New Network Models with EpiModel

This tutorial documents how to use EpiModel to solve new types of stochastic network models. New stochastic models require specifying new module functions. These functions may either replace the existing modules routinely called in `netsim`, or they may supplement them by adding new processes into the chain of operations at each time step in the simulation.

The Modular Framework

Modules for network model simulations are specified in the `control.net` helper function. Within that function, modules are identified by the `.FUN` parameters, the input of which is a standardized R function described below. Users can replace existing modules, for example `births.FUN` by writing a new function for births and passing it into `control.net`.

```
function (type, nsteps, start = 1, nsims = 1, ncores = 1, depend,
    rec.rand = TRUE, b.rand = TRUE, d.rand = TRUE, tea.status = TRUE,
    attr.rules, epi.by, use.pids = TRUE, pid.prefix, initialize.FUN = initialize.net,
    deaths.FUN = deaths.net, births.FUN = births.net, recovery.FUN = recovery.net,
    edges_correct.FUN = edges_correct, resim_nets.FUN = resim_nets,
    infection.FUN = infection.net, get_prev.FUN = get_prev.net,
    verbose.FUN = verbose.net, module.order = NULL, set.control.stergm,
    save.nwstats = TRUE, nwstats.formula = "formation", delete.nodes = FALSE,
    save.transmat = TRUE, save.network = TRUE, save.other, verbose = TRUE,
    verbose.int = 1, skip.check = FALSE, ...)
NULL
```

Each of the arguments listed with a `.FUN` suffix is a pointer to tell `netsim` which function to use to simulate those processes. Each call to `netsim` begins by initializing the simulation with the steps specified in `initialize.FUN`, the function for which is `initialize.net`.

Additional modules may be added into this mix by setting arguments in `control.net` with names ending in `.FUN` but not among those listed. These new functions are passed through into the `control.net` through the `...` argument. We provide examples of building new modules below.

Module Structure

To develop new module functions, one must understand the structure of the objects created internally during a call to `netsim`. Some of the information in these data is returned at the end of the call as part of the output (within the returned object of class `netsim`), while some other is used only internally and discarded. Some of the features that are helpful to know include:

1. The main data structure storing the information passed among the modules is `dat`. Every module has only two input arguments: `dat` and `at`, the current time step. All the inputs, such as parameters, and the outputs, such as summary epidemiological statistics, are contained on `dat`. Each module reads `dat`, updates internal data within `dat`, and then outputs `dat` for the next module to work on.
2. Attributes of individuals in the module are stored in a sublist to `dat` called `attr`. This is a named list, with each element of the list as a separate attribute in a vector. All vectors are exactly the same length and correspond to one attribute per individual in the population. `netsim` always creates five attributes by default: `active`, `status`, and `infTime`, `entrTime`, and `exitTime`. The `active` attribute keeps track of whether an individual is currently alive in the the population or has left it through some process such as death or out-migration. The `status` attribute indicates current infection status as a character ("s", "i", "r", for susceptible, infected, and recovered); `infTime` records the time at which each individual was infected (`NA` for susceptible nodes). The `entrTime` and `exitTime` contain the time steps at which the person enters or leaves the population.
3. Summary model statistics are contained in a sublist to `dat` called `epi`. This list stores information on the sizes of the population with each disease status at each time point, as well as the size of flows among those attribute states. The default approach for built-in models is to calculate summary *prevalence* statistics, such as the size or frequency of infection in the population, in the `get_prev.FUN` module

as the last step in each time step; summary *incidence* statistics are typically calculated within the modules in which the event, such as births or infections, has occurred. But for simplicity, it is possible to embed all summary statistics within an action module.

4. In EpiModel time loops, the initial network is coded as time 1. The only module that is run at this time step is the `initialize.FUN` module for initialization of things like network structure or disease status. All other functions will start running at time 2 and again for the number of time steps specified in `control.net`. In the the module examples below, you will notice that some data are created at time step 2 and then later continually updated for the following time steps: the coding is of the form `if (at == 2) {create something for time steps 1 and 2} else {edit that thing for time steps 3+}`. That is a shorthand way of creating new data structures without having the edit the initialization module, which is a little unwieldy (it handles a lot of tasks). It is your choice where and when to create and update data structures within `dat`.
5. `netsim` handles attributes differently depending on whether they are embedded within the formation formula for the ERGM that was estimated in `netest`. By default, network attributes that are contained within the formula will automatically be transferred to `dat$attr`, but left at their fixed state for the duration of the simulation; the except is the attribute `status`, which of course can vary over time as a function of disease transmission. Passing an attribute on the initial network as a vertex attribute but not using within the ERGM will result in that attribute being removed from the network object. Therefore, attributes that are necessary for the epidemic mechanics but not the ERGM model (for example, age in Example 1 below) must be initialized on the `dat` network structure independent of the network object passed in through the output of `netest` in the `est` argument to `netsim`.

