DiffEqBiological Tutorial I: Introduction

Samuel Isaacson

October 18, 2019

DiffEqBiological.jl is a domain specific language (DSL) for writing chemical reaction networks in Julia. The generated chemical reaction network model can then be translated into a variety of mathematical models which can be solved using components of the broader DifferentialEquations.jl ecosystem.

In this tutorial we'll provide an introduction to using DiffEqBiological to specify chemical reaction networks, and then to solve ODE, jump, tau-leaping and SDE models generated from them. Let's start by using the DiffEqBiological `reaction_network` macro to specify a simply chemical reaction network; the well-known Repressilator.

We first import the basic packages we'll need, and use Plots.jl for making figures:

```
# If not already installed, first hit "]" within a Julia REPL. Then type:
# add DifferentialEquations DiffEqBiological PyPlot Plots Latexify

using DifferentialEquations, DiffEqBiological, Plots, Latexify
pyplot(fmt=:svg);
```

We now construct the reaction network. The basic types of arrows and predefined rate laws one can use are discussed in detail within the DiffEqBiological Chemical Reaction Models documentation. Here we use a mix of first order, zero order and repressive Hill function rate laws. Note, $\varnothing$ corresponds to the empty state, and is used for zeroth order production and first order degradation reactions:

```
repressilator = @reaction_network begin
    hillr(P_3,α,K,n), ∅ --> m_1
    hillr(P_1,α,K,n), ∅ --> m_2
    hillr(P_2,α,K,n), ∅ --> m_3
    (δ,γ), m_1 ↔ ∅
    (δ,γ), m_2 ↔ ∅
    (δ,γ), m_3 ↔ ∅
    β, m_1 --> m_1 + P_1
    β, m_2 --> m_2 + P_2
    β, m_3 --> m_3 + P_3
    μ, P_1 --> ∅
    μ, P_2 --> ∅
    μ, P_3 --> ∅
end α K n δ γ β μ;
```

We can use Latexify to look at the corresponding reactions and understand the generated rate laws for each reaction

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

$$\varnothing \xrightarrow{\frac{\alpha \cdot \mathrm{K}^n}{\mathrm{K}^n + \mathrm{P_3}^n}} \mathrm{m_1}$$
$$\varnothing \xrightarrow{\frac{\alpha \cdot \mathrm{K}^n}{\mathrm{K}^n + \mathrm{P_1}^n}} \mathrm{m_2}$$
$$\varnothing \xrightarrow{\frac{\alpha \cdot \mathrm{K}^n}{\mathrm{K}^n + \mathrm{P_2}^n}} \mathrm{m_3}$$
$$\mathrm{m_1} \underset{\gamma}{\overset{\delta}{\rightleftharpoons}} \varnothing$$
$$\mathrm{m_2} \underset{\gamma}{\overset{\delta}{\rightleftharpoons}} \varnothing$$
$$\mathrm{m_3} \underset{\gamma}{\overset{\delta}{\rightleftharpoons}} \varnothing$$
$$\mathrm{m_1} \xrightarrow{\beta} \mathrm{m_1} + \mathrm{P_1}$$
$$\mathrm{m_2} \xrightarrow{\beta} \mathrm{m_2} + \mathrm{P_2}$$
$$\mathrm{m_3} \xrightarrow{\beta} \mathrm{m_3} + \mathrm{P_3}$$
$$\mathrm{P_1} \xrightarrow{\mu} \varnothing$$
$$\mathrm{P_2} \xrightarrow{\mu} \varnothing$$
$$\mathrm{P_3} \xrightarrow{\mu} \varnothing$$

We can also use Latexify to look at the corresponding ODE model for the chemical system

```
latexify(repressilator, cdot=false)
```

$$\begin{aligned}
\frac{dm(t)}{dt} &= \frac{\alpha K^n}{K^n + P_3^n} - \delta m_1 + \gamma \\
\frac{dm(t)}{dt} &= \frac{\alpha K^n}{K^n + P_1^n} - \delta m_2 + \gamma \\
\frac{dm(t)}{dt} &= \frac{\alpha K^n}{K^n + P_2^n} - \delta m_3 + \gamma \\
\frac{dP(t)}{dt} &= \beta m_1 - \mu P_1 \\
\frac{dP(t)}{dt} &= \beta m_2 - \mu P_2 \\
\frac{dP(t)}{dt} &= \beta m_3 - \mu P_3
\end{aligned}$$

