Advanced Computing

- In reality, we typically only get one computing class with our students.
- Students need more exposure to internalize the ideas and make expressing computations 2nd nature.
- So ideally 2 courses
  - either an undergraduate course & graduate course
  - or 2 undergrad. courses.

- Encourage grad. students to take undergrad. class and then graduate class.
- 8 – 10 years of math concepts & vocabulary 1 course of computing !!!
- But need strong culture and support within a dept. to encourage students to see computing as important and not as a distraction from "real" purpose.

Advanced Computing

- In practice, the "Advanced" course covers the fundamentals covered in the intro. class since only one course.
  - getting started with R, language design and concepts, writing functions & programming
  - data input/output, text manipulation
  - shell tools and remote login

New topics

- Then turn to
  - efficient code
  - profiling
  - linking to existing C code (not writing it)
  - Parallel/Distributed computing concepts & packages
  - basics of C programming
  - interfacing to other languages/systems
    - e.g. Java, Python, BUGS
  - Software design & development
    - Object Oriented programming
    - Writing R packages
  - Important environment tools
    - Version control (Subversion, mercurial)
    - LaTeX, Sweave, Docbook (XML authoring system for richer dynamic documents)
  - Strategies for working with big data
  - Symbolic math. software

- Too much in 10 weeks, but whet their appetite and invite them to explore topics that interest them.
  - Give them the pitch why these things are useful for them and why they need to learn them for their professional development and career.
  - Give them the concepts and some details to get them started and have them explore themselves
    - with support on the mailing list
    - advertises that some people are pursuing the topics and helps encourage others to also.

Optional Topics or Programming Exercises

- Web scraping
- Web graphics
- More computational statistics topics
  - i.e. algorithms used in implementing statistical methods
    - numerical optimization
    - bootstrap
    - cross-validation
    - Random number generation techniques
    - MCMC
    - Computer experiments & simulation
    - …

Profiling

Efficiency – timing code

- system.time(expression) returns the amount of time it took to evaluate the expression
- 3 measurements (or 5 if sub-processes)
  - User, system and total elapsed time.
  - Want user + system = total, or else something else consuming the machine so problematic measurement.
- Students should explore how time changes with size of the inputs.

- Issues with resolution and measuring time for small increments.
- Plot time versus input size and see if algorithm is linear, polynomial, exponential
  - Empirical algorithmic complexity

- To improve speed, use different algorithm.
- Or refine code
  - move expressions within loops that are invariant to compute just once and assigned to variable
  - Avoid concatenating vectors, but pre-allocate and assign to i-th element.
    - i.e. x = c(); for(i in seq(along = y)) x = c(x, g(y[i]))

Profiling

- Identify bottlenecks by measuring what functions are called the most often and take the most time.
- Rprof() function turns on measuring in the evaluator, producing data about function calls.
- Rprof("myProfilingData")
  Evaluate expressions
  Rprof(NULL)
- Data is now available in the file myProfilingData.

Analyzing profiling data

- (See the code in rw1.R, courtesy of Ross Ihaka)
- Read profiling data into R via summaryRprof()
  prof = summaryRprof("myProfilingData")
- List with 3 elements
  - Look at prof$by.self
  - self.time self.pct total.time total.pct

"sample"	0.94	52.2	1.06	58.9
"rw2d1"	0.70	38.9	1.80	100.0
"length"	0.08	4.4	0.08	4.4
"=="	0.04	2.2	0.04	2.2
"+"	0.02	1.1	0.02	1.1
"-"	0.02	1.1	0.02	1.1

2-D Random Walk

- First version (rw2d1) implements the algorithm in the most obvious way that mirrors the natural description of the process
  - good idea to implement using the obvious approach and "bank" that so one can check more subtle implementations.
  - Use as a benchmark/validator
  - Also, if it is sufficiently fast, stop and don't over optimize.

- Problem with comparing result to benchmark when random number generation
- Use set.seed() to have a common starting point
  - But still issues if generate random numbers in different ways, order, etc.

- Profiling tells us sample() is taking all the time.
- We have many calls to sample – 2 per iteration of the loop
  - in both cases, we generate just a single value, and both are from a binary distribution, i.e. up or down, left or right.
  - sample() is very powerful, but overkill. The overhead of setting up the values to sample from is too much for a single value.
  - Can replace with a Bernoulli, or a simple Uniform and whether it is > .5 or not.

- Interestingly, while we remove sample() as a bottleneck, we will later bring it back!
  - Because we will generate many values from a given set of values in a vectorized operation, not a single scalar returned each time.
  - In that context, the overhead of establishing the possible sample values becomes very small part of the overall operation.

- As we progress through the different implementations, we gradually move away from the loop and towards vectorized operations.
- Until R can recognize potential for vectorization and do it automatically for us, students need to learn to write code using vectorized operations where possible
- This also means that they should write functions that are vectorized also, so that they and others can use those functions in vectorized ways.

- Focus energy on improving the calls to the functions in the first rows of this data frame.
- Rewrite algorithm, adjust expressions.

