

Getting Started with golden

Contents

golden	1
Installation	1
Philosophy & design	1
Scalar Functions	2
Understanding the classes	2
Running a simulation: simple examples	4
Example 0: exponential decay	4
Example 1: ageing as a trajectory; parametric uncertainty	7
Example 2: a stochastic multivariate trajectory	9
Example 3: timed events & multiple transitions	11
A more realistic example: globorisk	14
Initial population	14
Hazards	17
Trajectories & history	19
Running & plotting	20
Modelling interdependence and constraints	23
An SIR model	23

golden

golden is a flexible patient-level microsimulation modelling framework focussed on trajectories of patient risk factors and hazards of events, authored by Pete Dodd and Robert Chisholm.

Installation

golden is not currently available via CRAN, so must be installed via either devtools:

```
devtools::install_github("RSE-Sheffield/golden")
```

or by manually downloading a .zip from the golden GitHub repository to be installed:

```
devtools::install.packages("<path to .zip>", repos = NULL, type="source")
```

Following either of the actions, you should now be able to load golden like any other package.

```
library(golden)
```

Philosophy & design

golden is a framework which provides the glue, allowing R functions either defined by the user or selected from other packages to be combined into a comprehensive patient trajectory model.

The four conceptual components of the framework are **hazards**, **transitions**, **trajectories**, and **histories**. The population state is conceptualized as a `data.frame` or `data.table`, with one row per patient, and columns representing and storing information about the patients.

Hazards are functions that determine the likelihood of an event occurring for a patient in a given time step, given their current state (column values). If the random event occurs, an associated set of *transitions* will be applied that change the patient's state. An example hazard might be the mortality rate, with death as an associated transition.

Trajectories are rules that determine the evolution of patients' states. An example might be ageing, where ages advance with time while the patient is alive.

Histories define what aggregate summaries from the population to record as time series. The final population state will automatically be returned, which can be useful for computing cumulative or final quantities.

These concepts are represented by S3 objects in R, with constructors defined below. In order to create these objects, users will typically need to define

- Hazard functions
- Trajectory functions
- Transition functions
- History column functions
- History column filter functions

Whilst the exact usage of each case differs slightly, they all share a common core whereby users must specify both a function (often shortened to `fn` in the constructor argument names) and the function's arguments (often shortened to `args`).

The arguments passed to a function will typically be the names of columns (within the population `data.table`) to be passed to the function. The name “~STEP” is a reserved name created to represent the simulation time step.

Within this document, *Note* will preface some remark that explains how an example illustrates some important functionality that could be used in other ways.

Scalar Functions

Currently **golden** assumes that all functions are *designed to process vectors*, rather than individual scalars. This is so that columns in the data objects representing the population are operated on efficiently. If scalar handling is required, we recommend that you wrap your scalar function inside a parent function which calls the scalar function for each element of the vector arguments, e.g. by using **Vectorize**.

In the examples below, we have tended to use `ifelse(test, x, y)` for clarity. Often, patterns like

```
ans <- x
ans[test] <- y
```

may be faster than `ifelse` patterns.

Understanding the classes

Hazards and Transitions

Hazards represent an instantaneous chance of a patient transitioning between states. A hazard is passed the requested vector of population data and returns a hazard value which is converted to a probability via the equation below. A random number is then generated for each patient, which is tested to decide whether the hazard has occurred for that patient. Patients that are selected to be impacted by the hazard, then execute linked transition functions to update one or more of their states.

$$p = 1 - \exp(-h \, dt) \tag{1}$$

- p is the probability of the hazard occurring
- h is the value returned by a hazard function
- dt is the times differential, this value is always currently 1

Each transition is created via `new_transition()` which requires the transition function, it's arguments and the state (population column) that it updates.

```
example_transition <- new_transition(  
  ## arguments here  
)
```

Each hazard is created via `new_hazard()` which requires the hazard function, it's arguments, and a list of dependent transitions. Hazards also provide optional configuration to decrease the frequency or range of steps that a hazard is active for.

```
example_hazard <- new_hazard(  
  ## arguments here  
)
```

Trajectories

Trajectories represent regularly changing states, such as a patient's age or disease progression. They operate very similar to transitions, however trajectories allow multiple states may be updated simultaneously. Multivariate trajectories can return a list of vectors instead of a single vector, one for each state to be updated.

Each trajectory is created via `new_trajectory()` which requires the trajectory function, it's arguments, and the states to be updated.

```
example_trajectory <- new_trajectory(  
  ## arguments here  
)
```

History and Columns

History is the **golden** approach to logging aggregate population information (e.g. timeseries) during simulation execution. Individual columns of data to be collected are specified with a corresponding reduction function, with the optional facility to filter down to a sub-population to be reduced (e.g. the average age of living male patients).

Each column is created via `new_column()` which requires a name for the column, the reduction function and arguments to use and an optional filter function and respective arguments.

```
example_column <- new_column(  
  ## arguments here  
)
```

One or more columns are then passed to `new_history()`, alongside the frequency of data collection (which defaults to 1, every step).

```
example_history <- new_history(  
  ## arguments here  
)
```

Parameters

With all the components of the simulation defined, they can now be assembled into the main parameter list to pass to the simulation.

The parameters are created by passing lists of hazards and trajectories, whilst also specifying the number of steps to execute an optional items such as random seed or history.

```
example_parameters <- new_parameters(
  ## arguments here
)
```

Running a simulation: simple examples

To execute a simulation `run_simulation()` is called, passing the initial population table and parameters structure. In the next few sections we will step through a series of related, artificial examples to illustrate particular features.

First, let's load some relevant libraries and set the random seed:

```
library(golden)      # this package
library(data.table)  # enhanced data handling
library(ggplot2)     # plotting
set.seed(1234)       # PRNG seed
```

Example 0: exponential decay

One of the arguments for `run_simulation` is the initial population. **golden** does not provide utilities to create populations. Users are expected to create their own by resampling data, using packages such as **X** and **Y**, or writing their own methods. For these simple examples, we will create an artificial population as a `data.table` with one row per patient and columns containing their data. Here, we are considering ages, systolic blood pressure (SBP), and total cholesterol (TC).

```
## initial population
N <- 1e4L # population size
pop0 <- data.table(
  id = 1:N,                # patient ids
  death = rep(-1, N),      # time of death
  age = rep(40, N),        # age
  sbp = rlnorm(N, log(140), .05), # systolic BP
  tc = rlnorm(N, log(4.5), .01) # total cholesterol
)
```

For our first example, we will see how to apply a constant hazard of death. We need to write a function that returns this hazard:

```
## constant mortality hazard
deathrate0 <- function() {
  0.1
}
```

This is actually a special case: usually functions need to return vectors, and should be programmed to return vectors of the same length as their input arguments. This special constant function has no arguments and its scalar return value will be recycled to create a vector of identical element values, with length equal to the population size.