To solve the ODEs we need to specify the values of the parameters in the model, the initial condition, and the time interval to solve the model on. To do this it helps to know the orderings of the parameters and the species. Parameters are ordered in the same order they appear after the `end` statement in the `@reaction_network` macro. Species are ordered in the order they first appear within the `@reaction_network` macro. We can see these orderings using the `speciesmap` and `paramsmap` functions:

```
speciesmap(repressilator)
```

```
OrderedCollections.OrderedDict{Symbol,Int64} with 6 entries:
```

```
  :m_1 => 1
  :m_2 => 2
  :m_3 => 3
  :P_1 => 4
  :P_2 => 5
  :P_3 => 6
```

```
paramsmap(repressilator)
```

```
OrderedCollections.OrderedDict{Symbol,Int64} with 7 entries:
  :α => 1
  :K => 2
  :n => 3
  :δ => 4
  :γ => 5
  :β => 6
  :μ => 7
```

0.1 Solving the ODEs:

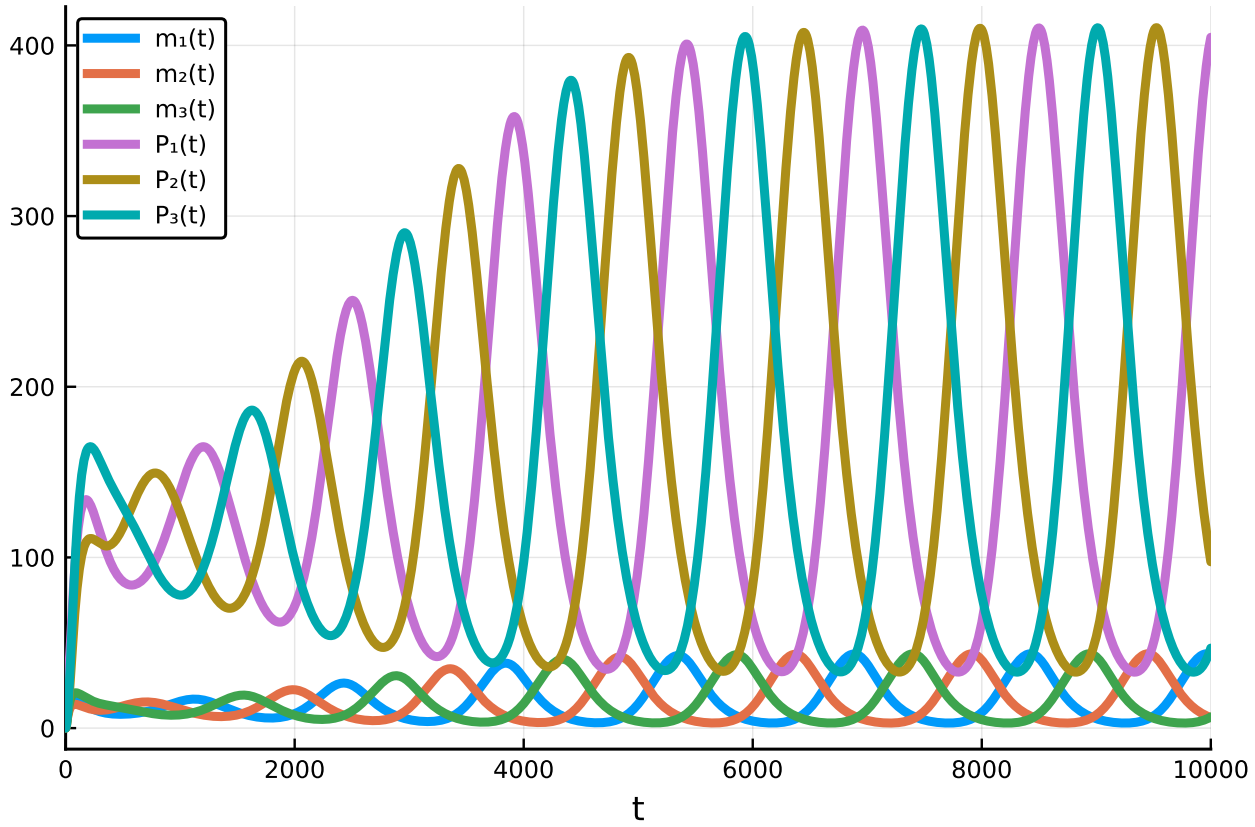
Knowing these orderings, we can create parameter and initial condition vectors, and setup the `ODEProblem` we want to solve:

```
# parameters [α,K,n,δ,γ,β,μ]
p = (.5, 40, 2, log(2)/120, 5e-3, 20*log(2)/120, log(2)/60)

# initial condition [m_1,m_2,m_3,P_1,P_2,P_3]
u_0 = [0.,0.,0.,20.,0.,0.]

# time interval to solve on
tspan = (0., 10000.)

# create the ODEProblem we want to solve
oprob = ODEProblem(repressilator, u_0, tspan, p)
```

```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 10000.0)
u0: [0.0, 0.0, 0.0, 20.0, 0.0, 0.0]
```