The examples below provide practice with this functionality. You can also explore these features and more both by reading through the code for `netsim` and for the other functions that it calls, or by stepping through a call to `netsim` using a browser tool like `debug`. Examining the code through the debugger works beautifully and seamlessly in an IDE like Rstudio.

Example 1: A Different Demographic Approach

In this example, we show how to build in an aging process, age-specific mortality, and a constant growth model for births. This will require one *new* module function and two *replacement* module functions, along with the needed parameters.

Aging Module

To introduce aging into a model, we need to write a new module function, here called `aging`, that performs the necessary processes. Writing this illustrates some key requirements of any module added to `netsim`. First, to describe what the function actually does: at the initial time step in the loop, which is 2 for the reasons noted above, persons in the population are randomly assigned an age between 18 and 49 years. At the subsequent time steps, their age is incremented by one month, as our time unit for the simulation is a month.

```
aging <- function(dat, at) {

  ## Attributes
  if (at == 2) {
    n <- sum(dat$attr$active == 1)
    dat$attr$age <- sample(18:49, n, replace = TRUE)
  } else {
    dat$attr$age <- dat$attr$age + 1/12
  }

  ## Summary statistics
  if (at == 2) {
    dat$epi$meanAge <- rep(mean(dat$attr$age, na.rm = TRUE), 2)
  } else {
    dat$epi$meanAge[at] <- mean(dat$attr$age, na.rm = TRUE)
  }

  return(dat)
}
```

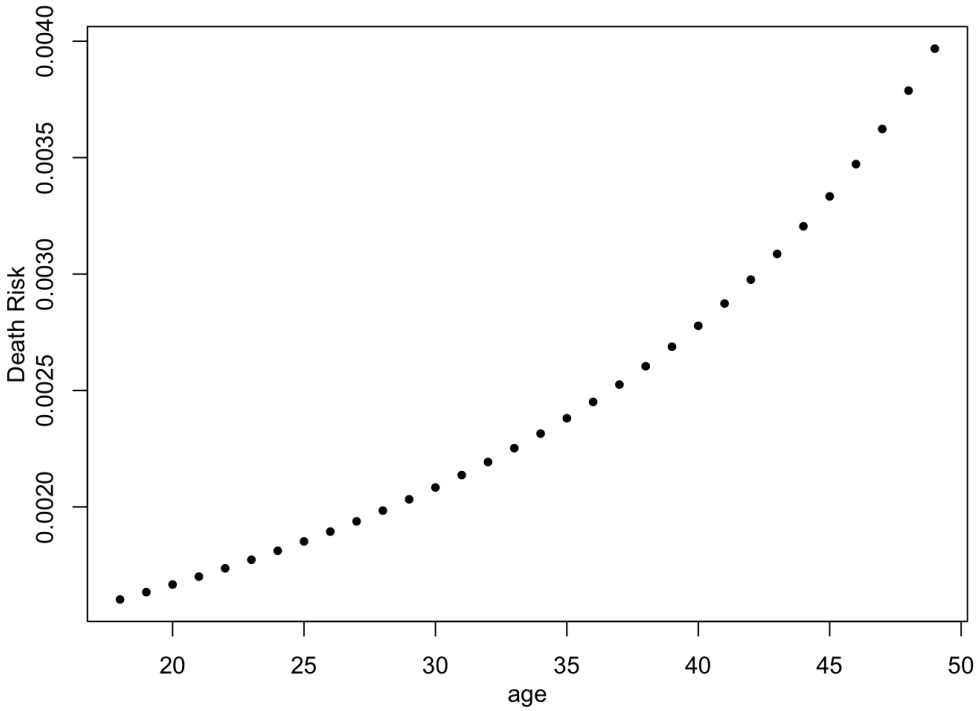
As described above, all of the modules in `netsim` use the same two functional arguments, `dat`, and `at`. The former is the master data object that carries around individual attributes, summary outcomes, and other data to be output at the end of the simulation. All individual-level attributes are saved in the `dat` object in a sublist called `attr`. The `active` vector of attributes is a binary attribute indicating whether each person is active or not. Therefore, `n` in time step 2 queries the size of the active population so the `sample` function knows how many ages to produce. Everyone's age is stored on that `attr` list in a new vector called `age`. That vector is modified for everyone at each subsequent time step. We can add summary statistics to any module by saving them onto `dat$epi`. Here, we create a vector called `meanAge` at time step 2 that contains the mean of everyone's age in the population; at time steps 3 and onward, a new summary statistic and appended on that vector.

