M+E+C:
Computation with **R**

Keith O'Hara

05/21/2018

R in a Nutshell

- **R** is a modern, open-source implementation of the **S** programming language.
- Written mostly in `C`
- **R** is designed to be a mixture of interactive and OO-style programming
  - 'Extreme dynamism'
  - Classes and methods
- It's a mature project... which is both a good and bad thing
  - *e.g.*, not designed with parallel programming in mind
  - started in 1993 by **R**obert Gentleman and **R**oss Ihaka
  - version 1.0.0 released in 2000; version 3.5.0 released in April 2018
- **R**-Core group; mailing lists; SVN
- RStudio

R vs Matlab

- Syntax comparison:

R	Matlab
C <- t(A)%*%B	C = A′ * B
C <- A * B	C = A. * B
C <- solve(A)	C = inv(A)
C <- solve(A,B)	C = A\B
n <- nrow(A)	n = size(A,1)
C <- matrix(rnorm(n*k),n,k)	C = normrnd(0,1,[n,k])

- Speed?

R features

OOP and **R**

- Not quite OOP
- Usual OOP style:

  `object.method(input)`

  For example:

  `model.solve(parameters)`

  That is, the `solve` method depends on the model type.
- **R** S3 format:

  `method(object, input)`

OOP and **R**

- There are multiple approaches to OO programming in **R**:
  - S3: informal; ad hoc; most commonly used
  - S4: more formal; multiple dispatch; awkward
  - Reference Class: conforms more to usual message-passing OO systems; looks like `object$method(input)`
- S4:

  `object@member`

- Not as widely used as S3. Bioconductor is a well-known collection of S4 packages.

Peculiarities of **R**

- "**R** is slow..." limited BLAS and LAPACK
  - Build using OpenBLAS or system libraries (such as vecLib)
- The number of function inputs affects performance (even if they're not used); *e.g.*,

  ```
  inner_product(x,y,blah,blah,blah,blah,blah,blah)
  ```

  This is generally not a feature of compiled programming languages.
- This is due to 'lazy evaluation.' Example:

  ```
  foo <- function(x, y=z) {z <- x*x; y*log(z)/x}
  foo(2); foo(2,1)
  ```
- Vectors and matrices (v.s. Matlab)
- `C` API
- Assignment operators: `<-` vs `=`

R Packages

- Great package system!

  The Comprehensive R Archive Network (CRAN): 12584 packages
- Bioconductor
- Some useful packages:
  - devtools
  - Rcpp
  - RcppArmadillo
  - ggplot2

R Packages: Building from Source

- Building **R** packages from source requires some tools.
  - Windows users should install Rtools.
  - macOS users should install provided clang and gfortran binaries.
- Example:

```
install.packages("devtools")
library(devtools)
install_github("TraME-Project/Shortest-Path-R")
```

- **Let's look at the structure of a package!**

R Internals

S-Expressions (SEXP)

- All R objects are declared as SEXP objects when passed as inputs on a C-level.
- You can pass pretty much anything as a SEXP object; you can even use it to call **R** functions from C/C++ code.

```
SEXP add_one (SEXP a_R, SEXP func)
{
    try {
        Function myFunc = as<Function>(func);
        NumericVector a = as<NumericVector>(a_R);
        NumericVector b = myFunc(a);
        //
        return wrap(b);
    } catch( std::exception &ex ) {
        forward_exception_to_r( ex );
    } catch(...) {
        ::Rf_error( "C++ exception (unknown reason)" );
    }
    return R_NilValue;
}
```

Rcpp and RcppArmadillo

- Rcpp is a great package with an easy to use (and abuse) API for working with `C++` and **R**.
- RcppArmadillo is essentially a skeleton package that contains the Armadillo header files.
- Use Rcpp; avoid using **R**'s `C` API directly.
- Load the Rcpp package and call dynamic loaded code using

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

.Call("my_C_function",input_1,input_2)
```

Parallel Computing

- parallel package; combines snow and others
- How it works; memory issues
- `foreach` function
- Passing current environment and other functions
- Tip for working with **R** on NYU's HPC (nodes vs cores)

Parallel Computing: Example

