Biological Networks – Dynamic Aspects

Preconditions

Make sure that the following software is installed on your `Linux` computer.

1. `emacs` or any other text-editor.
2. **The two input files (`brusselator.jl`, `simpleFBA.jl`) used in this tutorial can be found online under the following URL**
   http://www.tbi.univie.ac.at/∼xtof/Leere/269020

The following directories and files are supposed to exist on your Linux computer.

```
$ test -d $HOME/CCBiol2/local/bin
$ test -d $HOME/CCBiol2/local/share
$ echo $PATH
/usr/local/bin:/usr/bin:$HOME/CCBiol2/local/bin
```

Prerequisites: Installing Software

`Julia`

Download the latest stable precompiled version (Generic Linux binaries 64-bit X86) of `julia`, a high-level, high-performance dynamic programming language for numerical computing, from the URL https://julialang.org/downloads/

1. Use `wget` to download julia-1.7.2-linux-x86_64.tar.gz
2. Unpack `julia`'s tar-archive and make a symbolic link pointing from the executible to `$HOME/CCBiol2/local/bin/julia`

```
$ cd $HOME/CCBiol2/local
$ tar zxvf julia-1.1.0-linux-x86_64.tar.gz
$ ln -s $HOME/CCBiol2/local/julia-1.7.2/bin/julia $HOME/CCBiol2/local/bin/julia
$ julia --version
julia version 1.7.2
```

3. Install the following julia addon packages (`DifferentialEquations`, `JuMP`, `GLPK`, `Plots`, and `Catalyst`) from the the command-line prompt of `julia`

```
$ julia
julia> import Pkg
julia> Pkg.update()
julia> Pkg.add("Plots"); import Plots
julia> Pkg.add("DifferentialEquations"); import DifferentialEquations
julia> Pkg.add("ParameterizedFunctions"); import ParameterizedFunctions
```

```
julia> Pkg.add("Catalyst"); import Catalyst
julia> Pkg.add("GLPK"); import GLPK
julia> Pkg.add("JuMP"); import JuMP

julia> Pkg.installed()
Dict{String,Union{Nothing, VersionNumber}} with 8 entries:
  "Catalyst"               => v"10.8.0"
  "GLPK"                   => v"1.0.1"
  "DifferentialEquations"  => v"7.1.0"
  "IJulia"                 => v"1.23.3"
  "DiffEqNoiseProcess"     => v"5.9.0"
  "Plots"                  => v"1.28.2"
  "ParameterizedFunctions" => v"5.13.1"
  "JuMP"                   => v"1.0.0"

julia> exit()
```

1 Studying a System's Dynamics using `Julia`

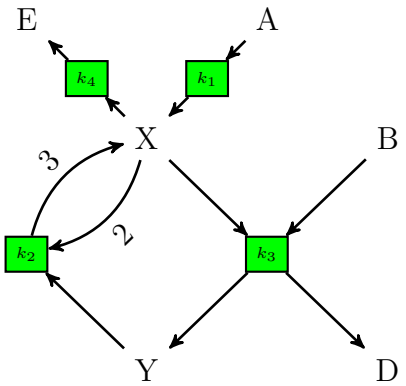
1. Make a working directory

```
$ mkdir Exercise
$ cd Exercise
```

From ODEs to the system dynamics

2. **Brusselator a step-by-step example:**

   The *brusselator* is a famous theoretical model for a particular type of an autocatalytic reaction proposed by *Ilya Prigogine*. It possesses a fixed point at $(A, B/A)$ which becomes unstable when $B > 1 + A^2$ resulting in oscillatory behaviour. The chemical reaction equations are as follows:



$$
\begin{aligned}
\mathrm{A} &\xrightarrow{k_1} \mathrm{X} \\
2\cdot\mathrm{X} + \mathrm{Y} &\xrightarrow{k_2} 3\cdot\mathrm{X} \\
\mathrm{X} + \mathrm{B} &\xrightarrow{k_3} \mathrm{Y} + \mathrm{D} \\
\mathrm{X} &\xrightarrow{k_4} \mathrm{E}
\end{aligned}
$$

$$
\begin{aligned}
\tfrac{\mathrm{d}}{\mathrm{d}t}X &= +k_1\cdot A + k_2\cdot Y\cdot X^2 \\
&\quad -k_3\cdot B\cdot X - k_4\cdot X \\
\tfrac{\mathrm{d}}{\mathrm{d}t}Y &= +k_3\cdot B\cdot X \\
&\quad -k_2\cdot X^2\cdot Y
\end{aligned}
$$