At this point we are all set to solve the ODEs. We can now use any ODE solver from within the DiffEq package. We'll just use the default DifferentialEquations solver for now, and then plot the solutions:

```
sol = solve(oprob, saveat=10.)
plot(sol, fmt=:svg)
```

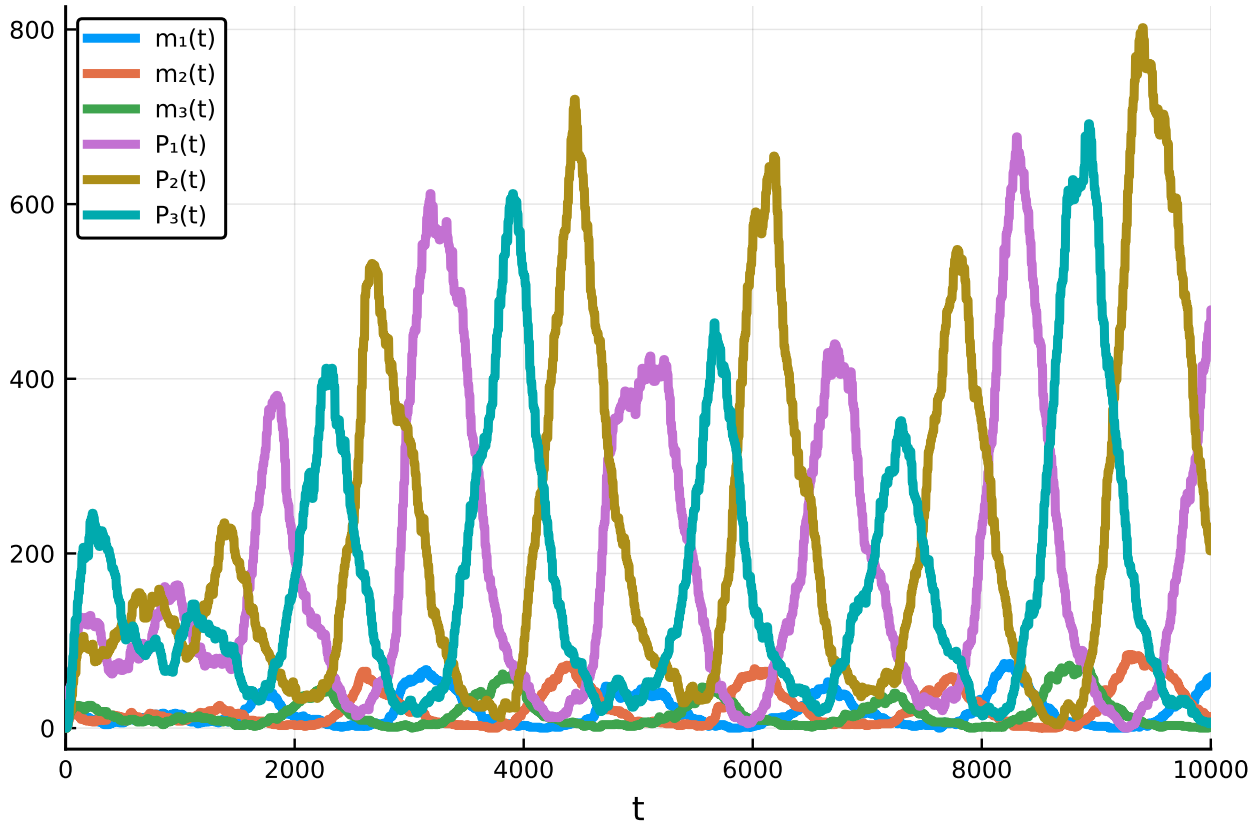


We see the well-known oscillatory behavior of the repressilator! For more on choices of ODE solvers, see the JuliaDiffEq documentation.

0.2 Stochastic Simulation Algorithms (SSAs) for Stochastic Chemical Kinetics

Let's now look at a stochastic chemical kinetics model of the repressilator, modeling it with jump processes. Here we will construct a DiffEqJump `JumpProblem` that uses Gillespie's `Direct` method, and then solve it to generate one realization of the jump process:

```
# first we redefine the initial condition to be integer valued
u_0 = [0,0,0,20,0,0]

# next we create a discrete problem to encode that our species are integer valued:
dprob = DiscreteProblem(repressilator, u_0, tspan, p)

# now we create a JumpProblem, and specify Gillespie's Direct Method as the solver:
jprob = JumpProblem(dprob, Direct(), repressilator, save_positions=(false,false))

# now let's solve and plot the jump process:
sol = solve(jprob, SSAStepper(), saveat=10.)
plot(sol, fmt=:svg)
```