Hazards control how likely events are to happen at each time step. We define what events look like by specifying (a list of) transition functions that will be called to enact changes on the data if a hazard triggers an event. In our initial population we have chosen to include a variable `death` for time of death, which is -1 when people are alive. We can therefore use a generic transition to enact mortality by recording the time of death:

```
## generic transition: returns state as input value
transition_fn <- function(state, i) {
  # If result is true, and state is -1, update state to current time
}
```

```

  ifelse(state == -1, rep(i, length(state)), state)
}

```

We will need to create a hazard that understands more context about how apply the hazard function and associated transitions to the data:

```

## new_hazard creates a hazard
morthaz0 <- new_hazard(
  fn = deathrate0,           # function returning hazard
  args = c(),               # arguments for hazard function
  transitions = new_transition( # creates a transition
    fn = transition_fn,      # 'kills' by death -1 -> time of death
    args = c("death", "~STEP"), # args for transition fn
    states = "death"        # col transition acts on
  )
)

```

Note that the generic transition will record time of death: the input arguments are `death` (to check not already dead) and `~STEP` (the special current step variable) and, where applied, the function will set `death` to the current step value.

If we want to record aggregate time series data about our population during the simulation, we will need to construct a history object. For this we will need to specify columns, what summary they are computing, and (optionally) what population subset they are considering by providing a filter function that selects rows. As with above, we need to provide vectors of column names that specify which inputs functions are taking as arguments. You can also choose the frequency with which time series statistics are computed and recorded.

```

## filter to restrict to alive
filter_alive <- function(x) {
  x == -1 # tests alive applied to 'death'
}

## new_history creates a history
noalive <- new_history(
  columns = list(
    new_column(
      name = "no. alive",      # creates a col in history
      fn = length,            # column name
      args = "age",           # summary function
      filter_fn = filter_alive, # input args for summary fn
      filter_args = "death"    # filter to row-restrict
    )
  ),
  frequency = 1               # input variables for filter
                                # record history every 1 steps
)

```

Now we are in a position to create a parameters object, which will take lists of *hazards*, *trajectories*, and a *history*, as well as other unformation like how many steps to run the model for. Note that we haven't said how long a step is: the hazards need to return values in units of per step. If steps are meant to represent years, mortality hazards should return in units of per year; if step are meant to represent weeks; hazards should use units of per week.

```

parms0 <- new_parameters(
  hazards = morthaz0,
  steps = 10,
  debug = FALSE,
  history = noalive
)

```

```
)
```

Printing parameters, hazards, etc. will generate a summary of their ingredients.

Given an initial population and a parameters object, we can run the model:

```
## run the simulation
result0 <- run_simulation(pop0, parms0)
#> Simulation Complete in 0.01 seconds!
```

This outputs a list of 3 things:

- the final population
- a 'history': time series summary data
- timing data

```
## output list
print(result0) # explicit print unnecessary
#> $pop
#>      id death  age      sbp      tc
#>      <int> <num> <num>    <num>    <num>
#> 1:      1     0   40 131.8005 4.418978
#> 2:      2     0   40 141.9555 4.528311
#> 3:      3    -1   40 147.8007 4.523375
#> 4:      4     2   40 124.5065 4.506346
#> 5:      5    -1   40 143.0363 4.566057
#> ---
#> 9996: 9996     4   40 140.1382 4.519607
#> 9997: 9997    -1   40 125.8770 4.487413
#> 9998: 9998     1   40 139.6489 4.529636
#> 9999: 9999    -1   40 138.3427 4.470808
#> 10000: 10000    -1   40 145.5417 4.573960
#>
#> $history
#>      ~STEP no. alive
#>      <int>    <num>
#> 1:      1      9059
#> 2:      2      8162
#> 3:      3      7375
#> 4:      4      6702
#> 5:      5      6068
#> 6:      6      5495
#> 7:      7      4943
#> 8:      8      4463
#> 9:      9      4035
#> 10:     10      3635
#>
#> $timing
#> <golden_timing>
#>
#> Hazards:
#>      Hazard Total Time (s) Avg Time (s) % Runtime
#>      <char>          <num>          <num>          <num>
#> 1: deathrate0              0              0              0
#>
#> Transitions:
```

```

#>      Transition Total Time (s) Avg Time (s) % Runtime
#>      <char>          <num>      <num>      <num>
#> 1: transition_fn      0.001      1e-04      12.5
#>
#> Columns:
#>      Column Total Time (s) Avg Time (s) % Runtime
#>      <char>          <num>      <num>      <num>
#> 1: no. alive          0          0          0

```

Final population data can be used to understand final status and to record cumulative data at an individual level. Conversely, history data captures specified summaries, but does so over time.

Here, we can plot the fraction of the population surviving at each time step and compare it to the deterministic expectation:

```

## plot
ggplot(result0$history, aes(`~STEP`, `no. alive`)) +
  geom_line() +
  geom_point() +
  geom_function(fun = function(x) N * exp(-x / 10), col = 2)

```

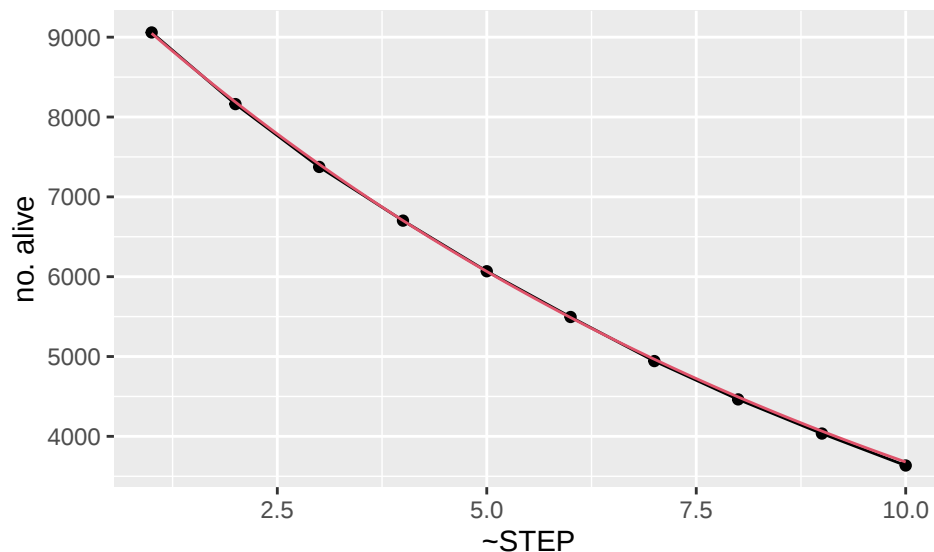


Figure 1: Comparison with analytical result (red)

Example 1: ageing as a trajectory; parametric uncertainty

You may have noticed that the example above did not include a trajectory. That is, none of the data associated with individuals evolved over time, except as a result of events triggered by hazards. In this example we will introduce some simple trajectories: deterministic or stochastic rules that specify how continuous variables associated with patients evolve. In realistic use cases, these would represent patient-level risk factors that influence the likelihood of health events. Perhaps the simplest example one could imagine is age.

