Gruyere quickstart (0.1-10)

Lawrence Hudson

2012-07-30

Contents

1 Modular approach

Gruyere's modular approach is shown in Fig. 1. The `RunSimulation` function glues everything together. It evolves the system as follows:

1. get the next time-series chunk from the simulation object
2. show the chunk to each of the observers
3. show the chunk to the controller
4. if the controller says to keep going, go back to step 1
5. if the controller says to terminate the simulation, return the final system state and reason for terminating (both given by the controller) and the final system time

2 Walk-through of a simulation of a resource–consumer system

Each block in Fig. 1 is explained with reference to Gruyere's 'resource.consumer' demo - a simulation of the dynamics of a community containing a single producer and a single consumer.

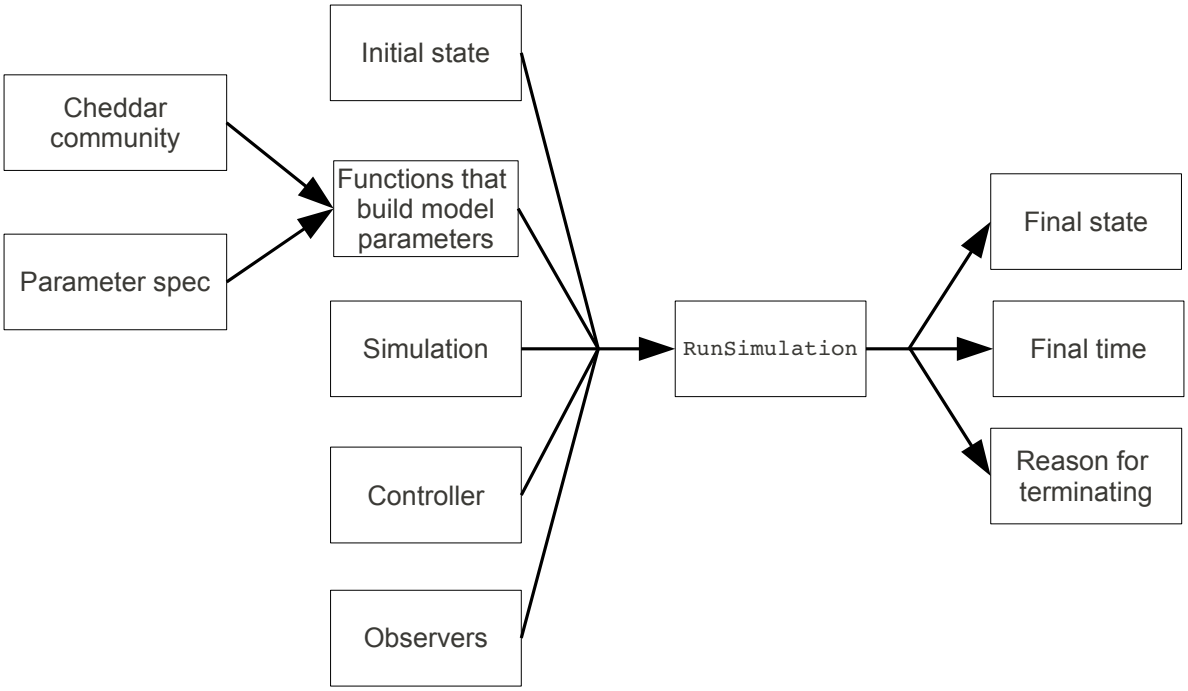


Figure 1: Gruyere's modular design.

2.1 Preamble

Load the package.

```
> library(gruyere)    # Cheddar and deSolve are loaded automatically
```

2.2 Community representation

Gruyere uses the Cheddar package to provide a community representation.

```
> # Create the Cheddar community whose dynamics will be simulated.
> community <- Community(nodes=data.frame(node=c('R','C'),
                                          category=c('producer', 'invertebrate'),
                                          M=c(0.1, 1),
                                          N=c(100, 1)),
                         trophic.links=data.frame(resource='R', consumer='C'),
                         properties=c(title='Resource-consumer',
                                      M.units='kg',
                                      N.units='m^-2'))
```

The Yodzis and Innes model requires that all populations have a valid M (with 'M.units' in kg) and that all populations have a 'category' belonging to one of 'producer', 'invertebrate', 'vert.ecto' and 'vert.endo'. The community object also defines the set of trophic links. In this case, the community also contains numerical abundance, N.

