Everyone knows that loops in R are to be avoided but vectorization is not always possible

It goes without saying that there are always many ways to solve a problem in R, but clearly some ways are better (for example, faster) than others. Recently, I found myself in a situation where I could not find a way to avoid using a loop, and I was immediately concerned, knowing that I would want this code to be flexible enough to run with a very large number of observations, possibly over many observations. Two tools immediately came to mind: data.table and Rcpp . This brief description explains the background of the simulation problem I was working on and walks through the evolution of ideas to address the problems I ran into when I tried to simulate a large number of individuals. In particular, when I tried to simulate a very large number of individuals, say over 1 million, running the simulation over night wasn't enough.

Setting up the problem

The task in question here is not the focus, but needs a little explanation to understand what is motivating the programming issue. I am conducting a series of simulations that involve generating an individual-level stochastic (Markov) process for any number of individuals. For the data generation, I am using the simstudy package developed to help facilitate simulated data. The functions `defDataAdd` and `genData` are both from `simstudy`. The first part of the simulation involves specifying the transition matrix `P` that determine a state I am calling `status`, and then defining the probability of an event that are based on a particular status level at a particular time point. For each individual, I generate 36 months of data and a status and event for each month.

```
library(data.table)
library(simstudy)

set.seed(123)
```

```
P <- matrix(c(0.985, 0.015, 0.000, 0.000,
              0.000, 0.950, 0.050, 0.000,
              0.000, 0.000, 0.850, 0.150,
              0.000, 0.000, 0.000, 1.000),
            nrow = 4, byrow = TRUE)

form <- "(status == 1) * 0.02 + (status == 2) * 0.10 + (status == 3) * 0.

dtDef <- defDataAdd(varname = "event",
            formula = form,
            dist = "binary",
            link = "identity")

N = 5000
did <- genData(N)
```

In order to simulate the Markov process, I decided immediately that Rcpp would be most appropriate because I knew I could not avoid looping. Since each state of a Markov process depends on the state immediately preceding, states need to be generated sequentially, which means no obvious way to vectorize (if someone has figured that out, let me know.)

```
#include  < RcppArmadilloExtensions/sample.h >;

// [[Rcpp::depends(RcppArmadillo)]]

using namespace Rcpp;

// [[Rcpp::export]]

IntegerVector MCsim( unsigned int nMonths, NumericMatrix P,
                     int startStatus, unsigned int startMonth ) {

  IntegerVector sim( nMonths );
  IntegerVector m( P.ncol());
  NumericVector currentP;
  IntegerVector newstate;

  unsigned int q = P.ncol();

  m = Rcpp::seq(1, q);

  sim[startMonth - 1] = startStatus;

  for (unsigned int i = startMonth; i < nMonths; i++) {

    newstate = RcppArmadillo::sample(m, 1, TRUE, P.row(sim(i-1) - 1));
    sim(i) = newstate(0);

  }

  return sim;
}
```

The process is simulated for each individual using the **Rcpp** function `MCsim`, but is done in the context of a `data.table` statement. The key here is that each individual is processed separately through the `keyby = id` statement. This obviates the requirement to loop through individuals even though I still need to loop within individuals for the stochastic process. This algorithm is quite fast, even with very large numbers of individuals and large numbers of observations (in this case months) per individual.

```
dt <- did[, .(status = MCsim(36, P, 1, 1)),
          keyby = id]
dt[, month := 1 : .N, keyby = id]
dt <- addColumns(dtDefs = dtDef, dtOld = dt)

dt
```

```
##            id status month event
##      1:    1      1     1     0
##      2:    1      1     2     0
##      3:    1      1     3     0
##      4:    1      1     4     0
##      5:    1      1     5     0
##     ---
## 179996: 5000      4    32     0
## 179997: 5000      4    33     0
## 179998: 5000      4    34     0
## 179999: 5000      4    35     0
## 180000: 5000      4    36     0
```