A variety of quantities may increment with time or in order to record event counts, but age should only advance in those who are alive. Let's define functions for both these possibilities:

```

## Define a function to increment by 1
generic_update <- function(x) {
  x + 1
}

```

```

}

## If dead, age is not changed
age_update <- function(age, death_time) {
  ifelse(death_time == -1, age + 1, age)
}

```

As with *hazards*, we will create a *trajectory* using our update function, but will need to provide information about what the function uses as inputs, and what column it acts upon:

```

## new_trajectory creates a trajectory
age_traj <- new_trajectory(
  fn = age_update,          # update function
  args = c("age", "death"), # args for update function
  states = "age"            # column(s) to be updated
)

```

Let's also consider a marginally more realistic mortality hazard: we'll make our mortality risk increase with SBP and TC:

```

## mortality hazard depending on individual characteristics
deathrate1 <- function(death, sbp, tc) {
  ifelse(death < 0, 0.1 + sbp / 1000 + tc / 100, 0)
}

morthaz1 <- new_hazard(
  fn = deathrate1,          # hazard function
  args = c("death", "sbp", "tc"), # args for hazard fn
  ## (list of) transitions to apply
  transitions = new_transition(
    fn = transition_fn,      # generic transition
    args = c("death", "~STEP"), # input col args
    states = "death"         # col outputted
  )
)

```

This time the hazard construction is less trivial, with a vector of column names to be used as input arguments by the mortality rate.

Note also that conceptually, this shows how to include parametric uncertainty in analyses. Often health economic analyses will perform Probabilistic Sensitivity Analysis (PSA), which means sampling across uncertain parameters to reflect their uncertainty in outputs. Here SBP and TC are patient characteristics that have been sampled to vary between patients, but they could equally represent uncertain global parameters, e.g. an uncertain treatment effect or natural history parameter. In this framework, 'parameters' (and any uncertainty in them) should be included in the 'population'.

In the same way as before, we can create a parameter object and run the simulation (unless they died in the first step).

```

## full parameters
parms1 <- new_parameters(
  hazards = morthaz1,
  trajectories = age_traj,
  steps = 10,
  debug = FALSE,
  history = noalive
)

```

```
## run the simulation
result1 <- run_simulation(pop0, parms1)
#> Simulation Complete in 0.01 seconds!
```

This time, one can see that people are no longer 40 year old at the end of the simulation (unless they died in the first step):

```
result1$pop
#>      id death  age    sbp    tc
#>      <int> <num> <num>  <num>  <num>
#> 1:      1     4   44 131.8005 4.418978
#> 2:      2     0   40 141.9555 4.528311
#> 3:      3     8   48 147.8007 4.523375
#> 4:      4    -1   50 124.5065 4.506346
#> 5:      5     0   40 143.0363 4.566057
#> ---
#> 9996: 9996     1   41 140.1382 4.519607
#> 9997: 9997     2   42 125.8770 4.487413
#> 9998: 9998     4   44 139.6489 4.529636
#> 9999: 9999     3   43 138.3427 4.470808
#> 10000: 10000     3   43 145.5417 4.573960
```

Example 2: a stochastic multivariate trajectory

In this section we will illustrate how to implement a stochastic multivariate trajectory. However, to keep things simple for inspecting longer runs, let's turn off death:

```
## keep everyone alive here to have a look at RW dynamics
deathrate2 <- function() {
  0
}
morthaz2 <- new_hazard(deathrate2, args = c(), transitions = list())
```

It may be the case that certain patient-characteristics evolve in a non-independent way. Modern statistical approaches and longitudinal data sets allow development of models that can simultaneously represent the correlated evolution of multiple variables, i.e. are multivariate. Here, we consider a toy example of a bivariate trajectory model for SBP and TC. We represent their evolution as a correlated geometric random walk (i.e. a correlated random walk in log transformed space):

```
## sbp & tc as correlated geometric random walk
bptc_update <- function(sbp, tc, death_time) {
  lsbp <- log(sbp)
  ltc <- log(tc)
  alive <- death_time < 0
  if (sum(alive) > 1) {
    U <- matrix(rnorm(2 * sum(alive)), nrow = sum(alive), ncol = 2)
    U[, 2] <- U[, 2] + U[, 1] / 2 # correlation
    U <- U / 100 # rescale
    lsbp[alive] <- lsbp[alive] + U[, 1]
    ltc[alive] <- ltc[alive] + U[, 2]
  }
  list(sbp = exp(lsbp), tc = exp(ltc))
}

bptc_update(rep(140, 5), rep(4.5, 5), rep(-1, 5))
```

```
#> $sbp
#> [1] 142.4213 141.2728 139.7552 141.2690 144.1367
#>
#> $tc
#> [1] 4.473751 4.576176 4.433001 4.521404 4.504199
```

The two columns are returned as a list of vectors. *Note* that even if a group of variables are not evolving in a correlated way, it may be more logical, convenient, or economical of code to group them together rather than writing separate update functions for each of them.

Note also that this example illustrates use of a random function. There is no restriction that the functions provided for use in trajectory updates, or transitions cannot include stochasticity. This is also true of hazard functions, although this would be less common.

Having defined our update, we can create a trajectory as before, but now with vectors to specify which columns are affected. Now we have multiple trajectories, we will need to pass a list of them to parameter the constructor.

```
bptc_traj <- new_trajectory(
  fn = bptc_update,          # trajectory fn
  args = c("sbp", "tc", "death"), # trajectory args
  states = c("sbp", "tc")    # outputs (list of vectors from fn)
)

## make list of trajectories for use below
trajlist2 <- list(
  ## age
  age_traj,
  ## SBP & TC handled in bivariate fashion
  bptc_traj
)
```

In this case, it may convey more to construct a history that picks out a single individual patient trajectory. There are different ways this could be done, since `sum` and `mean` and the identity function `function(x) x` all return the same when applied to a single individual (vector of length 1).

```
## filter to 1
filter_1 <- function(x) {
  x == 1
}

## history to also include random walk variables for id==1
noalive2 <- new_history(
  columns = list(
    new_column("no. alive", length, c("age"), filter_alive, c("death")),
    ## 3 ways of returning an individual's value:
    new_column("sbp1", sum, c("sbp"), filter_1, c("id")),
    new_column("tc1", mean, c("tc"), filter_1, c("id")),
    new_column("tc1 v2", function(x) x, c("tc"), filter_1, c("id"))
  ),
  frequency = 1
)
```

For this illustration, let's run the simulation for 1000 steps, but only for the first 50 individuals:

```
## full parameters
parms2 <- new_parameters(
```

```

hazards = morthaz2,
trajectories = trajlist2,
steps = 1000, #longer
debug = FALSE,
history = noalive2
)

## run the simulation
result2 <- run_simulation(pop0[1:50], parms2) #look at a limited population
#> Simulation Complete in 0.03 seconds!

