Continuous time Markov chains

R for modellers – Vignette 21

Julien Arino

June 2024

Outline

Continuous time Markov chains

ODE $\leftrightarrow$ CTMC

Simulating CTMC (in theory)

Simulating CTMC (in practice)

Parallelising your code in R

Continuous time Markov chains

ODE $\leftrightarrow$ CTMC

Simulating CTMC (in theory)

Simulating CTMC (in practice)

Parallelising your code in R

Continuous-time Markov chains

CTMC similar to DTMC except in way they handle time between events (transitions)

DTMC: transitions occur each Δt

CTMC: $\Delta t \to 0$ and transition times follow an exponential distribution parametrised by the state of the system

CTMC are roughly equivalent to ODE

Continuous time Markov chains

ODE $\leftrightarrow$ CTMC

Simulating CTMC (in theory)

Simulating CTMC (in practice)

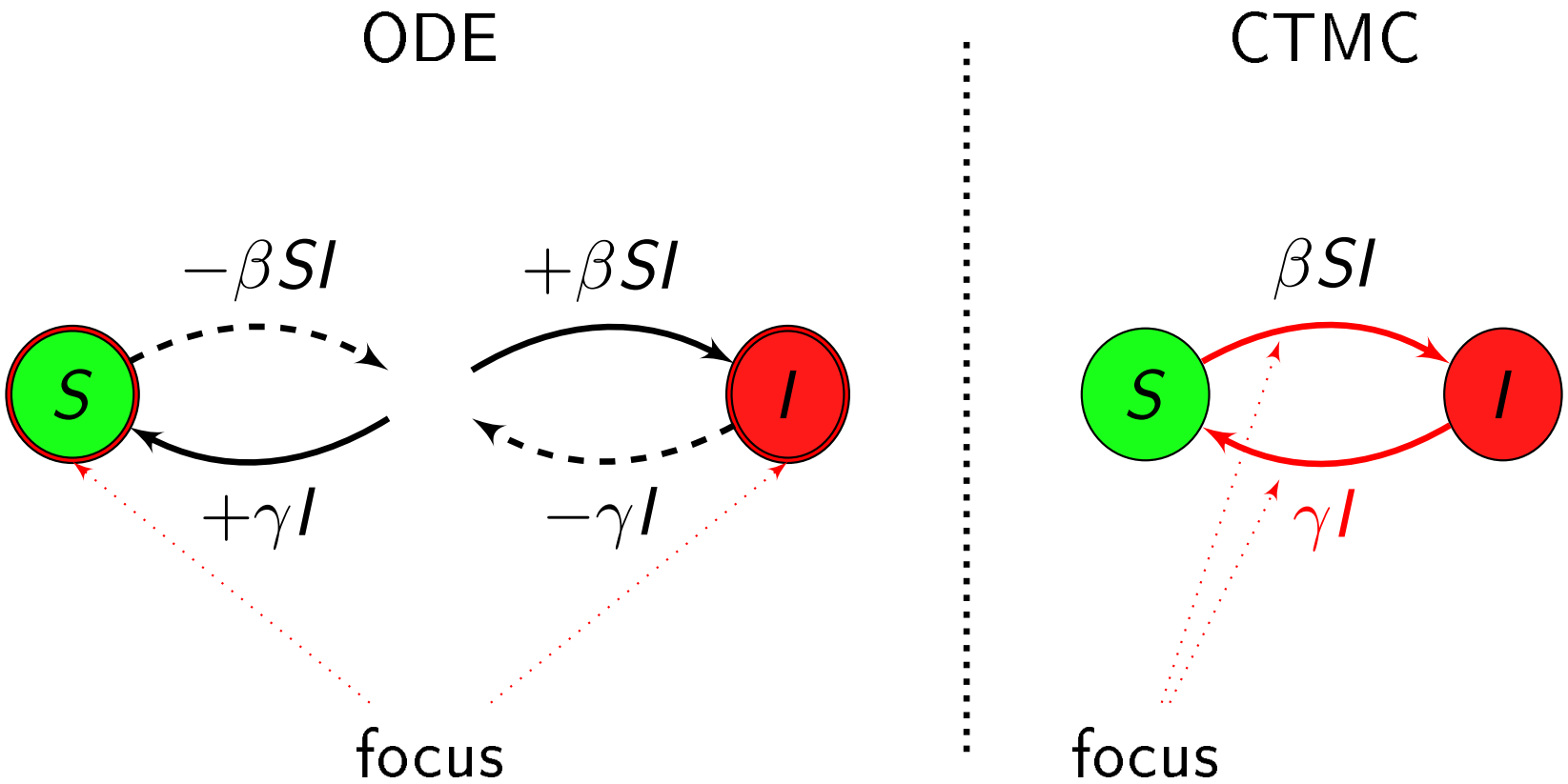
Parallelising your code in R

Converting your compartmental ODE model to CTMC

Easy as π :)

- Compartmental ODE model focuses on flows into and out of compartments
- ODE model has as many equations as there are compartments
- Compartmental CTMC model focuses on transitions
- CTMC model has as many transitions as there are arrows between (or into or out of) compartments

ODE to CTMC : focus on different components



SIS without demography

Transition	Effect	Weight	Probability
$S \to S-1,\ I \to I+1$	new infection	βSI	$\frac{\beta SI}{\beta SI + \gamma I}$
$S \to S+1,\ I \to I-1$	recovery of an infectious	γI	$\frac{\gamma I}{\beta SI + \gamma I}$

States are S, I

SIS with demography

Transition	Effect	Weight	Probability
$S \to S+1$	birth of a susceptible	b	$\frac{b}{b+d(S+I)+\beta SI+\gamma I}$
$S \to S-1$	death of a susceptible	dS	$\frac{dS}{b+d(S+I)+\beta SI+\gamma I}$
$S \to S-1,\ I \to I+1$	new infection	βSI	$\frac{\beta SI}{b+d(S+I)+\beta SI+\gamma I}$
$I \to I-1$	death of an infectious	dI	$\frac{dI}{b+d(S+I)+\beta SI+\gamma I}$
$S \to S+1,\ I \to I-1$	recovery of an infectious	γI	$\frac{\gamma I}{b+d(S+I)+\beta SI+\gamma I}$

States are S, I

Kermack & McKendrick model

Transition	Effect	Weight	Probability
$S \to S-1,\ I \to I+1$	new infection	βSI	$\frac{\beta SI}{\beta SI + \gamma I}$
$I \to I-1,\ R \to R+1$	recovery of an infectious	γI	$\frac{\gamma I}{\beta SI + \gamma I}$

States are S, I, R

Continuous time Markov chains

ODE $\leftrightarrow$ CTMC

Simulating CTMC (in theory)

Simulating CTMC (in practice)