2.3 Model parameters

There are three steps to assembling model parameters. Firstly, the parameters specification: the `ModelParamsSpec` function returns a vector of single values defining for the model, functional response and growth mode parameters; values of the *f constants* and *a constants*. If called with no parameters, `ModelParamsSpec` uses values of *f constants* and *a constants* given by Yodzis and Innes (1992) (also shown in Williams et al., 2007, first table on p 44). Values can be specified as parameters to `ModelParamsSpec` to provide deviations from the default values. The example below using all *f constants* set to 1:

```
> spec <- ModelParamsSpec(f.constants=AllFConstantsEqual())
```

See also `YodzisInnes92AConstants`, `YodzisInnes92FConstants`, `BroseEtAl06AConstants`, `OttoEtAl07AConstants`. Let's examine the spec:

```
> spec
```

```
    ar.producer aT.invertebrate    aT.vert.ecto    aT.vert.endo aJ.invertebrate
          0.386           0.500           2.300          54.900           9.700
   aJ.vert.ecto    aJ.vert.endo     fr.producer fT.invertebrate    fT.vert.ecto
          8.900          89.200           1.000           1.000           1.000
   fT.vert.endo fJ.invertebrate    fJ.vert.ecto    fJ.vert.endo      e.producer
          1.000           1.000           1.000           1.000           0.450
     e.consumer              fe               W               d               q
          0.850           1.000        1000.000           0.000           0.000
              K               a
        500.000           1.000
```

`IntermediateModelParams` combines the parameter spec with the community and returns a list containing *a constants f constants*, f_J, fe, e and the functional response and growth model parameters W, d, q, K and a (Williams et al., 2007, equations 2.8–2.9 and 2.10–2.12).

```
> params <- IntermediateModelParams(community, spec)
> names(params)
```

```
 [1] "ar" "aT" "aJ" "fr" "fT" "fJ" "m"  "e"  "fe" "W"  "d"  "q"  "K"  "a"
```

The purpose of the intermediate stage is to allow deviations from the values given in the `spec`. Changes could be made per-species or per-trophic-link. For example, we might want to investigate the effect of increasing a single species' respiration rate relative to other consumers or we might want to make cannibalistic trophic links more efficient than other trophic links. It is convenient to make these kinds of changes when the parameters are in this easy-to-understand form.

`BuildModelParams` takes the list returned by `IntermediateModelParams` and returns a list containing the parameters required by the normalised model (Williams et al., 2007, equations 2.17–2.18): ρ, x, y and f_e, e and the functional response and growth model parameters W, d, q, K and a.

```
> params <- BuildModelParams(community, params) # containing rho,x,z etc
```

2.4 Simulation and model function

There is no one 'correct' way of solving systems of ordinary differential equations for numerical simulation. Gruyere's `ODESimulation` object delegates the task of solving to `ode` function provided by the deSolve package (Soetaert et al., 2010), which offers a wide range of methods. By default, the `lsoda` method used, which uses the mature and sophisticated ODEPACK fortran library (Hindmarsh, 1983; Petzold, 1983). This method detects on-the-fly whether or not the system is stiff or non-stiff and selects the appropriate method and step size. When the community being simulated exhibits body-mass ratios where consumers are much smaller than their resources (e.g. Brose et al., 2006; Otto et al., 2007), the sytem will exhibit very fast transients, which `lsoda` will not be able to guarantee to solve and the function will generate errors. In such cases, other methods may be more appropriate, such as Runge-Kutta (implemented by e.g. the `rk45dp7` method in `ode`), which pays less attention to such problems.

Gruyere defines two models functions, both of which implement the normalised model, functional response and growth model equations. The (`YodzisInnesDyDt`) function is written in C and is the implementation than will normally be used. An R implementation (`YodzisInnesDyDt_R`) is also provided. This is relatively easy to understand and is included as a reference and for testing the faster C implementation.

```
> simulation <- ODESimulation(model=YodzisInnesDyDt,
                              params=params,
                              sampling.interval=0.1)
```

The time resolution is given by 'sampling.interval'. Extinction thresholds are optional and can be set can be set per-population. If a population's biomass density drops below its extinction threshold within a simulation chunk, its biomass density is set to zero, the remainder of the simulation chunk is discarded and the simulation is restarted at the point of extinction. In this example, no extinctions will take place. By default, the simulation is run in chunks of 100 time units; we could change this by passing a value for the 'chunk.time' argument. Other parameters to the `ode` function, such as 'atol' and 'rtol' can be passed into `ODESimulation`; see the help pages for `ode` and `lsoda` for more information.