```

We can now transform and plot the SBP and TC trajectories for the first individual, which do indeed look like a correlated geometric random walk:

```

plot_data <- melt(result2$history[, .(t = `~STEP`, sbp1, tc1)], id = "t")

ggplot(plot_data, aes(t, value, col = variable)) +
  geom_line() +
  facet_wrap(~variable, scales = "free_y") +
  theme(legend.position = "none")

```

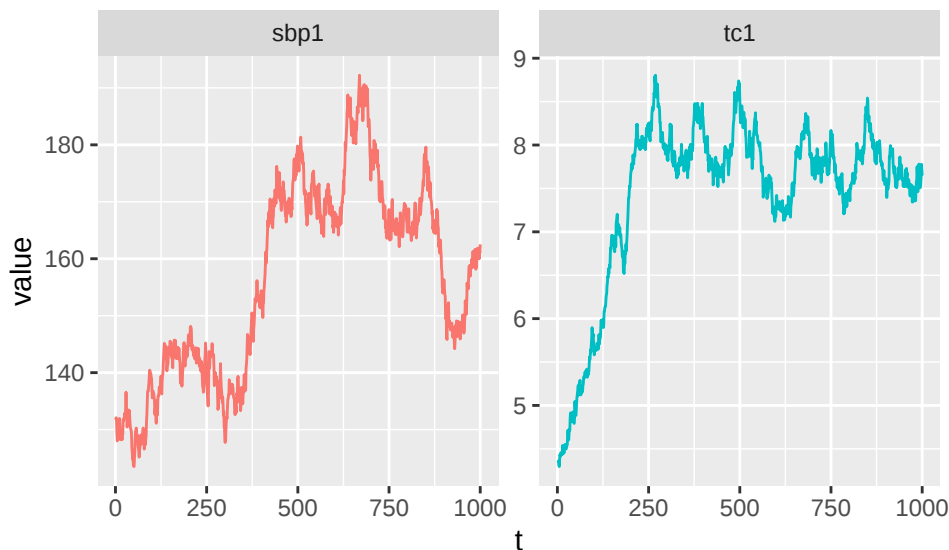


Figure 2: *Correlated geometric random walk trajectory for SBP & TC of a single patient*

Example 3: timed events & multiple transitions

Sometimes, it may be necessary to represent ‘deterministic’ events, which happen with certainty in response to some change. These are dealt with in **golden** by allowing hazard values to be `Inf`, which guarantees events will occur.

As a simple example, let’s suppose we want to put people on ‘treatment’ if their SBP exceeds 150 or their TC exceeds 4.5. First, let’s introduce some new variables to our population to describe related states:

```

## extra variables
pop0[, on_treatment := 0L]      # time on treatment: 0 means not on treatment
pop0[, treatment_counter := 0L] # counter for treatment initiation

```

We will say we want treatment to start with certainty if the SBP/TC condition above is met, and to finish

after 5 steps. This means defining hazard functions that become Inf when the relevant conditions are met:

```
## definitely go on treatment if sbp >= 150 or tc >= 4.5
treatment_start_haz <- function(death, on_treatment, sbp, tc) {
  ifelse(death > 0 | on_treatment > 0 | (sbp < 150 & tc < 4.5), 0, Inf)
}

## definitely end treatment after 5 steps if alive
treatment_end_haz <- function(death, on_treatment) {
  ifelse(death > 0 | on_treatment < 5, 0, Inf)
}
```

Of course we will also need a trajectory rule to track time-on-treatment, and add this to our list of trajectories:

```
## treatment trajectory
treatment_time_update <- function(death, on_treatment) {
  ifelse(death < 0 & on_treatment > 0, on_treatment + 1, 0)
}

## trajectory list
trajlist3 <- list(
  ## age
  age_traj,
  ## SBP & TC handled in bivariate fashion
  bptc_traj,
  ## new trajectory for treatment
  new_trajectory(
    fn = treatment_time_update,
    args = c("death", "on_treatment"),
    states = "on_treatment"
  )
)
```

And we will need to define the transitions associated with starting and finishing treatment:

```
## start off
treatment_start_transition <- function(on_treatment) {
  rep(1, length(on_treatment))
}

## finish off
treatment_end_transition <- function(on_treatment) {
  rep(0, length(on_treatment))
}
```

These will be included in a list of transitions associated with the hazard for starting treatment:

```
hazlist3 <- list(
  ## death
  morthaz0,
  ## starting treatment
  new_hazard(
    fn = treatment_start_haz,
    args = c("death", "on_treatment", "sbp", "tc"),
    ## (which triggers two transitions)
    list(
      new_transition(
```

```

    fn = treatment_start_transition,
    args = "on_treatment",
    states = "on_treatment"
  ),
  new_transition(
    fn = generic_update,
    args = "treatment_counter",
    states = "treatment_counter"
  )
),
## finishing treatment
new_hazard(
  fn = treatment_end_haz,
  args = c("death", "on_treatment"),
  ## (triggering one transition)
  new_transition(
    fn = treatment_end_transition,
    args = "on_treatment",
    states = "on_treatment"
  )
)
)

```

Note: as with trajectories, we can code transitions to act on multiple rows by defining functions that return lists of vectors, and specifying which columns these are via a vector of column names in `new_transition`. This could allow for things like correlated random transitions, but more often may be a more convenient way to specify things. E.g. in the above one could have defined a single `treatment_start_transition` that both turned on treatment and incremented the treatment counter, as opposed to defining these separately and grouping them with the `treatment_start_haz` using a list.

For this example, let's add a time series capturing the number on treatment at each time to the history:

```

history3 <- new_history(
  columns = list(
    new_column("no. alive", length, "age", filter_alive, "death"),
    ## alive and on treatment:
    new_column(
      name = "no. alive on tx",
      fn = function(x) sum(x > 0),
      args = "on_treatment",
      filter_fn = filter_alive,
      filter_args = "death"
    ),
    ## an individual's value:
    new_column("sbp1", sum, "sbp", filter_1, "id"),
    new_column("tc1", mean, "tc", filter_1, "id")
  ),
  frequency = 1
)

```

Finally, let's assemble the parameters and run the model:

```

## full parameters
parms3 <- new_parameters(
  hazards = hazlist3,

```

```

trajectories = trajlist3,
steps = 20, # longer
debug = FALSE,
history = history3
)

## run the simulation
result3 <- run_simulation(pop0, parms3) # look at a limited population
#> Simulation Complete in 0.05 seconds!
result3$pop
#>      id death   age    sbp      tc on_treatment treatment_counter
#>      <int> <num> <num>   <num>   <num>         <num>         <num>
#> 1:      1    -1    60 134.2391 4.478562           0           0
#> 2:      2    11    51 137.9237 4.881861           0           3
#> 3:      3     9    49 148.8540 4.585957           0           2
#> 4:      4     7    47 126.8567 4.582231           0           2
#> 5:      5     0    40 143.0363 4.566057           0          20
#> ---
#> 9996: 9996    -1    60 136.6982 4.515968           0           4
#> 9997: 9997     0    40 125.8770 4.487413           0           0
#> 9998: 9998     0    40 139.6489 4.529636           0          20
#> 9999: 9999     2    42 140.8224 4.525010           0           1
#> 10000: 10000   -1    60 145.7149 4.419947           0           4

```

You can see the number of times patients have started treatment in the final population data. This is an example of recording a cumulative variable, and this approach could be used to capture cumulative (discounted) costs etc in health economic applications.

Let's inspect the numbers alive and on treatment:

```

plot_data <- melt(result3$history[, .(
  t = `~STEP`,
  `no. alive`, `no. alive on tx`
)], id = "t")

ggplot(plot_data, aes(t, value, col = variable)) +
  geom_line() +
  geom_point() +
  theme(legend.position = "top")

```

This is a very artificial example, but one can see the cyclical pattern with people finishing treatment roughly in sync after 5 steps before restarting; faster evolution of SBP/TC would probably soften this synchronicity.