Parallelising your code in R

Gillespie's algorithm

- A.k.a. the stochastic simulation algorithm (SSA)
- Derived in 1976 by Daniel Gillespie
- Generates possible solutions for CTMC
- Extremely simple, so worth learning how to implement; there are however packages that you can use (see later)

Gillespie's algorithm

Suppose system has state $\mathrm{x}(t)$ with initial condition $\mathrm{x}(t_0) = \mathrm{x}_0$ and *propensity functions* a_i of elementary reactions

set $t \leftarrow t_0$ and $\mathrm{x}(t) \leftarrow \mathrm{x}_0$
`while` $t \leq t_f$

- $\xi_t \leftarrow \sum_j a_j(\mathrm{x}(t))$
- Draw τ_t from $T \sim \mathcal{E}(\xi_t)$
- Draw ζ_t from $\mathcal{U}([0,1])$
- Find r, smallest integer s.t.

$$\sum_{k=1}^{r} a_k(\mathrm{x}(t)) > \zeta_t \sum_j a_j(\mathrm{x}(t)) = \zeta_t \xi_t$$

- Effect the next reaction (the one indexed r)
- $t \leftarrow t + \tau_t$

Drawing at random from an exponential distribution

If you do not have an exponential distribution random number generator.. We want τ_t from $T \sim \mathcal{E}(\xi_t)$, i.e., T has probability density function

$$f(x, \xi_t) = \xi_t e^{-\xi_t x} 1_{x \geq 0}$$

Use cumulative distribution function $F(x, \xi_t) = \int_{-\infty}^{x} f(s, \xi_t)\, ds$

$$F(x, \xi_t) = (1 - e^{-\xi_t x}) 1_{x \geq 0}$$

which has values in $[0, 1]$. So draw ζ from $\mathcal{U}([0, 1])$ and solve $F(x, \xi_t) = \zeta$ for x

$$\begin{aligned} F(x, \xi_t) = \zeta &\Leftrightarrow 1 - e^{-\xi_t x} = \zeta \\ &\Leftrightarrow e^{-\xi_t x} = 1 - \zeta \\ &\Leftrightarrow \xi_t x = -\ln(1 - \zeta) \\ &\Leftrightarrow \boxed{x = \frac{-\ln(1 - \zeta)}{\xi_t}} \end{aligned}$$

Gillespie's algorithm (SIS model with only I eq.)

set $t \leftarrow t_0$ and $I(t) \leftarrow I(t_0)$

`while` $t \leq t_f$

- $\xi_t \leftarrow \beta(P^\star - i)i + \gamma i$
- Draw τ_t from $T \sim \mathcal{E}(\xi_t)$
- $v \leftarrow [\beta(P^\star - i)i, \xi_t] / \xi_t$
- Draw ζ_t from $\mathcal{U}([0,1])$
- Find pos such that $v_{pos-1} \leq \zeta_t \leq v_{pos}$
- switch pos
  1. New infection, $I(t + \tau_t) = I(t) + 1$
  2. End of infectious period, $I(t + \tau_t) = I(t) - 1$
- $t \leftarrow t + \tau_t$

Sometimes Gillespie goes bad

- Recall that the inter-event time is exponentially distributed
- Critical step of the Gillespie algorithm:
  - $\xi_t \leftarrow$ weight of all possible events (*propensity*)
  - Draw τ_t from $T \sim \mathcal{E}(\xi_t)$
- So inter-event time $\tau_t \to 0$ if ξ_t very large for some t
- This can cause the simulation to grind to a halt

Example: a birth and death process

- Individuals born at *per capita* rate b
- Individuals die at *per capita* rate d
- Let's implement this using classic Gillespie

(`CODE/simulate-birth-death-CTMC-classic-Gillespie.R` on course GitHub repo)

Gillespie's algorithm (birth-death model)

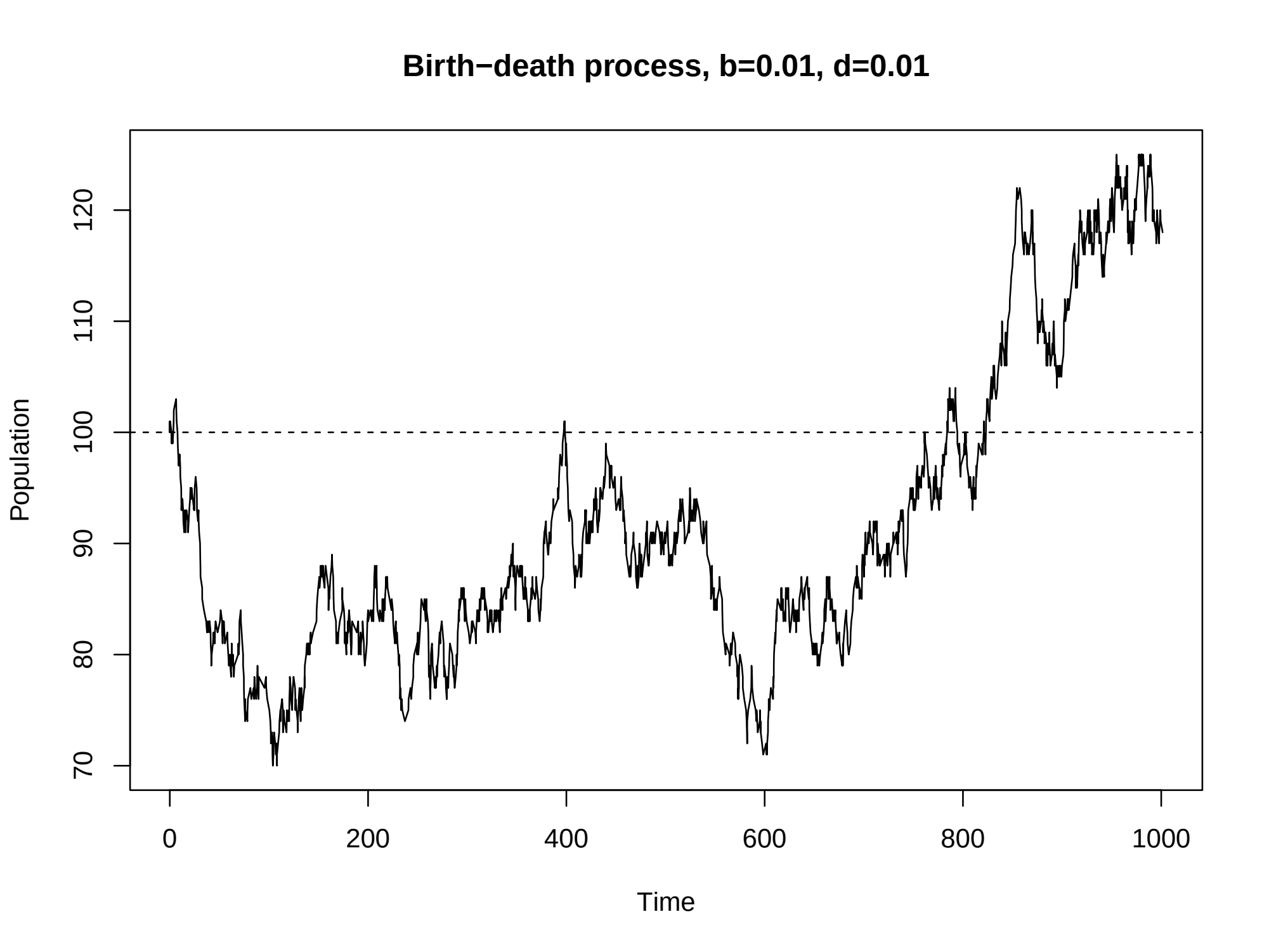
set $t \leftarrow t_0$ and $N(t) \leftarrow N(t_0)$

