Stockholm R useR Group: Microsimulation

Mark Clements

Department of Medical Epidemiology and Biostatistics,
Karolinska Institutet

2013-02-04

Who am I?

- Lektor in Biostatistics at Karolinska Institutet (colleagues with Alex Ploner)
- I did my undergraduate in statistics at the Department of Auckland in New Zealand (where R was developed)
- R user since 1998

Who am I?

- Lektor in Biostatistics at Karolinska Institutet (colleagues with Alex Ploner)
- I did my undergraduate in statistics at the Department of Auckland in New Zealand (where R was developed)
- R user since 1998
- Currently using microsimulation to model prostate cancer screening for a large randomised controlled trial planned for Stockholm

Disclaimers

1. This is work in progress!
2. This is a health-centric (chronic diseases) view of microsimulation

I welcome comments during the presentation.

Table of Contents

Simulation: model taxonomy

Cross-classification by the following factors

- Group-level (G) versus individual-level (I)
- Markov (M) versus non-Markov (nM)
- Discrete time (DT) versus continuous time (CT)

Simulation: model taxonomy

Cross-classification by the following factors

- Group-level (G) versus individual-level (I)
- Markov (M) versus non-Markov (nM)
- Discrete time (DT) versus continuous time (CT)

Some examples

- (G,M,CT) =

Simulation: model taxonomy

Cross-classification by the following factors

- Group-level (G) versus individual-level (I)
- Markov (M) versus non-Markov (nM)
- Discrete time (DT) versus continuous time (CT)

Some examples

- (G,M,CT) = System dynamics (deSolve package on CRAN; see also the Differential Equations TaskView on CRAN)
- (G,M,DT) =

Simulation: model taxonomy

Cross-classification by the following factors

- Group-level (G) versus individual-level (I)
- Markov (M) versus non-Markov (nM)
- Discrete time (DT) versus continuous time (CT)

Some examples

- (G,M,CT) = System dynamics (deSolve package on CRAN; see also the Differential Equations TaskView on CRAN)
- (G,M,DT) = Markov chains (HiddenMarkov, pomp, nnet::multinom())
- (I,nM,DT) =

Simulation: model taxonomy

Cross-classification by the following factors

- Group-level (G) versus individual-level (I)
- Markov (M) versus non-Markov (nM)
- Discrete time (DT) versus continuous time (CT)

Some examples

- (G,M,CT) = System dynamics (deSolve package on CRAN; see also the Differential Equations TaskView on CRAN)
- (G,M,DT) = Markov chains (HiddenMarkov, pomp, nnet::multinom())
- (I,nM,DT) = Agent-based simulation
- (I,nM,CT) =

Simulation: model taxonomy

Cross-classification by the following factors

- Group-level (G) versus individual-level (I)
- Markov (M) versus non-Markov (nM)
- Discrete time (DT) versus continuous time (CT)

Some examples

- (G,M,CT) = System dynamics (deSolve package on CRAN; see also the Differential Equations TaskView on CRAN)
- (G,M,DT) = Markov chains (HiddenMarkov, pomp, nnet::multinom())
- (I,nM,DT) = Agent-based simulation
- (I,nM,CT) = Discrete event simulation

What is Microsimulation?

What is Microsimulation?

Wikipedia:

- The International Microsimulation Association defines microsimulation as a modelling technique that operates at the level of individual units such as persons, households, vehicles or firms.

What is Microsimulation?

Wikipedia:

- The International Microsimulation Association defines microsimulation as a modelling technique that operates at the level of individual units such as persons, households, vehicles or firms.
- In health sciences, microsimulation refers to a type of simulation modeling that generates individual life histories.
  - The technique is used when 'stock-and-flow' type modeling of proportions (macrosimulation) of the population cannot sufficiently describe the system of interest.
  - This type of modeling does not necessarily involve interaction between individuals and in that case can generate individuals independently of each other, and can easily work with continuous time instead of discrete time steps.