Death Module

Whereas the aging module defined above is an original process added to `netsim` that did not exist in built-in models, death already has a module. In this case, we will replace the existing death module for susceptibles with our new one. In the existing module, the probability of death is based on a fixed risk that may vary by disease status; the parameters that control this, `ds.rate` and `di.rate` and so on, are input in the

parameters set through `param.net`. Here, we will replace approach with a simple death module that takes advantage of our new age attribute: we will model the probability of death as a nonlinear function of increasing age, like this:

```
ages <- 18:49
death.rates <- 1/(70*12 - ages*12)
par(mar = c(3.2, 3.2, 1, 1), mgp = c(2, 1, 0))
plot(ages, death.rates, pch = 20, xlab = "age", ylab = "Death Risk")
```



Since we are using monthly time units, the death rates need to be specified with that same scale. Our life expectancy will be 70 years: each month closer to that age one gets, the risk of death increases. This is not necessarily meant to fit a real-world demographic life table but the age-specific mortality rates generally *do* follow a similar form.

In our death module, life expectancy will be a variable parameter, with the death rate calculation to be set in the module function. Who dies at any time step will be based on random draws from binomial distributions which probabilities equal to their age-specific risks.

```
dfunc <- function(dat, at) {

  # Parameters
  idsElig <- which(dat$attr$active == 1)
  nElig <- length(idsElig)
  nDeaths <- 0

  # Processes
  if (nElig > 0) {
    ages <- dat$attr$age[idsElig]
    life.expt <- dat$param$life.expt
    death.rates <- pmin(1, 1/(life.expt*12 - ages*12))
    vecDeaths <- which(rbinom(nElig, 1, death.rates) == 1)
    idsDeaths <- idsElig[vecDeaths]
    nDeaths <- length(idsDeaths)

    # Update nodal attributes on attr and networkDynamic object
    if (nDeaths > 0) {
      dat$attr$active[idsDeaths] <- 0
      dat$attr$exitTime[idsDeaths] <- at
      dat$nw <- deactivate.vertices(dat$nw, onset = at, terminus = Inf,
                                    v = idsDeaths, deactivate.edges = TRUE)
    }
  }

  # Summary statistics
  if (at == 2) {
    dat$epi$d.flow <- c(0, nDeaths)
  } else {
    dat$epi$d.flow[at] <- nDeaths
  }

  return(dat)
}
```

The IDs of those eligible for the transition are determined by their current value of the active variable, again held in the `attr` list on `dat`. If there are any persons eligible, then the process will proceed. The ages of only those eligible are abstracted from the `attr` list. The life expectancy parameter is queried from the parameter list, `param`, containing the parameters passed into the model from `param.net` as described below. An individual death rate is calculated for each person based on their age and that life expectancy. With those rates, a Bernoulli coin flip is performed for each death-eligible person. If there are any deaths, the active attribute must be toggled from 1 to 0 for those people.

Within the network object, located at `dat$nw`, each of the dying nodes are “deactivated” from the network. This process removes their existing edges, if they are currently within a partnership, and disallows them from having future partnerships. Similar to our aging module, we count the number of deaths that have occurred within the time step and save that as a flow size in the `d.flow` vector within the `epi` sublist. At time step 2, that vector is created; at time steps 3 and onward, the summary statistic is appended onto that vector.

Birth Module

The built-in birth module for `netsim` simulates the number of new births at each time step as a function of a fixed rate, contained in the `b.rate` parameter specified in `param.net`, and the current population size. In the case of bipartite models, it is possible to specify that the size of the first group only should be considered (e.g., in case that mode 1 represents women in the population). In this example, we will use a different approach by using a constant growth model: the size of the population is expected to grow linearly at each time step based on a fixed growth rate at the outset of the simulation. Therefore, the number of new births that must be generated at each time step is the difference between the expected population size and the current population size.

```
bfunc <- function(dat, at) {

  # Variables
  growth.rate <- dat$param$growth.rate
  exptPopSize <- dat$epi$num[1] * (1 + growth.rate * at)
  n <- network.size(dat$nw)
  tea.status <- dat$control$tea.status

  numNeeded <- exptPopSize - sum(dat$attr$active == 1)
  if (numNeeded > 0) {
    nBirths <- rpois(1, numNeeded)
  } else {
    nBirths <- 0
  }
  if (nBirths > 0) {
    dat$nw <- add.vertices(dat$nw, nv = nBirths)
    newNodes <- (n + 1):(n + nBirths)
    dat$nw <- activate.vertices(dat$nw, onset = at, terminus = Inf, v = newNodes)
  }

  # Update attributes
  if (nBirths > 0) {
    dat$attr$active <- c(dat$attr$active, rep(1, nBirths))
    dat$attr$status <- c(dat$attr$status, rep("s", nBirths))
    dat$attr$infTime <- c(dat$attr$infTime, rep(NA, nBirths))
    dat$attr$entrTime <- c(dat$attr$entrTime, rep(at, nBirths))
    dat$attr$exitTime <- c(dat$attr$exitTime, rep(NA, nBirths))
    dat$attr$age <- c(dat$attr$age, rep(18, nBirths))
    if (tea.status == TRUE) {
      dat$nw <- activate.vertex.attribute(dat$nw, prefix = "testatus",
                                          value = 0, onset = at,
                                          terminus = Inf, v = newNodes)
    }
  }

  # Summary statistics
  if (at == 2) {
    dat$epi$b.flow <- c(0, nBirths)
  } else {
    dat$epi$b.flow[at] <- nBirths
  }

  return(dat)
}
```

