# DiffEqBiological Tutorial II: Network Properties API

Samuel Isaacson

September 3, 2021

The DiffEqBiological API provides a collection of functions for easily accessing network properties, and for incrementally building and extending a network. In this tutorial we'll go through the API, and then illustrate how to programmatically construct a network.

We'll illustrate the API using a toggle-switch like network that contains a variety of different reaction types:

```
using DifferentialEquations, DiffEqBiological, Latexify, Plots
fmt = :svg
pyplot(fmt=fmt)
rn = @reaction_network begin
    hillr(D_2,α,K,n), ∅ --> m_1
    hillr(D_1,α,K,n), ∅ --> m_2
    (δ,γ), m_1 ↔ ∅
    (δ,γ), m_2 ↔ ∅
    β, m_1 --> m_1 + P_1
    β, m_2 --> m_2 + P_2
    μ, P_1 --> ∅
    μ, P_2 --> ∅
    (k_+,k_-), 2P_1 ↔ D_1
    (k_+,k_-), 2P_2 ↔ D_2
    (k_+,k_-), P_1+P_2 ↔ T
end α K n δ γ β μ k_+ k_-;
```

```
Error: ArgumentError: Package DiffEqBiological not found in current path:
- Run `import Pkg; Pkg.add("DiffEqBiological")` to install the DiffEqBiolog
ical package.
```

This corresponds to the chemical reaction network given by

```
latexify(rn; env=:chemical)
```

```
Error: UndefVarError: latexify not defined
```

---

## 0.1 Network Properties

Basic properties of the generated network include the `speciesmap` and `paramsmap` functions we examined in the last tutorial, along with the corresponding `species` and `params` functions:

```
species(rn)
```

```
Error: UndefVarError: species not defined
```

```
params(rn)
```

```
Error: UndefVarError: params not defined
```

The numbers of species, parameters and reactions can be accessed using `numspecies(rn)`, `numparams(rn)` and `numreactions(rn)`.

A number of functions are available to access properties of reactions within the generated network, including `substrates`, `products`, `dependents`, `ismassaction`, `substratestoich`, `substratesymstoich`, `productstoich`, `productsymstoich`, and `netstoich`. Each of these functions takes two arguments, the reaction network `rn` and the index of the reaction to query information about. For example, to find the substrate symbols and their corresponding stoichiometries for the 11th reaction, `2P_1 --> D_1`, we would use

```
substratesymstoich(rn, 11)
```

```
Error: UndefVarError: substratesymstoich not defined
```

Broadcasting works on all these functions, allowing the construction of a vector holding the queried information across all reactions, i.e.

```
substratesymstoich.(rn, 1:numreactions(rn))
```

```
Error: UndefVarError: numreactions not defined
```

To see the net stoichiometries for all reactions we would use

```
netstoich.(rn, 1:numreactions(rn))
```

```
Error: UndefVarError: numreactions not defined
```

Here the first integer in each pair corresponds to the index of the species (with symbol `species(rn)[index]`). The second integer corresponds to the net stoichiometric coefficient of the species within the reaction. `substratestoich` and `productstoich` are defined similarly.

Several functions are also provided that calculate different types of dependency graphs. These include `rxtospecies_depgraph`, which provides a mapping from reaction index to the indices of species whose population changes when the reaction occurs:

```
rxtospecies_depgraph(rn)
```

```
Error: UndefVarError: rxtospecies_depgraph not defined
```

Here the last row indicates that the species with indices `[3,4,7]` will change values when the reaction `T --> P_1 + P_2` occurs. To confirm these are the correct species we can look at

```
species(rn)[[3,4,7]]
```

```
Error: UndefVarError: species not defined
```

The `speciestorx_depgraph` similarly provides a mapping from species to reactions for which their *rate laws* depend on that species. These correspond to all reactions for which the given species is in the `dependent` set of the reaction. We can verify this for the first species, `m_1`:

```
speciestorx_depgraph(rn)[1]
```

```
Error: UndefVarError: speciestorx_depgraph not defined
```

```
findall(depset -> in(:m_1, depset), dependents.(rn, 1:numreactions(rn)))
```

```
Error: UndefVarError: numreactions not defined
```

Finally, `rxtorx_depgraph` provides a mapping that shows when a given reaction occurs, which other reactions have rate laws that involve species whose value would have changed:

```
rxtorx_depgraph(rn)
```

```
Error: UndefVarError: rxtorx_depgraph not defined
```

**Note on Using Network Property API Functions** Many basic network query and reaction property functions are simply accessors, returning information that is already stored within the generated `reaction_network`. For these functions, modifying the returned data structures may lead to inconsistent internal state within the network. As such, they should be used for accessing, but not modifying, network properties. The API documentation indicates which functions return newly allocated data structures and which return data stored within the `reaction_network`.