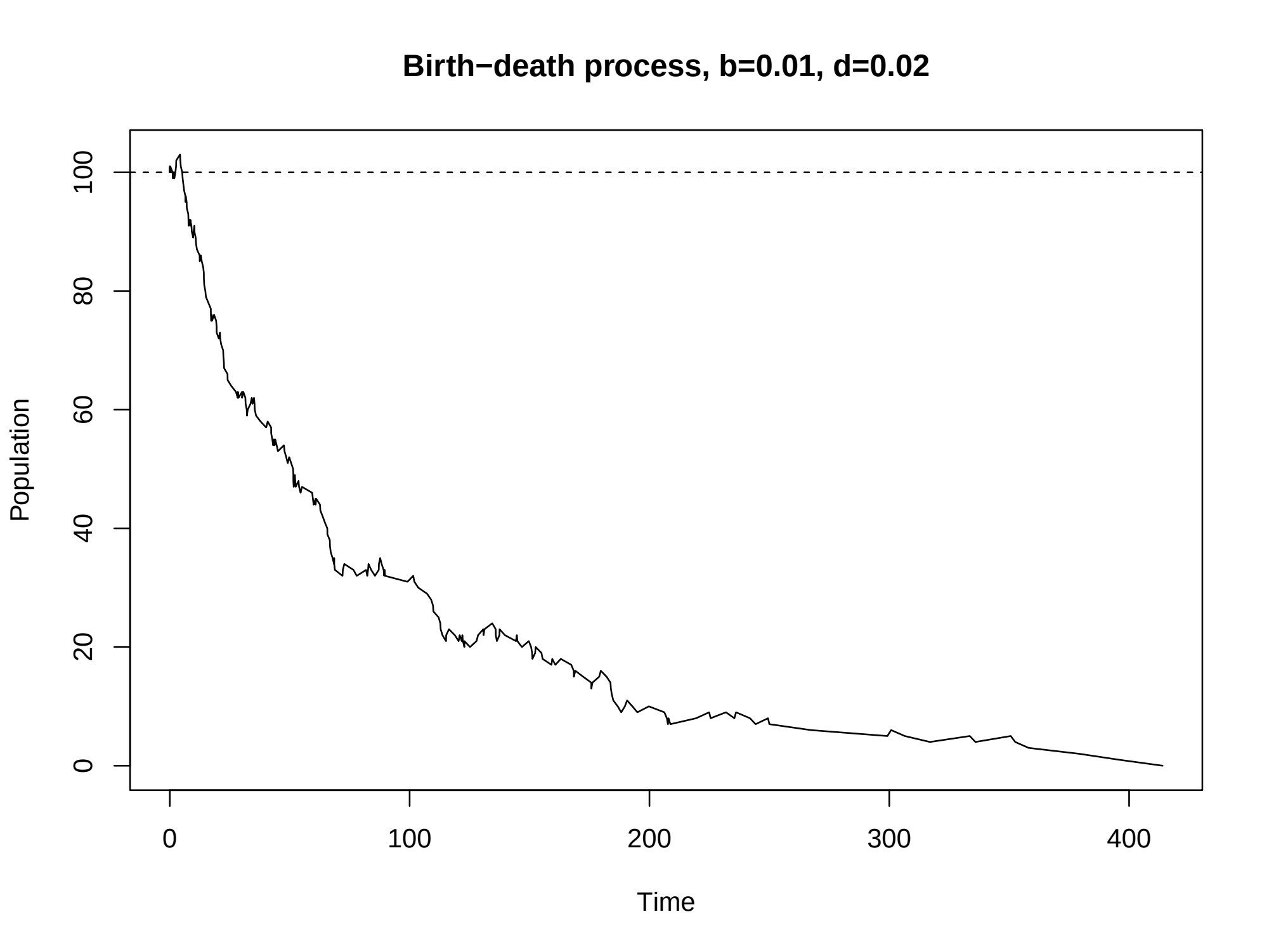
`while` $t \leq t_f$

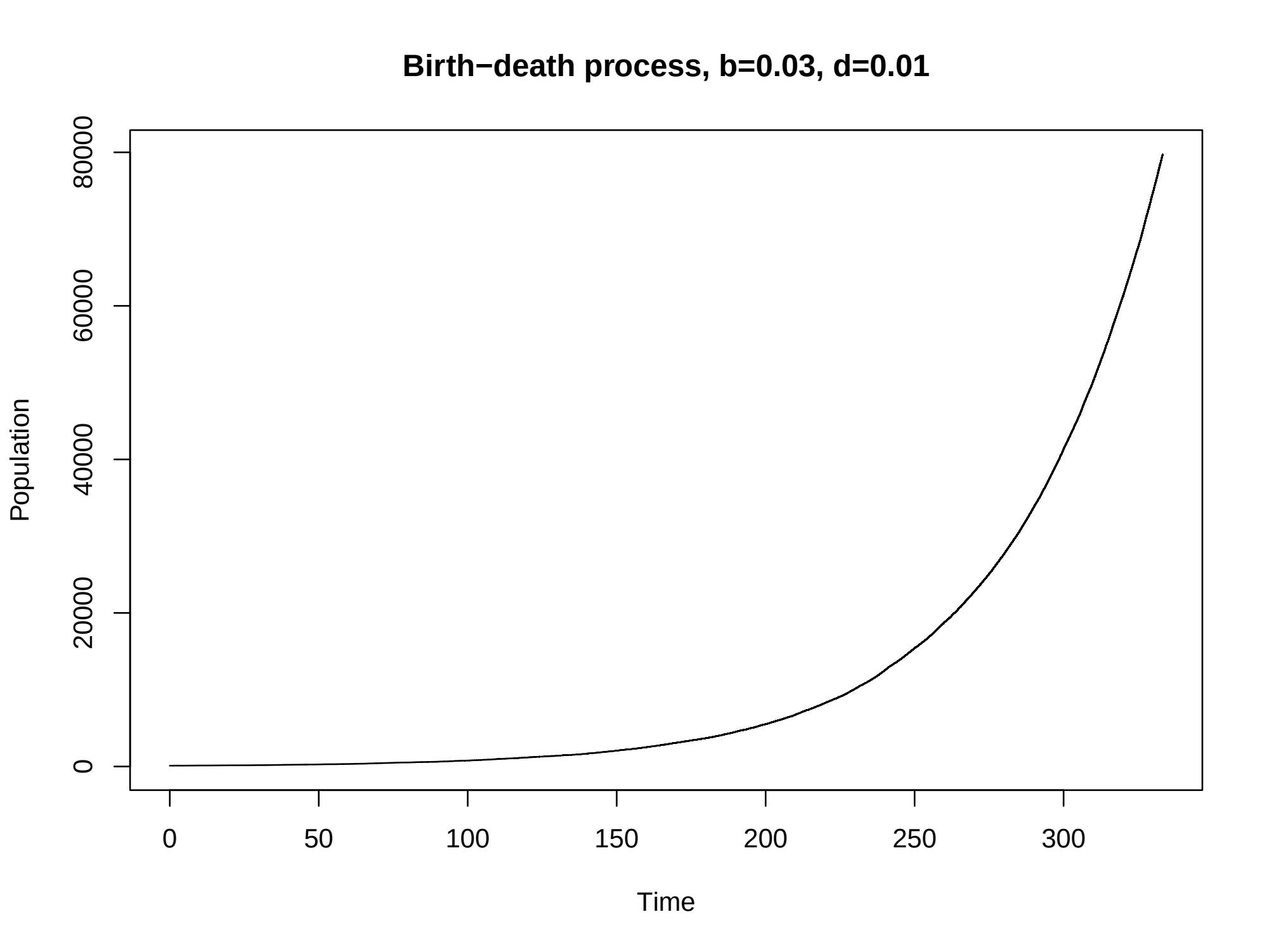
- $\xi_t \leftarrow (b+d)N(t)$
- Draw τ_t from $T \sim \mathcal{E}(\xi_t)$
- $v \leftarrow [bN(t), \xi_t] / \xi_t$
- Draw ζ_t from $\mathcal{U}([0,1])$
- Find pos such that $v_{pos-1} \leq \zeta_t \leq v_{pos}$
- switch pos
  1. Birth, $N(t+\tau_t) = N(t)+1$
  2. Death, $N(t+\tau_t) = N(t)-1$
- $t \leftarrow t + \tau_t$

```
birth_death = function(b = 0.01, d = 0.01,
                       N_0 = 100,
                       t_0 = 0, t_f = 1000) {

  # Vectors to store time and state.
  # Initialise with initial condition.
  t = t_0 # Initial time
  N = N_0 # Initial population

  # We'll track the current time and state (could also just check
  # last entry in t and N, but will take more operations)
  t_curr = t_0
  N_curr = N_0
```

```
  while (t_curr<=t_f) {
    xi_t = (b+d)*N_curr
    if (N_curr == 0) {
      break
    }
    tau_t = rexp(1, rate = xi_t)
    t_curr = t_curr+tau_t
    v = c(b*N_curr, xi_t)/xi_t
    zeta_t = runif(n = 1)
    pos = findInterval(zeta_t, v)+1
    switch(pos,
           {
             N_curr = N_curr+1 # Birth
           },
           {
             N_curr = N_curr-1 # Death
           })
    N = c(N, N_curr)
    t = c(t, t_curr)
  }
  return(data.frame(t = t, N = N))
}
```


Birth−death process, b=0.01, d=0.01
Population
70
80
90
100
110
120
0
200
400
600
800
1000
Time


Birth−death process, b=0.01, d=0.02
Population
0
20
40
60
80
100
0
100
200
300
400
Time

