Beware Data Frames -vs- Matrices

Tip 1

Writes to data frames are *slow*

```
for(i in 1:10000) { x[i,1] = 1 }
```

- Where `x` is a data frame: 3.5 sec
- Where `x` is a matrix: 0.04 sec

```
for(i in 1:10000) { x$V1[i] = 1 }
```

- Where `x` is a data frame: 3.5 sec

Beware Data Frames -vs- Matrices

Tip 2

When reading data frames, don't implicitly cast them to matrices.

```
for(i in 1:10000) { a = x[i,1] }
```

- Where `x` is a data frame: 0.7 sec
- Where `x` is a matrix: 0.02 sec

```
for(i in 1:10000) { a = x$V1[i] }
```

- Where `x` is a data frame: 0.05 sec

Efficient Memory Allocation

Tip 3

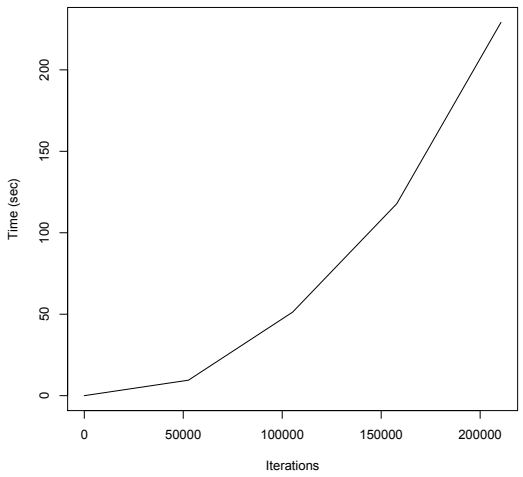
Pre-allocate all the memory you will use: don't expand vectors/matrices/data frames on the fly

```
x = NULL
for(i in 1:10000) { x = rbind(x, rnorm(20)) }
x = NULL
for(i in 1:10000) { x[i] = rnorm(1) } }
```

- 16.3 sec / 0.4 sec

```
x = matrix(nrow=10000,ncol=20)
for(i in 1:10000) { x[i,] = rnorm(20) }
x = vector("numeric", 10000)
system.time({ for(i in 1:10000) { x[i] = rnorm(1) } })
```

0.1 sec / 0.09 sec



Vectorize!

Tip 4

Ruthlessly Vectorize everything possible – `for` loops are evil!

```
x = seq(-6, 6, length.out=1e5)
area = 0
for(i in 2:1e5) { area = area+dnorm(x[i])*(x[i]-x[i-1]) }
```

- 0.63 sec

```
sum((x[2:100000]-x[1:99999])*dnorm(x[2:100000]))
```

- 0.02 sec

Optimised Basic Linear Algebra Subprograms (BLAS)

Tip 5 (Linux/Windows only)

Use a proper BLAS library for your architecture.

See cran.r-project.org/bin/windows/contrib/ATLAS as a starting point, though it often lags behind.

Find the `bin` subdirectory of your R installation and, after backing up the existing copy, replace the file `Rblas.so` (Linux)/`Rblas.dll` (Windows)

For example, squaring a 1000×1000 matrix takes 3 times longer on the default BLAS that comes with R compared to an optimized BLAS.

All matrix operations will be faster "for free": inversion, SVD, QR, Choleski, ...

Full Example - 2D Random Walk

Version A

```
rand_walk_A = function(n) {
   walk = data.frame(x = c(0), y = c(0))
   for(i in 2:n) {
      if(sample(c(TRUE, FALSE), 1)) {
         walk[i,1] = walk[i-1,1] + sample(c(-1,1), 1)
         walk[i,2] = walk[i-1,2]
      } else {
         walk[i,1] = walk[i-1,1]
         walk[i,2] = walk[i-1,2] + sample(c(-1,1), 1)
      }
   }
   walk
}
```


