PRACTICAL: LEARNING 'SIMMER'

K Degeling, M van de Ven, H Koffijberg

INTRODUCTION

This practical session will introduce the `simmer` package that will be used for defining and running discrete event simulations in R using RStudio. All `simmer` package related functions that will be used throughout the course will be introduced, explained, and illustrated. **It is important to read everything carefully, as all text contains important information. We strongly recommend you not to copy-and-paste any code from this description, but to write all lines of code yourself**.

Before introducing the functions that will be used, and before jumping into programming, it is important to take notice of the packages (and versions) that will be used throughout this and future practical assignment:

- `simmer` (version 4.3.0)
- `simmer.plot` (version 0.1.15)

The `simmer` package will be used to define and run discrete event simulation models. An overview of all functions that will be used from this package is provided in this tutorial. The `simmer.plot` package will be used to visualize the model structures defined using the `simmer` package. If you have a different version of one of these packages installed, please use the files provided in the `/practical_sessions/2_simmer/` folder to install the right package versions. For additional information about the `simmer` or `simmer.plot` package, or about other packages that can be used together with them, please visit https://r-simmer.org/. A previous presentation from Koen for R-HTA for provides a good primer for the `simmer` package, so that is recommended viewing before you start with this tutorial.

STEP 1: INITIALIZATION

The first step in any R script should be to initialize the document. For now, this includes two activities as illustrated in the code below:

1. Removing all objects from the work space, to ensure you're starting with a clean slate, which helps prevent errors when you run the code
2. Loading the required packages, being `simmer` and `simmer.plot` in this case

Time to start programming! Create a new R script in R Studio and write the initialization code.

```r
# CLEAR THE WORKSPACE
rm(list = ls()); gc();

# LOAD THE REQUIRED PACKAGES
library(simmer);
library(simmer.plot);
```

STEP 2: DEFINING THE MAIN OBJECTS

So far so good. Time to get into the `simmer` package. When defining and running discrete event simulations using the `simmer` package, two main objects are used: 1) the trajectory and 2) the simulation environment. Trajectories describe the pathway that is simulated, i.e., the structure of the model. The simulation environment defines the entities (e.g., patients) and resources (e.g., nurses and doctors) that will be simulated in the trajectories. In `simmer`, trajectories are defined by the `trajectory()` function, and simulation environments are defined by the `simmer()` function. By providing the defined trajectory object as an argument to the `plot()` function, the defined trajectory can be visualized using the `simmer.plot` package.

Step 2.1: Defining the trajectory

You will now first define a basic trajectory called `traj` and visualize it. The trajectory will resemble the pathway of patients who are being consulted by a doctor for 15 minutes. As illustrated in the code below, the following functions will be used to do so:

- `trajectory()` to initialize the trajectory
- `seize()` to seize a resource
- `timeout()` to implement a 'delay' (i.e., duration or time-to-event)
- `release()` to release a resource
- `%>%` to connect multiple functions (i.e., piping)

`trajectory()` The `trajectory()` function is used to initialize a trajectory through which entities (e.g., patients) can be simulated and resources (e.g., doctors) can be utilized. This function can be used together with several other functions, which will be introduced next, to further specify the trajectory. The most important argument to the `trajectory()` function itself is (see `?trajectory` for additional arguments):

- `name = "traj"` a character to provide the trajectory with a name.

seize() The `seize()` function is used to seize a resource. Although resources are not typically considered in health economic analyses, they can be of interest in such analyses and are an essential part of the application of discrete event simulation in other fields, such as operations research. Some key arguments to the function are (see `?seize` for additional arguments):

- `resource = ...` a character to define the resource that is to be seized.
- `amount = ...` a numerical to define the amount of resources to be seized.

timeout() The `timeout()` function is used to delay an entity while it is flowing through the trajectory. This function is used to implement so called 'time-to-events', such as the duration of a consultation or progression-free survival. The key argument to the function is (see `?timeout` for additional arguments):

- `task = ...` a numerical to define the amount the entity should be delayed. Please note that it does not require the unit of time to be defined, hence it is up to modeler to make sure delays are defined in the same unit of time throughout the trajectory.

release() The `release()` function is used to release a resource that previously has been seized using the `seize()` function. Hence, the key arguments to the function are the same as those for the `seize()` function (see `?release` for additional arguments):

- `resource = ...` a character to define the resource that is to be released.
- `amount = ...` a numerical to define the amount of resources to be released.

%>% The so-called 'pipe' operator `%>%` is used to connect multiple functions that are called to define a trajectory (or simulation as will be explained later). The `simmer` package uses it in a way that is similar to its use in the `tidyverse` package, for example.

In the example code below, a trajectory is defined in which entities seize a resource called `"doctor"` for `15` minutes. The `plot()` function is used to visualize the resulting trajectory structure. Move your mouse over the different blocks in the flowchart to see what information is provided.

```r
# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # SEIZE THE 'doctor' RESOURCE
  seize(resource = "doctor", amount = 1) %>%

  # DELAY THE 'patient' ENTITY FOR THE DURATION OF A CONSULATION
  timeout(task = 15) %>%

  # RELEASE THE 'doctor' RESOURCE
  release(resource = "doctor", amount = 1);

# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot
```

Step 2.2: Defining the simulation

After defining the basic trajectory, it is now time to initialize the simulation environment `sim`, and define the entities (i.e., patients) that will be simulated and resources (i.e., doctors) that will be utilized. The simulation environment will be defined so that `2` doctors are available to consult `100` patients. As illustrated in the code below, the following functions will be used to do so:

- `simmer()` to initialize the simulation environment
- `add_resource()` to define the resources
- `add_generator()` to define the entities
- `%>%` to connect multiple functions (i.e., piping)

simmer() The `simmer()` function is used to initialize a simulation environment. This function can be used together with several other functions, which will be introduced next, to define the entities and resources. The most important argument to the `simmer()` function itself is (see `?simmer` for additional arguments):

- `name = ...` a character to provide the simulation environment with a name

add_resource() The `add_resource()` function is used to define the resources for the simulation. Different types of resources can be defined in one simulation environment, by calling the function multiple times. The key arguments to the function are (see `?add_resource` for additional arguments, such as `queue_size`):

- `name = ...` a character to define the name of the resource (to be used in the `seize()` and `release()` functions)
- `capacity = ...` a numerical to define the amount of resources to be available in the simulation (`Inf` can be used to resemble the scenario in which unlimited resources are available, i.e. patients do not have to wait before being consulted)
- `mon = ...` a logical (`TRUE/FALSE`) to define whether the utilization of the resource needs to be monitored by the simulation

add_generator() The `add_generator()` function is used to define the entities that will be simulated. In most simulations that are performed to resemble a healthcare context, the entities will be hypothetical patients. The `add_generator()` function defines the number of entities and the time at which they enter the simulation. The key arguments to the function are (see `?add_generator` for additional arguments):

- `name_prefix = ...` a character to define the name that will be used to identify the entities (for example, the prefix "patient" will result in the generation of entities named "patient0", "patient1", "patient2", etc.)
- `trajectory = ...` an object in the global environment to define the trajectory in which the entities are to be simulated
- `distribution = ...` a function that determines the interarrival times. This determines the number of entities that are to be simulated and the time at which they enter the trajectory. By calling the `at()` function with a numerical vector, entities will be simulated at the times indicated by the vector. In the example below, the vector includes one hundred zero's (i.e., `c(0,0,0,...,0)`), so `100` entities will enter the trajectory at time zero. The `at()` function is used to turn the numerical vector into a function object that is required. Please note that the generator will stop once the function defined for the argument `distribution` returns a negative value.
- `mon = ...` a numerical to define the level at which the entities need to be monitored throughout the simulation. Although the use of attributes will be discussed later, the information provided in the help pane explains that three values can be provided: 0) no monitoring, 1) simple monitoring, and 2) detailed monitoring. A monitoring level of `2` tracks the attributes that are provided to the entities throughout the simulation and is critical in most simulations. Hence, the monitor level of entities is typically set to `2`.

```r
# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%

  # ADD 2 'doctor' RESOURCES TO THE ENVIRONMENT
  add_resource(name = "doctor", capacity = 2) %>%

  # ADD THE 'patient' ENTITIES
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 100)),
                mon = 2);
```

STEP 3: RUNNING THE FIRST SIMULATION

So far, we have defined a trajectory and a simulation environment. It is now time to run the first simulation of the course! Three functions will be discussed regarding running the simulation:

- `run()` to start a simulation
- `reset()` to reset a simulation environment
- `now()` to get the current simulation time

run() A simulation is performed using the `run()` function from the `simmer` package. As illustrated in the code below, this function can be called like a regular function or using the piping function `%>%`. Depending on how the `run()` function is being called, there are one or two important arguments (see `?run` for additional arguments):

- `.env` an object in the global environment to define the simulation environment that is to be simulated (if the piping function `%>%` is used, this argument is not to be provided, as the piping function already connects the function to the simulation environment).
- `until = ...` a numerical to define for how long the simulation needs to run (the default value `Inf` will make the simulation run until all entities have completed the whole trajectory, i.e. a terminating simulation).

reset() There are two additional important remarks with regard to running simulations using the `run()` function. First, it is important to always reset the simulation environment using the `reset()` function before running the simulation (this function also requires an `.env` argument when it is not connected to a simulation environment by a pipe). The reason for this is that the `run()` will not automatically run a new simulation from zero if it is being called, but runs the simulation until the specified time. Hence, if that time was already achieved in a previous simulation, nothing actually will be simulated. Second, although technically possible, it is better not to connect the `reset()` and `run()` function when initializing and defining the simulation (Step 2).

now() Another function that relates to the simulation environment, but can also be used within trajectories, is the `now()` function. This function gives the current time within the simulation, which is mainly useful for two purposes: 1) to get and use the current time within a trajectory, for example to determine the discount rate, and 2) to get the past simulation time after the simulation (unintendedly) stopped. When you print a simulation environment, you will also see the "now: ... " argument in the output, so it is not always necessary to separately call the `now()` function for the second purpose. Similar to the `reset()` function, the `now()` function requires an `.env = ...` argument when it is not connected to a simulation environment by a pipe.

Please take some time to understand the output below. How long did it take for all 100 patients to be seen by a doctor?

```r
# RUN THE SIMULATION USING THE 'reset()' AND 'run()' FUNCTIONS AS NORMAL FUNCTIONS
reset(.env = sim);
```

```
## simmer environment: sim | now: 0 | next: 0
## { Monitor: in memory }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 0 }
```

```r
run(.env = sim);
```

```
## simmer environment: sim | now: 750 | next:
## { Monitor: in memory }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 100 }
```

```r
# RUN THE SIMULATION BY PIPING THE 'reset()' AND 'run()' FUNCTIONS TO 'sim'
sim %>%
  reset();
```

```
## simmer environment: sim | now: 0 | next: 0
## { Monitor: in memory }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 0 }
```

```r
sim %>%
  run();
```

```
## simmer environment: sim | now: 750 | next:
## { Monitor: in memory }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 100 }
```

```r
# ALSO POSSIBLE, BUT LESS INFORMATIVE IN TERMS OF OUTPUT:
sim %>% reset() %>% run();
```

```
## simmer environment: sim | now: 750 | next:
## { Monitor: in memory }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 100 }
```

STEP 4: EXPANDING THE BASIC MODEL STRUCTURE

Now we have introduced the basic building blocks of trajectories and simulation environments to define and run a simple simulation study. Use these skills to expand the pathway that is being modeled and run a new simulation. Expand the trajectory and simulation environment in such a way that a `"nurse"` resource does an intake with all patients before they are being consulted by the `"doctor"`. Assume that a nurse takes `20` minutes to do the intake and that `3` nurses are available to perform this task.

Some tips to get you going:

- Use the code provided below as a starting point.
- Add comments wherever you think is appropriate.
- First expand the trajectory, while visualizing it after every change you make to verify your changes.
- You may also want to use the 'Show in new window' button to open the flowchart in a new window and be able to zoom in.
- Once you are confident your changes to the trajectory are correct, start adapting the simulation environment.

Based on the updated pathway, how long did it take for all 100 patients to be seen by a doctor?

```
# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  ... WRITE CODE HERE ...

  # DOCTOR CONSULTATION
  seize(resource = "doctor", amount = 1) %>%
  timeout(task = 15) %>%
  release(resource = "doctor", amount = 1);


# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot


# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%
  ... WRITE CODE HERE ...
  add_resource(name = "doctor", capacity = 2) %>%
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 100)),
                mon = 2);


# RUN THE SIMULATION
sim %>% reset() %>% run();
```

STEP 5: BRANCHING THE TRAJECTORY

An important aspect of defining trajectories is branching, which allows multiple different pathways to be reflected in one model structure. In the current context, we may assume that 20% of the patients do no longer need to see a doctor after they have been seen by a nurse, but of the patients who do see a doctor 40% need to see the doctor two times. To implement such a conditional pathway in which some patients are seen by a doctor and others are not, we will use the `branch()` and `rollback()` functions of the `simmer()` package within the trajectory.

branch() The `branch()` function has the following arguments (see `?branch` for further information):

- `option = ...` an integer number to define which trajectory in the branch the entity needs to go through. A value of `1` will direct the entity through the first trajectory, a value of `2` through the second trajectory, and so on. Importantly, a value of `0` will skip the branch. So in our case, if the patient does not need to see a doctor, a value of `0` should be returned, and thus the branch is skipped. Otherwise, a value of `1` is required so that the patient is directed to the first (and only) sub-trajectory in the branch. In the example below, the `sample(x = c(0, 1), size = 1, prob = c(0.20, 0.80))` statement returns a value of `0` with a 20% probability and a value of `1` with a 80% probability. The use of `function()` in the `option = ...` argument will be discussed later. You can also call an external function to provide the value, as will be illustrated.
- `continue = ...` a logical vector indicating for every trajectory in the branch whether the entity should continue beyond the branch after they completed the trajectory they were directed to. A `TRUE` value indicates that the entity should continue beyond the branch and a `FALSE` value indicates the entity should not continue after completing the trajectory in the branch. In our case, it does not matter whether `continue = ...` is `T` (i.e `TRUE`) or `F` (i.e. `FALSE`) because the overall trajectory stops after the branch.
- `...` the trajectories included in the branch, separated by commas `,`. The number of trajectories should be equal to the length of the logical vector provided to the `continue = ...` argument and to the numbers that can theoretically be returned by the function used for the `option = ...` argument. The trajectories can either be defined using the `trajectory()` function within the branch or by referring to a separately defined trajectory (both will be illustrated).

rollback() The `rollback()` function can be used to rmove an entity backward to a previous point in the trajectory, so it will repeat some or all actions. The `rollback()` function has the following arguments (see `?rollback` for further information):

- `amount = ...` an integer number indicating the number of activities (i.e., functions in the trajectory) to rollback.
- `times = ...` an integer number indicating how ofter the entity can rollback. `Inf` can be used in scenarios where there is no limit, or where the number of rollbacks is controlled by the `check = ...` argument or differently in the trajectory.
- `check = ...` a function that must return a logical value (i.e., boolean) that indicates whether a rollback is to be performed. If the `check = ...` argument is used, the `times = ...` argument is ignored. In the example, a patient is to "rollback" with a 40% probability to see the doctor for a second time. To do so, we use an external function and an "attribute". How these attributes work will be discussed later.

Step 5.1: Defining the trajectory

The code below illustrates how a trajectory can be defined within the branch using the `trajectory()` function. It also demonstrates an implementation of the `rollback()` function. Pay close attention to the `plot(traj)` output, since you can now observe the branch and the "back arrow" of the `rollback()` function to check whether the `amount = ...` argument was set correctly.

```
# PARAMETER AND FUNCTION TO HANDLE THE ROLL BACK
p_second_doctor_visit <- 0.40;
fun_rollback_doctor <- function(n_doctor_visits) {

  # INPUT:
  # - n_doctor_visits   a counter of the number of doctor visits
  #
  # OUTPUT:
  # - another_visit     a logical indicating whether another visit should occur, i.e. the
  #                     patient should roll back

  # If the patient already visited two times, no additional visit should occur
  if(n_doctor_visits == 2) {

    another_visit <- FALSE;

  # Otherwise, sample whether a second visit occurs
  } else if(n_doctor_visits == 1) {

    another_visit <- if(runif(1) < p_second_doctor_visit) {TRUE} else {FALSE}

  }

  # Return the outcome
  return(another_visit)

}

# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  timeout(task = 20) %>%
  release(resource = "nurse", amount = 1) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() sample(x = c(0, 1), size = 1, prob = c(0.20, 0.80)),
         continue = c(T),

    # DOCTOR CONSULTATION
    trajectory(name = "traj_doctor") %>%
      seize(resource = "doctor", amount = 1) %>%
      timeout(task = 15) %>%
      release(resource = "doctor", amount = 1) %>%

      # ATTRIBUTE TO COUNT THE NUMBER OF VISITS
      set_attribute(key = "n_doctor_visits", value = 1, mod = "+") %>%

      # ROLLBACK FOR A POTENTIAL SECOND VISIT
      rollback(amount = 4,
               check = function() {
                 fun_rollback_doctor(n_doctor_visits = get_attribute(.env = sim,
                                                                     keys = "n_doctor_visits"))
               })

  )


# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot
```

Step 5.2: Running the simulation

Given the updated trajectory, we can now simulate how long it would take to see all the patients. The use of the `sample()` function introduces randomness to the model. Therefore, for reproducibility purposes, we will set the random number seed before running the simulation to assure that we can regenerate our results.

