Writing Efficient Programs in R (and Beyond)

Ross Ihaka*, Duncan Temple Lang**, Brendan McArdle*

*The University of Auckland  
**The University of California, Davis
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.
The sample Function

A virtual lotto ticket line.

```r
> sample(1:40, 7)
[1]  34  15   3  32   4  39  37
```

Sampling with replacement.

```r
> sample(1:40, 7, replace = TRUE)
[1]  13  21  23  39  23   6  38
```

Sample random directions and step sizes.

```r
> sample(c(TRUE, FALSE), 1)
[1] TRUE
> sample(c(-1, 1), 1)
[1] -1
```
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 step-size (either $+1$ or $-1$).

These random choices are made using the `sample` function.
> rw2d1 =
function(n) {
  xpos = ypos = numeric(n)
  xdir = c(TRUE, FALSE)
  pm1 = c(1, -1)
  for(i in 2:n)
    if (sample(xdir, 1)) {
      xpos[i] = xpos[i-1] + sample(pm1, 1)
      ypos[i] = ypos[i-1]
    }
    else {
      xpos[i] = xpos[i-1]
      ypos[i] = ypos[i-1] + sample(pm1, 1)
    }
  list(x = xpos, y = ypos)
}
Performance

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

```r
> system.time(rw2d1(100000))
    user  system elapsed
   2.587   0.002   2.591
```

We’ll use this figure as a baseline for comparison with other methods we’ll develop later.
Version Two: Vectorisation

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

$$\text{steps} = \text{sample}(c(-1, 1), n - 1, \text{replace} = \text{TRUE})$$

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

$$\text{xdir} = \text{sample}(c(\text{TRUE}, \text{FALSE}), n - 1, \text{replace} = \text{TRUE})$$
> rw2d2 =
  function(n) {
    steps = sample(c(-1, 1), n - 1,
                   replace = TRUE)
    xdir = sample(c(TRUE, FALSE), n - 1,
                  replace = TRUE)
    xpos = c(0, cumsum(ifelse(xdir, steps, 0)))
    ypos = c(0, cumsum(ifelse(xdir, 0, steps)))
    list(x = xpos, y = ypos)
  }
Version Two: R Code

```r
> rw2d2 =
function(n) {
  steps = sample(c(-1, 1), n - 1,
                 replace = TRUE)
  xdir = sample(c(TRUE, FALSE), n - 1,
                replace = TRUE)
  xpos = c(0, cumsum(ifelse(xdir, steps, 0)))
  ypos = c(0, cumsum(ifelse(xdir, 0, steps)))
  list(x = xpos, y = ypos)
}
> system.time(rw2d2(100000))
  user  system elapsed
 0.103  0.011  0.114

This is 1/23 the elapsed time taken by the baseline version.

Vectorisation clearly makes a huge difference to run times.
Version Three: 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

```r
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 Three: R Code

> rw2d3 =
function(n) {
  xsteps = c(-1, 1, 0, 0)
  ysteps = c(0, 0, -1, 1)
  dir = sample(1:4, n - 1, replace = TRUE)
  xpos = c(0, cumsum(xsteps[dir]))
  ypos = c(0, cumsum(ysteps[dir]))
  list(x = xpos, y = ypos)
}
Version Three: R Code

```r
> rw2d3 =
function(n) {
  xsteps = c(-1, 1, 0, 0)
  ysteps = c(0, 0, -1, 1)
  dir = sample(1:4, n - 1, replace = TRUE)
  xpos = c(0, cumsum(xsteps[dir]))
  ypos = c(0, cumsum(ysteps[dir]))
  list(x = xpos, y = ypos)
}
```

```r
> system.time(rw2d3(100000))
user  system elapsed
0.011 0.001 0.013
```

This has cut the running time to about 1/9 of the previous version and 1/200 of the baseline version.
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 \texttt{rw2d2} function.

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

\begin{verbatim}
> Rprof()
> for(i in 1:100)
>     pos = rw2d2(100000)
> Rprof(NULL)
\end{verbatim}
Profiling Analysis

> prof = summaryRprof()
> prof$by.self[1:5,]

<table>
<thead>
<tr>
<th></th>
<th>self.time</th>
<th>self.pct</th>
<th>total.time</th>
<th>total.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ifelse&quot;</td>
<td>5.08</td>
<td>52.2</td>
<td>7.94</td>
<td>81.5</td>
</tr>
<tr>
<td>&quot;&amp;&quot;</td>
<td>1.30</td>
<td>13.3</td>
<td>1.30</td>
<td>13.3</td>
</tr>
<tr>
<td>&quot;sample&quot;</td>
<td>1.10</td>
<td>11.3</td>
<td>1.10</td>
<td>11.3</td>
</tr>
<tr>
<td>&quot;!&quot;</td>
<td>1.08</td>
<td>11.1</td>
<td>1.08</td>
<td>11.1</td>
</tr>
<tr>
<td>&quot;cumsum&quot;</td>
<td>0.34</td>
<td>3.5</td>
<td>0.34</td>
<td>3.5</td>
</tr>
</tbody>
</table>

81.5% of the time is being spent in the `ifelse` function (and calls made to other R functions from inside the `ifelse` function).

This explains why removing the `ifelse` calls has such a big effect.
Lessons

- Producing efficient programs in R requires thought and experimentation.
- 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.
- Unfortunately, it is not always possible to produce efficient programs using vectorisation.
Directions for New Research

- There are new high-level languages which produce very efficient code by using careful code analysis and transformation.
  - SaC — Single assignment C (University of Kiel)
  - CT — C for Throughput Computing (Intel)
- These languages are not interactive.
- Whether it is possible to bring the techniques used by these languages to an interactive languages is an open question.
- The other alternative is to try to make naively written programs run fast.
- How to do this in an interactive language is an open question.
A Quick Progress Report

• We believe that it is possible to make naively specified programs in a language not unlike R run much faster than R (up to 600 times faster for some problems).

• Integrating this with method-dispatch in object-oriented languages is tricky, but looks possible.

• This is not going to be enough to take advantage of the potential offered by the parallel processing architectures now becoming available.

• To harness that potential, the techniques used in languages like Sac and CT must be used.

• It is not clear whether this is possible in interactive languages.