R packages

- Writing R packages is simple
  - requires following straightforward conventions about folders and meta-data in specific files
- Easy for anyone to do, even on Windows
  - tool chain for building & installing packages getting easier there.
- Several important benefits
  - Easy to send self-installing code to self or others, including instructor
    - no issue with source individual files, in correct order, etc.
  - Can include data so others can reproduce issues, help debug
  - Can contain tests to verify the code is functional
  - Documentation
  - NAMESPACES

- Structure is a folder with several sub-folders and a DESCRIPTION file
- package.skeleton() function will create this for you
  - but better to understand the details directly
- Create a folder, typically with the name being the name of the package (not essential)
- Create and populate a DESCRIPTION file in the package folder.
- Create an R/ directory under that.
- Put R code files (.R, .r, .q, …) in the R/ directory.
- DONE!

DESCRIPTION file

- Package: RNYTimes
- Title: Access to the New York Times REST services
- Description: This provides R functions to access content
- from the New York Times using its REST services.
- Version: 0.1-0
- Author: Duncan Temple Lang
- Imports: XML, RJSONIO, RCurl
- Depends: methods
- Maintainer: Duncan Temple Lang <duncan@r-project.org>
- License: BSD

Processing the package

- One needs to install the package before loading it
  - Shell command
    - R CMD INSTALL myPackage
    - myPackage is the directory/folder containing your code
    - I often use R CMD INSTALL . when located in the package directory.
  - Can control where the installed package is located
    - via the –l libraryDirectory option
    - R_LIBS or R_USER_LIBS environment variable (preferable)
  - Then within R, library(myPackage)

Validating a package

- R CMD check myPackage
  - checks the package can be installed
  - has all the relevant pieces
  - is consistent with the meta data
  - runs any examples in the documentation files
  - processes any tests in the tests/ directory
  - checks the code for references to non-defined/ global variables
- Just a good thing to run!

Build a package to Distribute

- R CMD build myPackage
  - creates a tar.gz file with the source of the package others can install
- R CMD build –binary myPackage
  - builds a binary version of the package for the type of machine on which you run the command
    - people can install without needing additional tools on their machine
    - especially useful for building packages with C/FORTRAN code.

C/FORTRAN code

- To add C/FORTRAN code for use in a package, put the files in a directory named src/ under the package directory, i.e. parallel to the R/ directory.
- R CMD INSTALL will automatically compile & link into a DLL/Shared Object.
- Have to explicitly load it using dyn.load()/ library.dynam()
- OR preferably, use a NAMESPACE file and useDynLib() within that.

NAME SPACEs

- Add a NAMESPACE file to a package
- File that contains meta-information directives about what in the package the user should be able to see, and what should be hidden.
- 2 class of objects in the package
  - those that are exported, those that are not
  - export(myFun, myOtherFun)
    - leave out the internal implementation functions that I don't want people to be able to call directly.
  - Alternatively, export using a regular expression to match variable names
    - exportPattern(".*")
- Also, use directives to load DLL/SO
  - useDynLib(myPackage)

name space concept

- When you load a package into the search path, it might contain functions that conflict with or hide others later in the search path.
- This can be disastrous and is not good software design.
- So we want to hide some variables/objects which are not to be exported.
- So good hygiene to protect others from your naming scheme

What about imports?

- But what about functions we use in our package that come from other packages?
- What if some other package in our search path provides, e.g. plot or lm ?
- We want graphics::plot and stats::lm
  - (The :: is for specifying the namespace in which the variable is located.)
- So in our NAMESPACE file, we would use
  - importFrom(graphics, plot)
    importFrom(stats, lm)
- The code in the package will look in its imports and not along the search path.

Documentation

- Don't need to write documentation to use package mechanism
- Do need to write documentation to publish package on CRAN and have it pass R CMD check.
- Use the prompt() function to generate template doc. files for each function/object you want to document.
- Then edit. LaTeX-like markup language.
  - See Writing R Extensions manual.

Data for a package

- Put rda, csv, etc. files in a data/ directory
- Then can load into R via
  - data(myObject)

Big Data

- There are several approaches to dealing with large amounts of data in R
- Several packages
  - filehash, ff, bigmemory, biglm, bigglm, externalptr data types
  - Some deviate from the common semantics of the language and so are not ideal

Big Data

- Challenge on small computers – period.
- So have to have students use big, centralized hardware in dept.
- Need to know UNIX shell, ssh and be able to transfer code and data.
  - R package mechanism is excellent for this
  - So too distributed version control
- So my focus is more on these fundamentals rather than specialized approaches to large data, i.e. need to get the students to the stage where we can focus on these more interesting details.

- Parallel/distributed computing on a cluster or with multiple cores is perhaps more pedagogically interesting.
- Concepts that are more transferable across other languages and less "ad hoc", R-specific.
- Focuses them on thinking about parallel algorithms
  - in practice, most students will work with "embarassingly parallel" problems and so not have to think about changing the algorithm.

Packages for Distributed Computing

- See the High Performance Computing Task View on CRAN
- e.g. snow/snowfall, foreach, Rmpi, multicore
  - mapReduce
  - biocep

Issues

- Some issues in configuring cluster and firewalls to enable students to start tasks on multiple nodes.
- But programming interface quite simple.
- Distributing data-intensive tasks across nodes can greatly reduce performance improvement due to overhead of copying data.
- Similarly, short tasks are overwhelmed by proportion of time to distribute, start and harvest results.

- Students must have seen (or see for 1st time) the fundamentals.