Based on the updated pathway, how long did it take for all 100 patients to be seen?

```
# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%
  add_resource(name = "nurse", capacity = 3) %>%
  add_resource(name = "doctor", capacity = 2) %>%
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 100)),
                mon = 2);


# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);


# RUN THE SIMULATION
sim %>% reset() %>% run();
```

```
## simmer environment: sim | now: 915 | next:
## { Monitor: in memory }
## { Resource: nurse | monitored: TRUE | server status: 0(3) | queue status: 0(Inf) }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 100 }
```

Further illustrations

Sub-trajectories can also be defined separately and later be merged into one trajectory. The following code illustrates how a trajectory can be defined within the branch using an externally defined trajectory:

```r
# DEFINE THE DOCTOR TRAJECTORY
traj_doctor <-     trajectory(name = "traj_doctor") %>%
      seize(resource = "doctor", amount = 1) %>%
      timeout(task = 15) %>%
      release(resource = "doctor", amount = 1) %>%

      # ATTRIBUTE TO CHECK THE NUMBER OF VISITS
      set_attribute(key = "n_doctor_visits", value = 1, mod = "+") %>%

      # ROLLBACK FOR A POTENTIAL SECOND VISIT
      rollback(amount = 4,
               check = function() {
                 fun_rollback_doctor(n_doctor_visits = get_attribute(.env = sim,
                                                                      keys = "n_doctor_visits"))
               })


# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  timeout(task = 20) %>%
  release(resource = "nurse", amount = 1) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() sample(x = c(0, 1), size = 1, prob = c(0.20, 0.80)),
         continue = c(T),

    # DOCTOR CONSULTATION
    traj_doctor

  )


# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot
```

The following code illustrates how you can use an external function to provide values to the `option = ...` argument of the `branch()` function:*

```r
# EXTERNAL FUNCTION THAT DETERMINES WHETHER THE PATIENT IS TO SEE A DOCTOR AFTER THE INTAKE
fun_see_doctor <- function() {

  # OUTCOMES
  # 1   yes, see a doctor
  # 0   no, no need to see a doctor

  # SAMPLE THE VALUES 0 AND 1 WITH PROBABILITIES OF 20% AND 80%, RESPECTIVELY
  out <- sample(x = c(0, 1), size = 1, prob = c(0.20, 0.80));

  # RETURN THE OUTCOME
  return(out);
}


# TRY THE FUNCTION BY CALLING IT A FEW TIMES (HERE WE CALL IT ONLY ONCE)
fun_see_doctor();
```

```
## [1] 1
```

```r
# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  timeout(task = 20) %>%
  release(resource = "nurse", amount = 1) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() fun_see_doctor(), continue = c(T),

    # DOCTOR CONSULTATION
    traj_doctor

  )


# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%
  add_resource(name = "nurse", capacity = 3) %>%
  add_resource(name = "doctor", capacity = 2) %>%
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 100)),
                mon = 2);


# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);
```

```
# RUN THE SIMULATION
sim %>% reset() %>% run();
```

```
## simmer environment: sim | now: 915 | next:
## { Monitor: in memory }
## { Resource: nurse | monitored: TRUE | server status: 0(3) | queue status: 0(Inf) }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 100 }
```

For illustration purposes, we also demonstrate how the `branch()` is used including two (or more) trajectories. Two different sub-trajectories will be defined for the doctor consultation: one that takes 15 minutes and another that takes 30 minutes. Both trajectories will have the same probability of 40%. This trajectory will not be used beyond this illustration. Carefully look how the code for the `option = ...`, `continue = ...`, and `...` (trajectories) arguments changed, and how this impacted the trajectory structure (see plot).

```
# DEFINE TWO DOCTOR TRAJECTORIES
# - traj_doctor_short     including a 15 minute timeout
# - traj_doctor_long      including a 30 minute timeout
traj_doctor_short <- trajectory(name = "traj_doctor_short") %>%
  seize(resource = "doctor", amount = 1) %>%
  timeout(task = 15) %>%
  release(resource = "doctor", amount = 1)

traj_doctor_long <- trajectory(name = "traj_doctor_long") %>%
  seize(resource = "doctor", amount = 1) %>%
  timeout(task = 30) %>%
  release(resource = "doctor", amount = 1)


# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  timeout(task = 20) %>%
  release(resource = "nurse", amount = 1) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() sample(x = c(0, 1, 2), size = 1, prob = c(0.20, 0.40, 0.40)),
         continue = c(T, T),

    # TRAJECTORY 1: SHORT DOCTOR CONSULTATION (15 MINUTES)
    traj_doctor_short,

    # TRAJECTORY 2: LONG DOCTOR CONSULTATION (30 MINUTES)
    traj_doctor_long

  )


# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot
```

STEP 6: IMPLEMENTING PATIENT-LEVEL VARIATION

An important reason for using discrete event simulation is its flexibility towards including patient-level variation. This variation may either be due to heterogeneity in the patient population or due to stochastic uncertainty (i.e., first-order uncertainty). We will now work on further implementing stochastic uncertainty in the model. The fact that some patients will see a doctor and some do not, already is a form of stochastic uncertainty in the simulation. Another typical example of stochastic uncertainty is that in time-to-event. We will implement stochastic uncertainty in the duration of the nurse intake and doctor consultation using parametric distributions fitted to observed data. These parametric distributions are often referred to as "survival models" in health economic literature.

Step 6.1: Defining the parametric time-to-event distributions

Assume that data was collected for 15 patients on how long their intake and consultation took. Twelve of the 15 patients had a doctor consultation after the intake. The recorded times in minutes are recorded in the `data_time_nurse` and `data_time_doctor` vectors. The fitting of distributions will be discussed later in the course. For now it is sufficient to know that a Weibull distribution was fitted to the data, resulting in the parameter values as recorded in the `weibull_time_nurse` and `weibull_time_doctor` vectors. To illustrate how the fitted distributions match with the observed data, we draw `10,000` random samples from these, resulting in the `sim_time_nurse` and `sim_time_doctor` vectors, which we then compare numerically and visually to the observed data. The `rweibull()` function is used together with the fitted parameters to perform random draws from a Weibull distribution. This will be discussed in more detail later in the course.