Microsimulation in health: Two common approaches

1. Discrete time, Markov
   - Popular with health economists, who like their Markov models (e.g. TreeAge software)
   - Alex Ploner is working on such a framework for modelling cervical cancer, with model specification and post-processing in R and the simulation engine in C
   - The TraMineR package is very useful for visualising discrete time life histories

Microsimulation in health: Two common approaches

1. Discrete time, Markov
   - Popular with health economists, who like their Markov models (e.g. TreeAge software)
   - Alex Ploner is working on such a framework for modelling cervical cancer, with model specification and post-processing in R and the simulation engine in C
   - The TraMineR package is very useful for visualising discrete time life histories
2. Continuous time, non-Markov
   - Most generally implemented as a discrete event simulation
   - We are working on an R and C++ implementation for prostate cancer

What is discrete event simulation?

Conceptually, we have an event queue which is ordered by event times, where events are defined by their type and time[1].

[1] Law AM. (2007) *Simulation Modeling and Analysis*. Fourth edition. New York: McGraw-Hill.

What is discrete event simulation?

Conceptually, we have an event queue which is ordered by event times, where events are defined by their type and time[1].
The algorithm is:

1. Initialise an empty event queue; insert initial events into the queue

[1] Law AM. (2007) *Simulation Modeling and Analysis*. Fourth edition. New York: McGraw-Hill.

What is discrete event simulation?

Conceptually, we have an event queue which is ordered by event times, where events are defined by their type and time[1].
The algorithm is:

1. Initialise an empty event queue; insert initial events into the queue
2. Repeat until the queue is empty:
   (i) Retrieve the event at the head of the queue
   (ii) Process the event, possibly updating any variables, or insert/delete events from the queue

[1] Law AM. (2007) *Simulation Modeling and Analysis*. Fourth edition. New York: McGraw-Hill.

What is discrete event simulation?

Conceptually, we have an event queue which is ordered by event times, where events are defined by their type and time[1].
The algorithm is:

1. Initialise an empty event queue; insert initial events into the queue
2. Repeat until the queue is empty:
   (i) Retrieve the event at the head of the queue
   (ii) Process the event, possibly updating any variables, or insert/delete events from the queue
3. Finalise the simulation (e.g. return statistics).

[1] Law AM. (2007) *Simulation Modeling and Analysis*. Fourth edition. New York: McGraw-Hill.

Discrete event simulation: Software

- Proprietary: Arena, Extend, GPSS, SIMSCRIPT, etc.
- Open source: NS-2, Omnet++, Simpy, SSJ, etc.
- R: Only one simple example by Norm Matlof (`http://www.esg.montana.edu/R/revent.pdf`)

Table of Contents

R: Event queue (example)

R

```
> eq <- EventQueue$new() # R5 class (to be defined)
> eq$insert(3, "Clear drains")
> eq$insert(1, "Solve RC tasks")
> eq$insert(2, "Tax return")
> while(!eq$empty()) {
+   print(eq$pop())
+ }

[1] "Solve RC tasks"
attr(,"time")
[1] 1
[1] "Tax return"
attr(,"time")
[1] 2
[1] "Clear drains"
attr(,"time")
[1] 3
```

R: Event queue (using a closure)

In computer science, this is a priority queue or a heap.

```r
EventQueue <- function() {
  times <- events <- NULL
  insert <- function(time, event) {
    ord <- order(newtimes <- c(times, time))
    times <<- newtimes[ord]
    events <<- c(events, list(event))[ord]
  }
  pop <- function() {
    head <- structure(events[[1]], time=times[1])
    events <<- events[-1]
    times <<- times[-1]
    return(head)
  }
  empty <- function() length(events) == 0
  list(insert = insert, pop = pop, empty = empty)
}
```

R: Event queue (using a closure)

In computer science, this is a priority queue or a heap.

```r
EventQueue <- function() {
  times <- events <- NULL
  insert <- function(time, event) {
    ord <- order(newtimes <- c(times, time))
    times <<- newtimes[ord]
    events <<- c(events, list(event))[ord]
  }
  pop <- function() {
    head <- structure(events[[1]], time=times[1])
    events <<- events[-1]
    times <<- times[-1]
    return(head)
  }
  empty <- function() length(events) == 0
  list(insert = insert, pop = pop, empty = empty)
}
```

R: Event queue (using a closure)