Execution Time (10,000 steps): 28.0 secs

Full Example - 2D Random Walk

Version B - eliminate data frames

```
rand_walk_B = function(n) {
   x = 0; y = 0
   for(i in 2:n) {
      if(sample(c(TRUE, FALSE), 1)) {
         x[i] = x[i-1] + sample(c(-1,1), 1)
         y[i] = y[i-1]
      } else {
         x[i] = x[i-1]
         y[i] = y[i-1] + sample(c(-1,1), 1)
      }
   }
   list(x=x, y=y)
}
```


Execution Time (10,000 steps): 1.8 secs

Full Example - 2D Random Walk

Version C - eliminate dynamically expanded memory

```
rand_walk_C = function(n) {
   x = vector("numeric", n); y = vector("numeric", n)
   for(i in 2:n) {
      if(sample(c(TRUE, FALSE), 1)) {
         x[i] = x[i-1] + sample(c(-1,1), 1)
         y[i] = y[i-1]
      } else {
         x[i] = x[i-1]
         y[i] = y[i-1] + sample(c(-1,1), 1)
      }
   }
   list(x=x, y=y)
}
```

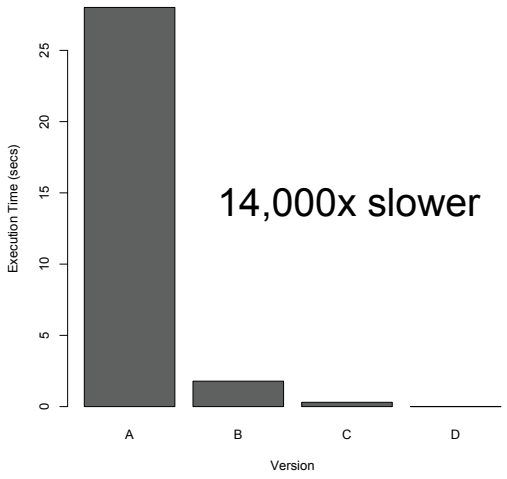

Execution Time (10,000 steps): 0.3 secs

Full Example - 2D Random Walk

Version D - fully vectorize

```
rand_walk_D = function(n) {
   move = sample(1:4, n-1, replace=TRUE)
   x = c(0, cumsum(c(-1,1,0,0)[move]))
   y = c(0, cumsum(c(0,0,-1,1)[move]))
   list(x=x, y=y)
}
```

Execution Time (10,000 steps): 0.002 secs



Full Example - 2D Random Walk

Can we go even faster without resorting to C?

Version E - forked processes (Extra tip for Mac OS X & Linux)

```
rand_walk_E = function(n) {
  move = sample(1:4, n-1, replace=TRUE)
  a = parallel(
        parse(text="c(0, cumsum(c(-1,1,0,0)[move]))"),
        "x")
  b = parallel(
        parse(text="c(0, cumsum(c(0,0,-1,1)[move]))"),
        "y", TRUE)
  collect(list(a, b))
}
```

Here actually much slower ... forking only benefits if: (a) slow running; and (b) small return size

Identifying Bottlenecks : `system.time`

For simple speed comparisons just use `system.time()`.

Tip: if you want to time a complex expression, just wrap in { }

```
system.time(dnorm(2); dnorm(3))
system.time(dnorm(2)
dnorm(3))
```

Both give syntax error, instead:

```
system.time({dnorm(2); dnorm(3)})
```

Also, output of `system.time` gives user/system/elapsed times. If system is a large percentage of elapsed time this indicates heavy OS management and possibly wasteful memory accesses.

Identifying Bottlenecks : `Rprof`

`system.time()` is very limited, so better to do code profiling to identify what's slowing you down.

Rprof - built-in profiling of R code

```
Rprof()
a = rand_walk_A(10000)
Rprof(NULL)
summaryRprof()
```

```
$by.self
                        self.time self.pct
"[<-.data.frame"             8.72     31.6
"xpdrows.data.frame"         8.64     31.3
"rand_walk_A"                5.12     18.6
"length<-"                   1.78      6.5
```


NB: `help("[<-.data.frame")` *not* `?[<-.data.frame` *etc*

Identifying Bottlenecks : Advanced Memory Options

`Rprof` would have helped spot the issues raised in tips 1, 2, 4 and 5. Identifying the slowdown caused by massive memory copying may be harder to spot.