A more realistic example: globorisk

For a slightly more realistic application of these principles, we will consider globorisk cardiovascular disease (CVD) model, which commendably has an R package including available data to reconstruct baseline hazard for combined CVD event risk. We include data for the US from this package, along with body mass index (BMI) estimates based on NCD-RisC and population demographic data from the UN World Population Prospects 2024 (WPP2024) estimates, courtesy of the R package wpp2024.

Initial population

First, let's construct a population. We provide in the package population demographic structure estimates for the US in 2023: we write a short function to enable sampling a cohort to match these patterns, and apply

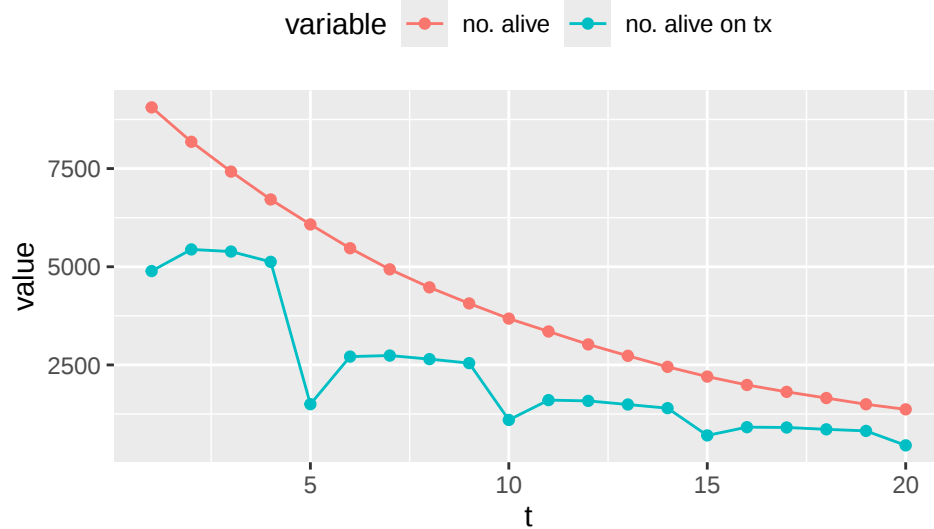


Figure 3: *Numbers alive and on treatment*

it to create our initial base cohort, including: age, sex, death, and a CVD event counter:

```
## population structure data
head(pop_snapshot)
#>      popM      popF
#> [1,] 2014.861 1925.018
#> [2,] 1957.782 1869.524
#> [3,] 1943.898 1857.020
#> [4,] 1923.427 1835.382
#> [5,] 1936.948 1846.899
#> [6,] 1955.784 1859.816

## function to sample a population using above data
make_cohort <- function(N) {
  popcounts <- rmultinom(1, size = N, prob = c(pop_snapshot))
  sexes <- rep(1, sum(popcounts[1:nrow(pop_snapshot)]))
  sexes <- c(sexes, rep(0, N - length(sexes)))
  initPop <- data.table(
    male = as.integer(sexes),
    age = as.integer(0),
    death = as.integer(rep(-1, N)),
    cvd_count = as.integer(rep(0, N))
  )
  ## do ages
  ageref <- rep(0:(nrow(pop_snapshot) - 1), 2)
  k <- 1
  for (i in 1:length(popcounts)) {
    if (popcounts[i] > 0) {
      initPop[k:(popcounts[i] + k - 1), age := ageref[i]]
      k <- k + popcounts[i]
    }
  }
  initPop
}
```

```
## sample the initial population
initPop <- make_cohort(N)
```

We need to do a bit of extra work to facilitate merging with other data. We add age categories to merge in BMI fit data, and then sample a distribution of BMIs. Then we add the globorisk age categories and merge in central values of SBP and TC:

```
## age categories used in the BMI fits data
initPop[, acat := fcase(
  age >= 25 & age < 30, "25-29",
  age >= 30 & age < 35, "30-34",
  age >= 35 & age < 40, "35-39",
  age >= 40 & age < 45, "40-44",
  age >= 45 & age < 50, "45-49",
  age >= 50 & age < 55, "50-54",
  age >= 55 & age < 60, "55-59",
  age >= 60 & age < 65, "60-64",
  age >= 65 & age < 70, "65-69",
  age >= 70 & age < 75, "70-74",
  age >= 75 & age < 80, "75-79",
  age >= 80 & age < 85, "80-84",
  age >= 85, "85plus",
  default = "20-24"
)]

## add male convention to BMI fits data
bmi_fits$male <- ifelse(bmi_fits$sex == "Men", 1, 0)

## merge in BMI fits data
initPop <- merge(
  initPop,
  bmi_fits[, .(male, acat, k, theta)],
  by = c("male", "acat"),
  all.x = TRUE, all.y = FALSE
)

## sample some BMIs
initPop[, bmi := rgamma(nrow(initPop), shape = k, scale = theta)]

## add globorisk age categories
initPop[, agec := as.integer(ifelse(age < 85, trunc(age / 5) - 7, 10))]
initPop[agec < 1, agec := 1]
initPop[agec > 9, agec := 9]

## merge in central SBP and TC for US from globorisk
initPop <- merge(
  initPop,
  globorisk_rf[, .(agec, male = sex, sbp = 10 * mean_sbp, tc = mean_tc)],
  by = c("agec", "male")
)
```

We can check this is looking plausible:

```
## initial pop
ggplot(initPop, aes(x = age, fill = factor(male), group = male)) +
  geom_histogram() +
  theme(legend.position = "top")
#> `stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```

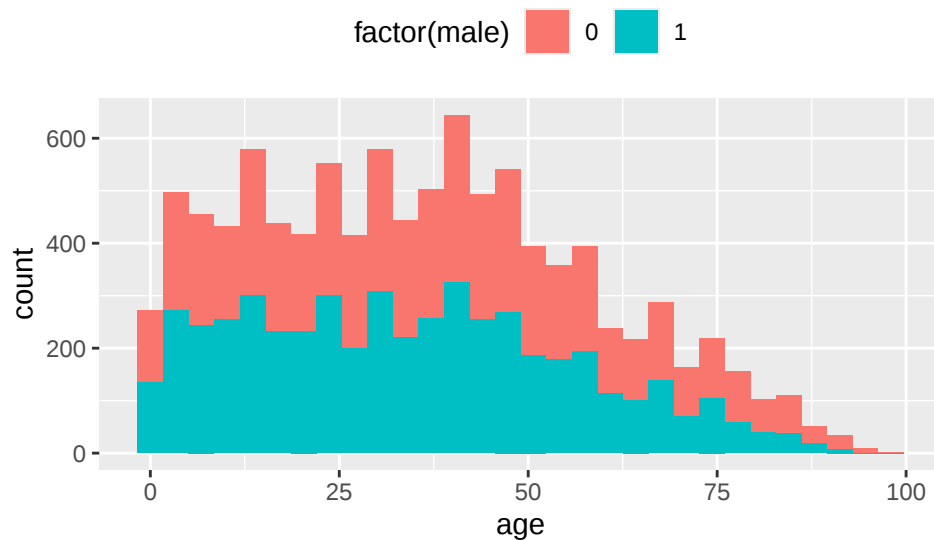


Figure 4: Initial population demography

Hazards

We create a mortality hazard function that looks up death rates in a given year, age, and sex from WPP2024 data:

```
## data prep
n_ages <- 101
n_years <- nrow(lifetable_data) / n_ages
qx_array <- array(0, dim = c(2, n_ages, n_years))
qx_array[1, , ] <- lifetable_data$mxM
qx_array[2, , ] <- lifetable_data$mxF

## mortality hazard function
mort_fn <- function(sex, age, year) {
  ## Convert to 1-indexed and clamp in bounds
  d <- dim(qx_array)
  n_rows <- d[2]
  n_cols <- d[3]
  row_index <- pmin(pmax(age + 1, 1), n_rows)
  col_index <- pmin(pmax(year + 1, 1), n_cols)
  qx_array[cbind(sex + 1, row_index, col_index)] # zero index female
}

## test
mort_fn(rep(1, 10), rep(50, 10), rep(70, 10))
#> [1] 0.00157482 0.00157482 0.00157482 0.00157482 0.00157482 0.00157482 0.00157482
#> [7] 0.00157482 0.00157482 0.00157482 0.00157482 0.00157482 0.00157482 0.00157482
```

The we write a hazard for CVD events that depends on updated age, sex, BMI, SBP and TC based on the globorisk R package.

```
globohaz <- function(death, sex, age, bmi, sbp = 140, tc = 4.5) {
  n <- length(sex)
  if (length(sex) != length(age) || length(sex) != length(bmi)) {
    stop("Arguments must be equal length!\n")
  }
  ## checks/additions
  age <- pmin(pmax(age, 41), 75)
  dm <- rep(0, n)
  smk <- rep(0, n)
  ## globorisk transforms
  agec <- as.integer(ifelse(age < 85, trunc(age / 5) - 7, 10))
  sbp <- sbp / 10
  dm <- as.integer(dm)
  smk <- as.integer(smk)
  bmi <- bmi / 5
  ## center
  agc_sex <- paste(agec, sex, sep = "_")
  sbp_c <- sbp - globorisk_rf[agc_sex][, mean_sbp]
  tc_c <- tc - globorisk_rf[agc_sex][, mean_tc]
  dm_c <- dm - globorisk_rf[agc_sex][, mean_dm]
  smk_c <- smk - globorisk_rf[agc_sex][, mean_smk]
  bmi_c <- bmi - globorisk_rf[agc_sex][, mean_bmi]
  ## compute hazard ratios
  HR <- sbp_c * globorisk_coefs[["main_sbpc"]] +
    bmi_c * globorisk_coefs[["main_bmi5c"]] +
    smk_c * globorisk_coefs[["main_smokc"]] +
    sex * smk_c * globorisk_coefs[["main_sexsmokc"]] +
    age * sbp_c * globorisk_coefs[["tvc_sbpc"]] +
    age * smk_c * globorisk_coefs[["tvc_smokc"]] +
    age * bmi_c * globorisk_coefs[["tvc_bmi5c"]]
  HR <- exp(HR) # un-log
  ## baseline hazard
  h <- globorisk_cvdr[agc_sex][, cvd_0]
  ## return
  ans <- h * HR
  ans[death > 0] <- 0 # no effect on dead
  ans[!is.finite(ans)] <- 0 # safety
  ans
}

## test: note returns vector
globohaz(
  c(-1, -1, 1), #alive
  c(0, 1, 1), #sex
  c(00, 50, 90), #age
  rep(25, 3), #BMI
  rep(145, 3), #SBP
  rep(5, 3) #TC
)
#> [1] 0.003613247 0.004592897 0.000000000
```

We combine these to create a list of hazards. CVD events will just increment the `cvd_count` variable, but in

more realistic applications would trigger costs and complications.

```
## hazards
hazlist <- list(
  ## CVD event
  new_hazard(
    fn = globohaz,
    ## input args for hazard
    args = c("death", "male", "age", "bmi", "sbp", "tc"),
    transitions = new_transition(
      fn = generic_update,      # transition function for event
      args = "cvd_count",      # input args for transition
      states = "cvd_count"     # vars affected by transition
    )
  ),
  ## deaths
  new_hazard(
    fn = mort_fn,
    args = c("male", "age", "~STEP"),
    transitions = new_transition(transition_fn, c("death", "~STEP"), "death")
  )
)
```

Trajectories & history

We already have trajectories for age, and (SBP and TC) from above, but would like to consider BMI as something that changes with age. We will make the simplistic assumption that we can use the cross-sectional pattern seen in our baselin population as a way to evolve BMI, and based this on a regression. *Note* there is no problem with using `predict` methods from regressions to define hazards or trajectories:

```
## linear model of BMI vs age in our data
mm <- lm(bmi ~ 1 + age + I(age^2),
  data = initPop
) # regression

## pretend that cross-sectional BMIs are a reasonable way to inform trajectories
bmi_update <- function(age) {
  predict(mm, newdata = data.table(age = age))
}

bmi_update(25) # test
#>      1
#> 28.91341
```

We group these into a list of trajectories, creating one for BMI based on the update function as we go:

```
## trajectories
trajlist <- list(
  ## age
  age_traj,
  ## BMI
  new_trajectory(bmi_update, "age", "bmi"),
  ## SBP & TC handled in bivariate fashion
  bptc_traj      #trajectory fn
)
```

Finally, we will monitor the living population size, average age (as a check), and the average number of CVD events:

```
history <- new_history(
  columns = list(
    new_column("no. alive", length, c("age"), filter_alive, c("death")),
    new_column("av. age alive", mean, c("age"), filter_alive, c("death")),
    new_column("av. cvd events", mean, c("cvd_count"), filter_alive, c("death"))
  ),
  frequency = 1
)
```

Running & plotting

Having defined our *hazards*, *trajectories*, and *history*, we can now create our parameter object and run the model:

```
parms <- new_parameters(
  hazards = hazlist,
  trajectories = trajlist,
  steps = n_years,
  debug = FALSE,
  history = history
)

## run the simulation
ret <- run_simulation(initPop, parms)
#> Simulation Complete in 1.91 seconds!
#> <golden_timing>
#>
#> Hazards:
#> Hazard Total Time (s) Avg Time (s) % Runtime
#> <char> <num> <num> <num>
#> 1: globohaz 1.485 0.0147029702 77.5862038
#> 2: mort_fn 0.011 0.0001089109 0.5747126
#>
#> Transitions:
#> Transition Total Time (s) Avg Time (s) % Runtime
#> <char> <num> <num> <num>
#> 1: generic_update 0.000 0.000000e+00 0.0000000
#> 2: transition_fn 0.008 7.920792e-05 0.4179728
#>
#> Columns:
#> Column Total Time (s) Avg Time (s) % Runtime
#> <char> <num> <num> <num>
#> 1: no. alive 0 0 0
#> 2: av. age alive 0 0 0
#> 3: av. cvd events 0 0 0
```

Note for the first time in these examples, the timing information is potentially useful: it shows that **globohaz** is where much of the time is going for this simulation, and if run time were an issue suggests where efforts to improve efficiency should be directed.

Let's have a look at a few of the outputs.

Have we run the model long enough that most people have died?