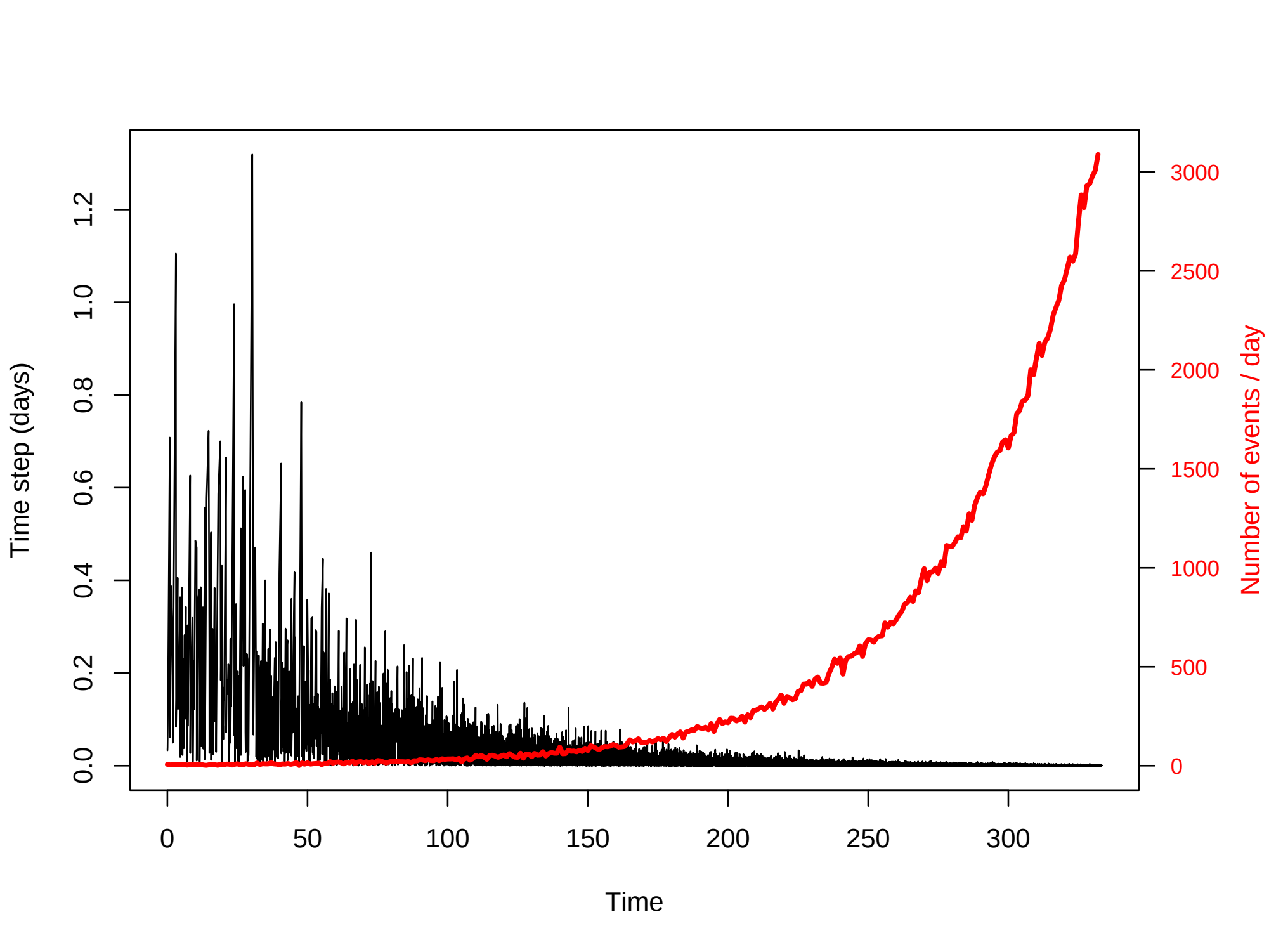

Birth−death process, b=0.03, d=0.01
Population
0
20000
40000
60000
80000
0
50
100
150
200
250
300
Time

Last one did not go well

- Penultimate slide: sim stopped because the population went extinct, I did not stop it!
- Last slide: I wanted 1,000 time units (days?)

- Interrupted at $t = 333.25$ because I "lost patience" (added something to check step size, see code for the slide)

- At stop time
  - $N = 79,707$
  - $|t| = |N| = 159,782$
  - time was moving slowly


Time step (days)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Number of events / day
0
500
1000
1500
2000
2500
3000
0
50
100
150
200
250
300
Time

Continuous time Markov chains

ODE $\leftrightarrow$ CTMC

Simulating CTMC (in theory)

Simulating CTMC (in practice)

Parallelising your code in R

Tau-leaping (and packages) to the rescue!

- *Approximation* method (compared to classic Gillespie, which is exact)
- Roughly: consider "groups" of events instead of individual events
- Good news: `GillespieSSA2` and `adaptivetau`, two standard packages for SSA in `R`, implement tau leaping

`adaptivetau` or `GillespieSSA2`?

- Both packages do roughly the same thing now (in the past, `GillespieSSA2` was the only only one exporting "events", but now both do)
- `GillespieSSA2` can precompile stuff, which is faster. Also has a slightly more compact syntax