The corresponding `julia` input file (`brusselator.jl`) looks as follows:

```
$ wget -v http://www.tbi.univie.ac.at/~xtof/Leere/269020/brusselator.jl
$ cat brusselator.jl
```

```
# reaction equations of bruesselator
#
#       A -> X     : k1
# 2 X + Y -> 3 X   : k2
#   X + B -> Y + D : k3
#       X -> E     : k4
#

# loading some usefull julia addon packages
using DifferentialEquations
using ParameterizedFunctions
using Plots

# define ODEs
f = @ode_def Bruesselator begin
  dX = k1*A + k2*X^2*Y - k3*B*X - k4*X
  dY = k3*B*X - k2*X^2*Y
end k1 k2 k3 k4 A B

# initial value array [X, Y]
u0 = [1.0, 1.0]

# parameter array [k1, k2, k3, k4, A, B]
p = [1.0, 1.0, 1.0, 1.0, 1.0, 3.0]

# simulation time (start, stop)
tspan = (0.0, 50.0)

# setup ODE problem
odes = ODEProblem(f, u0, tspan, p)

# solve ODES problem
sol = solve(odes)

# plot time course
plot(sol)
```

The file `brusselator.jl` possesses several sections. It starts with an optional comment block indicating the name and reaction equations of the model (lines 1-7). Then useful `julia` addon packages are loaded (lines 9-12). Next a *function with the ODE equations* is defined (lines 14-18) followed by two arrays assigning *initial values and parameters* (lines 21 & 24). Simulation *start and stop times* are specified in line 27. The information is integrated into an `ODEProblem` (line 30), which is solved in line 33. Finally the time course is plotted (line 36).

3. In the following we assume that the file you generated / downloaded is called `brusselator.jl`.

4. `Julia`'s interactive command-line REPL (read-eval-print loop) will be used, making it easy to change a model's structure and parameters on the fly. Start `julia`, load and execute `brusselator.jl`. After some time a window, showing the time course (compair with Figure 1), will pop-up and the command prompt will become active again.

```
$ julia
julia> include("bruesselator.jl")
```

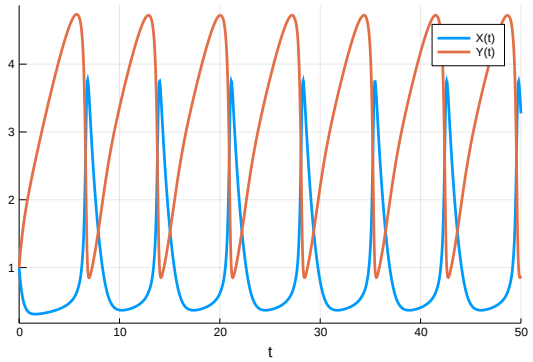


Figure 1: **Time course of the brusselator model.** The following value settings were used: $X(0) = Y(0) = 1$; $k_1 = k_2 = k_3 = k_4 = 1$; $A = 1$ and $B = 3$. Species X and Y oscillate, but their phases are shifted against each other.

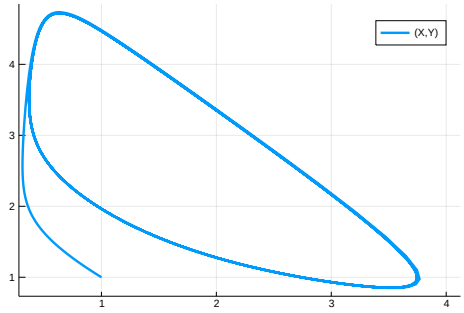


Figure 2: **Phase plane of the brusselator model.** The same value settings as in Figure 1 were used. The trajectory starts at phase plane point $(1, 1)$ and is attracted towards the limit cycle.