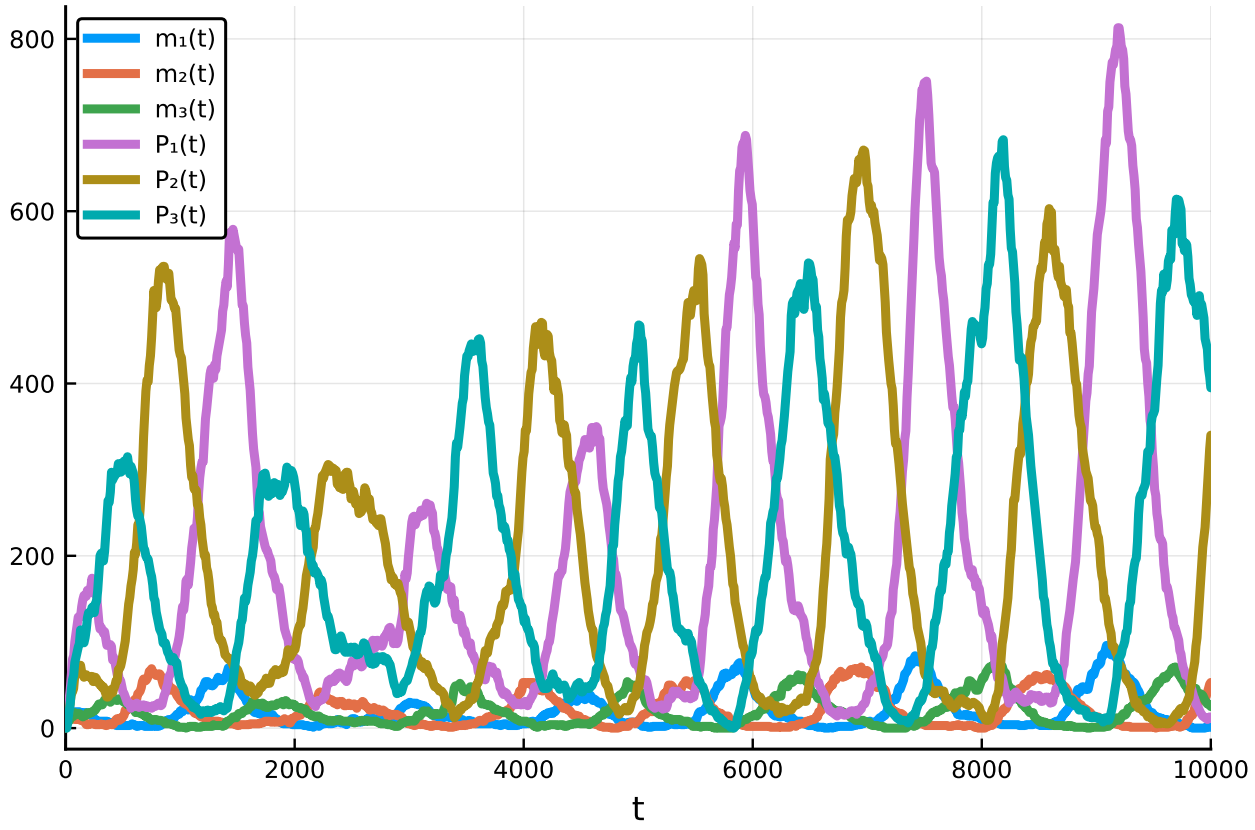


Here we see that oscillations remain, but become much noiser. Note, in constructing the `JumpProblem` we could have used any of the SSAs that are part of DiffEqJump instead of the `Direct` method, see the list of SSAs (i.e. constant rate jump aggregators) in the documentation.

0.3 τ-leaping Methods:

While SSAs generate exact realizations for stochastic chemical kinetics jump process models, τ-leaping methods offer a performant alternative by discretizing in time the underlying time-change representation of the stochastic process. The DiffEqJump package has limited support for τ-leaping methods in the form of the basic Euler's method type approximation proposed by Gillespie. We can simulate a τ-leap approximation to the repressilator by using the `RegularJump` representation of the network to construct a `JumpProblem`:

```
rjs = regularjumps(repressilator)
lprob = JumpProblem(dprob, Direct(), rjs)
lsol = solve(lprob, SimpleTauLeaping(), dt=.1)
plot(lsol, plotdensity=1000, fmt=:svg)
```



0.4 Chemical Langevin Equation (CLE) Stochastic Differential Equation (SDE) Models:

At an intermediary physical scale between macroscopic ODE models and microscopic stochastic chemical kinetic models lies the CLE, a SDE version of the model. The SDEs add to each ODE above a noise term. As the repressilator has species that get very close to zero in size, it is not a good candidate to model with the CLE (where solutions can then go negative and become unphysical). Let's create a simpler reaction network for a birth-death process that will stay non-negative:

```
bdp = @reaction_network begin
  c_1, X --> 2X
  c_2, X --> 0
  c_3, 0 --> X
end c_1 c_2 c_3
p = (1.0,2.0,50.)
u_0 = [5.]
tspan = (0.,4.);
```