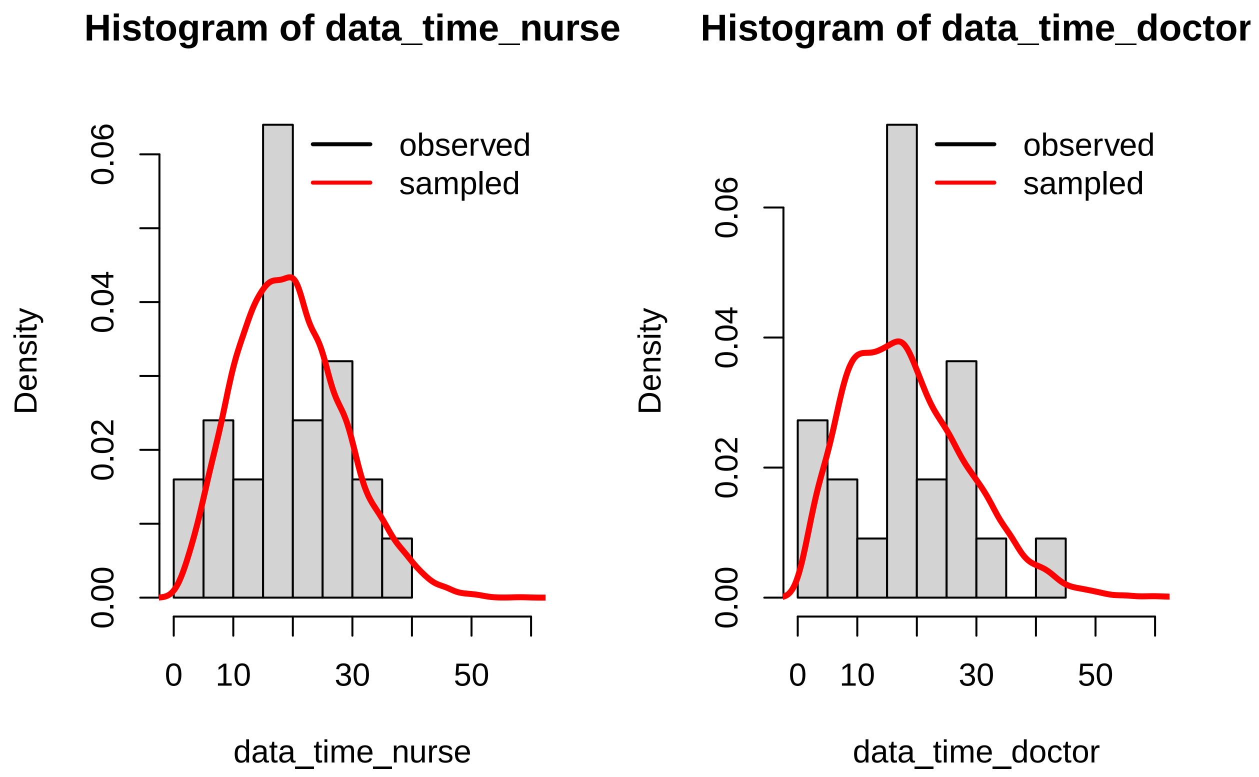
```r
# COLLECTED DATA ON THE MINUTES THAT 25 PATIENTS SPENT AT THE NURSE AND DOCTOR
data_time_nurse <- c(10, 4, 27, 39, 18, 31, 25, 27, 18, 18, 20, 23,
                     17, 19, 12, 30, 9, 4, 12, 19, 24, 33, 7, 17, 28);
data_time_doctor <- c(3, 13, 20,  6, 16, 34, 18, 26, 29, 41, 27, 21,
                      5, 22, 16, 7, 17, 18, 28, 2, 20, 17);


# DISTRIBUTIONS FITTED TO THE DATA
weibull_time_nurse <- c(shape = 2.325996, scale = 22.137460);
weibull_time_doctor <- c(shape = 1.887695, scale = 20.677025);


# PERFORM A SIMPLE SIMULATION OF HYPOTHETICAL PATIENTS TO ASSESS THE FIT
# 10.000 SIMULATIONS FOR THE NURSE
samples_time_nurse <- rweibull(n = 10000,
                               shape = weibull_time_nurse["shape"],
                               scale = weibull_time_nurse["scale"]);
# 10.000 SIMULATIONS FOR THE DOCTOR
samples_time_doctor <- rweibull(n = 10000,
                                shape = weibull_time_doctor["shape"],
                                scale = weibull_time_doctor["scale"]);


# COMPARE THE OBSERVED MEAN MINUTES SPENT TO THE SIMULATED MEAN MINUTES SPENT
c(nurse_observed = mean(data_time_nurse), nurse_sampled = mean(samples_time_nurse));
```

```
## nurse_observed  nurse_sampled
##       19.64000       19.68847
```

```r
c(doctor_observed = mean(data_time_doctor), doctor_sampled = mean(samples_time_doctor));
```

```
## doctor_observed  doctor_sampled
##        18.45455        18.46671
```

```r
# VISUALIZE THE FIT OF THE DISTRIBUTIONS IN A HISTOGRAM
{
  # SETTING TO PLOT TWO GRAPHS NEXT TO EACH OTHER
  par(mfrow = c(1, 2));

  # PLOT THE OBSERVED AND SAMPLED DATA FOR THE NURSE TIME
  hist(data_time_nurse, prob = TRUE, xlim = c(0, 60));
  lines(density(samples_time_nurse), col = "red", lwd = 3);
  legend("topright", legend = c("observed", "sampled"),
         col = c("black", "red"), lty = 1, lwd = 2, bty = 'n');

  # PLOT THE OBSERVED AND SAMPLED DATA FOR THE DOCTOR TIME
  hist(data_time_doctor, prob = TRUE, xlim = c(0, 60));
  lines(density(samples_time_doctor), col = "red", lwd = 3);
  legend("topright", legend = c("observed", "sampled"),
         col = c("black", "red"), lty = 1, lwd = 2, bty = 'n');

}
```



Step 6.2: Implementing the distributions in the trajectory

After the data analysis has been performed, the fitted distributions can be implemented in the trajectory. Each distribution will be used to define the `task = ...` argument of the corresponding `timeout()` call. We will again use the `rweibull()` function to generate a random draw from the distribution, though now we only sample 1 value each time a value is required. The use of the `function()` statement before the `rweibull()` function will be explained after the next step. Based on the illustration for the nurse, implement the distribution for the duration of the doctor consultation.

```r
# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  timeout(task = function() rweibull(n = 1,
                                     shape = weibull_time_nurse["shape"],
                                     scale = weibull_time_nurse["scale"])
          ) %>%
  release(resource = "nurse", amount = 1) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() sample(x = c(0, 1), size = 1, prob = c(0.20, 0.80)),
         continue = c(T),

         # DOCTOR CONSULTATION
         trajectory(name = "traj_doctor") %>%
           seize(resource = "doctor", amount = 1) %>%
           timeout(...WRITE CODE HERE...) %>%
           release(resource = "doctor", amount = 1)

  )


# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot
```

Step 6.3: Running the simulation

Now the trajectory has been updated to reflect stochastic uncertainty in the duration of the intake and the consultation, it is time to see what the impact of this change is on the outcome. Again, we need to set the random number seed for reproducibility.

Based on the updated pathway, how long did it take for all 100 patients to be seen?

```
# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%
  add_resource(name = "nurse", capacity = 3) %>%
  add_resource(name = "doctor", capacity = 2) %>%
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 100)),
                mon = 2);

# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);

# RUN THE SIMULATION
sim %>% reset() %>% run();
```

```
## simmer environment: sim | now: 742.829275576585 | next:
## { Monitor: in memory }
## { Resource: nurse | monitored: TRUE | server status: 0(3) | queue status: 0(Inf) }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 100 }
```

Remark: Use of "function()" within the trajectory

When providing dynamic values to an argument of a function within a trajectory, the statement `function()` needs to be used to instruct the simulation to evaluate the statement for every entity. A dynamic value refers to a value that may be different among entities. In the example below, we illustrate the impact of the `function()` statement for drawing a random value from the `weibull_time_nurse` Weibull distribution. If the `function()` statement is not used, only one random draw will be performed and used for **all** entities. If a unique value is to be sampled for every entity, the `function()` statement needs to be used. In this illustration, we use the `log_()` function to print to the console.

log_() The `log_()` function logs a character to the console. In reporting the message, it includes the simulation time at that point and the entity that generated the message. This function is extremely helpful for debugging, but be aware that logging can significantly impact the computation time of your model. The `log_()` function has only one important argument (see `?log_` for additional information):

- `message = ...` a character representing the message to be printed. To define this message, other functions can be called or different characters can be combined using, for example, the `paste()` function. Typically, the process will include reporting on numerical values and, hence, the `as.character()` function is often required to transform these into characters.