As with the life expectancy parameter, the growth rate will be a parameter that we pass into the module function through `param.net` and which will be stored on `dat$param`. The expected population size uses that the starting population size, stored in the summary statistics list `epi`, where the number needed is the difference between the expected and current population size, calculated again with the active attribute on `dat`.

Any birth function must do two things: add new nodes onto the network object and then update all of their attributes with their initial values. In our function, the `add.vertices` function will add new nodes onto the network, and the `activate.vertices` function will allow them to actively form edges within the model. We will need to set the 5 standard attributes for these new incoming births, as well as our new age attribute, on the `attr` list. The data for the new births will positionally be located at the end of the vector, so we append the values specified above using the `c` function. So new births will be active, disease susceptible, have no infection time, have an entry time equal to `at`, have no exit time, and enter the population at age 18. We additionally use a time-varying attribute on the network object for disease status, `testatus` for temporally-extended status, to track the changes in disease status for each node at each time point. This attribute value will automatically be changed to 1 if the person becomes infected.

Network Model Parameterization

For these examples, we will simplify things by fitting a basic random-mixing model for our ERGM. For the dissolution coefficient adjustment we take the mean of the death rates as a basic approximation to the overall mortality schedule for the population.

```
nw <- network.initialize(500, directed = FALSE)
est <- netest(nw, formation = ~edges, target.stats = 150,
              coef.diss = dissolution_coefs(~offset(edges), 60, mean(death.rates)))
```

Epidemic Model Parameterization

To parameterize the epidemic model, it is necessary to collect all the new elements that we created. For the model parameters, the transmission modules are unchanged and use the same infection probability and act rate as the built-in models. Our new birth module requires a growth rate parameter. Since our time units are months, we specify a 1% yearly growth rate in terms of monthly growth. The death module requires an input of a life expectancy. The module expects that in terms of years. Initial conditions are specified like a built-in SI model.

```
param <- param.net(inf.prob = 0.15, growth.rate = 0.00083, life.expt = 70)
init <- init.net(i.num = 50)
```

For controls settings, we specify the model type, number of simulations and time steps per simulation, similar to any built-in model. To replace the default death module, it is necessary to input the new `dfunc` function name as the `deaths.FUN` parameter. A similar approach is used for the birth module. Both of these functions must be sourced into memory before running `control.net` : the control function will try to look up this data and save it within the object output. To add a new module into `netsim` , give the module a name not among those modules available for replacement; but the module name must end with `.FUN` to register as a module. One important note on ordering: within a time step, replacement modules will be run in the order in which they are listed in the `control.net` help documentation; additional modules like `aging.FUN` will be run in the order in which you list them in `control.net` . Additional modules will run first and existing/replacement modules will run second. This ordering of all the modules may be changed and explicitly specified using the `module.order` argument in `control.net` .

```
control <- control.net(type = "SI", nsims = 5, nsteps = 250,
                       deaths.FUN = dfunc, births.FUN = bfunc, aging.FUN = aging,
                       depend = TRUE, save.network = FALSE)
```

Finally, the input for that module argument is the function itself. Because we have not input any parameters that automatically trigger a dependent network model (the vital dynamics parameters), it is necessary to explicitly set that `depend=TRUE` .

Model Results

The model is simulated using the standard method of inputting the fitted network model object, the parameters, initial conditions, and control settings.

```
mod <- netsim(est, param, init, control)
```