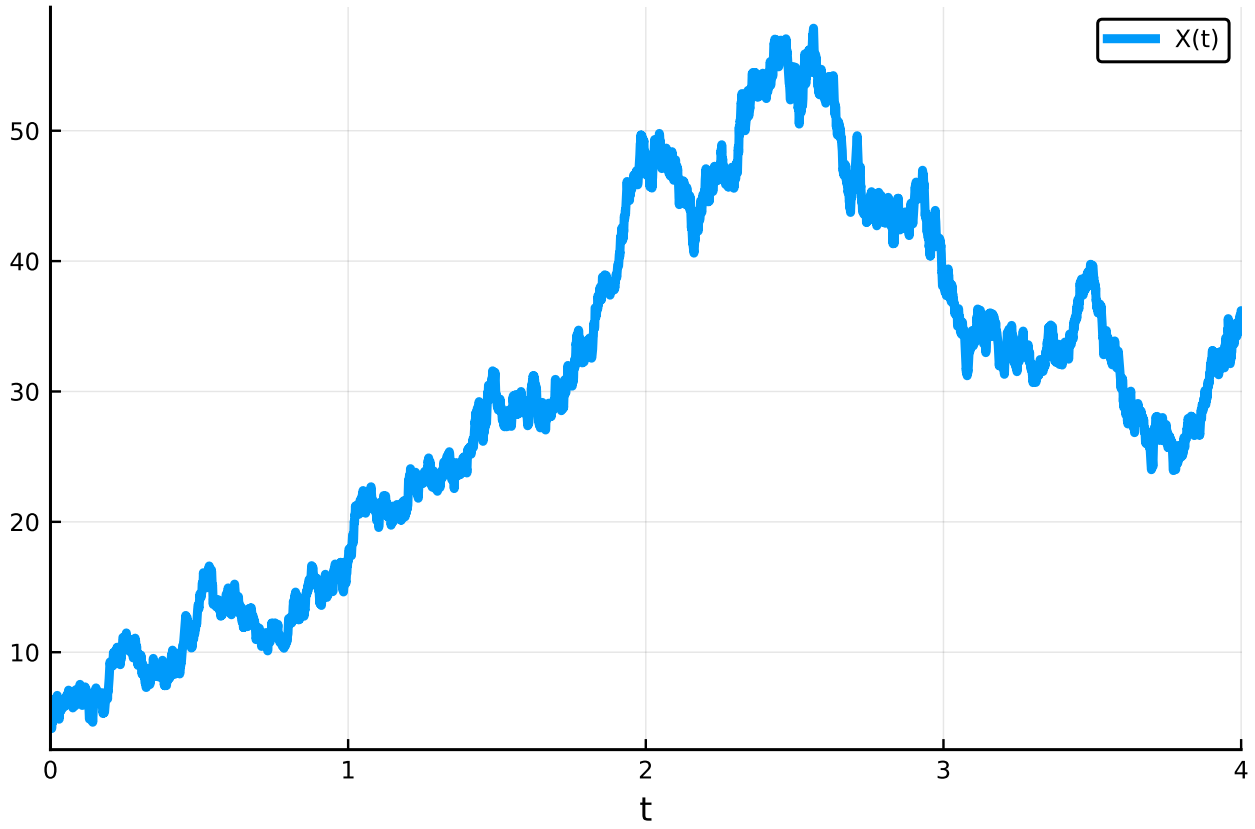
The corresponding Chemical Langevin Equation SDE is then

```
latexify(bdp, noise=true, cdot=false)
```

$$\mathrm{dX}\left(t\right) = \left(c_1 X - c_2 X + c_3\right) dt + \sqrt{\left\|c_1 X\right\|}\mathrm{dW}_1\left(t\right) - \sqrt{\left\|c_2 X\right\|}\mathrm{dW}_2\left(t\right) + \sqrt{\left\|c_3\right\|}\mathrm{dW}_3\left(t\right)$$

where each $W_i(t)$ denotes an independent Brownian Motion. We can solve the CLE SDE model by creating an `SDEProblem` and solving it similar to what we did for ODEs above:

```
# SDEProblem for CLE
sprob = SDEProblem(bdp, u_0, tspan, p)

# solve and plot, tstops is used to specify enough points
# that the plot looks well-resolved
sol = solve(sprob, tstops=range(0., step=4e-3, length=1001))
plot(sol, fmt=:svg)
```



We again have complete freedom to select any of the StochasticDifferentialEquations.jl SDE solvers, see the documentation.

0.5 What information can be queried from the reaction_network:

The generated `reaction_network` contains a lot of basic information. For example