The code on the next pages illustrates the impact of using the `function()` statement for 10 simulated entities. First, it will be illustrated what happens when the `function()` statement is not used, followed by an illustration of the correct use of the `function()` statement on the subsequent page.

```
# TEMPORARY TRAJECTORY ILLUSTRATING WHEN "function()" IS NOT USED
tmp.traj <- trajectory() %>%

  # THE ONLY PART OF THIS TRAJECTORY WILL BE A LOG OF THE SAMPLED VALUE
  log_(message = as.character(rweibull(n = 1,
                                       shape = weibull_time_nurse["shape"],
                                       scale = weibull_time_nurse["scale"])))


# TEMPORARY SIMUALTION ILLUSTRATING WHEN "function()" IS NOT USED
tmp.sim <- simmer() %>%
  add_generator(name_prefix = "patient",
                trajectory = tmp.traj,
                distribution = at(rep(x = 0, times = 10)),
                mon = 2);


# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);


# RUN THE SIMULATION
tmp.sim %>% reset() %>% run();
```

```
## 0: patient0: 33.3754640096593
## 0: patient1: 33.3754640096593
## 0: patient2: 33.3754640096593
## 0: patient3: 33.3754640096593
## 0: patient4: 33.3754640096593
## 0: patient5: 33.3754640096593
## 0: patient6: 33.3754640096593
## 0: patient7: 33.3754640096593
## 0: patient8: 33.3754640096593
## 0: patient9: 33.3754640096593

## simmer environment: anonymous | now: 0 | next:
## { Monitor: in memory }
## { Source: patient | monitored: 2 | n_generated: 10 }
```

```r
# TEMPORARY TRAJECTORY ILLUSTRATING WHEN "function()" IS USED
tmp.traj <- trajectory() %>%

  # THE ONLY PART OF THIS TRAJECTORY WILL BE A LOG OF THE SAMPLED VALUE
  log_(message = function() as.character(rweibull(n = 1,
                                                  shape = weibull_time_nurse["shape"],
                                                  scale = weibull_time_nurse["scale"])))


# TEMPORARY SIMUALTION ILLUSTRATING WHEN "function()" IS USED
tmp.sim <- simmer() %>%
  add_generator(name_prefix = "patient",
                trajectory = tmp.traj,
                distribution = at(rep(x = 0, times = 10)),
                mon = 2);


# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);


# RUN THE SIMULATION
tmp.sim %>% reset() %>% run();
```

```
## 0: patient0: 24.3350703446991
## 0: patient1: 11.9399271978457
## 0: patient2: 21.0972969631651
## 0: patient3: 9.03619290370118
## 0: patient4: 6.66910301457832
## 0: patient5: 35.9500896214398
## 0: patient6: 18.2536612891309
## 0: patient7: 8.69707584789342
## 0: patient8: 17.7116870574821
## 0: patient9: 19.9375421100046

## simmer environment: anonymous | now: 0 | next:
## { Monitor: in memory }
## { Source: patient | monitored: 2 | n_generated: 10 }
```

STEP 7: USING ATTRIBUTES WITHIN TRAJECTORIES

The power of patient-level modeling techniques like discrete event simulation is that they allow entity-level variables to be tracked and updated throughout simulations. In a healthcare context, entity variables typically are patient characteristics, such as age and gender, but can also be treatment history, test results, and treatment outcomes. Variables can either be static (i.e., fixed throughout the simulation) or dynamic (i.e., updated throughout the simulation based on events). The `simmer` package includes the `set_attribute()` and `get_attribute()` functions that can be used to write and read entity-level variables, respectively. In the example that we are working on, we will use these functions to store the times and events.

set_attribute() The `set_attribute()` function is used to set variables on an entity level. Each variable is defined by a name that is a character, which the `simmer` package refers to as a `key`. The value of a variable can be numbers only. For example, it is not possible to have an attribute `"gender"` that is defined by the character values `"female"` and `"male"`. To have a variable that defines the gender, you would rather name it `"gender_female"` with a value of `1` for females and `0` for males. It is important to realize that you can set multiple attributes at the same time, as will be discussed below. All arguments to the `set_attribute()` function are relevant (see `?set_attribute` for further information):

- `keys = ...` a character vector that defines the names of the attributes that you want to set or update.
- `values = function() ...` a numerical vector that defines the values to be used to set or update the attributes. When multiple attributes are set or updated, the order of the values should match the order of the keys. Notice that, again, we need to use the `function()` statement to allow for the value to be different for every entity.
- `mod = ...` a character vector of mathematical expressions that are used to modify the pre-updated values of the attributes using the values provided. For example, if the value of an attribute was previously set to `10` and the current call of the `set_attribute()` function includes `value = 5` and `mod = "+"`, the updated value of the attribute will be `10 + 5 = 15`. Similarly, if the pre-updated values of two attributes are both `10` and the current call of the `set_attribute()` function includes `value = c(5, 5)` and `mod = c("+", "-")`, the resulting values will be `15` and `5`. The `mod = ...` argument is typically useful for adding costs to an attribute that tracks the total costs, for example.
- `init = ...` a numerical vector that defines what the initial values for the attributes should be if the `mod = ...` argument is used but the attribute has not been defined previously. In that scenario, it is not known what value should be 'modified', so that information is provided by this `init = ...` argument. For example, if an attribute has not yet been set and the current call of the `set_attribute()` function includes `value = 5`, `mod = "+"`, and `init = 20`, the set value of the attribute will be `20 + 5 = 25`.

get_attribute() The `get_attribute()` function allows you to read the current value of one or multiple attributes from within the trajectory. This enables the use of attribute values to determine what happens in the simulation. For example, `get_attribute()` can be used to set the value of other attributes by using it in the `value = function() ...` argument of the `set_attribute()` function. Also, as will be illustrated, it can be used to define the `option = function() ...` argument in the `branch()` function or the `task = function()...` argument of the `timeout()` function. Both two arguments of the `get_attribute()` are essential (see `?get_attribute()` for more information):

- `.env = ...` the simulation environment that contains the entity from which the attribute is to be read. This should correspond to the simulation environment that you defined using the `simmer()` function and in which you simulate entities using the `add_generator()` function.
- `keys = ...` a character vector that defines the names of the attributes for which you want to read the values. If multiple names are provided in a vector, a vector will be returned with the value in the corresponding order. Unfortunately, it is up to you to track the order of the attributes, as the returned vector is not named in the current version of the `simmer` package.

Similar to the `set_attribute()` and `get_attribute()` functions that concern variables on an entity level, `get_global()` and `get_local()` can be used to define variables in the global simulation environment. These global variables can be used by every entity, whereas the entity variables can only be set and read by each entity itself. However, since we will not use global variables, these functions are not discussed in further detail. You may consult the help function for more information.

In the example below, it is illustrated how the `set_attribute()` function can be used to set an attribute called `"see_doctor"` to indicate whether the patient needs to see the doctor after the intake by the nurse. We will again use the `log_()` function to print the value to the console. However, this time we read the value to be logged from the attribute using the `get_attribute()` function. The `get_attribute()` function will subsequently also be used to define the `option = function ...` argument to the `branch()` function. The corresponding flowchart illustrates how the `set_attribute()` statements are visible as part of the trajectory. The simulation is only run for `10` patients for the sake of readability.

```
# DEFINE THE FUNCTIONS TO SAMPLE FROM THE DISTRIBUTIONS
fun_time_nurse <- function() rweibull(n = 1,
                                      shape = weibull_time_nurse["shape"],
                                      scale = weibull_time_nurse["scale"]);
fun_time_doctor <- function() rweibull(n = 1,
                                       shape = weibull_time_doctor["shape"],
                                       scale = weibull_time_doctor["scale"]);


# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  timeout(task = function() fun_time_nurse()) %>%
  release(resource = "nurse", amount = 1) %>%

  # SET AN ATTRIBUTE "see_doctor" TO RECORD WHETHER THE PATIENT WILL NEED TO SEE A DOCTOR
  set_attribute(keys = "see_doctor",
                values = function() sample(x = c(0, 1),
                                           size = 1,
                                           prob = c(0.20, 0.80))
                ) %>%

  # LOG THE "see_doctor" ATTRIBUTE TO THE CONSOLE
  # OBSERVE HOW WE USE THE "paste()" FUNCTION TO ADD TEXT FOR INTERPRETATION
  log_(message = function() as.character(paste("see_doctor =",
                                               get_attribute(.env = sim,
                                                             keys = "see_doctor")))
       ) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() get_attribute(.env = sim, keys = "see_doctor"),
         continue = c(T),

         # DOCTOR CONSULTATION
         trajectory(name = "traj_doctor") %>%
           seize(resource = "doctor", amount = 1) %>%
           timeout(task = function() fun_time_doctor()) %>%
           release(resource = "doctor", amount = 1)
  )
```

```r
# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot


# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%
  add_resource(name = "nurse", capacity = 3) %>%
  add_resource(name = "doctor", capacity = 2) %>%
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 10)),
                mon = 2);

# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);

# RUN THE SIMULATION
sim %>% reset() %>% run();
```

```
## 11.9399: patient1: see_doctor = 0
## 18.609: patient3: see_doctor = 1
## 21.0973: patient2: see_doctor = 1
## 24.3351: patient0: see_doctor = 1
## 36.8627: patient4: see_doctor = 1
## 39.0906: patient6: see_doctor = 1
## 41.0348: patient5: see_doctor = 0
## 45.2824: patient7: see_doctor = 1
## 55.4219: patient9: see_doctor = 1
## 62.2885: patient8: see_doctor = 1
```

```
## simmer environment: sim | now: 76.921165448362 | next:
## { Monitor: in memory }
## { Resource: nurse | monitored: TRUE | server status: 0(3) | queue status: 0(Inf) }
## { Resource: doctor | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Source: patient | monitored: 2 | n_generated: 10 }
```

After the simulation is completed, you can extract information on all attributes of all entities and how these changed over time. How to do so will be discussed in the subsequent section. For now, it is sufficient to know that it is possible. In our example, it would be interesting to be able to see after the simulation which patients needed to see a doctor after their intake and how much time all patients spent at the nurse and doctor. Hence, a useful way of utilizing attributes is to store the value for the `branch()` to see a doctor (like was illustrated above) and the times spent at both resources.