In computer science, this is a priority queue or a heap.

```r
EventQueue <- function() {
  times <- events <- NULL
  insert <- function(time, event) {
    ord <- order(newtimes <- c(times, time))
    times <<- newtimes[ord]
    events <<- c(events, list(event))[ord]
  }
  pop <- function() {
    head <- structure(events[[1]], time=times[1])
    events <<- events[-1]
    times <<- times[-1]
    return(head)
  }
  empty <- function() length(events) == 0
  list(insert = insert, pop = pop, empty = empty)
}
```

R: Event queue (using a closure)

In computer science, this is a priority queue or a heap.

```
EventQueue <- function() {
  times <- events <- NULL
  insert <- function(time, event) {
    ord <- order(newtimes <- c(times, time))
    times <<- newtimes[ord]
    events <<- c(events, list(event))[ord]
  }
  pop <- function() {
    head <- structure(events[[1]], time=times[1])
    events <<- events[-1]
    times <<- times[-1]
    return(head)
  }
  empty <- function() length(events) == 0
  list(insert = insert, pop = pop, empty = empty)
}
```

R: Event queue (using an R5 class)

```
EventQueue <-
  setRefClass("EventQueue",
              fields = list(times = "numeric", events = "list"),
              methods = list(
                insert = function(time,event) {
                  ord <- order(newtimes <- c(times,time))
                  times <<- newtimes[ord]
                  events <<- c(events,list(event))[ord]
                },
                pop = function() {
                  head <- structure(events[[1]], time=times[1])
                  times <<- times[-1]
                  events <<- events[-1]
                  return(head)
                },
                empty = function() length(times) == 0
                ))
```

Comments

- R5 classes and closures look very similar!
- Closures provide a simple approach to working with fields and methods
- R5 classes allow for inheritance (see next), but they are slow.

R: Running the simulation

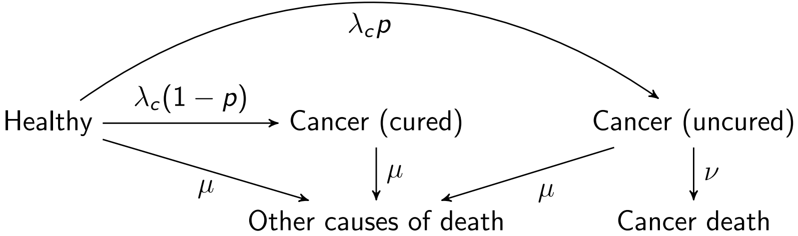
R

```
> set.seed(123)
> sim <- Simulation$new()
> sim$run()

[1] "Cancer diagnosis"
attr(,"time")
[1] 55.76424
[1] "Cancer death"
attr(,"time")
[1] 59.29142

> sim$run()

[1] "Death due to other causes"
attr(,"time")
[1] 59.96818
```

Outline of the BaseDiscreteEventSimulation class

- Inherit from the EventQueue class (with methods: insert, pop and empty)
- Define init() to set up the initial events
- Schedule events using scheduleAt(time, event), where event can be any object (e.g. a list or a character string)
- Define handleMessage(event) to respond to different events, possibly scheduling other events or clear()ing the queue
- Define final() to finish the simulation (if required)
- After the model is defined, run() the simulation

Simulation: Concrete example



where $p = 0.5$, and μ, λ_c and μ are rates for Weibull distributions. In practise for competing risks, we sample from the event time distributions and take the first event.

: Concrete class example

```
Simulation <-
  setRefClass("Simulation",
              contains = "BaseDiscreteEventSimulation")
Simulation$methods(init = function() {
  clear()
  scheduleAt(rweibull(1,8,85), "Death due to other causes")
  scheduleAt(rweibull(1,3,90), "Cancer diagnosis")
})
Simulation$methods(handleMessage = function(event) {
  now <- attr(event, "time")
  if (event %in% c("Death due to other causes", "Cancer death")) {
    clear()
    print(event)
  }
  else if (event == "Cancer diagnosis") {
    if (runif(1) < 0.5)
      scheduleAt(now + rweibull(1,2,10), "Cancer death")
    print(event)
  }
})
```

