R Programming

Efficiency

Efficiency Considerations

In order to use R effectively, it is important to understand which operations are likely to be expensive and how the expense can be avoided.

The main expensive operations are:

- Boxing and unboxing
- Data copying
- Argument matching

Boxing and Unboxing

- Every value in R must be stored in a vector.
- This is true for values which look like scalars such as `TRUE`, `27.3` and `"hello"`.
- When you type a value like `27.3` to R, the result is that value stored in a vector of length 1.
- Operations which are specified elementwise on vectors incur overhead because the elements must be extracted into a vector of length 1, processed and then transferred back to a longer vector.

Data Copying

- When a value is passed into a function, R works hard to make it appear that it is a copy of the value which has been handed to the function.
- Sometimes it is possible to avoid the overhead of making a copy, but often it is not. Many of the copy operations which R performs are unnecessary because code which decides whether it is necessary to copy errs on the conservative side.

Argument Matching

- The presence of optional arguments to R functions means that function calls are relatively expensive. (Three passes are made over the argument list).
- Some functions (mainly the simple mathematical functions) avoid the overhead of argument processing.
- These functions are called "primitive" and can be recognised by their appearance.

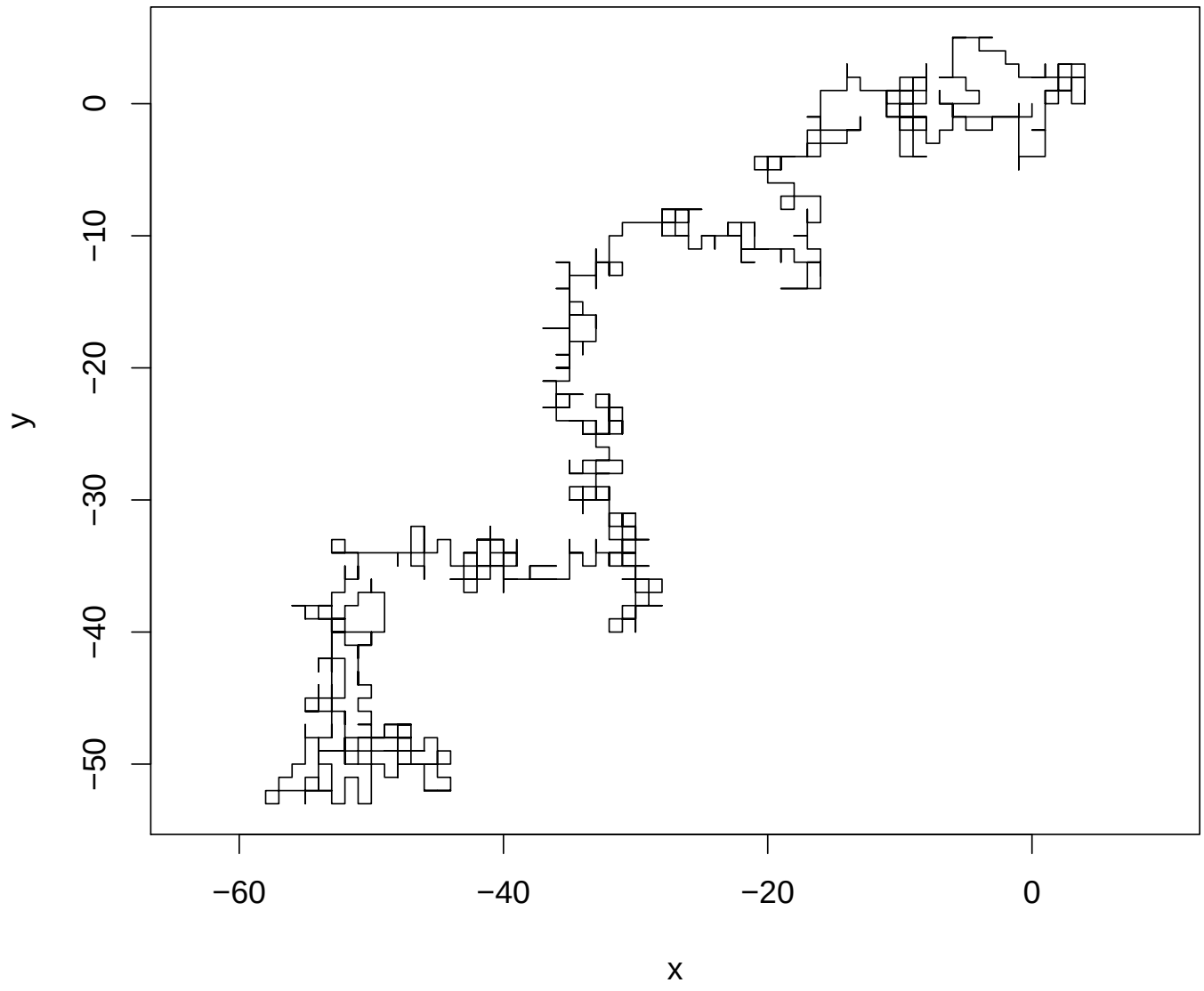
```
> sin
function (x)  .Primitive("sin")
```