```
library(doParallel)
#
n_cores <- 2
#
cl <- makeCluster(n_cores)
registerDoParallel(cl)

kk <- foreach(i=1:8, .combine=c) %dopar% rnorm(i*10)

stopCluster(cl)
```

- Options: `.inorder`, `.packages`, `.export`, `.noexport`

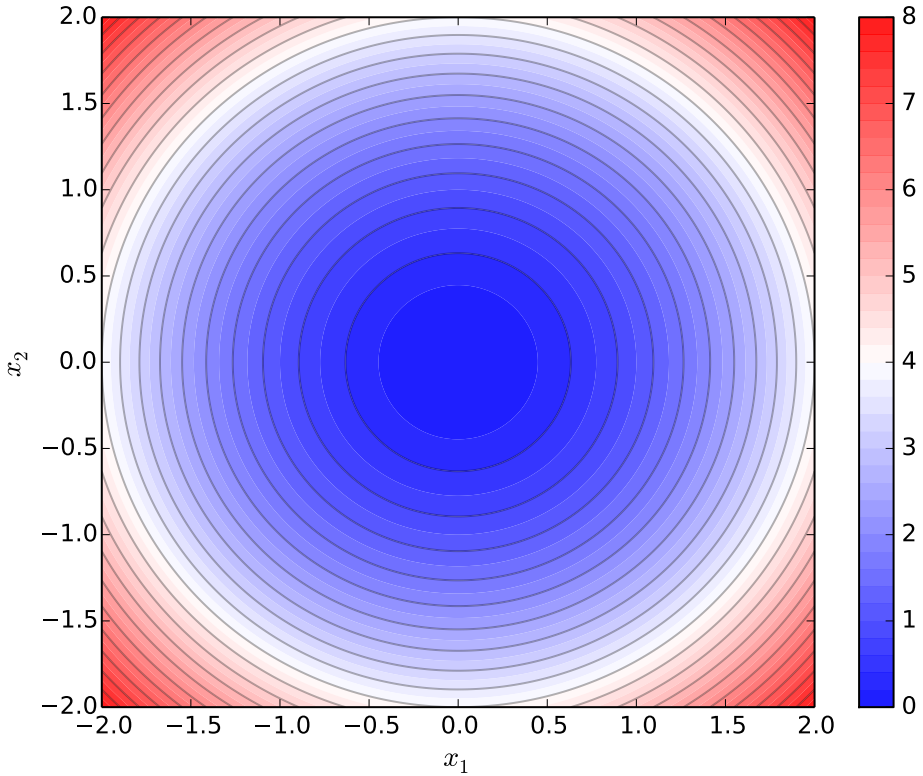
Application: Numerical Optimization

Example: Sphere Function

- Minimize

$$\min_{\mathbf{x}} \left\{ \sum_{k=1}^{n} x_k^2 \right\}, \quad \nabla f(\mathbf{x}) = 2\mathbf{x}$$

- In two dimensions:



Example: Sphere Function

R code:

```
fn <- function(x) { return(sum(x*x)) }
gn <- function(x) { return(2*x) }

n <- 10
x0 <- rep(2,n)
optim(x0,fn,gn,method="BFGS")
```

Example: Logistic Regression

- Data:

$$y_i|\mathbf{x}_i, \boldsymbol{\theta} \sim \text{Bern}(\mu_i)$$

  where $\mu_i := \text{sigm}(\boldsymbol{\theta}'\mathbf{x}_i)$, $\text{sigm}(x) = \frac{1}{1+\exp(-x)}$.

- Objective function:

$$f(\boldsymbol{\theta}) := -\sum_{i=1}^{N} [y_i \ln(\mu_i) + (1 - y_i) \ln(1 - \mu_i)]$$

- Gradient and Hessian:

$$\nabla_{\boldsymbol{\theta}} f := \mathbf{X}'(\boldsymbol{\mu} - \mathbf{y})$$

$$\mathbf{H} := \nabla_{\boldsymbol{\theta}}[\nabla_{\boldsymbol{\theta}} f]' = \mathbf{X}'\mathbf{S}\mathbf{X}$$

  $\mathbf{S} := \text{diag}(\mu_i(1 - \mu_i))$.

Example: Logistic Regression

```
sigm <- function(x) 1/(1+exp(-x));
n <- 10000; k <- 10;
X <- matrix(rnorm(n*k),n,k); theta <- runif(k);

mu <- sigm(X%*%theta);
y <- numeric(n); for(i in 1:n) { y[i] <- rbinom(1,1,mu[i]) };

fn <- function(x,y,X) {
  mu <- sigm(X%*%x);
  return(-sum( y*log(mu) + (1-y)*log(1-mu) )) };
gn <- function(x,y,X) {
  mu <- sigm(X%*%x);
  return( t(X) %*%(mu - y) ) };

x0 <- rep(2,k);
optim(x0,fn,gn,X=X,y=y,method="BFGS")
```