R: Concrete class example

```r
Simulation <-
  setRefClass("Simulation",
              contains = "BaseDiscreteEventSimulation")
Simulation$methods(init = function() {
  clear()
  scheduleAt(rweibull(1,8,85), "Death due to other causes")
  scheduleAt(rweibull(1,3,90), "Cancer diagnosis")
})
Simulation$methods(handleMessage = function(event) {
  now <- attr(event, "time")
  if (event %in% c("Death due to other causes", "Cancer death")) {
    clear()
    print(event)
  }
  else if (event == "Cancer diagnosis") {
    if (runif(1) < 0.5)
      scheduleAt(now + rweibull(1,2,10), "Cancer death")
    print(event)
  }
})
```

R: Concrete class example

```r
Simulation <-
  setRefClass("Simulation",
              contains = "BaseDiscreteEventSimulation")
Simulation$methods(init = function() {
  clear()
  scheduleAt(rweibull(1,8,85), "Death due to other causes")
  scheduleAt(rweibull(1,3,90), "Cancer diagnosis")
})
Simulation$methods(handleMessage = function(event) {
  now <- attr(event, "time")
  if (event %in% c("Death due to other causes", "Cancer death")) {
    clear()
    print(event)
  }
  else if (event == "Cancer diagnosis") {
    if (runif(1) < 0.5)
      scheduleAt(now + rweibull(1,2,10), "Cancer death")
    print(event)
  }
})
```

R: Concrete class example

```r
Simulation <-
  setRefClass("Simulation",
              contains = "BaseDiscreteEventSimulation")
Simulation$methods(init = function() {
  clear()
  scheduleAt(rweibull(1,8,85), "Death due to other causes")
  scheduleAt(rweibull(1,3,90), "Cancer diagnosis")
})
Simulation$methods(handleMessage = function(event) {
  now <- attr(event, "time")
  if (event %in% c("Death due to other causes", "Cancer death")) {
    clear()
    print(event)
  }
  else if (event == "Cancer diagnosis") {
    if (runif(1) < 0.5)
      scheduleAt(now + rweibull(1,2,10), "Cancer death")
    print(event)
  }
})
```

R: Discrete event simulation (R5)

```
BaseDiscreteEventSimulation <-
  setRefClass("BaseDiscreteEventSimulation",
              contains = "EventQueue",
              methods = list(
                clear = function() {
                  times <<- numeric()
                  events <<- list()
                  },
                scheduleAt = function(time, event) insert(time,
                    event),
                init = function() stop("VIRTUAL!"),
                handleMessage = function(event) stop("VIRTUAL!"),
                final = function() {},
                run = function() {
                  init()
                  while(!empty()) {
                    event <- pop()
                    handleMessage(event)
                  }
                  final()
                }))
```

Some class extensions

- Define fields for currentTime and previousEventTime
- Define a utility function now() for the current time
- Include a report field for returning statistics

R: Running the simulation

R

```
> set.seed(123)
> sim <- Simulation$new()
> system.time(for (i in 1:5000) sim$run())

   user  system elapsed
  14.96    0.02   15.03

> subset(sim$report,id<=4)

   id   state    begin      end                   event
1   1 Healthy  0.00000 55.76424        Cancer diagnosis
2   1  Cancer 55.76424 59.29142            Cancer death
3   2 Healthy  0.00000 59.96818 Death due to other causes
4   3 Healthy  0.00000 43.61622        Cancer diagnosis
5   3  Cancer 43.61622 80.36382 Death due to other causes
6   4 Healthy  0.00000 31.80345        Cancer diagnosis
7   4  Cancer 31.80345 38.04237            Cancer death
```

R: Concrete class example 2

```r
Simulation <-
  setRefClass("Simulation",
              contains = "BaseDiscreteEventSimulation2", # See Additional material
              fields = list(id = "numeric", state = "character", report = "data.frame"),
              methods = list(initialize = function(id = 0) callSuper(id = id)))
Simulation$methods(init = function() {
  clear()
  id <<- id + 1
  state <<- "Healthy"
  scheduleAt(rweibull(1,8,85), "Death due to other causes")
  scheduleAt(rweibull(1,3,90), "Cancer diagnosis")
})
```