Example: Generating a 2d Simple Random Walk

A two dimensional (discrete) random walk can be defined as follows:

- Start at the point $(0,0)$.
- For $i = 1,2,3,\ldots$ take a unit step in a randomly chosen direction; N, S, E, W.

It is possible to theory to study such a random walk, but it is also useful to use simulation to study the properties of random walks.


0
−10
−20
−30
−40
−50
y
−60
−40
−20
0
x

Questions about Random Walks

There are many questions which arise concerning two dimensional random walks. E.g.

1. What is the distribution of the number of steps taken before the walk returns to $(0,0)$?
2. What is the distribution of the number of steps taken before the walk encounters a point which has been visited before?
3. Does the random walk return to $(0,0)$ infinitely often or are there only a finite number of visits?

Some questions can be answered with some simple theory, some can not.

Simulation can be helpful when studying random processes like this.

Implementing a Random Walk Simulation

We'll use the problem of generating a random walk to show that different appoaches lead to a wide range of running times for the resulting program.

For big problems it can be important to find efficient ways of carrying out computations.

Often spending a little time studing alternative ways to implement a program can produce a huge savings in runtime.

Timing R Commands

The `system.time` function can be used to time how long it takes to evaluate an R expression.

```
> system.time(R-expression)
```

The value returned by `system.time` is a vector containing three values.

user	The time spent running user code.
system	The time spent in system calls (doing things like reading and writing files).
elapsed	The actual time it took for the expression to be evaluated.

It is usually the elapsed time which is important.

Version One — Naive Implementation

In this version we'll write the program the way a C, C++ or Java programmer might.

This means running a loop and generating the values one a time.

At the heart of the program we have to choose a direction (x or y) to step in and an stepsize (either $+1$ or -1).

These random choices are made using the `sample` function.

Version One — R Code

```
> rw2d1 =
    function(n) {
      x = y = numeric(n)
      xdir = c(TRUE, FALSE)
      step = c(1, -1)
      for(i in 2:n)
        if (sample(xdir, 1)) {
          x[i] = x[i-1] + sample(step, 1)
          y[i] = y[i-1]
        }
        else {
          x[i] = x[i-1]
          y[i] = y[i-1] + sample(step, 1)
        }
      list(x = x, y = y)
    }
```

Performance

We can time the performance of this algorithm using the `system.time` function.

```
> system.time(rw2d1(100000))
   user  system elapsed
  1.732   0.001   1.749
```

We'll use this figure as a baseline for comparison with other methods we'll develop later.

Version Two — Simplified Sampling

All the action in the initial version of the function goes on inside the loop.

The `sample` function is relatively complex and, because it might be expensive, let's replace the calls to it by something simpler.

A random `TRUE`/`FALSE` value can be generated as follows:

```
runif(1) > .5
```

and a random ± 1 can be generated with

```
2 * (runif(1) > .5) - 1
```

(Note the logical to numeric coercion here.)

Version Two — Code

```
> rw2d2 =
    function(n) {
      x = y = numeric(n)
      for(i in 2:n) {
        if (runif(1) > .5) {
          x[i] = x[i-1] + 2 * (runif(1) > .5) - 1
          y[i] = y[i-1]
        }
        else {
          x[i] = x[i-1]
          y[i] = y[i-1] + 2 * (runif(1) > .5) - 1
        }
      }
      list(x = x, y = y)
    }
```

Performance

Again, we judge performance using the `system.time` function.

```
> system.time(rw2d2(100000))
   user  system elapsed
  1.391   0.003   1.404
```

This is a 25% drop in elapsed time when compared with the baseline method.

Version Three — Vectorised Version

Given that R is designed to use vectorised operations it is natural to try replacing the loop in the function with vectorised operations.

Rather than computing the position element by element, this version computes the vectors of position changes and then uses `cumsum` to compute the positions.

To compute n positions we need $n - 1$ position changes.