Use the code on the next page as a starting point to store the time that will be spent by the patient at the nurse and doctor in attributes called `"time_nurse"` and `"time_doctor"`, and then read these values to define how long the intake and consultation will take. For the nurse, use the `timeout()` function to simulate the duration of the intake. For the doctor, use the `timeout_from_attribute()` function, which is very convenient for simulating delays based on attributes and was specifically implemented to do so. Consult `?timeout_from_attribute` to learn how this function works and implement it in the code. Check that this different implementation of defining and using the time-to-events does not impact the outcomes.

```r
# DEFINE THE FUNCTIONS TO KEEP THE TRAJECTORY CLEAN
fun_time_nurse <- function() rweibull(n = 1,
                                      shape = weibull_time_nurse["shape"],
                                      scale = weibull_time_nurse["scale"]);
fun_time_doctor <- function() rweibull(n = 1,
                                       shape = weibull_time_doctor["shape"],
                                       scale = weibull_time_doctor["scale"]);
fun_see_doctor <- function() sample(x = c(0, 1), size = 1, prob = c(0.20, 0.80));


# DEFINE THE TRAJECTORY
traj <- trajectory(name = "traj") %>%

  # NURSE INTAKE
  seize(resource = "nurse", amount = 1) %>%
  ...WRITE CODE HERE...
  timeout(...WRITE CODE HERE...) %>%
  release(resource = "nurse", amount = 1) %>%

  # SET AN ATTRIBUTE "see_doctor" TO RECORD WHETHER THE PATIENT WILL NEED TO SEE A DOCTOR
  set_attribute(keys = "see_doctor", values = function() fun_see_doctor()) %>%

  # BRANCH TO REFLECT THAT 20% OF PATIENTS DO NOT NEED TO SEE A DOCTOR AFTER THE INTAKE
  branch(option = function() get_attribute(.env = sim, keys = "see_doctor"),
         continue = c(T),

         # DOCTOR CONSULTATION
         trajectory(name = "traj_doctor") %>%
           seize(resource = "doctor", amount = 1) %>%
           ...WRITE CODE HERE...
           timeout_from_attribute(...WRITE CODE HERE...) %>%
           release(resource = "doctor", amount = 1)
  )


# VISUALIZE THE TRAJECTORY
# plot(traj); # uncomment if you want to plot

# DEFINE THE SIMULATION ENVIRONMENT
sim <- simmer(name = "sim") %>%
  add_resource(name = "nurse", capacity = 3) %>%
  add_resource(name = "doctor", capacity = 2) %>%
  add_generator(name_prefix = "patient",
                trajectory = traj,
                distribution = at(rep(x = 0, times = 100)),
                mon = 2);

# SET THE RANDOM NUMBER SEED FOR REPRODUCIBILITY
set.seed(123);

# RUN THE SIMULATION
sim %>% reset() %>% run();
```

STEP 8: OBSERVING MONITORED ATTRIBUTES

So far, we only looked at the total simulation time as an outcome. Although informative for the current exemplary case study, this is rarely an outcome of primary interest. From an operations management perspective, outcomes that are of higher interest typically are waiting times and resource utilization rates. From a health economic perspective, outcomes that are of primary interest are the time spent at different states and the costs accrued. There are three main functions that the `simmer` package provides to extract outcomes from the simulation environment: `get_mon_arrivals()`, `get_mon_resources()` and `get_mon_attributes()`.

get_mon_arrivals() The `get_mon_arrivals()` function returns a `data.frame` with very basic information for every simulated entity, including the time they entered the simulation, how much time they were served by resources, and when they finished. The most important arguments to this function are (see `?get_mon_arrivals` for more information):

- `.envs = ...` the simulation environment for which the outcomes are to be extracted.
- `per_resource = ...` a logical indicating whether the arrival times are to extracted on resource level, which results in multiple rows per entity.

get_mon_resources() The `get_mon_resources()` function returns a `data.frame` with information about the utilization and queue sizes of the resources. The function has only one argument (see `?get_mon_resources` for more information):

- `.envs = ...` the simulation environment for which the outcomes are to be extracted.

get_mon_attributes() The `get_mon_attributes()` function returns a `data.frame` with all changes to attributes for every simulated entity. The function has only one argument (see `?get_mon_attributes` for more information):

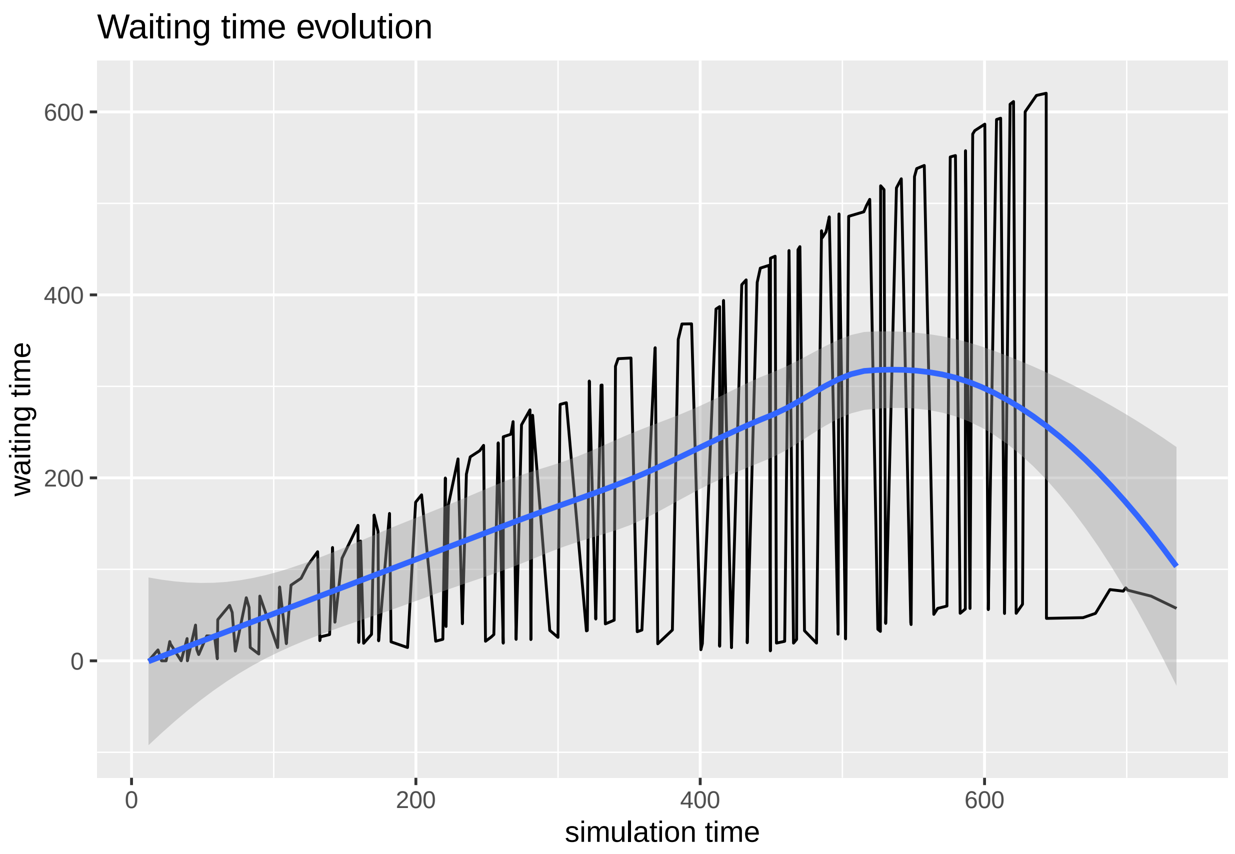
- `.envs = ...` the simulation environment for which the outcomes are to be extracted.