R: Concrete class example 2

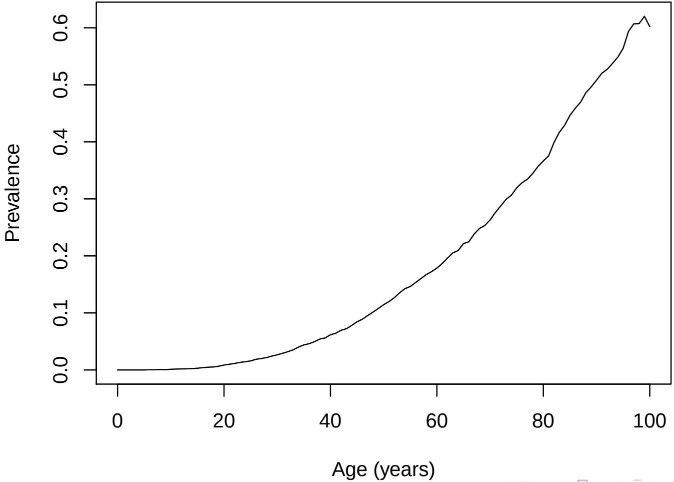
```
Simulation$methods(handleMessage = function(event) {
  report <<- rbind(report, data.frame(id = id,
                                      state = state,
                                      begin = previousEventTime,
                                      end = currentTime,
                                      event=event,
                                      stringsAsFactors = FALSE))
  if (event %in% c("Death due to other causes", "Cancer death")) {
    clear()
  }
  else if (event == "Cancer diagnosis") {
    state <<- "Cancer"
    if (runif(1) < 0.5)
      scheduleAt(now() + rweibull(1,2,10), "Cancer death")
  }
})
```

Life histories: calculating prevalence using SQL

R

```
> require(sqldf)
> report <- sim$report
> ages <- transform(data.frame(lhs = seq(0,100,1)),
+                   rhs = lhs + 1)
> prev <- sqldf("select t.*, a.lhs as age
+       from report as t
+       inner join ages as a on t.begin<=a.lhs and a.lhs<t.end")
> xtabs(~state+age, prev, subset = age %% 10 == 0)

         age
state        0   10   20   30   40   50   60   70   80   90  100
  Cancer     0    6   43  132  302  539  769  909  788  374   59
  Healthy 5000 4991 4947 4836 4589 4173 3536 2546 1362  362   39
```

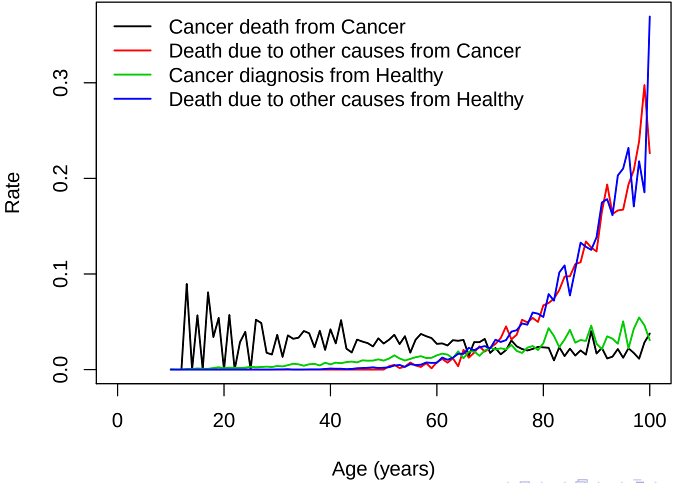
Life histories: Prevalence of cancer



Life histories: calculating rates using SQL

```
pt <- sqldf("select t.*, a.lhs as age, min(a.rhs,t.end)
    - max(a.lhs,t.begin) as pt, (a.lhs<=t.end and t.end<=
    a.rhs) as eventp from ages as a outer left join
    report as t on a.lhs<=t.end and a.rhs>t.begin")
rates <- sqldf("select event, state, age, events/
    persontime as rate from (select state, age, sum(pt)
    as persontime from pt group by state, age) natural
    join (select state, age, event, sum(eventp) as events
     from pt group by state, age, event) order by event,
    state, age")
```

Life histories: Rates



Microsimulation: Outline of tasks

- Define the microsimulation model
- For different scenarios:
  - Define the input parameters
    - Possibly initial histories based on observed individuals (e.g. a survey or registers)
    - Transition probabilities or time to event distributions
  - Run the microsimulation (in cancer, for 10^5 to 10^7 individuals)
  - Summarise the results