5. Store the plot as PDF-file (e.g. `brusselator-tc.pdf`) by executing the `savefig()` command (note this command closes the pop-up window).

```
julia> savefig("brusselator-tc.pdf")
```

6. Since the *brusselator model* is a two variable system, the phase plan X *versus* Y can be visualized. Plot the phase plain as follows

```
julia> plot(sol, vars=(1,2))
```

7. Change the initial values of X and Y to $X = 1.5, Y = 3$ or $X = 3, Y = 4$ and compare the time-course and phaseplane plots to the original value settings (What do you notice?).

```
julia> u0 = [3.0, 4.0]
julia> odes = ODEProblem(f, u0, tspan, p)
julia> sol = solve(odes)
julia> plot(sol)
julia> plot(sol, vars=(1,2))
```

8. Change the parameter B to $B = 1.7$ and look at the time-courses and phaseplanes for the same initial conditions of X and Y as in item 7. (What do you notice?)

9. **The Lorenz system:**

   The *Lorenz equations* (see below) are a simplified mathematical model for atmospheric convection developed by *Edward Lorenz*, that show chaotic solutions for certain parameter values and initial conditions. Use `emacs` to formulate a `julia` model of the Lorenz equations (use filename `lorenz.jl`).

$$\frac{dx}{dt} = \sigma \cdot (y - x)$$
$$\frac{dy}{dt} = x \cdot (\rho - z) - y$$
$$\frac{dz}{dt} = x \cdot y - \beta \cdot z$$

   Use $\sigma = 10.0$, $\rho = 28.0$, $\beta = \frac{8}{3}$ as parameters and $x = 1.0$, $y = z = 0.0$ for the inital conditions.

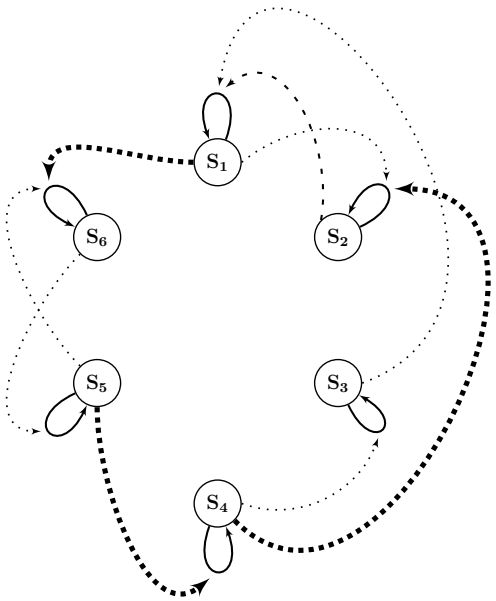
   (a) Integrate your formulation of the Lorenz model between *start time* 0.0 and *stop time* 100.0. Plot the resulting trajectory in the 3-dimensional state space (resempling a butterly or figure eight).

   (b) To convince yourself of the chaotic nature of the Lorenz equations plot the variables time `versus` y. Do x and z show a similar behavior?

From Reactions to the system dynamics

10. **Hypercycle a step-by-step example:**

    The hypercycle an abstract model of organization of self-replicating molecules was introduced by *Manfred Eigen* and *Peter Schuster* to solve the error threshold problem encountered during the modelling of replicative molecules that hypothetically existed on the primordial Earth. The n replicators are usually connected in a cycle and a replicator "helps" its succesor in the cycle to replicate. For $n \leq 4$ the hypercycles possess one stable fix point where



$$
\begin{aligned}
0 &\xrightarrow{k_f} F \\
F + S_1 + S_2 &\xrightarrow{k_2} 2 * S_1 + S_2 \\
F + S_2 + S_1 &\xrightarrow{k_1} 2 * S_2 + S_1 \\
F + S_2 + S_4 &\xrightarrow{k_3} 2 * S_2 + S_4 \\
F + S_6 + S_1 &\xrightarrow{k_3} 2 * S_6 + S_1 \\
F + S_1 + S_3 &\xrightarrow{k_1} 2 * S_1 + S_3 \\
F + S_3 + S_4 &\xrightarrow{k_1} 2 * S_3 + S_4 \\
F + S_4 + S_5 &\xrightarrow{k_3} 2 * S_4 + S_5 \\
F + S_5 + S_6 &\xrightarrow{k_1} 2 * S_5 + S_6 \\
F + S_6 + S_5 &\xrightarrow{k_1} 2 * S_6 + S_5 \\
S_i &\xrightarrow{k_d} 0 \quad i = 1, 2, \ldots, 6
\end{aligned}
$$