- `adaptivetau` is more robust: precompiling is great but runs into issues when you are parallelising your code

$\Longrightarrow$ I will illustrate using `adaptivetau`

CODE directory has some `GillespieSSA2` examples as well. Both are very similar!

adaptivetau to simulate an SIRS CTMC

Initial setup

```
library(adaptivetau)

Pop <- 1000
I_0 <- 2
IC <- c(S = (Pop-I_0), I = I_0, R = 0)

params <- list(gamma = 1/5, nu = 1/50)
params$beta <- params$gamma*1.5/(Pop-I_0)

t_f = 100
```

adaptivetau to simulate an SIRS CTMC

Reactions and reaction rates

```
reactions_names <- c("new_infection",
                     "recovery",
                     "loss_immunity")
reactions_effects <- list(
  c(S=-1, I=+1), # new infection
  c(I=-1, R=+1), # recovery
  c(R=-1, S=+1)  # loss of immunity
)
reactions_rates <- function(x, params, t) {
  with(as.list(c(x, params)), {
    rates <- c(
      beta*S*I, # new infection
      gamma*I,  # recovery
      nu*R      # loss of immunity
    )
    return(rates)
  })
}
```

adaptivetau to simulate an SIRS CTMC

Calling the "solver"

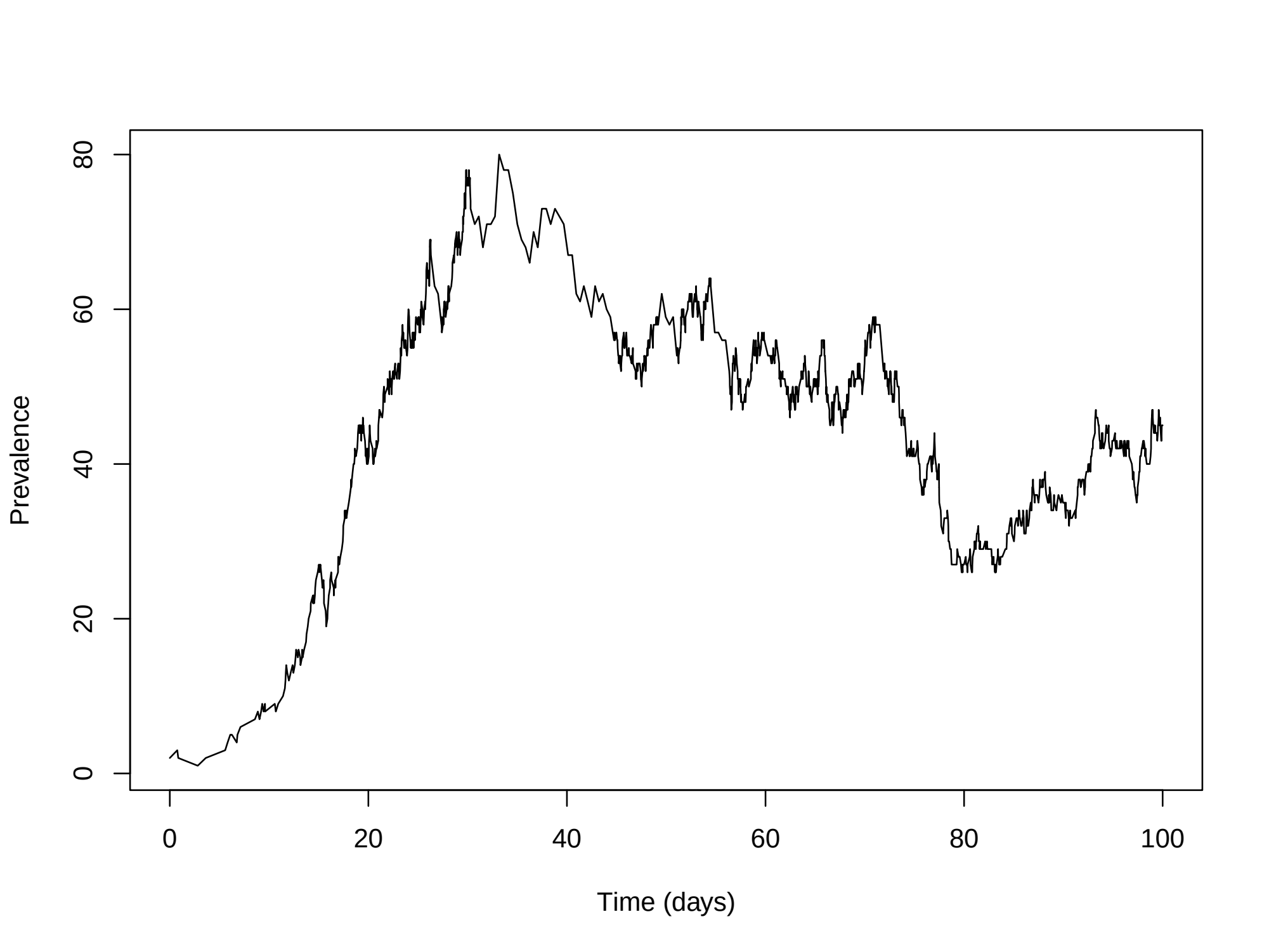
```
set.seed(1)

sol <- ssa.adaptivetau(
  init.values = IC,
  transitions = reactions_effects,
  rateFunc = reactions_rates,
  params = params,
  tf = t_f
)
```