Microsimulation: Calibration/estimation

- Define
  - Microsimulation model
  - Fixed input parameters
  - Prior distributions for the uncertain input parameters
  - Calibration targets (i.e. data to fit)
- Sample from the posterior distribution
  1. Run the microsimulation (in cancer, for 10^5 to 10^7 individuals)
  2. Calculate the likelihood for the calibration targets
- Summarise the posterior distribution

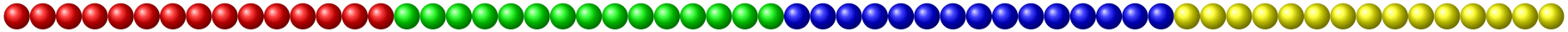
Table of Contents

Common random numbers and variance reduction

- For calibration/estimation and comparing scenarios in microsimulation, we want to reduce the Monte Carlo variation. Best practice advises the use of common random numbers
- The simplest approach for common random numbers is to have a different random seed for each individual
- A better approach is to have a different random seed for each individual for each "random process"

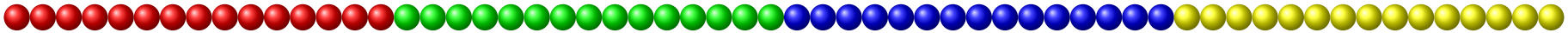
Random streams and sub-streams

Imagine that we have a random number generator that produces a long, independent series of random numbers:

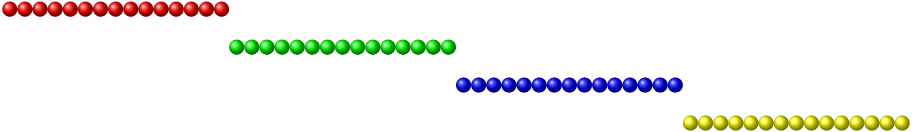


Random streams and sub-streams

Imagine that we have a random number generator that produces a long, independent series of random numbers:

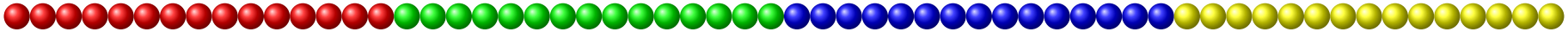


Now, we split this series into a set of streams:

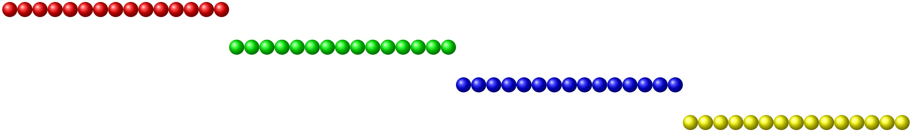


Random streams and sub-streams

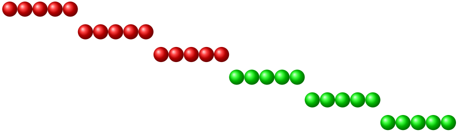
Imagine that we have a random number generator that produces a long, independent series of random numbers:



Now, we split this series into a set of streams:



And we can split the streams into a set of sub-streams:



Random streams and sub-streams: microsimulation

- Streams $\rightarrow$ random processes

Random streams and sub-streams: microsimulation

- Streams $\rightarrow$ random processes
- Sub-streams for a given stream $\rightarrow$ individuals

R: Random number streams

- Random number streams are implemented in the parallel core package (see also the rlecuyer, rstream and rsprng packages)
- parallel uses the `"L'Ecuyer-CMRG"` random number generator, which has a period of around 2^{191} (=3138550867693340381917894711603833208051177722232017256448; the default `"Mersenne-Twister"` RNG has a period of $2^{19937}-1$)
- Each stream is a subsequence of the period of length 2^{127} which is in turn divided into substreams of length 2^{76}
- The parallel package adapts and simplifies the RngStreams C package, losing some useful functionality

RNGStream object

- open, close and with methods for using an RNGStream object
- resetSubStream, resetStream, nextSubStream and nextStream methods for changing the object seed

R: Example of random number streams

R

```
> set.seed(101)
> s1 <- RNGStream(nextStream=FALSE)
> s2 <- RNGStream()
> with(s1,rnorm(1))

[1] 2.189189

> with(s2,rnorm(1))

[1] 0.5205894

> s1$nextSubStream()
> with(s1,rnorm(1))

[1] -1.461113
```