- `f=oderhsfun(repressilator)` is a function `f(du,u,p,t)` that given the current state vector `u` and time `t` fills `du` with the time derivatives of `u` (i.e. the right hand side of the ODEs).
- `jac=jacfun(repressilator)` is a function `jac(J,u,p,t)` that evaluates and returns the Jacobian of the ODEs in `J`. A corresponding Jacobian matrix of expressions can be accessed using the `jacobianexprs` function:

```
latexify(jacobianexprs(repressilator), cdot=false)
```

$$\begin{bmatrix} -\delta & 0 & 0 & 0 & 0 & \frac{-K^n n\alpha P_3^{-1+n}}{\left(K^n+P_3^n\right)^2} \\ 0 & -\delta & 0 & \frac{-K^n n\alpha P_1^{-1+n}}{\left(K^n+P_1^n\right)^2} & 0 & 0 \\ 0 & 0 & -\delta & 0 & \frac{-K^n n\alpha P_2^{-1+n}}{\left(K^n+P_2^n\right)^2} & 0 \\ \beta & 0 & 0 & -\mu & 0 & 0 \\ 0 & \beta & 0 & 0 & -\mu & 0 \\ 0 & 0 & \beta & 0 & 0 & -\mu \end{bmatrix}$$

- `pjac = paramjacfun(repressilator)` is a function `pjac(pJ,u,p,t)` that evaluates and returns the Jacobian, `pJ`, of the ODEs *with respect to the parameters*. This allows `reaction_networks` to be used in the DifferentialEquations.jl local sensitivity analysis package DiffEqSensitivity.

By default, generated `ODEProblems` will be passed the corresponding Jacobian function, which will then be used within implicit ODE/SDE methods.

The DiffEqBiological API documentation provides a thorough description of the many query functions that are provided to access network properties and generated functions. In DiffEqBiological Tutorial II we'll explore the API.

0.6 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 DiffEqTutorials Github site. Or, submit a pull request to DiffEqTutorials updating the tutorial!

0.7 Appendix

This tutorial is part of the DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/DiffEq

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.weave_file("models","03-diffeqbio_I_introduction.jmd")
```

Computer Information:

```
Julia Version 1.2.0
Commit c6da87ff4b (2019-08-20 00:03 UTC)
Platform Info:
```

```
  OS: macOS (x86_64-apple-darwin18.6.0)
  CPU: Intel(R) Core(TM) i7-6920HQ CPU @ 2.90GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
```

Package Information:

```
Status `~/.julia/environments/v1.2/Project.toml`
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.4.3
[a93c6f00-e57d-5684-b7b6-d8193f3e46c0] DataFrames 0.19.4
[2b5f629d-d688-5b77-993f-72d75c75574e] DiffEqBase 6.3.4
[eb300fae-53e8-50a0-950c-e21f52c2b7e0] DiffEqBiological 4.0.1
[c894b116-72e5-5b58-be3c-e6d8d4ac2b12] DiffEqJump 6.2.2
[a077e3f3-b75c-5d7f-a0c6-6bc4c8ec64a9] DiffEqProblemLibrary 4.5.1
[6d1b261a-3be8-11e9-3f2f-0b112a9a8436] DiffEqTutorials 0.1.0
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.8.0
[7073ff75-c697-5162-941a-fcdaad2a7d2a] IJulia 1.20.0
[42fd0dbc-a981-5370-80f2-aaf504508153] IterativeSolvers 0.8.1
[23fbe1c1-3f47-55db-b15f-69d7ec21a316] Latexify 0.11.0
[54ca160b-1b9f-5127-a996-1867f4bc2a2c] ODEInterface 0.4.6
[47be7bcc-f1a6-5447-8b36-7eeeff7534fd] ORCA 0.3.0
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.17.2
[f0f68f2c-4968-5e81-91da-67840de0976a] PlotlyJS 0.13.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 0.27.0
[438e738f-606a-5dbb-bf0a-cddfbfd45ab0] PyCall 1.91.2
[d330b81b-6aea-500a-939a-2ce795aea3ee] PyPlot 2.8.2
[b4db0fb7-de2a-5028-82bf-5021f5cfa881] ReactionNetworkImporters 0.1.5
[295af30f-e4ad-537b-8983-00126c2a3abe] Revise 2.2.0
[789caeaf-c7a9-5a7d-9973-96adeb23e2a0] StochasticDiffEq 6.11.2
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 3.7.0
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.9.1
[b77e0a4c-d291-57a0-90e8-8db25a27a240] InteractiveUtils
[d6f4376e-aef5-505a-96c1-9c027394607a] Markdown
```