Beware: `set.seed(1)` is used for reproducibility. Remove for real simulations! (E.g., use `set.seed(NULL)`)

Simulation output

time	S	I	R
0.0000000	998	2	0
0.7551818	997	3	0
0.8523781	997	2	1
2.8140734	997	1	2
3.6220576	996	2	2
5.5801782	995	3	2

Can be useful to convert to a `data.frame` for convenience (e.g., to use `sol$time` instead of `sol[,"time"]`)


Prevalence
0
20
40
60
80
0
20
40
60
80
100
Time (days)

Important options to `ssa.adaptivetau`

- Not an option *per se*: calling `ssa.exact` instead of `ssa.tauleap` uses the exact SSA algorithm (traditional Gillespie algorithm) instead of tau-leaping

- Both `ssa.exact` and `ssa.tauleap` have a `reportTransitions`, which, when set to `TRUE`, returns the transitions that occurred at each time step

adaptivetau to simulate an SIRS CTMC

Playing with options

```
set.seed(1)

sol <- ssa.exact(
  init.values = IC,
  transitions = reactions_effects,
  rateFunc = reactions_rates,
  params = params,
  tf = t_f,
  reportTransitions = TRUE
)
```

Transitions in the simulation

Calling with `reportTransitions = TRUE` returns the solution as a `list` with fields `dynamics` and `transitions`

`dynamics` is the output we had before

`transitions` is a matrix with the corresponding events. That's where having the transition names is useful:

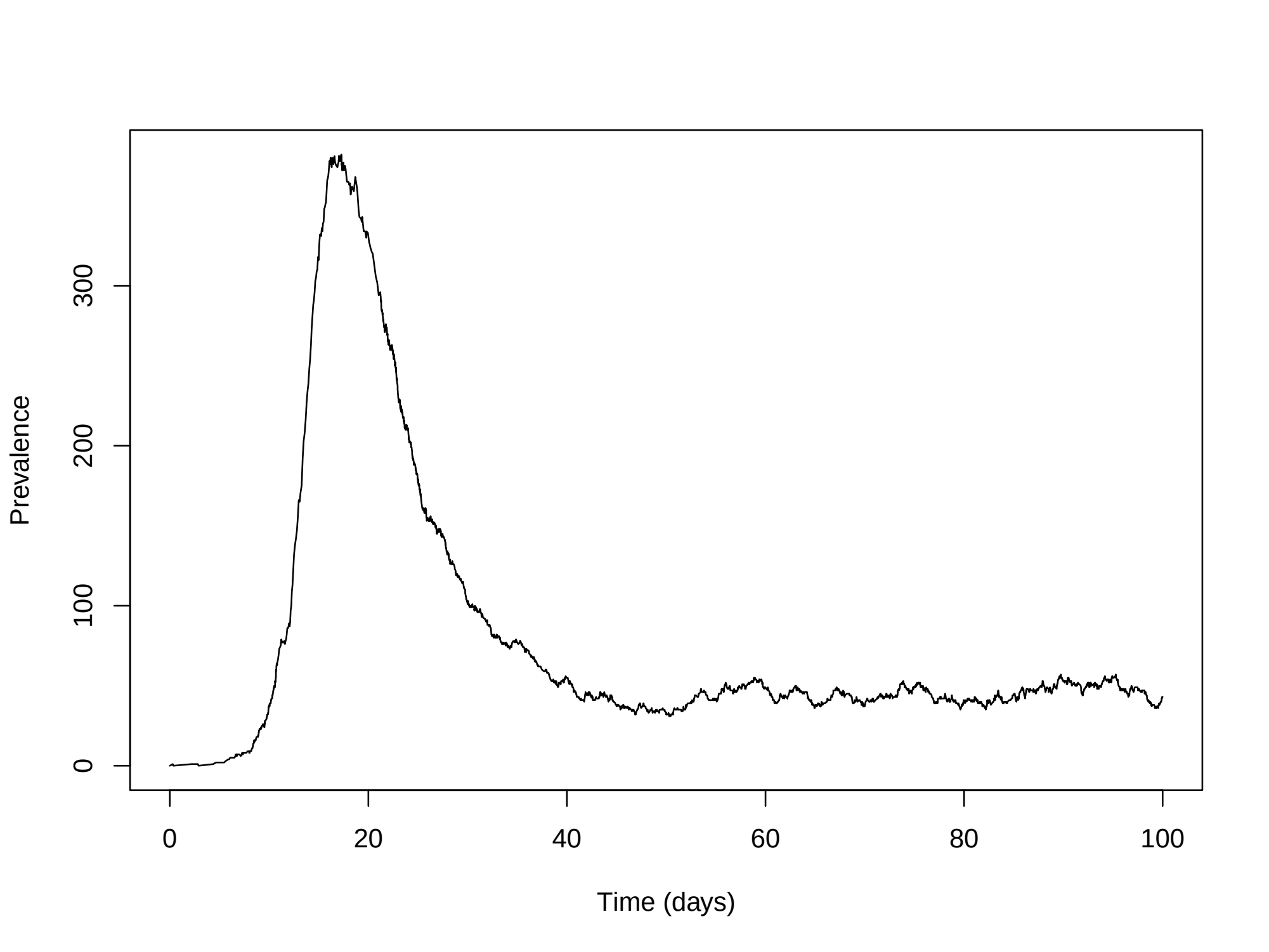
```
> colnames(sol$transitions) <- reactions_names
```

ensures we have the names of the transitions in the output matrix

Why are transitions useful?

Let's take the example of an SLIAR model

What is going on?


Prevalence
0
100
200
300
0
20
40
60
80
100
Time (days)

Who is doing the infecting?