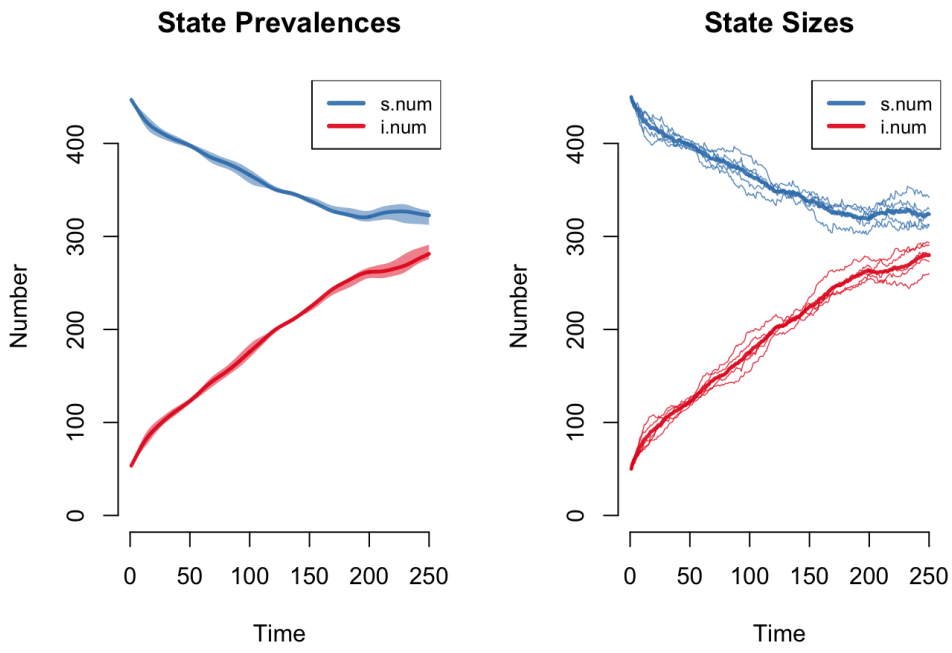
Printing the model object shows that we now have non-standard birth and death output for flows. Additionally, we have the `meanAge` data saved as other output (summary statistics saved in `epi` named ending with `.num` will automatically be classified as compartments and `.flow` as flows).

```
mod
```

```
EpiModel Simulation
=======================
Model class: netsim

Simulation Summary
----------------------
Model type: SI
No. simulations: 5
No. time steps: 250
No. NW modes: 1

Model Parameters
----------------------
inf.prob = 0.15
growth.rate = 0.00083
life.expt = 70
act.rate = 1

Model Output
----------------------
Variables: s.num i.num num meanAge d.flow b.flow si.flow
Transsmissions: sim1 ... sim5
```

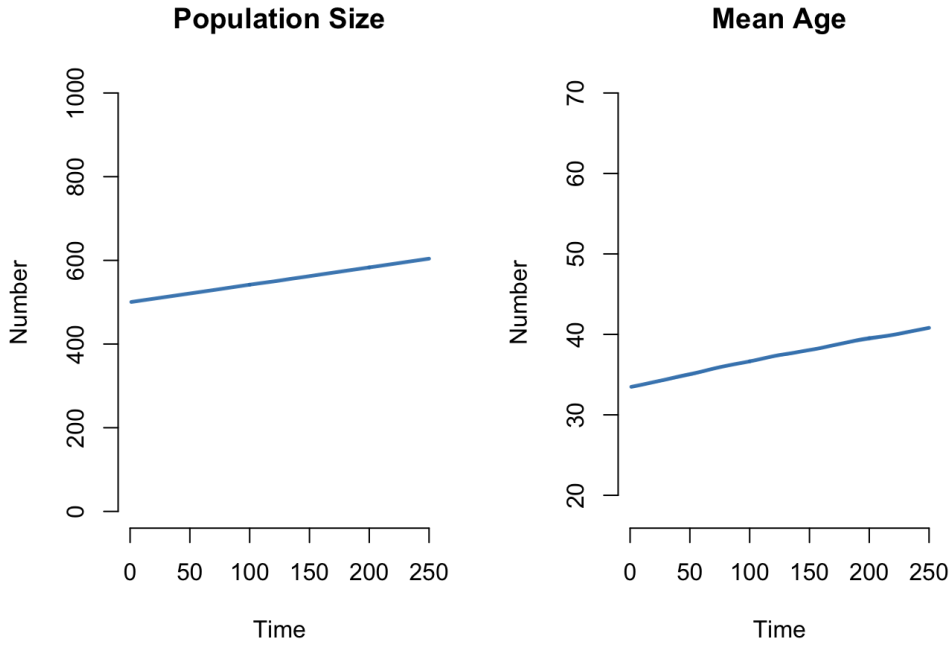
Basic model plots show the simulation results for the prevalence and absolute state sizes over time. The same sort of graphical plotting options available for any built-in model are also available for these new expansion models.

```
par(mfrow = c(1,2))
plot(mod, main = "State Prevalences")
plot(mod, popfrac = FALSE, main = "State Sizes", sim.lines = TRUE,
     qnts = FALSE, mean.smooth = FALSE)
```