---

## 0.2 Incremental Construction of Networks

The `@reaction_network` macro is monolithic, in that it not only constructs and stores basic network properties such as the reaction stoichiometries, but also generates **everything** needed to immediately solve ODE, SDE and jump models using the network. This includes Jacobian functions, noise functions, and jump functions for each reaction. While this allows for a compact interface to the DifferentialEquations.jl solvers, it can also be computationally expensive for large networks, where a user may only wish to solve one type of problem and/or have fine-grained control over what is generated. In addition, some types of reaction network structures are more amenable to being constructed programmatically, as opposed to writing out all reactions by hand within one macro. For these reasons DiffEqBiological provides two additional macros that only *initially* setup basic reaction network properties, and which can be extended through a programmatic interface: `@min_reaction_network` and `@empty_reaction_network`. We now give an introduction to constructing these more minimal network representations, and how they can be programmatically extended. See also the relevant API section.

The `@min_reaction_network` macro works identically to the `@reaction_network` macro, but the generated network will only be complete with respect to its representation of chemical network properties (i.e. species, parameters and reactions). No ODE, SDE or jump models are generated during the macro call. It can subsequently be extended with the addition of new species, parameters or reactions. The `@empty_reaction_network` allocates an empty network structure that can also be extended using the programmatic interface. For example, consider a partial version of the toggle-switch like network we defined above:

```
rnmin = @min_reaction_network begin
    (δ,γ), m_1 ↔ ∅
    (δ,γ), m_2 ↔ ∅
    β, m_1 --> m_1 + P_1
    β, m_2 --> m_2 + P_2
    μ, P_1 --> ∅
```

```
    μ, P_2 --> ∅
end δ γ β μ;
```

```
Error: LoadError: UndefVarError: @min_reaction_network not defined
in expression starting at /var/lib/buildkite-agent/builds/1-amdci4-julia-cs
ail-mit-edu/julialang/scimltutorials-dot-jl/tutorials/models/04-diffeqbio_I
I_networkproperties.jmd:2
```

Here we have left out the first two, and last three, reactions from the original `reaction_network`. To expand the network until it is functionally equivalent to the original model we add back in the missing species, parameters, and *finally* the missing reactions. Note, it is required that species and parameters be defined before any reactions using them are added. The necessary network extension functions are given by `addspecies!`, `addparam!` and `addreaction!`, and described in the API. To complete `rnmin` we first add the relevant species:

```
addspecies!(rnmin, :D_1)
addspecies!(rnmin, :D_2)
addspecies!(rnmin, :T)
```

```
Error: UndefVarError: addspecies! not defined
```

Next we add the needed parameters

```
addparam!(rnmin, :α)
addparam!(rnmin, :K)
addparam!(rnmin, :n)
addparam!(rnmin, :k_+)
addparam!(rnmin, :k_−)
```

```
Error: UndefVarError: addparam! not defined
```

Note, both `addspecies!` and `addparam!` also accept strings encoding the variable names (which are then converted to `Symbol`s internally).

We are now ready to add the missing reactions. The API provides two forms of the `addreaction!` function, one takes expressions analogous to what one would write in the macro:

```
addreaction!(rnmin, :(hillr(D_1,α,K,n)), :(∅ --> m_2))
addreaction!(rnmin, :((k_+,k_−)), :(2P_2 ↔ D_2))
addreaction!(rnmin, :k_+, :(2P_1 --> D_1))
addreaction!(rnmin, :k_−, :(D_1 --> 2P_1))
```

```
Error: UndefVarError: addreaction! not defined
```

The rate can be an expression or symbol as above, but can also just be a numeric value. The second form of `addreaction!` takes tuples of `Pair{Symbol,Int}` that encode the stoichiometric coefficients of substrates and reactants:

```
# signature is addreaction!(rnmin, paramexpr, substratestoich, productstoich)
addreaction!(rnmin, :(hillr(D_2,α,K,n)), (), (:m_1 => 1,))
addreaction!(rnmin, :k_+, (:P_1=>1, :P_2=>1), (:T=>1,))
addreaction!(rnmin, :k_−, (:T=>1,), (:P_1=>1, :P_2=>1))
```

```
Error: UndefVarError: addreaction! not defined
```

Let's check that `rn` and `rnmin` have the same set of species:

```
setdiff(species(rn), species(rnmin))
```

```
Error: UndefVarError: species not defined
```

the same set of params:

```
setdiff(params(rn), params(rnmin))
```

```
Error: UndefVarError: params not defined
```

and the final reaction has the same substrates, reactions, and rate expression:

```
rxidx = numreactions(rn)
setdiff(substrates(rn, rxidx), substrates(rnmin, rxidx))
```

```
Error: UndefVarError: numreactions not defined
```

```
setdiff(products(rn, rxidx), products(rnmin, rxidx))
```

```
Error: UndefVarError: products not defined
```

```
rateexpr(rn, rxidx) == rateexpr(rnmin, rxidx)
```

```
Error: UndefVarError: rateexpr not defined
```

