceCpp("t_test_cpp.cpp")
set.seed(1)
x1 <- rnorm(30)
x2 <- rnorm(50)

microbenchmark(
  t.test.me(x1, x2),
  t_test_cpp(x1, x2))
```

A 20x speedup using C++

Function	Mean
t.test.me in R	40.8
t_test_me in C++	2.8

Recap: how to Rcpp

1. Write C++ code using Rcpp. Annotate your function with `// [[Rcpp::export]]`
2. Compile the C++ code using 'Rcpp::sourceCpp'.
3. Use your C++ function in R.

Rcpp data types: Vectors

- `NumericVector`: can hold real-valued numbers (`double`)
- `IntegerVector`: can hold integer-valued numbers
- `LogicalVector`: binary (`bool`)
- `CharacterVector`: vector of strings

Rcpp data types: List

`List`: a general data type which can contain other types, similar to how `list` works in R.

```
Rcpp::List foo(Rcpp::List mod) {
  // Read in list from R
  double min = Rcpp::as<double>(mod["min"]);
  int size = Rcpp::as<int>(mod["size"]);

  // Return a list to R
  return Rcpp::List::create(
    Rcpp::Named("bar") = min,
    Rcpp::Named("spam") = size);
}
```

Rcpp data types: DataFrame

`DataFrame`: essentially the same thing as a list, but every column must have the same length.

```
DataFrame foo_dataframe(DataFrame mod) {
  // Read our dataframe
  NumericVector x = as<NumericVector>(mod["x"]);

  // Return a dataframe to R
  return DataFrame::create(
    Rcpp::Named("z") = x);
}
```

Rcpp syntactic sugar

Syntactic sugar brings a higher level of abstraction to your C++ code.

- Less code to write the same thing – easier to read, write, and maintain
- Provides a subset of the high-level R syntax to C++.

Rcpp syntactic sugar

Syntactic sugar brings a higher level of abstraction to your C++ code.

- Less code to write the same thing – easier to read, write, and maintain
- Provides a subset of the high-level R syntax to C++.

Examples:

- Math functions like `sin, log, pmin, var`...
- d/q/p/q Statistical Functions, e.g. `dnorm`

RcppArmadillo

`RcppArmadillo` is a C++ linear algebra library that balances speed and ease of use.

- To use it, add this to the top of your C++ file:

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
```

- Now you have access to a long list of Matrix/Vector operations:
  `http://arma.sourceforge.net/docs.html#part_fns`
  - Element-wise functions: `exp`, `log`, `sqrt`, ...
  - Matrix decompositions: `Cholesky`, `SVD` ...
  - Sparse matrices
  - Higher-dimensional arrays

Resources

- Hadley Wickham's site: a nice intro
  `http://adv-r.had.co.nz/Rcpp.html`
  - To get a handle on Rcpp, try out the simple exercises.
- Dirk Eddelbuettel's site: links to all things Rcpp
  `http://dirk.eddelbuettel.com/code/rcpp.html`
- "Seamless R and C++ Integration with Rcpp" book on SpringerLink, a comprehensive resource on Rcpp