R

```
>
> s1$resetStream()
> s2$resetStream()
> with(s1,rnorm(2))

[1]  2.1891887 -0.6045821

> with(s2,rnorm(2))

[1] 0.5205894 0.8820710

> s1$nextSubStream()
> with(s1,rnorm(2))

[1] -1.4611126 -0.8230108
```

R: RNGStream object

```r
require(parallel)
RNGkind("L'Ecuyer-CMRG")
RNGStream <- function(nextStream = TRUE) {
  current <- if (nextStream) nextRNGStream(.Random.seed) else .
      Random.seed
  .Random.seed <<- startOfStream <- startOfSubStream <- current
  structure(list(open = function() .Random.seed <<- current,
                 close = function() current <<- .Random.seed,
                 resetSubStream = function() .Random.seed <<-
                     current <<- startOfSubStream,
                 resetStream = function() .Random.seed <<- current
                   <<- startOfSubStream <<- startOfStream,
                 nextSubStream = function() .Random.seed <<-
                     current <<- startOfSubStream <<-
                     nextRNGSubStream(startOfSubStream),
                 nextStream = function() .Random.seed <<- current
                   <<- startOfSubStream <<- startOfStream <<-
                     nextRNGStream(startOfStream)),
            class="RNGStream")
  }
```

R: with method for RNGStream object

```
with.RNGStream <- function(stream,expr,...) {
  stream$open()
  out <- eval(substitute(expr), enclos = parent.frame(), ...)
  stream$close()
  out
}
```

Table of Contents

Why C++?

- R is too slow!

Why C++?

- R is too slow!
- Good GPL'd libraries in C and C++ available for:
  - Discrete event simulation (SSIM)
  - Random number streams (RngStreams - of course!)
- But...

Why C++?

- R is too slow!
- Good GPL'd libraries in C and C++ available for:
  - Discrete event simulation (SSIM)
  - Random number streams (RngStreams - of course!)
- But. . .
  - We like R
  - We may miss R's dynamic typing, closures, extensive packages, etc.
  - We may not really want to program in C++

Entrez: Rcpp

- An elegant C++ framework for passing data and structures between R and C++
- Increasingly popular solution for dealing with large and slow computational tasks in
- We can now do all of our pre- and post-processing with , and use C++ as the engine

Good advice

Do everything for two reasons

R: Running the simulation

R

```
> require(microsimulation)

Loading required package: microsimulation
Loading required package: Rcpp

> set.seed(123)
> system.time(df <- callSimplePerson(100000))

   user  system elapsed
  0.688   0.016   0.705

> head(df)

   endtime         event id startTime   state
1 55.76424      toCancer  0   0.00000 Healthy
2 59.29142 toCancerDeath  0  55.76424  Cancer
3 59.96818  toOtherDeath  1   0.00000 Healthy
4 43.61622      toCancer  2   0.00000 Healthy
5 80.36382  toOtherDeath  2  43.61622  Cancer
6 31.80345      toCancer  3   0.00000 Healthy
```

Discussion

- Ongoing work
  - Methods for calibration (with Alexandra Jauhiainen)
  - Application to prostate cancer (with Hatef Darabi)
  - Statistics collection in C++
  - Visualisation; the Biograph package looks very interesting
- Issues and challenges
  - User-defined random number generators use one C function (`double *user_unif_rand ()`) → name conflict??
  - How to encourage R programmers to learn C++?
- The microsimulation package is available at https://github.com/mclements/microsimulation

Table of Contents

“Process-oriented” discrete event simulation

- The concept on the previous slide describes event-oriented simulation
- At a higher level, we can consider a process-oriented simulation, where we have a process that includes a series of events. The processes run as “continuations”.