---

## 0.3 Extending Incrementally Generated Networks to Include ODEs, SDEs or Jumps

Once a network generated from `@min_reaction_network` or `@empty_reaction_network` has had all the associated species, parameters and reactions filled in, corresponding ODE, SDE or jump models can be constructed. The relevant API functions are `addodes!`, `addsdes!` and `addjumps!`. One benefit to contructing models with these functions is that they offer more fine-grained control over what actually gets constructed. For example, `addodes!` has the optional keyword argument, `build_jac`, which if set to `false` will disable construction of symbolic Jacobians and functions for evaluating Jacobians. For large networks this can give a significant speed-up in the time required for constructing an ODE model. Each function and its associated keyword arguments are described in the API section, Functions to add ODEs, SDEs or Jumps to a Network.

Let's extend `rnmin` to include the needed functions for use in ODE solvers:

```
addodes!(rnmin)
```

```
Error: UndefVarError: addodes! not defined
```

The Generated Functions for Models section of the API shows what functions have been generated. For ODEs these include `oderhsfun(rnmin)`, which returns a function of the form `f(du,u,p,t)` which evaluates the ODEs (i.e. the time derivatives of `u`) within `du`. For each generated function, the corresponding expressions from which it was generated can be retrieved using accessors from the Generated Expressions section of the API. The equations within `du` can be retrieved using the `odeexprs(rnmin)` function. For example:

```
odeexprs(rnmin)
```

```
Error: UndefVarError: odeexprs not defined
```

Using Latexify we can see the ODEs themselves to compare with these expressions:

```
latexify(rnmin)
```

```
Error: UndefVarError: latexify not defined
```

For ODEs two other functions are generated by `addodes!`. `jacfun(rnmin)` will return the generated Jacobian evaluation function, `fjac(dJ,u,p,t)`, which given the current solution `u` evaluates the Jacobian within `dJ`. `jacobianexprs(rnmin)` gives the corresponding matrix of expressions, which can be used with Latexify to see the Jacobian:

```
latexify(jacobianexprs(rnmin))
```

```
Error: UndefVarError: jacobianexprs not defined
```

`addodes!` also generates a function that evaluates the Jacobian of the ODE derivative functions with respect to the parameters. `paramjacfun(rnmin)` then returns the generated function. It has the form `fpjac(dPJ,u,p,t)`, which given the current solution `u` evaluates the Jacobian matrix with respect to parameters `p` within `dPJ`. For use in DifferentialEquations.jl solvers, an `ODEFunction` representation of the ODEs is available from `odefun(rnmin)`.

`addsdes!` and `addjumps!` work similarly to complete the network for use in StochasticDiffEq and DiffEqJump solvers.

**Note on Using Generated Function and Expression API Functions** The generated functions and expressions accessible through the API require first calling the appropriate `addodes!`, `addsdes` or `addjumps` function. These are responsible for actually constructing the underlying functions and expressions. The API accessors simply return already constructed functions and expressions that are stored within the `reaction_network` structure.

---

## 0.4 Example of Generating a Network Programmatically

For a user directly typing in a reaction network, it is generally easier to use the `@min_reaction_network` or `@reaction_network` macros to fully specify reactions. However, for large, structured networks it can be much easier to generate the network programmatically. For very large networks, with tens of thousands of reactions, the form of `addreaction!` that uses stoichiometric coefficients should be preferred as it offers substantially better performance. To put together everything we've seen, let's generate the network corresponding to a 1D continuous time random walk, approximating the diffusion of molecules within an interval.

The basic "reaction" network we wish to study is

$$u_1 \leftrightarrows u_2 \leftrightarrows u_3 \cdots \leftrightarrows u_N$$

for $N$ lattice sites on $[0, 1]$. For $h = 1/N$ the lattice spacing, we'll assume the rate molecules hop from their current site to any particular neighbor is just $h^{-2}$. We can interpret this hopping process as a collection of $2N - 2$ "reactions", with the form $u_i \to u_j$ for $j = i + 1$ or $j = i - 1$. We construct the corresponding reaction network as follows. First we set values for the basic parameters:

```
N = 64
h = 1 / N
```

```
0.015625
```

then we create an empty network, and add each species

```
rn = @empty_reaction_network

for i = 1:N
    addspecies!(rn, Symbol(:u, i))
end
```

```
Error: LoadError: UndefVarError: @empty_reaction_network not defined
in expression starting at /var/lib/buildkite-agent/builds/1-amdci4-julia-cs
ail-mit-edu/julialang/scimltutorials-dot-jl/tutorials/models/04-diffeqbio_I
I_networkproperties.jmd:2
```

