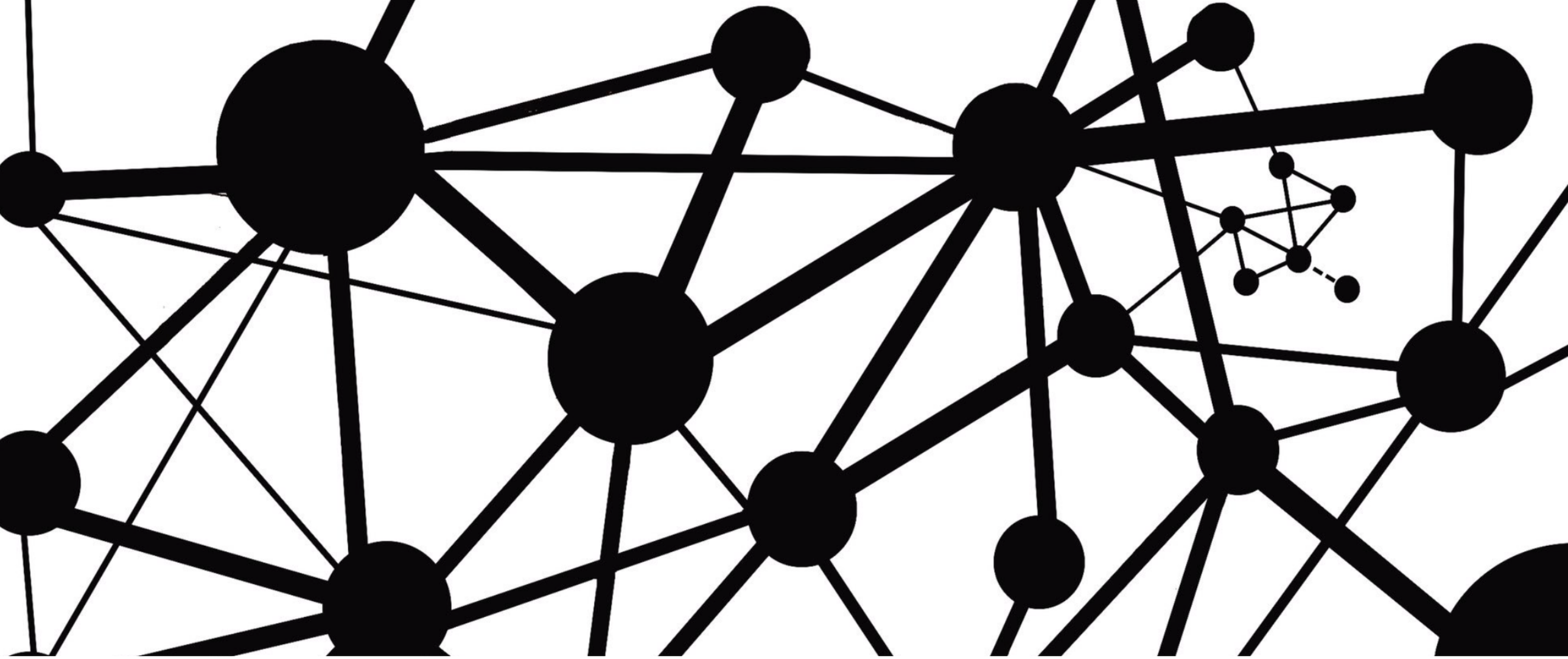


Extending EpiModel

Network Models for HIV/STI Transmission Dynamics with EpiModel

June 28, 2017

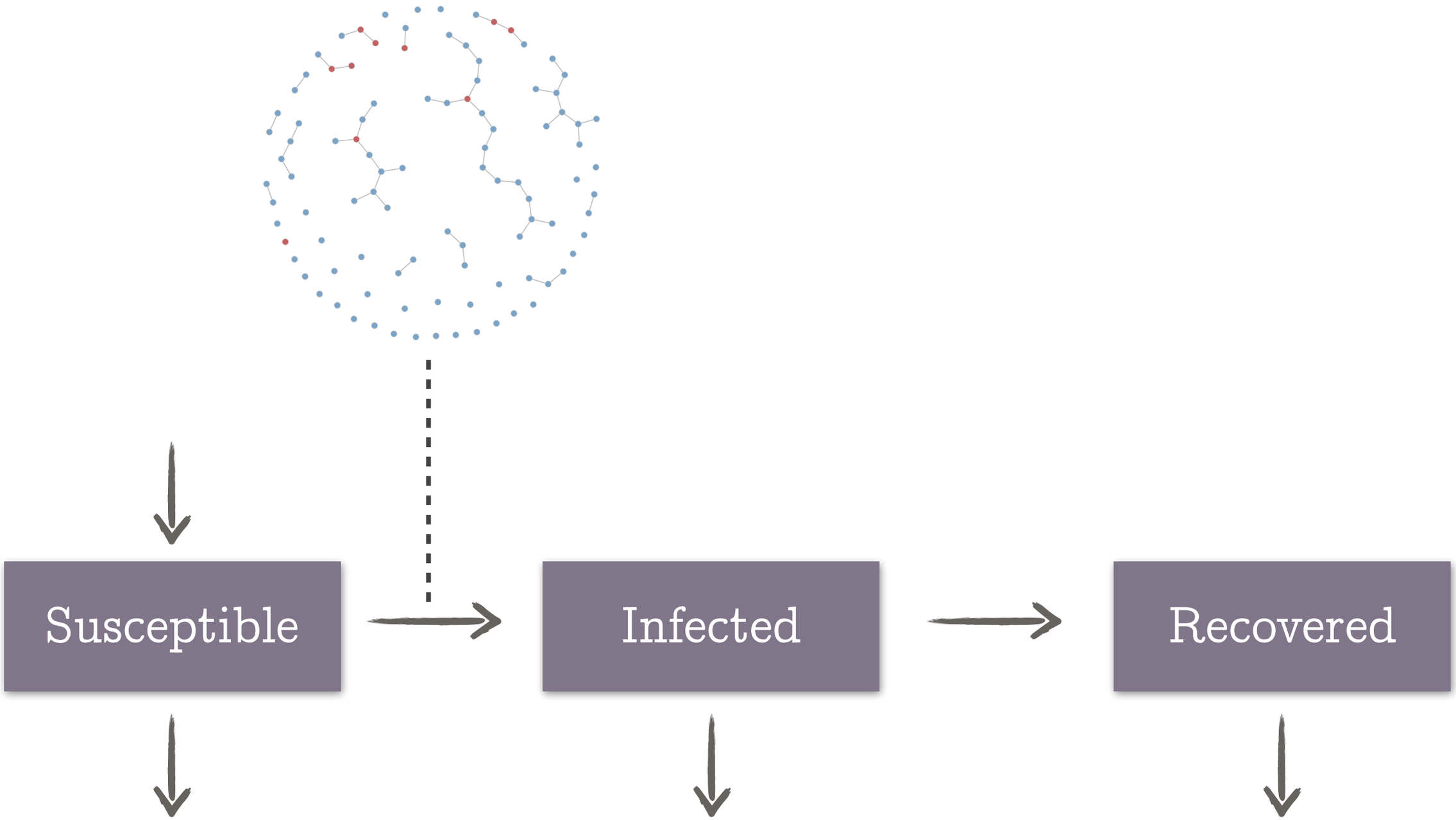
The EpiModel API

- So far, we have worked on “base” models: arbitrarily defined networks but fixed epidemiology
- Research applications require extendability of both
- EpiModel designed to provide (relatively) easy support to do this
- Our Application Programming Interface (API) provides a unified approach to doing this in “modules”
- The trickier parts — the contact network simulations — is handled by the `Statnet/ergm` methods under the hood
- You can use the same utility functions (`plot`, `as.data.frame`, `mutate`) as with the base models

What are Modules?