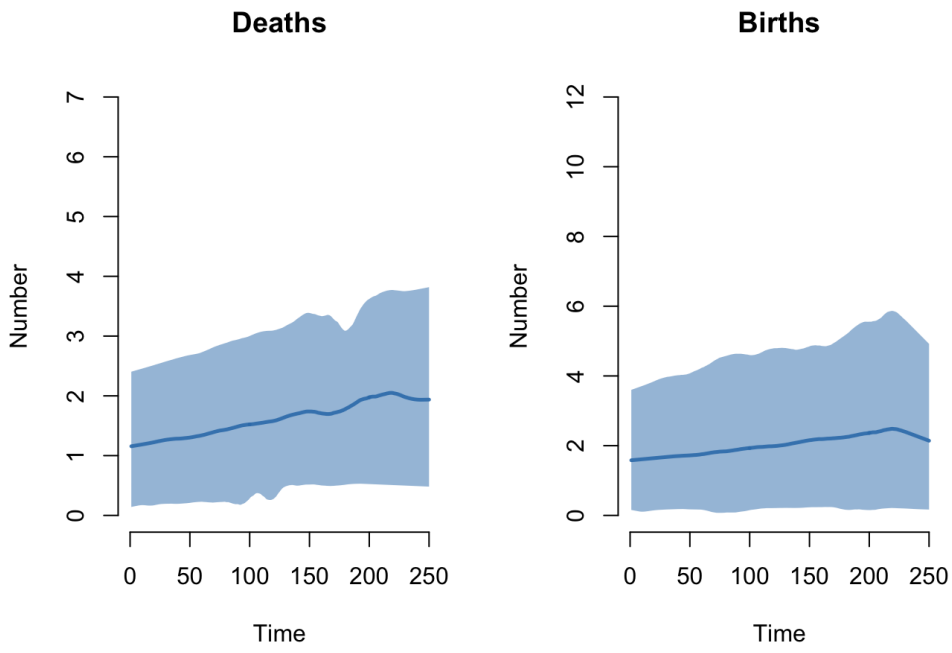
This plot shows the total population size and mean age of the population over time. The linear trend in population size is evidence of our constant growth model. The linear increase in mean age is due to initializing the population at a maximum age of 49 but allowing the life expectancy to reach 70.

```
par(mfrow = c(1, 2))
plot(mod, y = "num", popfrac = FALSE, main = "Population Size", ylim = c(0, 1000))
plot(mod, y = "meanAge", main = "Mean Age", ylim = c(18, 70))
```



The final plots show the variability and mean number of total deaths and births over time. Setting the quantiles to 1 shows the full extent of the data distribution.

```
par(mfrow = c(1, 2))
plot(mod, y = "d.flow", popfrac = FALSE, mean.smooth = TRUE, qnts = 1, main = "Deaths")
plot(mod, y = "b.flow", popfrac = FALSE, mean.smooth = TRUE, qnts = 1, main = "Births")
```



Example 2: Migration

In this example, the only demographic processes will be in and out migration. This will be an open population model in which there are no birth and deaths, but there are this other form of entry and exit from the network. Similar to the standard built-in death module, there will be different out-migration rates for susceptibles and infecteds in the population, considering that the travel plans of these two groups may differ. In contrast to the built-in birth module, in-migration will be a function of a constant expected count, regardless of the current population size of the network. In addition, whereas births were always into a susceptible state, now some proportion of in-migrants may be infected upon entry.

Out-Migration Module

The out-migration module looks quite similar to the death module because death and out-migration are functionally similar processes. In both, persons who leave the network are changed from active to inactive, and thereby “deactivated” from the network. The steps are to pull out all the respective rates, then run the processes, and then generate the output. The output here is the number of out-migrations per time step. Our prevalence module will then count the active number of members of each compartment given the rest of the processes in the network.

```
outmigrate <- function(dat, at) {

  # Variables
  active <- dat$attr$active
  status <- dat$attr$status
  rates <- dat$param$omig.rates

  # Process
  idsElig <- which(active == 1)
  nElig <- length(idsElig)

  nMig <- 0
  if (nElig > 0) {
    ratesElig <- rates[as.numeric(status == "i") + 1]
    vecMig <- which(rbinom(nElig, 1, ratesElig) == 1)
    if (length(vecMig) > 0) {
      idsMig <- idsElig[vecMig]
      nMig <- length(idsMig)

      dat$attr$active[idsMig] <- 0
      dat$attr$exitTime[idsMig] <- at
      dat$nw <- deactivate.vertices(dat$nw, onset = at, terminus = Inf,
                                    v = idsMig, deactivate.edges = TRUE)
    }
  }

  # Summary statistics
  if (at == 2) {
    dat$epi$omig.flow <- c(0, nMig)
  } else {
    dat$epi$omig.flow[at] <- nMig
  }

  return(dat)
}
```

The out-migration rates here are specified as a vector of length 2, where the first element is the rate for the susceptibles and the second the rate for the infecteds. We generate a vector for each eligible node for the transition by looking up those nodes disease status and applying the rates with `rates[as.numeric(status == "i") + 1]`. First, we convert the character values to numeric using a logical indicator, then add 1 to the value to properly index the rate values according to their place in the vector. We then record the size of the out-flows by adding new data to `dat$epi`, as mentioned in the introduction.