R: Discrete event simulation 2

```
BaseDiscreteEventSimulation2 <-
  setRefClass("BaseDiscreteEventSimulation2",
              contains = "BaseDiscreteEventSimulation",
              fields = list(currentTime = "numeric",
                previousEventTime = "numeric"),
              methods = list(
                now = function() currentTime,
                run = function() {
                  previousEventTime <<- 0.0
                  init()
                  while(!empty()) {
                    event <- pop()
                    currentTime <<- attr(event, "time")
                    handleMessage(event)
                    previousEventTime <<- currentTime
                  }
                  final()
                }))
```

R: Common random numbers in a microsimulation

```
setOldClass("RNGStream")
Simulation <-
  setRefClass("Simulation",
              contains = "BaseDiscreteEventSimulation2",
              fields = list(id = "numeric", state = "character",
                 report = "data.frame", rng = "RNGStream"),
              methods= list(initialize = function(id = 0) {
                callSuper(id = id)
                rng <<- RNGStream(nextStream = FALSE)
              }))
Simulation$methods(init = function() {
  clear()
  id <<- id + 1
  state <<- "Healthy"
  scheduleAt(with(rng,rweibull(1,8,85)), "Death due to other
      causes")
  scheduleAt(with(rng,rweibull(1,3,90)), "Cancer diagnosis")
})
Simulation$methods(final = function() rng$nextSubStream())
```

R: Common random numbers in a microsimulation II

```r
Simulation$methods(handleMessage = function(event) {
  report <<- rbind(report, data.frame(id = id,
                                      state = state,
                                      begin = previousEventTime,
                                      end = currentTime,
                                      event=event,
                                      stringsAsFactors = FALSE))
  if (event %in% c("Death due to other causes", "Cancer death")) {
    clear()
  }
  else if (event == "Cancer diagnosis") {
    state <<- "Cancer"
    if (with(rng,runif(1)) < 0.5)
      scheduleAt(now() + with(rng,rweibull(1,2,10)), "Cancer death")
  }
})
```

C++: A simple microsimulation example (R code)

```
enum <- function(obj, labels)
  factor(obj, levels=0:(length(labels)-1), labels=labels)

callSimplePerson <- function(n=100) {
  stateT <- c("Healthy","Cancer","Death")
  eventT <- c("toOtherDeath", "toCancer", "toCancerDeath")
  out <- .Call("callSimplePerson",
               parms=list(n=as.integer(n)),
               PACKAGE="microsimulation")
  out <- transform(as.data.frame(out),
                   state=enum(state,stateT),
                   event=enum(event,eventT))
  out
}
```

C++: A simple microsimulation example

```
// include headers for microsimulation and Rcpp
#include "microsimulation.h"
#include <Rcpp.h>

using namespace std;

enum state_t {Healthy, Cancer, Death};

enum event_t {toOtherDeath, toCancer, toCancerDeath};

// for returning life histories
map<string, vector<double> > report;

#define Reporting(name,value) report[name].push_back(value);
```

C++: A simple microsimulation example

```
class SimplePerson : public cProcess
{
public:
  state_t state;
  int id;
  SimplePerson(const int i = 0) : id(i) {};
  void init();
  virtual void handleMessage(const cMessage* msg);
};

void SimplePerson::init() {
  state = Healthy;
  scheduleAt(R::rweibull(8.0,85.0),toOtherDeath);
  scheduleAt(R::rweibull(3.0,90.0),toCancer);
}
```

C++: A simple microsimulation example

```
void SimplePerson::handleMessage(const cMessage* msg) {
  double dwellTime, pDx;
  Reporting("id",id);
  Reporting("startTime",previousEventTime);
  Reporting("endtime", now());
  Reporting("state", state);
  Reporting("event", msg->kind);

  switch(msg->kind) {
  case toOtherDeath:
  case toCancerDeath:
    Sim::stop_simulation();
    break;
  case toCancer:
    state = Cancer;
    if (R::runif(0.0,1.0) < 0.5)
      scheduleAt(now() + R::rweibull(2.0,10.0), toCancerDeath);
    break;
  default:
    REprintf("No valid kind of event\n");
    break;
  }
}
```

C++: A simple microsimulation example

```
RcppExport SEXP callSimplePerson(SEXP parms) {
  SimplePerson person;
  Rcpp::RNGScope scope;
  Rcpp::List parmsl(parms);
  int n = Rcpp::as<int>(parmsl["n"]);
  for (int i = 0; i < n; i++) {
    person = SimplePerson(i);
    Sim::create_process(&person);
    Sim::run_simulation();
    Sim::clear();
  }
  return Rcpp::wrap(report);
}
```