This is where things begin to slow down

It is the next phase of the simulation that started to cause me problems. For the simulation, I need to assign individuals to a group or cohort which is defined by a month and is based on several factors: (1) whether an event occurred in that month, (2) whether the status of that individual in that month exceeded a value of `1`, and (3) whether or not the individual experienced 2 or more events in the prior 12 months. An indivdual might be eligible for more than one cohort, but will be assigned to the first possible cohort (i.e. the earliest month where all three criteria are met.) Again, the specifics of the simulation are not important here. What is important, is the notion that the problem requires looking through individual data sequentially, something R is generally not so good at when the sequences get particularly long, and they must be repeated a large number of times. My first, naïve, approach was to create an **R** function

that loops through all the individuals and loops within each individual until a cohort is found:

```
rAssignCohortID <- function(id, month, status,
                            event, nInds,
                            startMonth, thresholdNum) {

  cohort <- rep(0, length(id));

  for (j in (1 : nInds))  {

    idMonth <- month[id == j];
    idEvent <- event[id == j];
    idStatus <- status[id == j];

    endMonth <- length(idMonth);

    done <- FALSE;
    i <- max(startMonth - idMonth[1], 13);

    while (i <= endMonth & !done) {

      if (idEvent[i] == 1 & idStatus[i] > 1) {

        begin = i-12;
        end = i-1;

        sumED = sum(idEvent[begin:end]);

        if (sumED <= thresholdNum) {

          cohort[id == j] <- i - 1 + month[1];
          done = TRUE;
        }
      }
      i = i + 1;
    }
  }

  return(cohort);
}
```

```
system.time(dt[, cohort1 := rAssignCohortID(id, month, status, event,
                    nInds = N, startMonth = 13, thresholdNum = 2)])
```

```
##    user  system elapsed
##   10.92    1.81   12.89
```

Working through possible solutions

The naïve approach works, but can we do better? I thought **Rcpp** might be a solution, because we know that loops in C++ are much more efficient. However, things did not turn out so well after I translated the function into C++; in fact, they got a little worse.

```
#include < Rcpp.h >;

using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector cAssignCohortID( IntegerVector id,
                      IntegerVector month,
                      IntegerVector status,
                      IntegerVector event,
                      int nInds,
                      int startMonth,
                      int thresholdNum) {

  IntegerVector cohort(id.length(), 0);

  IntegerVector idMonth;
  IntegerVector idEvent;
  IntegerVector idStatus;

  for (int j = 0; j < nInds; j++) {

    idMonth = month[id == j+1];
    idEvent = event[id == j+1];
    idStatus = status[id == j+1];

    int endMonth = idMonth.length();
    int sumED;
    bool done = FALSE;
    int i = std::max(startMonth - idMonth(0), 12);
    int begin;
    int end;

    while (i < endMonth & !done) {

      if (idEvent(i) == 1 & idStatus(i) > 1) {

        begin = i-12;
        end = i-1;

        sumED = sum(idEvent[Rcpp::seq(begin, end)]);

        if (sumED >= thresholdNum) {
          cohort[id == j + 1] = i + month(0);
          done = TRUE;
        }
      }
      i += 1;
    }
  }

  return(cohort);
```

```
}
```

```
system.time(dt[, cohort2 := cAssignCohortID(id, month, status, event,
                    nInds = N, startMonth = 13, thresholdNum = 2)])
```

```
##  user  system elapsed
## 13.88   2.03   16.05
```

I know that the function `cAssignCohortID` bogs down not in the loop, but in each phase where I need to subset the data set to work on a single `id`. For example, I need to execute the statement `idMonth = month[id == j+1]` for each `id`, and this apparently uses a lot of resources. I tried variations on this theme, alternatives to subset the data set within the Rcpp function, but could get no improvements. But a light bulb went off in my head (dim as it might be), telling me that this is one of the many things `data.table` is particularly good at. In fact, I used this trick earlier in generating the stochastic process data. So, rather than subsetting the data within the function, I created a regular R function that handles only a single individual `id` at a time, and let `data.table` do the hard work of splitting up the data set to process by individual. As you can see, things got markedly faster.

```
rAssignCohort <- function(id, month, status, event,
                    nInds, startMonth, thresholdNum) {

  cohort <- 0

  endMonth = length(month);

  done = FALSE;
  i = max(startMonth - month[1], 13);

  while (i <= endMonth & !done) {

    if (event[i] == 1 & status[i] > 1) {

      begin = i-12;
      end = i-1;

      sumED = sum(event[begin:end]);

      if (sumED >= thresholdNum) {

        cohort <- i - 1 + month[1];
        done = TRUE;
      }
```

```
    }
    i = i + 1;
  }

  return(cohort)
}
```

```
system.time(dt[, cohort3 := rAssignCohort(id, month, status, event,
                    nInds = N, startMonth = 13, thresholdNum = 2),
                keyby=id])
```

```
##    user  system elapsed
##     0.2     0.0     0.2
```