In-Migration Module

The in-migration process will not be a function of the current network size but an expected count per time step. Furthermore, there is a fixed probability that will determine whether the incoming nodes are infected (versus susceptible). This is functionally similar to births, since a number of new entries must be specified, those new nodes added to the network, and the nodal attributes of those new nodes specified. Finally, we keep track of the number of new in-migrations per unit time.

```
inmigrate <- function(dat, at) {

  # Variables
  nw <- dat$nw
  n <- network.size(nw)
  active <- dat$attr$active

  exp.inmig <- dat$param$exp.inmig
  risk <- dat$param$imig.risk
  tea.status <- dat$control$tea.status

  # Add Nodes
  nMig <- 0
  idsElig <- which(active == 1)
  nElig <- length(idsElig)
  if (nElig > 0) {
    nMig <- rpois(1, exp.inmig)
    if (nMig > 0) {
      dat$nw <- add.vertices(dat$nw, nv = nMig)
      newNodes <- (n + 1):(n + nMig)
      dat$nw <- activate.vertices(dat$nw, onset = at, terminus = Inf,
                                  v = newNodes)
    }
  }

  # Update attributes
  if (nMig > 0) {
    dat$attr$active <- c(dat$attr$active, rep(1, nMig))
    newStatus <- rbinom(nMig, 1, risk)
    newStatus <- ifelse(newStatus == 1, "i", "s")
    if (tea.status == TRUE) {
      dat$nw <- activate.vertex.attribute(dat$nw, prefix = "testatus", value = newStatus,
                                          onset = at, terminus = Inf, v = newNodes)
    }
    dat$attr$status <- c(dat$attr$status, newStatus)
    infTime <- ifelse(newStatus == "i", at, NA)
    dat$attr$infTime <- c(dat$attr$infTime, infTime)
    dat$attr$entrTime <- c(dat$attr$entrTime, rep(at, nMig))
    dat$attr$exitTime <- c(dat$attr$exitTime, rep(NA, nMig))
  }

  # Summary statistics
  if (at == 2) {
    dat$epi$imig.flow <- c(0, nMig)
  } else {
    dat$epi$imig.flow[at] <- nMig
  }

  return(dat)
}
```

For the newly arriving nodes, we must set all three of the core attributes mentioned in the introduction: `active`, `status`, `infTime`, `entrTime`, and `exitTime`.

Network Model

This will be same network model as before. The dissolution coefficient is now adjusted by a weighted average of the out-migration rates of the susceptible and infected at the outset.

```
nw <- network.initialize(500, directed = FALSE)
est <- netest(nw, formation = ~edges, target.stats = 150,
              coef.diss = dissolution_coefs(~offset(edges), 60, 0.0046))
```

Epidemic Model Parameters

For the model parameters, we will be simulating an SIS model so we will need an infection probability and recovery rate per unit time. These parameters will have the same function as all the built-in SIS models as we have not changed anything related to these two processes. There will also be three new parameters:

- `omig.rates` : a vector of length 2 containing the out-migration rates for the susceptibles and infecteds, respectively

- `exp.inmig` : the number of expected in-migrations at each time step. This is simply a fixed count; the actual number of in-migrations at each time step will be a draw from a Poisson distribution with rate set to that parameter.
- `inmig.risk` : the probability that each new in-migrant is infected at entry.

```
param <- param.net(inf.prob = 0.2, rec.rate = 0.02,
                   omig.rates = c(0.005, 0.001), exp.inmig = 3, imig.risk = 0.1)
init <- init.net(i.num = 50)
```

For the control settings, the usual settings for the model type, number of simulations, and number of time steps are used. In addition, we specify the new modules into EpiModel with the `.FUN` approach, in which each module is identified and refers to a specific module function. As noted above, additional modules will be run in the order in which you place them in `control.net` ; if it were necessary that the in-migration module be run first, their ordering could be switched below.

```
control <- control.net(type = "SIS", nsims = 5, nsteps = 500,
                       omig.FUN = outmigrate, imig.FUN = inmigrate,
                       depend = TRUE, save.network = FALSE)
```

Finally, the `depend` argument must again be set to `TRUE` to tell `netsim` to resimulate the network at each time step, because this does not happen automatically with expansion models like this.

Model Results

The model is simulated similar as the built-in models.