2.5 Controller

The controller governs when simulations will be terminated. Gruyere has three different controllers: `MaxTimeController`, which halts simulations when a time limit is reached, `EquilibriumController`, which halts simulations when all populations reach an equilibrium and `RunningAverageController`, which halts simulations when each population's biomass density reaches either an equilibrium or a fluctuating steady state. This example uses the simplest, which terminates simulations when a time limit is reached.

```
> controller <- MaxTimeController(100)
```

2.6 Observers

Observers are optional components that are shown time-series chunks of the simulation as it runs. An observer could potentially do anything with a time-series chunk. Gruyere contains many observers and users can write their own. Some pre-defined observers give feedback as the simulation runs, for example the `PlotNDeviationsObserver`, `PlotNvTObserver`, `PlotBDeviationsObserver` and `PlotBvTObserver` observers show the state of the simulation graphically. Two very useful observers are `CollectChunksObserver`, which records all of the simulation chunks in memory, and `WriteChunksObserver`, which writes each chunk of the time series to a file as the simulation runs. In this example, we use collect the simulation time series in memory and print out the total elapsed time of the simulation.

```
> collector <- CollectChunksObserver() # Collect simulation results in memory
> observers <- list(collector, ElapsedTimeObserver())
```

2.7 RunSimulation

We use the product of the community's M and N, computed by Cheddar's `Biomass` function, to get the initial biomass density of each species.

```
> res <- RunSimulation(initial.state=Biomass(community),
                       simulation=simulation,
                       controller=controller,
                       observers=observers)

[1] "Simulation time:"
   user  system elapsed
  0.136   0.000   0.137
```

The R `list` `res` holds the final simulation time and final biomasses for each species:

```
> res

$time
[1] 100

$final.state
       R        C
52.50601 32.79729

$terminate
[1] TRUE
```

2.8 Observers and results

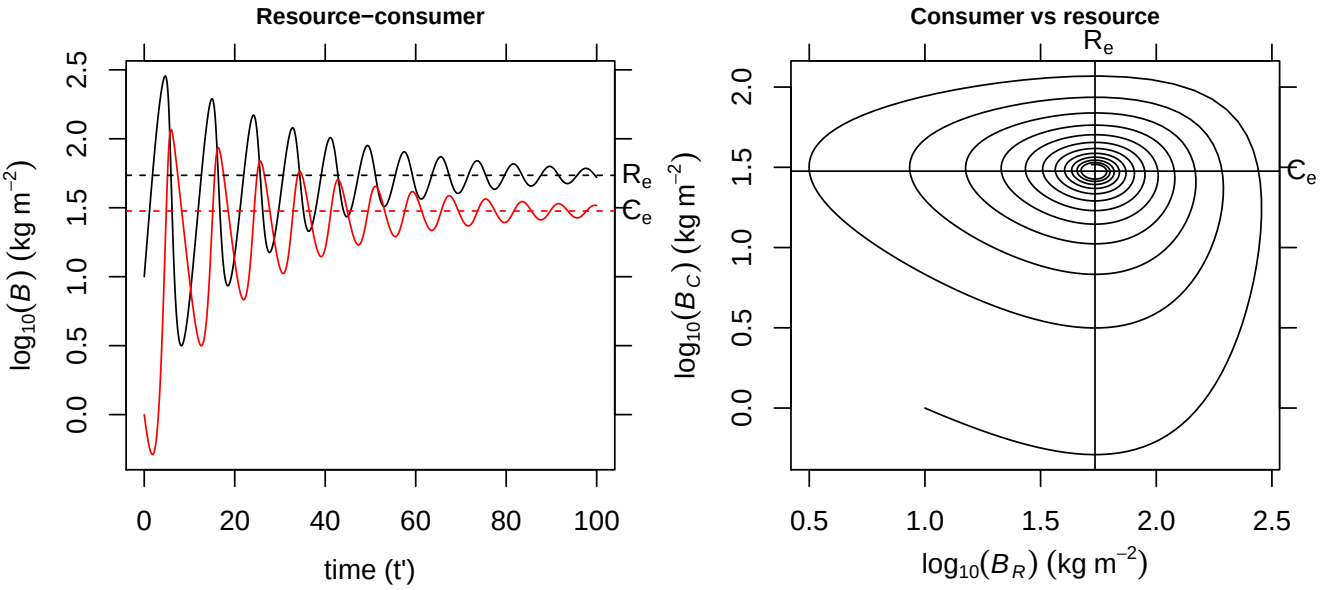
The `GetTimeSeries` function returns an R `matrix` containing the time series that was assembled by the `CollectChunksObserver` object. The first column is time, columns 2 and 3 are biomasses of the resource and consumer respectively.

```
> tseries <- GetTimeSeries(collector)
> head(tseries)

     time        R         C
[1,]  0.0 10.00000 1.0000000
[2,]  0.1 10.99530 0.9434874
[3,]  0.2 12.08922 0.8914540
[4,]  0.3 13.29103 0.8436240
[5,]  0.4 14.61075 0.7997456
[6,]  0.5 16.05920 0.7595900
```