We can go further if we look at transitions in detail

Using transitions – The not so good way

```
reactions_names <- c("new_infection",
                     "L_to_I",
..

reactions_effects <- list(
  c(S=-1, L=+1), # new infection
  c(L=-1, I=+1), # L to I
..

reactions_rates <- function(x, params, t) {
  with(as.list(c(x, params)), {
    rates <- c(
      beta*S*(I+eta*A), # new infection I
      (1-p)*epsilon*L,  # L to I
..
```

Using transitions – The good way

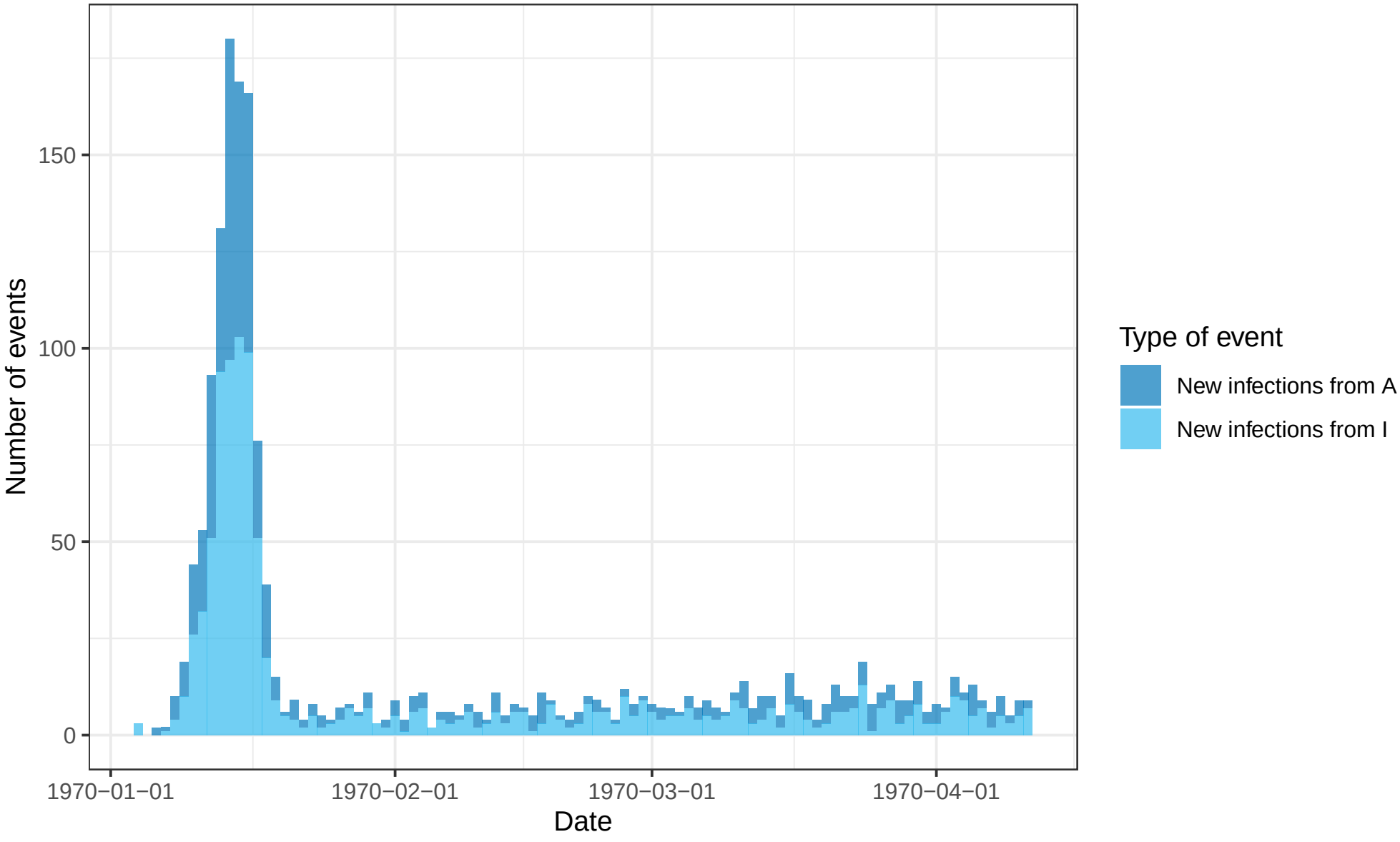
```
reactions_names <- c("new_infection_I",
                     "new_infection_A",
                     "L_to_I",
..

reactions_effects <- list(
  c(S=-1, L=+1), # new infection I
  c(S=-1, L=+1), # new infection A
  c(L=-1, I=+1), # L to I
..

reactions_rates <- function(x, params, t) {
  with(as.list(c(x, params)), {
    rates <- c(
      beta*S*I,          # new infection I
      beta*S*eta*A,      # new infection A
      (1-p)*epsilon*L,   # L to I
..
```

Representing transition results as epi diagrams

We use the libraries `incidence2` and `ggplot2`

Need to make a "line list" from the transitions, i.e., a matrix with columns `time`, `value` and `event`

See the code in the `Rnw` source of these slides and in `CODE/functions-useful.R`


Number of events
150
100
50
0
1970−01−01
1970−02−01
1970−03−01
1970−04−01
Date
Type of event
New infections from A
New infections from I

Continuous time Markov chains

ODE $\leftrightarrow$ CTMC

Simulating CTMC (in theory)

Simulating CTMC (in practice)

Parallelising your code in R

Parallelisation

To run multiple realisations, it is a good idea to parallelise your code, since CTMC simulations are *embarrassingly parallel*

Write a function, e.g., `run_one_sim` that .. runs one simulation, then call it from within a `parLapply` statement

Note: if you want to compute the mean trajectory of the realisations, you will need to interpolate solutions, since event times are different in each realisation