There are two options:

- `gctorture()`
  `Rprof(memory.profiling=TRUE)`
  `[...]`
  `Rprof(NULL)`
  `gctorture(on=FALSE)`
  `summaryRprof(memory="both")`
  `summaryRprof(memory="tseries")`
  `summaryRprof(memory="stats")`
- `tracemem(myvar)`
  `[...]`
  `untracemem(myvar)`

Getting Setup

What you'll need when resorting to C:

- R, obviously (www.r-project.org)
- Mac OS X: XCode (included on system DVD)
- Linux: GCC and other compiler tools (included in most repositories, *e.g.* build-essential & r-base-dev on Debian/Ubuntu)
- Windows: Rtools (www.murdoch-sutherland.com/Rtools)

Calling Into C Code

There are two methods:

1. `.C`
   - R types are coerced to C types
   - C side of the coin needs no special knowledge of R internals
   - Easy and fast to code
   - Can even interface to existing libraries that weren't written for R
2. `.Call`
   - Gives C level access directly to R types
   - Requires learning internal R storage types and memory allocation
   - Much harder and places more responsibility on coder
   - Useful if need to manipulate/create native R objects and types which have no natural C analogue

.C -vs- .Call

Technically, `.Call` has certain advantages over `.C`

- Less copying of arguments
- Ability to dynamically dimension results in C
- Access to R data types and easier ability to execute R code from C (i.e. reverse)
- Ability to handle NAs etc

But, if these aren't going to be issues then `.C` may be a lot less headache.

Only time to cover `.C` today, but once that's mastered, documentation for `.Call` straightforward.

.C - The Basics - "Hello world!" Example

Starting off in C:

helloworld.c

```
#include <R.h>

void helloworld() {
   Rprintf("Hello world!\n");
}
```

Then, from the command line when in the directory containing helloworld.c

Command Line

```
R CMD SHLIB helloworld.c
```

Which will create the file helloworld.so (Mac/Linux) or helloworld.dll (Windows)

.C - The Basics - "Hello world!" Example

Then in R,

R

```
> dyn.load("/path/to/helloworld.so")
> x = .C("helloworld")
Hello world!
```


.C - The Basics - Simple Example

Starting off in C:

addnums.c

```
#include <R.h>

void addnums(double *a, double *b, double *res) {
   *res = *a + *b;
}
```

Then, from the command line when in the directory containing helloworld.c

Command Line

```
R CMD SHLIB addnums.c
```

Which will create the file addnums.so (Mac/Linux) or addnums.dll (Windows)

.C - The Basics - Simple Example

R

```
> dyn.load("/path/to/addnums.so")
> n = 3.141
> m = 2.718
> x = .C("addnums", as.double(n), as.double(m),
         as.double(0))
> x
[[1]]
[1] 3.141

[[2]]
[1] 2.718

[[3]]
[1] 5.859
```


.C - The Basics - Simple Example

R

```
> dyn.load("/path/to/addnums.so")
> n = 3.141
> m = 2.718
> x = .C("addnums", a=as.double(n), b=as.double(m),
         res=as.double(0))
> x$a
[1] 3.141
> x$res
[1] 5.859
```


.C - The Basics - Summary

- Your C function must have no return value (`void`) and all arguments passed as pointers
  - include `R.h` for access to various R functions
  - use `Rprintf()`, not `printf()` for output
- Compile using `R CMD SHLIB file.c`
- Load the resulting shared library in R with `dyn.load()`
- Call the C function using `.C`
  - first argument is string containing C function name
  - subsequent arguments are the *ordered* list of arguments to the C function (*no name matching occurs*)
  - *explicitly cast all* these R variables to the correct C type (`as.double()`, `as.integer()`, `as.character()`)
  - you will be returned a list with item names matching the argument names you passed in
  - the returned items will be *copies* – although C is passed pointers you can't overwrite anything in the R environment

.C - Vectors/Matrices

Vectors and matrices require care:

- cast to double – will be flat 1D arrays in C
- pass the dimension of the vector/matrix explicitly!
- matrices are stored column-wise (quite standard in C) and the default for `matrix()` function in R
- the return will likewise be a flat array which must be re-formed into a matrix

sqmatmul

See handout lines ???