Show the results in two plots. The first uses the Gruyere function `PlotBvT` to show $\log_{10}-transformed$ biomass against time. The second shows the $\log_{10}$-transformed biomass of the resource against the consumer.

```
> # Plot the results
> par(mfrow=c(1,2))
> PlotBvT(community, tseries, col=c(1,2))
> # Equilibria: eqns 12 and 13 of Yodzis and Innes (1992) on p.1160 using x and
> # y given in eqns 10 and 11, p 1156.
> Re <- with(params, W[1,2] / ( (y[1,2]-1)^ (1/(q+1))))
> Ce <- as.numeric(with(params, (fe[1,2]*e[1,2] / x[2]) * Re * (1-Re/K)))
> abline(h=log10(Re), lty=2)
> mtext(~R[e], side=4, at=log10(Re), las=1, line=0)
> abline(h=log10(Ce), lty=2, col=2)
> mtext(~C[e], side=4, at=log10(Ce), las=1, line=0)
> plot(log10(tseries[,'R']), log10(tseries[,'C']),
      xlab=Log10BLabel(community, name="italic(B[R])"),
      ylab=Log10BLabel(community, name="italic(B[C])"),
      type="l", main="Consumer vs resource")
> axis(side=3, labels=FALSE)
> axis(side=4, labels=FALSE)
> abline(v=log10(Re))
> mtext(~R[e], side=3, at=log10(Re), las=1, line=0)
> abline(h=log10(Ce))
> mtext(~C[e], side=4, at=log10(Ce), las=1, line=0)
```



References

U. Brose, R.J. Williams, and N.D. Martinez. Allometric scaling enhances stability in complex food webs. 9(11):1228–1236, 2006. doi: 10.1111/j.1461-0248.2006.00978.x.

A C Hindmarsh. Odepack, a systematized collection of ode solvers. In R S Stepleman, editor, *IMACS Transactions on Scientific Computation*, volume 1, pages 55–64. North-Holland, Amsterdam, 1983.

S.B. Otto, B.C. Rall, and U. Brose. Allometric degree distributions facilitate food-web stability. 450 (7173):1226–1229, 2007. doi: 10.1038/nature06359.

L. Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. 4(1):136–148, 1983.

Karline Soetaert, Thomas Petzoldt, and R. Woodrow Setzer. *Solving Differential Equations in R: Package deSolve*, 2010. URL `http://www.jstatsoft.org/v33/i09`.

Richard J Williams, Ulrich Brose, and Neo D Martinez. Homage to Yodzis and Innes 1992: scaling up feeding-based population dynamics to complex ecological networks. In N Rooney, Kevin S McCann, and D L G Noakes, editors, *From energetics to ecosystems: the dynamics and structure of ecological systems*, chapter 2, pages 37–51. Springer, 2007.

Peter Yodzis and Stuart Innes. Body size and resource-consumer dynamics. 139(6):1151–1175, 1992. doi: 10.1086/285380.