- Perform **one** core process within the system
- Module = concept; module function = realization
- Inputs are data structure and parameters
- Outputs are **revised** data structure
- “Plug and play”

A Base SIR Model



What Happens Inside `netsim`

```
netsim(est, param, init, control)
    t1 initialization module
    for (at in 2 to tn) {
        module 1
        module 2
        network re-simulation module
        transmission module
        module 5
    }
    tn Clean up and save output
```

Base Modules

`initialize.FUN`	Module to initialize the model at time 1, with the default function of initialize.net.
`deaths.FUN`	Module to simulate death or exit, with the default function of deaths.net.
`births.FUN`	Module to simulate births or entries, with the default function of births.net.
`recovery.FUN`	Module to simulate disease recovery, with the default function of recovery.net.
`edges_correct.FUN`	Module to adjust the edges coefficient in response to changes to the population size, with the default function of edges_correct that preserves mean degree.
`resim_nets.FUN`	Module to resimulate the network at each time step, with the default function of resim_nets.
`infection.FUN`	Module to simulate disease infection, with the default function of infection.net.
`get_prev.FUN`	Module to calculate disease prevalence at each time step, with the default function of get_prev.net.
`verbose.FUN`	Module to print simulation progress to screen, with the default function of verbose.net.

Modules

initialize.FUN
deaths.FUN
births.FUN
recovery.FUN
edges_correct.FUN
resim_nets.FUN
infection.FUN
get_prev.FUN
verbose.FUN

SI models in closed populations

Modules

initialize.FUN
deaths.FUN
births.FUN
recovery.FUN
edges_correct.FUN
resim_nets.FUN
infection.FUN
get_prev.FUN
verbose.FUN

SIR models in open populations

Base Modules

Let’s take a peek inside the recovery module:

```
View(recovery.net)
```

Within EpiModel::netsim

```
netsim(est, param, init, control)
    t1 initialization module
    for (at in 2 to tn) {
        module 1
        module 2
        network re-simulation module
        transmission module
        module 5
    }
    tn Clean up and save output
```

Let’s Build a Module for Aging

- What’s involved in the aging process?
- What should the module do?
- Is it just one module or multiple?

Simple Aging Approach

- Let's assume a closed population so we don't have to worry about assigning age to incoming nodes
- Therefore, aging will just be one self-contained module in our workflow

Step 1: Define the Module Function

```
aging <- function(dat, at) {
    age <- dat$attr$age
    age <- age + 1/52
    dat$attr$age <- age
    return(dat)
}
```

Data structure that gets passed around, module to module

In discrete time, everything happens *at* a specific time step

Functions should contain both arguments (even, as in this case, one is not used)

Step 1: Define the Module Function

```
aging <- function(dat, at) {
    age <- dat$attr$age
    age <- age + 1/52
    dat$attr$age <- age
    return(dat)
}
```

Processes within modules usually involve four steps:

1. Pull out individual-level attributes (or parameters or summary stats) from `dat`
2. Do something to them
3. Write them back on `dat`
4. Return `dat`

Step 1: Define the Module Function

```
aging <- function(dat, at) {
    age <- dat$attr$age
    age <- age + 1/52
    dat$attr$age <- age
    return(dat)
}
```

All individual-level attributes read and written to `dat$attr`

`dat$attr` is a list within a list

Every element of `dat$attr` is a vector of length n

`dat$attr` contains all the attributes needed for your research

Step 1: Define the Module Function

```
aging <- function(dat, at) {
    age <- dat$attr$age
    tunit <- dat$param$tunit
    age <- age + 1/tunit
    dat$attr$age <- age
    return(dat)
}
```

Maybe the unit of time is not fixed in your model but varying (e.g., to check competing risk biases)