Step 8.1: Operational outcomes

Since this course focuses on the implementation of discrete event simulation to simulate health and economic outcomes, we will mainly use the `get_mon_attributes()` function in further practical sessions. The reason for this is that attributes (or variables) are a very convenient way of capturing all intermediate and final outcomes of interest. However, for illustration purposes, we will also demonstrate the use of the output of the `get_mon_resources()` and `get_mon_arrivals()` function to obtain insights in the utilization of resources and waiting times. You may consult `?plot.simmer` for more information on plotting more operations related outcomes. We will use functions from the `dplyr` package that is part of the `tidyverse` for data manipulation, so this package needs to be installed and loaded.

Can you find out how many nurses and doctors are required to reduce the total runtime to max. 3 hours?

```
# INSTALL THE REQUIRED PACKAGE
# UNCOMMENT LINE BELOW IF REQUIRED
# - ONLY NECESSARY ONCE ON YOUR OWN MACHINE
# - REQUIRED EVERY SESSION WHEN WORKING WITH R STUDIO CLOUD
# install.packages(pkgs = "tidyverse");


# LOAD THE REQUIRED PACKAGE
library(tidyverse);


# EXTRACT THE DATA FROM THE SIMULATION
mon_arrivals <- get_mon_arrivals(.envs = sim, per_resource = TRUE);
mon_resources <- get_mon_resources(.envs = sim);


# PLOT THE WAITING TIME
plot(x = mon_arrivals, metric = "waiting_time");
```



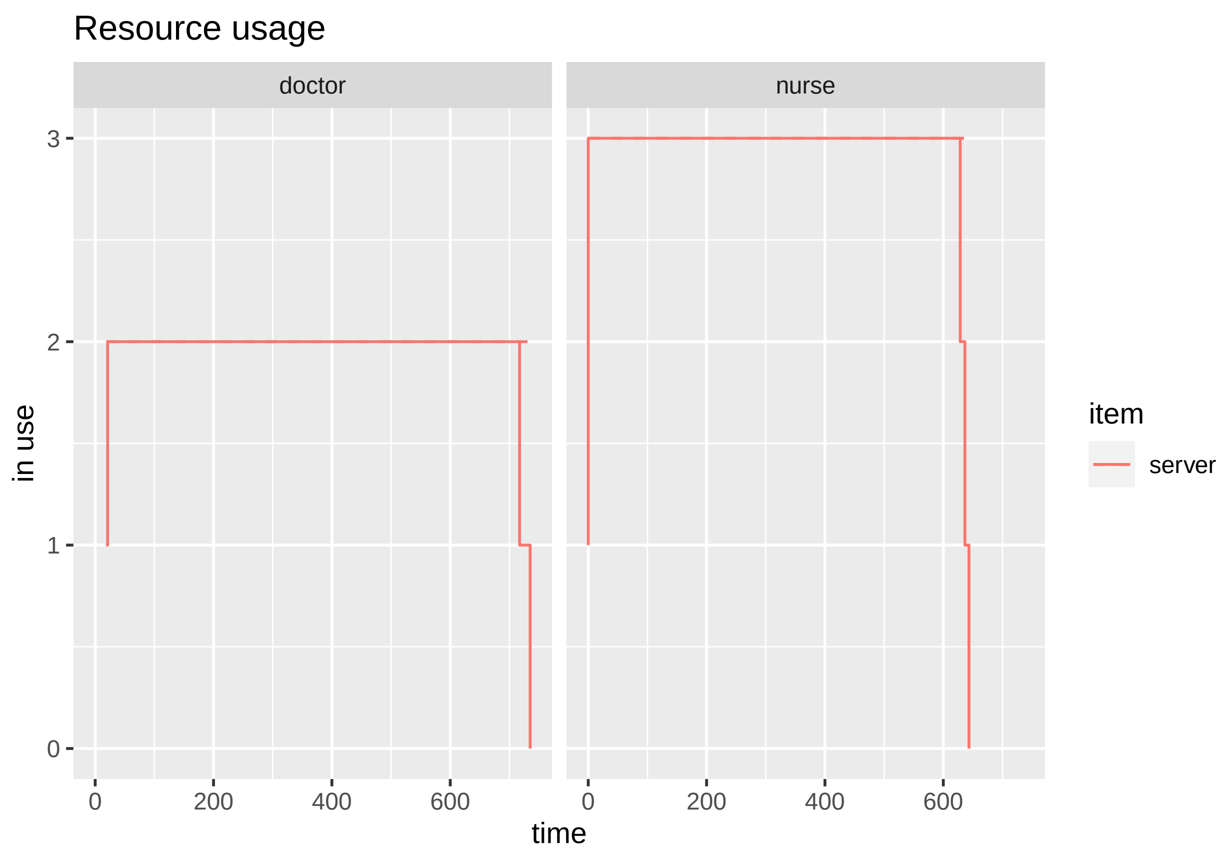
```
# CALCULATE THE MEAN WAITING TIME FOR EACH RESOURCE
mon_arrivals <- mon_arrivals %>%
  mutate(waiting_time = end_time - start_time - activity_time);

mean_waiting_time <- mon_arrivals %>%
  group_by(resource) %>%
  summarize(mean_waiting_time = mean(waiting_time));

mean_waiting_time;
```

```
## # A tibble: 2 x 2
##   resource mean_waiting_time
##   <chr>                <dbl>
## 1 doctor                32.6
## 2 nurse                313.
```

```
# AVERAGE TOTAL WAITING TIME
sum(mean_waiting_time$mean_waiting_time);
```

```
## [1] 345.544
```

```
# PLOT THE RESOURCE USAGE OVER THE SIMULATION TIME
plot(x = mon_resources, metric = "usage", item = "server", steps = TRUE);
```



Step 8.2: Patient outcomes

As will be illustrated below, the `get_mon_attributes()` function returns a data.frame with **all** changes to attributes for every entity, including timestamps. Although this provides a wealth of information and is great for debugging, it makes it hard to extract what is of main interest, which are the final outcomes for each entity. Therefore, we have written the custom `summarize_attributes()` function to extract the final values for each attribute on the entity level.

summarize_attributes() The `summarize_attributes()` function summarizes the output of the `get_mon_attributes()` function into a data.frame with a row for each entity and the final values of the attributes of interest. If an attribute is not defined for an entity, `NA` is returned deliberately. The `summarize_attributes()` function uses functions from the `tidyverse` package and has two arguments:

- `sim_out = ...` a data.frame returned by the `get_mon_attributes()` function.
- `keys = ...` a character vector defining the names of the attributes that are to be included in the entity-level summary.

The example below illustrates the (complex) structure of the `data.frame` that is returned by the `get_mon_attributes()` function and how the `summarize_attributes()` function can be used to summarize the variables of interest. The subsequent example will use the summary to validate the simulation.

```r
# LOAD THE REQUIRED PACKAGE
library(tidyverse);


# FUNCTION TO CONVERT THE SIMULATION OBJECT INTO A data.frame WITH A ROW FOR EACH ENTITY
summarize_attributes <- function(sim_out, keys) {

  # INPUTS:
  # - sim_out   the simulation object obtained using the 'get_mon_attributes()' function
  # - keys      character vectors defining the names of the attributes to be included

  # OUTPUT:
  # - df_out    data.frame with the name of the entity and all required variables

  df_out <- sim_out %>%
    # filter to include rows that have information attributes of interest
    filter(key %in% keys) %>%
    # group so that for every entity rows are grouped by the attributes
    group_by(name, key) %>%
    # for each entity and attribute, order the rows by the simulation time
    arrange(time) %>%
    # only select the last row, representing the final attribute value
    slice(n()) %>%
    # select columns including the entity name and attribute name and value
    dplyr::select(name, key, value) %>%
    # transform the long format data.frame into wide format
    spread(key = key, value = value)

  # RETURN THE df_out OBJECT WITH THE COLUMNS ORDERED ACCORDING TO THE keys INPUT
  return(df_out[, c("name", keys)])

}
```

```
# EXTRACT THE DATA FROM THE SIMULATION
mon_attributes <- get_mon_attributes(.envs = sim);


# PRINT LINES 95 TO 100 FROM THE mon_attributes OBJECT TO OBSERVE ITS STRUCTURE
mon_attributes[95:100, ];
```

```
##         time      name         key    value replication
## 95  100.9529 patient40  time_nurse 23.40500           1
## 96  103.0820 patient26 time_doctor 27.15679           1
## 97  104.9225 patient38  see_doctor  1.00000           1
## 98  104.9225 patient41  time_nurse 32.07337           1
## 99  111.3357 patient36  see_doctor  1.00000           1
## 100 111.3357 patient42  time_nurse 18.64077           1
```

```
# TRANSFORM THE mon_attributes OBJECT INTO A PATIENT-LEVEL SUMMARY OF THE FINAL
# VALUES OF ALL ATTRIBUTES
df_attributes <- summarize_attributes(sim_out = mon_attributes,
                                      keys = c("time_nurse", "see_doctor", "time_doctor"));


# PRINT THE FIRST 5 LINES OF THE df_attributes data.frame TO OBSERVE ITS STRUCTURE
head(x = df_attributes, n = 5);
```