The step sizes can be computed as

```
2 * (runif(n - 1) >  .5) - 1
```

and whether or not to step in the x direction can be determined as

```
runif(n - 1) >  .5
```

Version Three — Code

```
> rw2d3 =
    function(n) {
      xdir = runif(n - 1) >  .5
      step = 2 * (runif(n - 1) >  .5) - 1
      x = c(0, cumsum(ifelse(xdir, step, 0)))
      y = c(0, cumsum(ifelse(xdir, 0, step)))
      list(x = x, y = y)
  }
```

Performance

Again, we judge performance using the `system.time` function.

```
> system.time(rw2d3(100000))
   user  system elapsed
  0.090   0.009   0.100
```

This has cut the running time to $1/17$ of the baseline version.

Vectorisation clearly makes a huge difference to run times.

Version Four — Different Sampling Method

Previously, we changed our sampling method from using `sample` to using `runif`.

In the loop-based version this produced a saving because the cost associated with `sample` is mostly in the cost of making the call.

In the loop version of the function there are many calls to `sample` and this is expensive.

In the vectorised version there are only two calls to sample, so the effect of the overhead is very much reduced.

It may well be worth trying sample again.

Version Four — Code

```
> rw2d4 =
    function(n) {
      steps = sample(c(-1, 1), n - 1,
                     replace = TRUE)
      xdir = sample(c(TRUE, FALSE), n - 1,
                    replace = TRUE)
      x = c(0, cumsum(ifelse(xdir, steps, 0)))
      y = c(0, cumsum(ifelse(xdir, 0, steps)))
      list(x = x, y = y)
  }
```

Performance

Again, we judge performance using the `system.time` function.

```
> system.time(rw2d4(100000))
   user  system elapsed
  0.067   0.009   0.076
```

Here, using `sample` has gained a small drop in runtime.

Version Five — Heavy Vectorisation

A potential problem with the previous version is the use of the `ifelse` function to deal with the x and y directions separately.

As a final improvement let's deal with the four step directions separately and simply choose one of the four directions at random.

The directions can be chosen via

```
dirs = sample(1:4, n - 1, replace = TRUE)
```

and this can then be used to select the appropriate increments in the x and y directions from precomputed vectors.

Version Five — Code

```
> rw2d5 =
    function(n) {
      xsteps = c(-1,  1,  0,  0)
      ysteps = c( 0,  0, -1,  1)
      dir = sample(1:4, n - 1, replace = TRUE)
      x = c(0, cumsum(xsteps[dir]))
      y = c(0, cumsum(ysteps[dir]))
      list(x = x, y = y)
  }
```

Performance

Again, we judge performance using the `system.time` function.

```
> system.time(rw2d5(100000))
   user  system elapsed
  0.006   0.005   0.011
```

This has cut the running time to about $1/7$ of the previous version and $1/159$ of the baseline version.

Lessons

- Optimising R performance is a very dark art indeed.
  - Knowing about the detail of how functions work internally can be helpful but is not essential.
  - Experimentation with the code and timing the results with `system.time` can reduce run times by orders of magnitude.
- In general, vectorisation is a big win and converting loops into vectorised alternatives almost always pays off.
- Code profiling can give a way to locate those parts of a program which will benefit most from optimisation.

Profiling

Profiling is a useful tool which can be used to find out how much time is being spent inside each function when some R code is run.

When profiling is turned on, R gathers information on where the program is at regularly spaced time points (20 millisecond separation by default) and stores the information in a file.

After profiling is turned off the information stored in the file can be analysed to produce a summary of how much time is spent in each function.

It can be quite surprising to find out just where R is spending its time and this can help to find ways to make programs run faster.

Profiling Example

The following code will enable use to find out where R is spending its time when running the `rw2d4` function.

Because the process is statistical we'll run the function a number of times to ensure that enough data is being gathered.

```
> Rprof()
> for(i in 1:100)
      pos = rw2d4(100000)
> Rprof(NULL)
```

Profiling Analysis

```
> prof = summaryRprof()
> prof$by.self
          self.time self.pct total.time
"ifelse"       3.36    53.00       4.90
"sample"       1.00    15.77       1.00
"&"            0.92    14.51       0.92
"!"            0.56     8.83       0.56
"cumsum"       0.22     3.47       0.22
"list"         0.16     2.52       0.16
"c"            0.06     0.95       0.06
"is.na"        0.06     0.95       0.06
          total.pct
"ifelse"      77.29
"sample"      15.77
"&"           14.51
"!"            8.83
```

```
"cumsum"      3.47
"list"        2.52
"c"           0.95
"is.na"       0.95
```

More than 80% of the time is being spent in the `ifelse` function. This explains why removing the `ifelse` calls has such a big effect.