You can input the time unit, `tunit`, as a parameter within your model that is used by one or more modules

Parameters get stored on `dat$param`

Step 1: Define the Module Function

```
aging <- function(dat, at) {
    age <- dat$attr$age
    tunit <- dat$param$tunit
    age <- age + 1/tunit
    dat$attr$age <- age
    dat$epi$meanAge[at] <-
                    mean(age)
    return(dat)
}
```

Perhaps it is of interest to track some summary statistics specific to processes within this module, such as the mean of the ages

Summary statistics are indexed by time (bracket notation) and written on `dat$epi`

`dat$epi` is another list of vectors, each of length nsteps for the sim

Step 2: Parameterizing the Function

- Recall the three helper functions for netsim for network models are `param.net`, `init.net`, and `control.net`
- If you look inside the help files for these functions, they will look like this

```
param.net(inf.prob, inter.eff, inter.start, act.rate,
rec.rate, b.rate, ds.rate, di.rate, dr.rate, inf.prob.m2,
rec.rate.m2, b.rate.m2, ds.rate.m2, di.rate.m2, dr.rate.m2,
...)
```

- The … means you can put anything in there you need, on top of the individual parameters that are listed
- There's also no need to use any of the existing parameters if you don't want them

Step 2: Parameterizing the Function

- So for parameters, we can use the existing defined infection probability and act rate, but enter a new time unit

```
myparam <- param.net(inf.prob = 0.15, act.rate = 2,
                     tunit = 52)
```

- The name of the parameter, tunit, needs to correspond to what you have referenced inside your module (`dat$param$tunit`)
- The initial conditions (that is, everyone's initial age in the population) can be a little more complicated, so we will return to that in the tutorial

Step 3: Input the Module Function

- The module function gets input into the control settings in `control.net`
- You can replace an existing module there (e.g., sub out a new transmission function) or add a new one

```
control <- control.net(type = "SI", nsims = 5, nsteps = 250,
                       aging.FUN = aging, depend = TRUE)
```

- Modules are identified to EpiModel by a `.FUN` suffix
- The name of the module function is the input to this argument
- To force a dependent process model, use `depend = TRUE`

Step 4: Run the Model

- Once everything is defined and parameterized, the model may be run just like the base models

```
sim <- netsim(est, myparam, myinit, mycontrol)
```

- After the model simulation has completed, the data may be plotted and analyzed similar to base models