Finally, it occurred to me that an `Rcpp` function that is not required to subset the data might offer more yet improvements in speed. So, for the last iteration, I combined the strengths of looping in `Rcpp` with the strengths of subsetting in `data.table` to create a formidable combination. (Even when sample sizes exceed 1 million, the data are generated in a flash.)

```
#include < Rcpp.h >;

using namespace Rcpp;

// [[Rcpp::export]]
int cAssignCohort( IntegerVector month,
                   IntegerVector status,
                   IntegerVector event,
                   int startMonth, int thresholdNum) {

  int endMonth = month.length();
  int sumED;
  int cohort = 0;
  bool done = FALSE;
  int i = std::max(startMonth - month(0), 12);
  int begin;
  int end;

  while (i < endMonth & !done) {

    if (event(i) == 1 & status(i) > 1) {

      begin = i-12;
      end = i-1;

      sumED = sum(event[Rcpp::seq(begin, end)]);
```

```
      if (sumED >= thresholdNum) {
        cohort = i + month(0);
        done = TRUE;
      }
    }
    i += 1;
  }
  return(cohort);
}
```

```
system.time(dt[, cohort4 := cAssignCohort(month, status, event,
                                startMonth=13,  thresholdNum = 2), keyby=i
```

```
##    user  system elapsed
##    0.01    0.00    0.01
```

For a more robust comparison, let's use the `benchmark` function in package `rbenchmark`, and you can see how well `data.table` performs and how much `Rcpp` can add when used efficiently.

```
library(rbenchmark)

benchmark(
  dt[, cohort1 := rAssignCohortID(id, month, status, event,       # Naïve
                      nInds = N, startMonth = 13, thresholdNum = 2)],
  dt[, cohort2 := cAssignCohortID(id, month, status, event,        # Rcpp
                      nInds = N, startMonth = 13, thresholdNum = 2)],
  dt[, cohort3 := rAssignCohort(id, month, status, event,    # data.table
                      nInds = N, startMonth = 13, thresholdNum = 2), keyb
  dt[, cohort4 := cAssignCohort(month, status, event,   # combined data.t
                      startMonth=13,  thresholdNum = 2), keyby=id],
    replications = 5,
    columns = c("replications", "elapsed", "relative"))
```

```
##   replications elapsed relative
## 1            5   46.18    461.8
## 2            5   68.01    680.1
## 3            5    0.96      9.6
## 4            5    0.10      1.0
```

Postscript

I shared all of this with the incredibly helpful folks who have created data.table, and they offered a data.table only solution that avoids all looping, which I will share here for completeness. While it is an improvement over the third approach presented above (R function with data.table statment `keyby`), it is still no match for the fastest solution. (But, this all just goes to show you there will always be new approaches to consider, and I don't claim to have come any where near to trying them all out.)

```
dtfunc <- function(dx) {

  dx[, prev12 := Reduce(`+`, shift(event, 1:12)), by=id]
  map <- CJ(id=1:N, start=13L, end=36L, event=1L, statusx=1L, prev12x=1L)
  ans <- dx[map, on=.(id, event, status > statusx, prev12 > prev12x, mont
            .I, allow=TRUE, by=.EACHI, nomatch=0L][, .(id, I)]
  minans <- ans[, .(I=min(I)), by=id]

  dx <- dx[, cohort5 := 0L][minans, cohort5 := min(month) - 1L + dx$month

  return(dx)
}

system.time(dtfunc(dt))
```

```
##    user  system elapsed
##    0.18    0.00    0.17
```

And here is a more complete comparison of the fastest version with this additional approach:

```
benchmark(
  dt[, cohort6 := cAssignCohort(month, status, event,   # combined data.t
                    startMonth=13,  thresholdNum = 2), keyby=id],
  dt2 <- dtfunc(dt),
  replications = 5,
  columns = c("replications", "elapsed", "relative"))
```

```
##   replications elapsed relative
## 1            5    0.10      1.0
## 2            5    0.85      8.5
```

RELATED

It can be easy to explore data generating mechanisms with the simstudy package
May 16, 2017
In "R"

Composing reproducible manuscripts using R Markdown
April 17, 2017
In "R"

R Function Call with Ellipsis Trap/Pitfall
April 2, 2017
In "R"

PUBLISHED BY

Keith Goldfeld
I am an Assistant Professor of Biostatistics in the Department of Population Health at the New York University School of Medicine. View all posts by Keith Goldfeld →

May 4, 2017 Keith Goldfeld

One thought on "Everyone knows that loops in R are to be avoided but vectorization is not always possible"

Dulani
May 4, 2017 at 11:50 pm

Thanks so much for sharing this. I have discussions with colleagues all the time about the best ways to optimize R code and I will be referencing this post frequently.

The data.table "only" solution is interesting, but that syntax practically inscrutable!