```
## history: number alive
ggplot(ret$history, aes(`~STEP`, `no. alive`)) +
  geom_line()
```

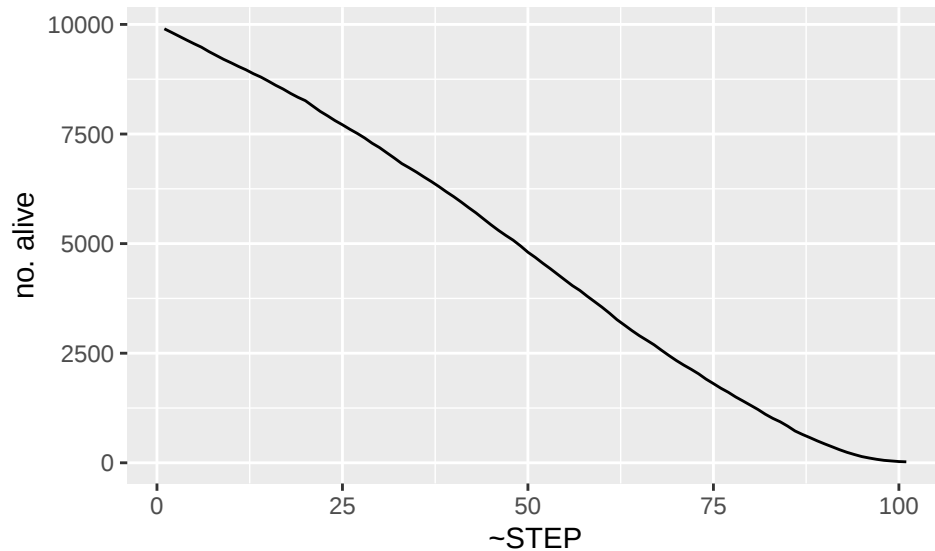


Figure 5: *Living cohort size over time*

What age did people die at?

```
## now dead
ggplot(
  ret$pop[death > 0],
  aes(x = age, fill = factor(male), group = male)
) +
  geom_histogram() +
  theme(legend.position = "top")
#> `stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```

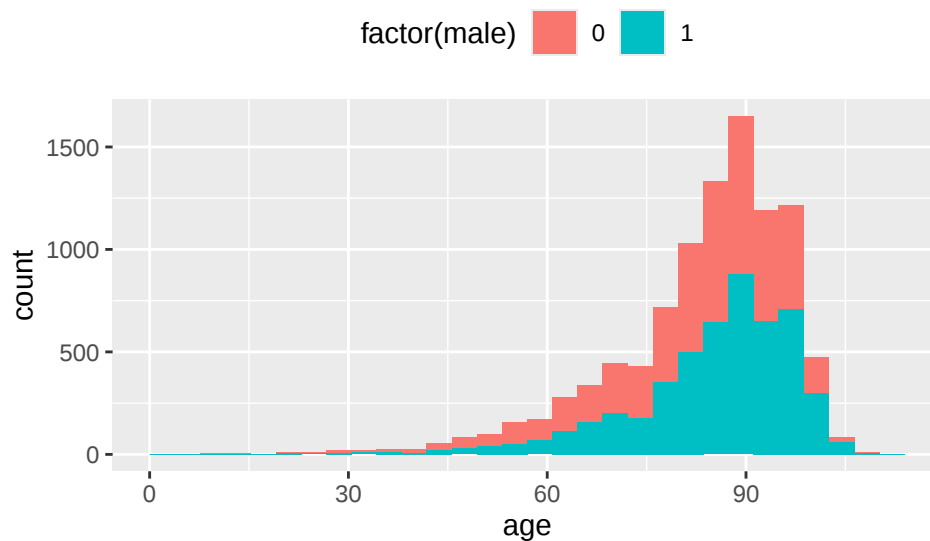


Figure 6: *Age at death*

How many CVD events have those alive typically experienced?

```
## history: mean CVD count
ggplot(ret$history, aes(`~STEP`, `av. cvd events`)) +
  geom_line()
```

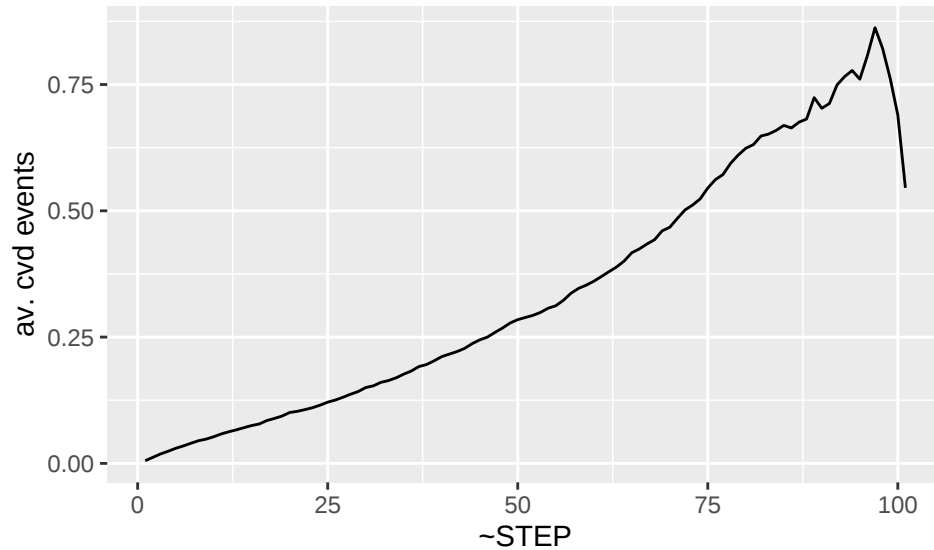


Figure 7: Average CVD event count in survivors

The third member of the output list provides `data.table` timing information, which will print like this:

```
ret$timing
#> <golden_timing>
#>
#> Hazards:
#>      Hazard Total Time (s) Avg Time (s) % Runtime
#>      <char>          <num>      <num>      <num>
#> 1: globohaz          1.485 0.0147029702 77.5862038
#> 2: mort_fn           0.011 0.0001089109  0.5747126
#>
#> Transitions:
#>      Transition Total Time (s) Avg Time (s) % Runtime
#>      <char>          <num>      <num>      <num>
#> 1: generic_update      0.000 0.000000e+00 0.0000000
#> 2: transition_fn       0.008 7.920792e-05 0.4179728
#>
#> Columns:
#>      Column Total Time (s) Avg Time (s) % Runtime
#>      <char>          <num>      <num>      <num>
#> 1: no. alive          0          0          0
#> 2: av. age alive      0          0          0
#> 3: av. cvd events      0          0          0
```

The intent here is to provide some information on where the simulation is spending most of its time to guide any optimizations. This information can also be printed automatically at the end of a run by setting the flag `print_timing = TRUE` (which is on by default) when creating `new_parameters`. Note though that this information is only printed in interactive mode, but not, e.g., when generating markdown reports.