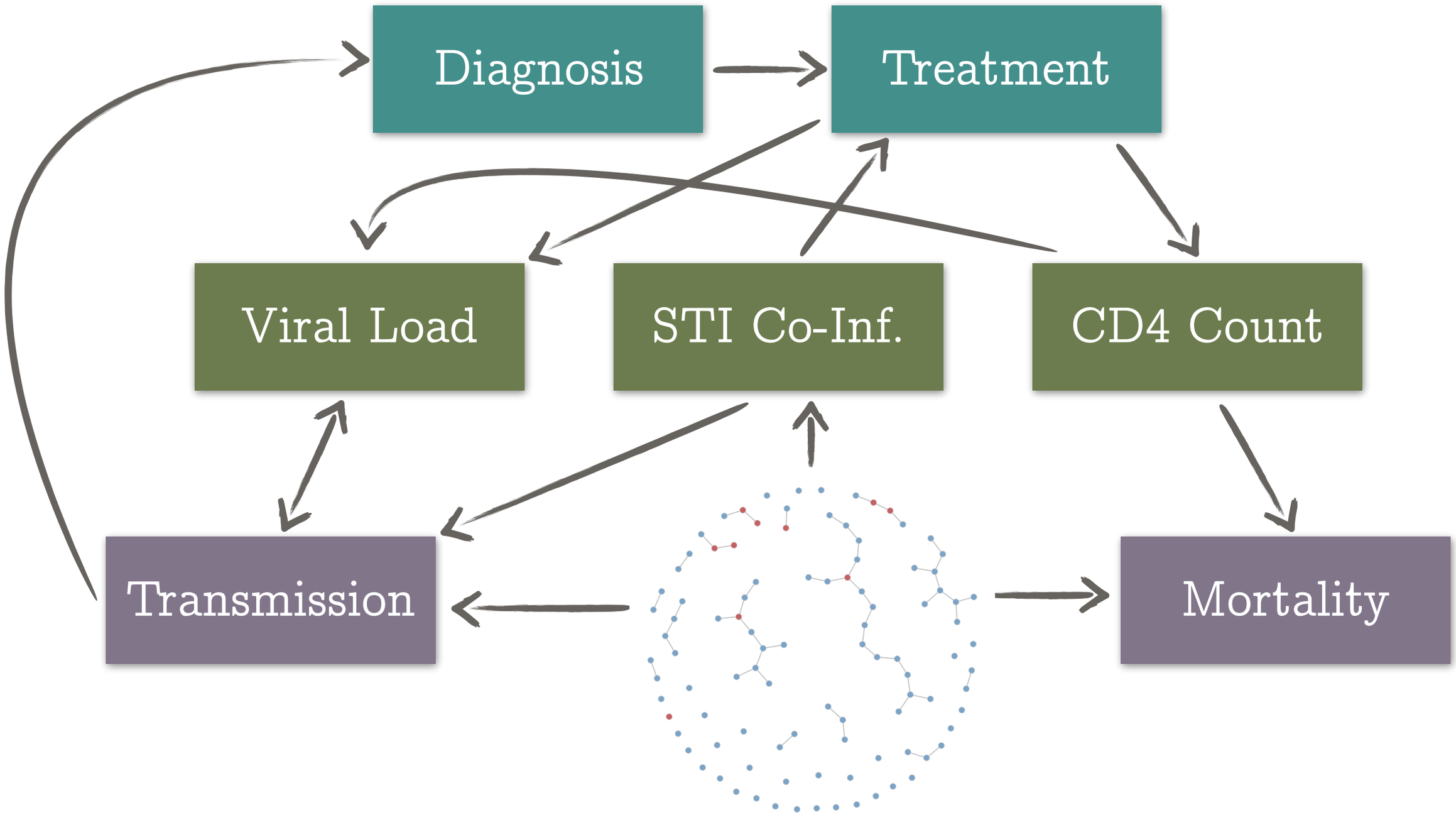
Caveats

- The devil is always in the details
- While EpiModel is meant to be a broad research platform, we're still working out the details about how to best support users like you
- EpiModel as a coding platform versus EpiModel as templates
- Your research cases may require a more substantial rewriting than we imply here, but we're here to help!

Base Modules

`initialize.FUN`	Module to initialize the model at time 1, with the default function of initialize.net.
`deaths.FUN`	Module to simulate death or exit, with the default function of deaths.net.
`births.FUN`	Module to simulate births or entries, with the default function of births.net.
`recovery.FUN`	Module to simulate disease recovery, with the default function of recovery.net.
`edges_correct.FUN`	Module to adjust the edges coefficient in response to changes to the population size, with the default function of edges_correct that preserves mean degree.
`resim_nets.FUN`	Module to resimulate the network at each time step, with the default function of resim_nets.
`infection.FUN`	Module to simulate disease infection, with the default function of infection.net.
`get_prev.FUN`	Module to calculate disease prevalence at each time step, with the default function of get_prev.net.
`verbose.FUN`	Module to print simulation progress to screen, with the default function of verbose.net.

Conceptual Diagram for HIV/STI Transmission Model



Modular Extension Exercise

- Let's take a current ID that you are interested in (but not HIV or STIs)
  - What are components of the system?
  - What type of modules do we need?
  - How do the modules interact?
- This is purely conceptual – no need to worry about the coding details yet
  - After we go through the tutorials, we'll code up together a test and treat intervention for an STI