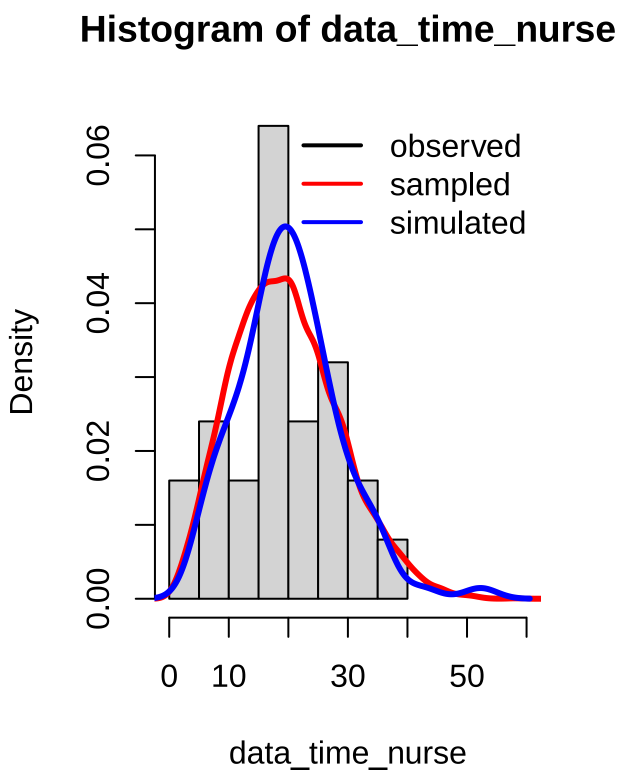
```
## # A tibble: 5 x 4
## # Groups:   name [5]
##   name      time_nurse see_doctor time_doctor
##   <chr>          <dbl>      <dbl>       <dbl>
## 1 patient0       24.3           0       NA
## 2 patient1       11.9           1       15.2
## 3 patient10      25.6           1        8.93
## 4 patient11       5.92          1       15.9
## 5 patient12      14.4           1       29.4
```

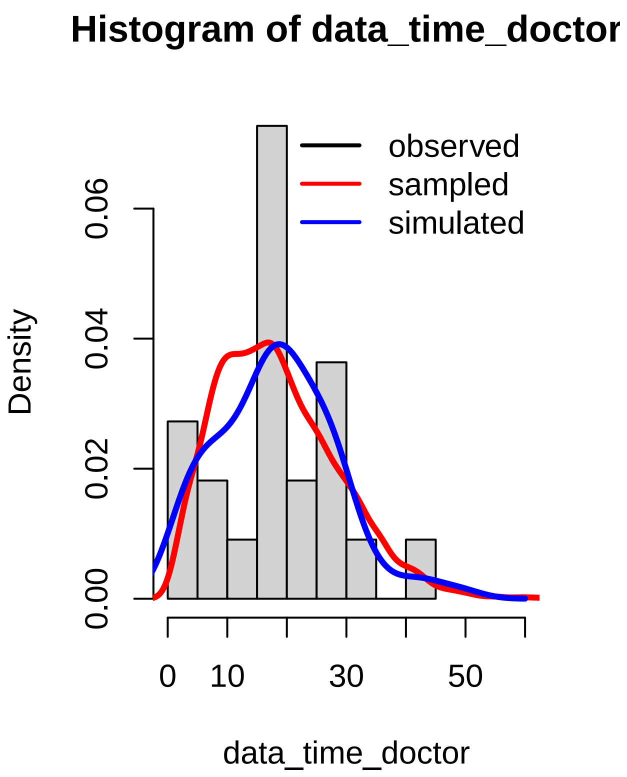
Now that the simulation output has been summarized, we can use the patient-level final values of the variables. We will do so to validate whether the simulated intake and consultation times match with those observed and sampled from the Weibull distributions that served as input to the simulation. To this end, the simulated times-to-event will be added to the previously plotted histogram. Also, we will assess whether the number of patients that need to see a doctor after the intake is approximately 80%. The small difference that can be observed in the percentage of consultations and distributions are caused by the low number of simulated patients. You can increase the number of simulated patients to see whether the outcomes become more accurate and stable.

```
# SEE WHETHER THE PROBABILITY OF SEEING A DOCTOR IS SIMILAR TO 80%
# - note that we use the mean of a vector of 0 and 1 values, which is equivalent to
#   taking the sum of the vector (i.e., counting the number of 1 values) and dividing
# it by the length of the vector
mean(x = df_attributes$see_doctor);
```

```
## [1] 0.79
```

```r
# EXTRACT THE SIMULATED TIMES
# - Alternatively, you could calculate this from the mon_arrivals data.frame
out_time_nurse <- df_attributes$time_nurse;
out_time_doctor <- df_attributes$time_doctor;


# VISUALIZE THE FIT OF THE DISTRIBUTIONS IN A HISTOGRAM
{
  # SETTING TO PLOT TWO GRAPHS NEXT TO EACH OTHER
  par(mfrow = c(1, 2));

  # PLOT THE OBSERVED AND SIMULATED DATA FOR THE NURSE TIME
  hist(data_time_nurse, prob = TRUE, xlim = c(0, 60));
  lines(density(samples_time_nurse), col = "red", lwd = 3);
  lines(density(out_time_nurse), col = "blue", lwd = 3);
  legend("topright", legend = c("observed", "sampled", "simulated"),
         col = c("black", "red", "blue"), lty = 1, lwd = 2, bty = 'n');

  # PLOT THE OBSERVED AND SIMULATED DATA FOR THE DOCTOR TIME
  hist(data_time_doctor, prob = TRUE, xlim = c(0, 60));
  lines(density(samples_time_doctor), col = "red", lwd = 3);
  lines(density(out_time_doctor, na.rm = T), col = "blue", lwd = 3);
  legend("topright", legend = c("observed", "sampled", "simulated"),
         col = c("black", "red", "blue"), lty = 1, lwd = 2, bty = 'n');

}
```


Histogram of data_time_nurse
observed
sampled
simulated
Density
0.00
0.02
0.04
0.06
0
10
30
50
data_time_nurse


Histogram of data_time_doctor
observed
sampled
simulated
Density
0.00
0.02
0.04
0.06
0
10
30
50
data_time_doctor

FINAL REMARKS

To finish this first practical session, let's recall the functions that the `simmer` package provides as basic building blocks for implementing discrete event simulations. Please take some time to reflect on the functions below.

Functions for defining trajectories:

- `trajectory()`
- `set_attribute()`
- `get_attribute()`
- `log_()`
- `seize()`
- `timeout()`
- `timeout_from_attribute()`
- `release()`
- `branch()`
- `rollback()`
- `%>%`

Functions for defining simulation environments:

- `simmer()`
- `add_resource()`
- `add_generator()`
- `reset()`
- `run()`
- `now()`
- `%>%`

Functions for defining to extract and summarize results:

- `get_mon_arrivals()`
- `get_mon_resources()`
- `get_mon_attributes()`
- `summarize_attributes()`

Also, reflect on the following aspects:

- the use of `plot()` (or `plot.simmer()`) to plot trajectories and outcomes
- the use of `function()` when defining values within trajectories
- the use of `set.seed()` for reproducibility

You now know all the building blocks required to implement discrete event simulations in R using the `simmer` package. There are many other useful functions that the `simmer` package provides, especially when building operations management models aiming to assess resource use etc. However, those functions are not crucial or applicable to the tutorials in this course and, therefore, not further discussed. If you want to know more about the functions that the `simmer` package provides, please check the website https://r-simmer.org/ or type `?simmer::` in the console, which will show a list of all functions in the package. In the following tutorials, we will use your newly obtained simulation programming skills for developing a health economic model.