We next add one parameter $\beta$, which we will set equal to the hopping rate of molecules, $h^{-2}$:

```
addparam!(rn, :β)
```

```
Error: UndefVarError: addparam! not defined
```

Finally, we add in the $2N-2$ possible hopping reactions:

```
for i = 1:N
    (i < N) && addreaction!(rn, :β, (Symbol(:u,i)=>1,), (Symbol(:u,i+1)=>1,))
    (i > 1) && addreaction!(rn, :β, (Symbol(:u,i)=>1,), (Symbol(:u,i-1)=>1,))
end
```

```
Error: UndefVarError: addreaction! not defined
```

Let's first construct an ODE model for the network

```
addodes!(rn)
```

```
Error: UndefVarError: addodes! not defined
```

We now need to specify the initial condition, parameter vector and time interval to solve on. We start with 10000 molecules placed at the center of the domain, and setup an `ODEProblem` to solve:

```
u_0 = zeros(N)
u_0[div(N,2)] = 10000
p = [1/(h*h)]
tspan = (0.,.01)
oprob = ODEProblem(rn, u_0, tspan, p)
```

```
Error: UndefVarError: rn not defined
```

We are now ready to solve the problem and plot the solution. Since we have essentially generated a method of lines discretization of the diffusion equation with a discontinuous initial condition, we'll use an A-L stable implicit ODE solver, `Rodas5`, and plot the solution at a few times:

```
sol = solve(oprob, Rodas5())
times = [0., .0001, .001, .01]
plt = plot()
for time in times
    plot!(plt, 1:N, sol(time), fmt=fmt, xlabel="i", ylabel="u_i", label=string("t = ",
time), lw=3)
end
plot(plt, ylims=(0.,10000.))
```

```
Error: UndefVarError: oprob not defined
```

Here we see the characteristic diffusion of molecules from the center of the domain, resulting in a shortening and widening of the solution as $t$ increases.

Let's now look at a stochastic chemical kinetics jump process version of the model, where $\beta$ gives the probability per time each molecule can hop from its current lattice site to an individual neighboring site. We first add in the jumps, disabling `regular_jumps` since they are not needed, and using the `minimal_jumps` flag to construct a minimal representation of the needed jumps. We then construct a `JumpProblem`, and use the Composition-Rejection Direct method, `DirectCR`, to simulate the process of the molecules hopping about on the lattice:

```
addjumps!(rn, build_regular_jumps=false, minimal_jumps=true)

# make the initial condition integer valued
u_0 = zeros(Int, N)
u_0[div(N,2)] = 10000

# setup and solve the problem
dprob = DiscreteProblem(rn, u_0, tspan, p)
jprob = JumpProblem(dprob, DirectCR(), rn, save_positions=(false,false))
jsol = solve(jprob, SSAStepper(), saveat=times)
```

```
Error: UndefVarError: addjumps! not defined
```

We can now plot bar graphs showing the locations of the molecules at the same set of times we examined the ODE solution. For comparison, we also plot the corresponding ODE solutions (red lines) that we found:

```
times = [0., .0001, .001, .01]
plts = []
for i = 1:4
    b = bar(1:N, jsol[i], legend=false, fmt=fmt, xlabel="i", ylabel="u_i",
title=string("t = ", times[i]))
    plot!(b,sol(times[i]))
    push!(plts,b)
end
plot(plts...)
```

```
Error: UndefVarError: jsol not defined
```

Similar to the ODE solutions, we see that the molecules spread out and become more and more well-mixed throughout the domain as $t$ increases. The simulation results are noisy due to the finite numbers of molecules present in the stochsatic simulation, but since the number of molecules is large they agree well with the ODE solution at each time.

---

## 0.5 Getting Help

Have a question related to DiffEqBiological or this tutorial? Feel free to ask in the DifferentialEquations.jl Gitter. If you think you've found a bug in DiffEqBiological, or would like to request/discuss new functionality, feel free to open an issue on Github (but please check there is no related issue already open). If you've found a bug in this tutorial, or have a

suggestion, feel free to open an issue on the SciMLTutorials Github site. Or, submit a pull request to SciMLTutorials updating the tutorial!

---

```
Error: MethodError: no method matching tutorial_footer(::String, ::String;
remove_homedir=true)
Closest candidates are:
  tutorial_footer(::Any, ::Any) at /var/lib/buildkite-agent/builds/1-amdci4
-julia-csail-mit-edu/julialang/scimltutorials-dot-jl/src/SciMLTutorials.jl:
79 got unsupported keyword argument "remove_homedir"
  tutorial_footer(::Any) at /var/lib/buildkite-agent/builds/1-amdci4-julia-
csail-mit-edu/julialang/scimltutorials-dot-jl/src/SciMLTutorials.jl:79 got
unsupported keyword argument "remove_homedir"
```