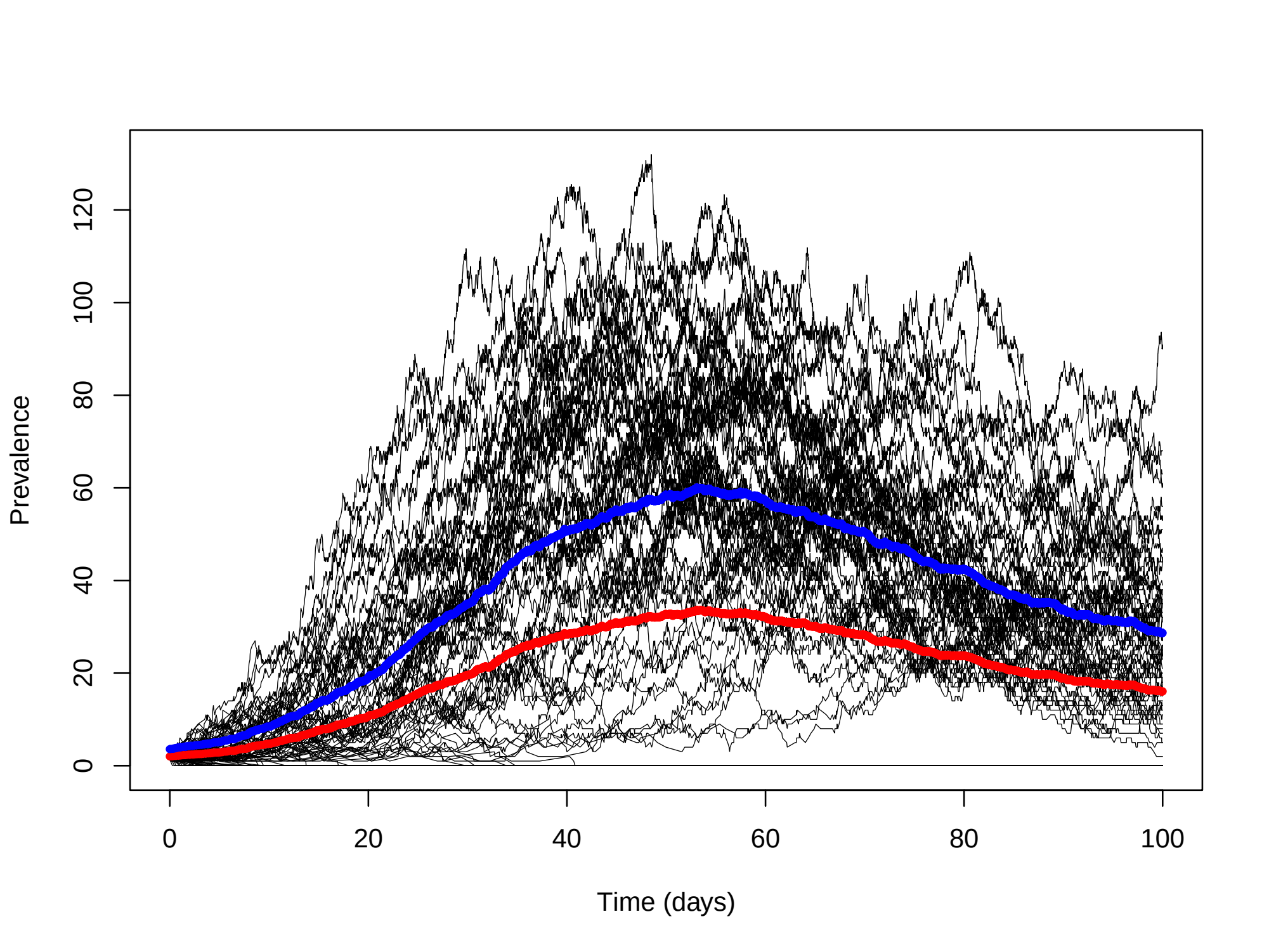
On the GitHub repo for the course, see

- `CODE/SIS-CTMC-parallel.R`
- `CODE/SIS-CTMC-parallel-multiple-R0.R`

```r
# Set parameters
# We store even the IC in there.. In parallel computations, this
# can be useful
params = list(
  Pop = 1000,
  I_0 = 2,
  t_f = 100,
  gamma = 1/5,
  nu = 1/50,
  R0 = 1.5)
params$beta <- params$gamma*params$R0/(params$Pop-params$I_0)
params$IC <- c(S = (params$Pop-params$I_0),
               I = params$I_0, R = 0)
params$reactions_names <- c("new_infection",
                            "recovery",
                            "loss_immunity")
params$reactions_effects <- list(
  c(S=-1, I=+1), # new infection
  c(I=-1, R=+1), # recovery
  c(R=-1, S=+1)  # loss of immunity
)
```

```r
library(parallel)
run_one_sim = function(params) {
  with(as.list(params), {
    set.seed(NULL) # avoid reproducibility here
    sol <- ssa.exact(
      init.values = IC,
      transitions = reactions_effects,
      rateFunc = reactions_rates,
      params = params,
      tf = t_f,
      reportTransitions = TRUE
    )
    # Interpolate result (just I will do)
    wanted_t = seq(from = 0, to = t_f, by = 0.01)
    sol$interp_I = approx(x = sol$dynamics[,"time"],
                          y = sol$dynamics[,"I"],
                          xout = wanted_t)
    names(sol$interp_I) = c("time", "I")
    # Return result
    return(sol)
  })
}
```

```r
if (FALSE) {
  SIMS = lapply(1:50, function(x) run_one_sim(params))
} else {
  nb_cores <- detectCores()-1
  if (nb_cores > 124) {
    nb_cores = 124
  }
  cl <- makeCluster(nb_cores)
  clusterEvalQ(cl,{
    library(adaptivetau)
  })
  clusterExport(cl,
                c("params",
                  "run_one_sim",
                  "reactions_rates"),
                envir = .GlobalEnv)
  SIMS = parLapply(cl = cl,
                   X = 1:100,
                   fun =  function(x) run_one_sim(params))
  stopCluster(cl)
}
```


Prevalence
0
20
40
60
80
100
120
0
20
40
60
80
100
Time (days)

Benefit of parallelisation

Run the parallel code for 100 sims between `tictoc::tic()` and `tictoc::toc()`, giving 0.353 sec elapsed, then the sequential version

```
tictoc::tic()
SIMS = lapply(X = 1:params$number_sims,
              FUN =  function(x) run_one_sim(params))
tictoc::toc()
```

which gives 1.015 sec elapsed on a 128T AMD Ryzen Threadripper 3990X 64-Core Processor (parallel is 2.88× faster)

Some words of caution – Overheads

- Overheads (setting up the cluster, providing data to workers, etc.) can be significant. For small numbers of simulations, overheads can be larger than the gains
- Be careful in particular if the function to be parallelised is very fast
- Setup time increases with more cores

In the example, we have the following

- Serial version (all included): 1.015 sec elapsed
- Parallel version:
  - Time for the parallel part: 0.353 sec elapsed
  - Overall, including setup: 1.698 sec elapsed

Some words of caution – RAM usage

- Beware of RAM usage: each worker will have to have a copy of the data, so if the data is large or you have many workers (cores/threads on your computer), this can be a problem

- Also, about RAM usage: results can quickly become large, so you may have to select what to keep and what to discard or save intermediate results to disk (it is fine to write to disk from a worker or to run a certain number of simulations, save the results and then run the next batch, without restarting the cluster)

Amdahl's law

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + p/s}$$

where

- S_{latency} is the theoretical speedup of the execution of the whole task
- s is the speedup of the part of the task that benefits from improved system resources
- p is the proportion of execution time that the part benefiting from improved resources originally occupied