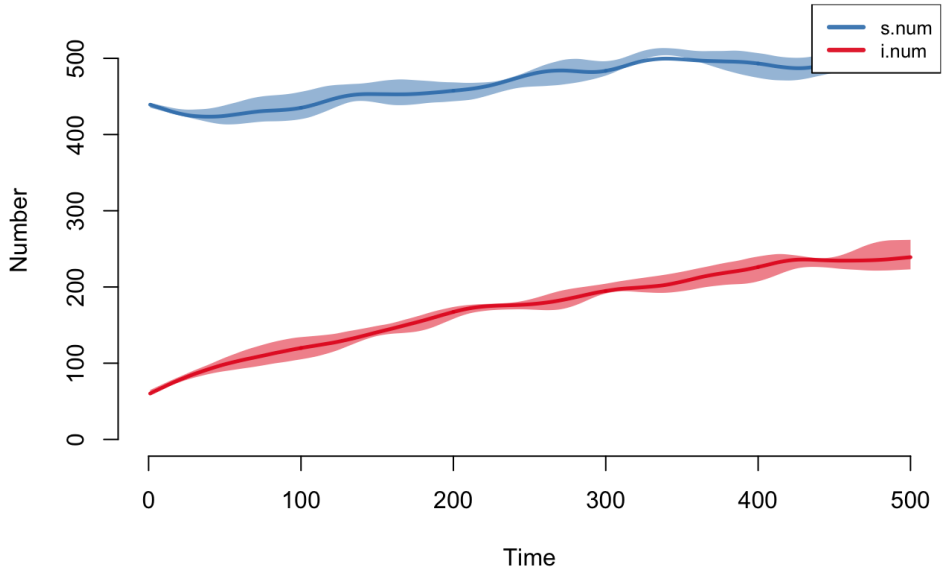
```
mod <- netsim(est, param, init, control)
```

The mode output is as follows. Note that the flows now contain the in and out-migration flows that were stored in these modules, in addition to the incidence and recovery flows.

```
mod
```

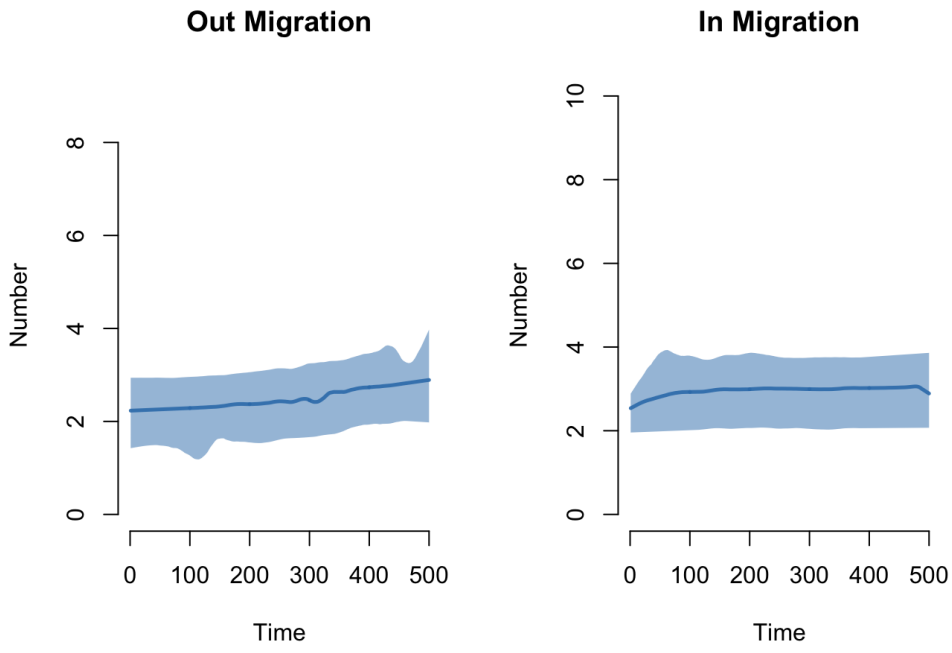
```
EpiModel Simulation
=======================
Model class: netsim

Simulation Summary
----------------------
Model type: SIS
No. simulations: 5
No. time steps: 500
No. NW modes: 1

Model Parameters
----------------------
inf.prob = 0.2
rec.rate = 0.02
omig.rates = 0.005 0.001
exp.inmig = 3
imig.risk = 0.1
act.rate = 1

Model Output
----------------------
Variables: s.num i.num num omig.flow imig.flow is.flow
si.flow
Transsmissions: sim1 ... sim5
```

```
plot(mod)
```



These are the smoothed means for the migration flows over time.

```
par(mfrow = c(1, 2))
plot(mod, y = "omig.flow", main = "Out Migration")
plot(mod, y = "imig.flow", main = "In Migration")
```



Last updated: 2017-06-01 with EpiModel v1.5.0

Back to Top (NewNet.html) | Back to epimodel.org (http://www.epimodel.org/)