Debug

The `debug` flag passed to `run_simulation()`, which defaults `True`, enables potentially expensive runtime validation. For example if maths within your model returns `NaN`, it will likely propagate and cause dependent values to also equal `NaN`. This behaviour can be difficult to track back to the source. When `debug=True` `NaN` and similar erroneous return values will be detected at return and raise a corresponding exception. Once a model is deemed stable, `debug=False` can be enabled to remove any performance overhead of expensive runtime validation. *No benchmarking has been carried out to estimate the impact of the `debug` flag.*

Modelling interdependence and constraints

While this was not the motivation for the package, it is possible to model interdependence between patients (rows), for example infection processes, or constrained resources (e.g. hospital beds) that may affect multiple patients simultaneously.

Fundamentally, this is possible because our functions can aggregate over the columns/vectors that are their inputs, allowing individual rates or events to be influenced by the state of other rows. So, e.g., resources could be modelled with explicit columns, with potentially duplicated data (e.g. a bed count that is the same in each row). Resources could also be represented by rows: there is no reason rows should only represent patients, although not all columns would necessarily be applicable across very different types of represented object. The latter approach may be better-suited to large numbers of separate resources, but would require care to avoid introducing overheads when querying the relevant rows in determining their values.

Below, we illustrate a simple example of interdependence by implementing a transmission model of an infectious disease, where the rate of infection depends on the overall state of the population.

An SIR model

In this section, we implement a simple SIR (Susceptible-Infected-Recovered) model to illustrate how interdependence can be modeled: in this case by the dependence of infection rates on the overall infection prevalence in the population. In this model, individuals can transition between three states: susceptible, infected, and recovered. We represent these states by the integers 1, 2, and 3, respectively. Since individuals progress through these states 1->2->3, we need a function to advance their state.

```
advance <- function(x){  
  x <- x + 1  
  x  
}
```

We'll also need a history function to record these states over time:

```
prevs <- new_history(  
  list(  
    new_column(  
      name = "Susceptible",  
      fn = function(x) mean(x == 1),  
      args = "state"  
    ),  
    new_column(  
      name = "Infected",  
      fn = function(x) mean(x == 2),  
      args = "state"  
    ),  
    new_column(  
      name = "Recovered",  
      fn = function(x) mean(x == 3),  
      args = "state"  
    )  
  )  
)
```

```
)
)
```

We'll have a recovery rate that only applies to those in state 2 (Infected):

```
## recovery hazard
recreate <- function(x) {
  ans <- (x == 2) * 1 / 7 # only applies to I
  ans
}

## create relevant hazard object
recover_hazard <- new_hazard(
  fn = recreate, # function returning hazard
  args = "state", # arguments for hazard function
  transitions = new_transition( # creates a transition
    fn = advance, # 1->2->3
    args = "state", # args for transition fn
    states = "state" # col transition acts on
  )
)
```

Then we need an infection process that will be driven by the prevalence of infection in the cohort:

```
## force of infection hazard
foi <- function(x) {
  R0 <- 2
  beta <- R0 / 7 # R0 * recovery rate
  foi <- beta * mean(x == 2) # beta x prevalence
  foi * (x == 1) # only applies to S
}

## as hazard object
infect_hazard <- new_hazard(
  fn = foi, # function returning hazard
  args = "state", # arguments for hazard function
  transitions = new_transition( # creates a transition
    fn = advance, # 1->2->3
    args = "state", # args for transition fn
    states = "state" # col transition acts on
  )
)
```

Now we can gather these into a parameter object and run:

```
## parameter object
parms <- new_parameters(
  hazards = list(
    infect_hazard,
    recover_hazard
  ),
  steps = 100,
  history = prevs,
  debug = FALSE
)
```

```
## initial population
N <- 1e3L
pop <- data.frame(id = seq_len(N), state = rep(1, N))
pop[1:10, "state"] <- 2 #10 infected individuals

## run the simulation
result <- run_simulation(pop, parms)
#> Simulation Complete in 0.01 seconds!
```

And then, finally, plot:

```
## plot the results
matplot(result$history$`~STEP`, result$history[, -1], type = "l", lty = 1, lwd = 2)
legend("topright", legend = names(result$history)[-1], col = 1:3, lty = 1, lwd = 2)
```

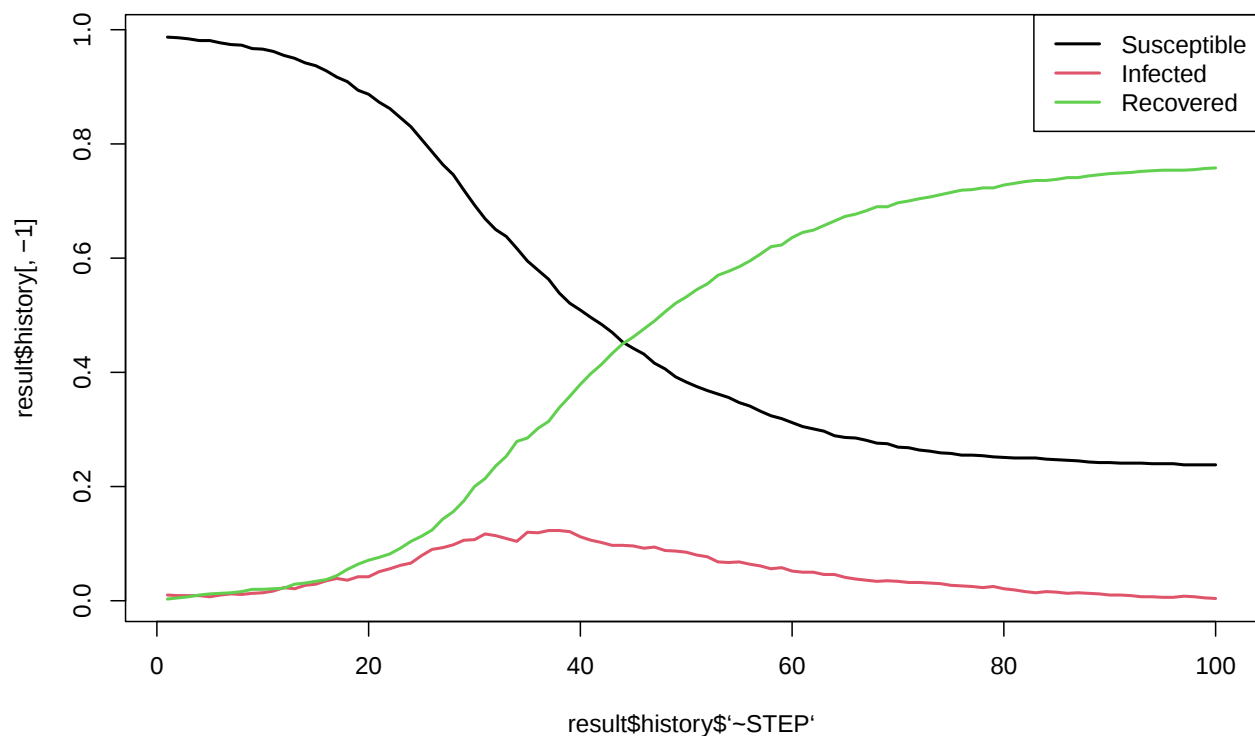


Figure 8: *SIR dynamics*