R

See handout lines ???

Don't do this!! Will return to BLAS/LAPACK soon

.C - Distributions & Random Number Generation

To access the distribution calculations and random number generators available in R is a straight-forward C call.

The procedure is:

- include the header file `Rmath.h`
- before the first generation of a random number, call `GetRNGstate();`
- to find function, see page 106 in *"Writing R Extensions"*. Note:
  - *all* arguments are double (even df etc)
  - there is no `n` for random number generation
- make appropriate calls in C
- after all random number generation and before function exits, call `PutRNGstate();`

.C - Distributions & Random Number Generation

Revisit the 2D random walk

2drw

See handout lines ???

R

See handout lines ???

Aside: when we increase the number of steps to 1,000,000 we find the fastest native R version earlier takes 0.3 sec and this C version takes 0.15 sec.

.C - Using BLAS/LAPACK

Don't implement *any* vector/matrix algebra if you don't have to: rely on BLAS & LAPACK.
Defy anyone to beat it in 99.9% of general cases!

References:

- BLAS Quick Reference (www.netlib.org/blas/blasqr.pdf)
- BLAS website (www.netlib.org/blas)
- `R_ext/BLAS.h` and `R_ext/Lapack.h`
- LAPACK website (www.netlib.org/lapack and www.netlib.org/lapack/double for example)

Usage:

- Include `<R_ext/BLAS.h>` and `<R_ext/Lapack.h>` in your C
- Lookup required function
- Call by wrapping Fortran call, *e.g.*: `F77_CALL(dgeevx)(...);`

.C - Using BLAS/LAPACK

Beware! Everything is passed as pointers and some BLAS/LAPACK routines will change your data. However, behaviour is very well documented.

linmod

See handout lines ???

R

See handout lines ???

.C - Miscellaneous Extras

Arguably the most useful 6 pages of the *"Writing R Extensions"* manual are pages 106-112. Cover the stable APIs for such things as:

- Mathematical functions (§6.7.2, p.107)
  - gamma, di/tri/tetra/... gamma functions
  - beta function
  - ${}^nC_k$, ${}^nP_k$
  - bessel functions
- Numerical Utilities (§6.7.3, p.107-8)
  - efficient powers (recall, no `x^n` in C!)
  - accurate $\log(1+x)$, $\log(1+x)-x$, e^x-1, ... for $|x| \ll 1$
  - and more
- Mathematical Constants (§6.7.4, p.108-9) to 30 decimal places $\pi, \frac{\pi}{2}, \frac{1}{\pi}, \sqrt{\pi}, e, \ldots$
- Optimization (§6.8, p.109-10) for direct C-level access to `optim` for Nelder Mead, BFGS, simulated annealing, ...

.C - Miscellaneous Extras

- Integration (§6.9, p.110-11) for direct C-level access to finite and infinite support integrals by quadrature
- Utility functions (§6.10, p.111-12)
  - extensive sorting and searching
  - temporary file functions
  - and more

Last 'extra' comment: you'll find hitting the Esc key won't interrupt anything running in C the way it does for things running in native R.

To overcome this annoyance, include the header `<R_ext/Utils.h>` and periodically call `R_CheckUserInterrupt();` at safe/suitable places in your code.

Going Even Faster – The Future

The current trend for extra speed is looking towards GPUs.

This means learning a new programming paradigm:

- NVIDIA CUDA (www.nvidia.com/cuda)
- ATI Stream (www.amd.com/stream)
- OpenCL (www.khronos.org/opencl and www.apple.com/macosx/technology)

Even this 3 year old MacBook Pro has 32 475MHz graphics cores $\Longrightarrow$ theoretical 15GHz untapped processing power.

The current MacBook Pro has 48 1265MHz graphics cores $\Longrightarrow$ theoretical 60GHz untapped processing power.

Current best desktop NVIDIA card has 480 700MHz graphics cores $\Longrightarrow$ theoretical 336GHz untapped processing power.