Figure 3: **6-species hypercycle.** The hypercycle is persistent although it does not contain a *hamilton cycle*. (lhs) Wireing diagram of the catalytic interactions (dashed arrows) in the hypercycle. Each species $S_1 - S_6$ is a replicator (solid self loop), that consumes food F (not shown), useing some other species as "catalyst" (indicated by the dashed arrows) to make a copy of itself (rhs) Shows the 17 chemical equations that describe the system. Note there are 3 catalytic coupling constants k_1 weak $\cdots$, k_2 medium - - -, and k_3 strong ▪▪▪, one feeding rate k_f and one degradation rate k_d.

all the species are present. For $n \geq 5$ the solution is a stable limit cycle and the n species oscillate. Figure 3 shows the smallest hypercycle with a complex topology, that is not a simple hamilton cycle. The corresponding `julia` input file (`6hypercyc.jl`) looks as follows:

```
$ wget -v http://www.tbi.univie.ac.at/~xtof/Leere/269020/6hypercyc.jl
$ cat 6hypercyc.jl
```

```
# 6 species hypercycle with non hamiltonian cycle structure
#
# F + S1 + S2 --> 2 * S1 + S2 : k2
# F + S2 + S1 --> 2 * S2 + S1 : k1
# F + S2 + S4 --> 2 * S2 + S4 : k3
# F + S6 + S1 --> 2 * S6 + S1 : k3
```

```
# F + S1 + S3 --> 2 * S1 + S3 : k1
# F + S3 + S4 --> 2 * S3 + S4 : k1
# F + S4 + S5 --> 2 * S4 + S5 : k3
# F + S5 + S6 --> 2 * S5 + S6 : k1
# F + S6 + S5 --> 2 * S6 + S5 : k1
#          S1 --> 0           : kd
#          S2 --> 0           : kd
#          S3 --> 0           : kd
#          S4 --> 0           : kd
#          S5 --> 0           : kd
#          S6 --> 0           : kd
#           F --> 0           : kd
#           0 --> F           : kf
#

# loading some usefull julia addon packages
using DifferentialEquations
using Catalyst
using Plots

# define reaction network
rn = @reaction_network SixHyperCyc begin
     k2, F + S1 + S2 --> 2 * S1 + S2
     k1, F + S2 + S1 --> 2 * S2 + S1
     k3, F + S2 + S4 --> 2 * S2 + S4
     k3, F + S6 + S1 --> 2 * S6 + S1
     k1, F + S1 + S3 --> 2 * S1 + S3
     k1, F + S3 + S4 --> 2 * S3 + S4
     k3, F + S4 + S5 --> 2 * S4 + S5
     k1, F + S5 + S6 --> 2 * S5 + S6
     k1, F + S6 + S5 --> 2 * S6 + S5
     kd, (F, S1, S2, S3, S4, S5, S6) --> 0
     kf,  0 --> F
end k1 k2 k3 kd kf

# initial value array [S1, S2, S3, S4, S5, S6, F]
u0 = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

# parameter array [k1, k2, k3, kd, kf]
p = [1.0, 2.0, 3.0, 1.0, 3.8]

# simulation time (start, stop)
tspan = (0.0, 200.0)

# setup ODE problem
odes = ODEProblem(rn, u0, tspan, p)

# solve ODES problem
sol = solve(odes)

# plot time course
```

```
58 plot(sol)
```

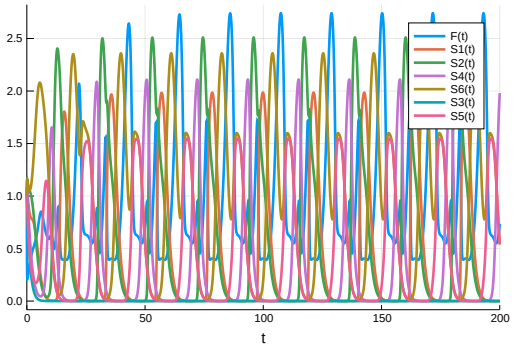


Figure 4: **Hypercycle for 6 species.** For $k_f \geq 3.8$ stable limit cycle oscillation are observed. Imagine waves going arround the system, that constructively interfere.

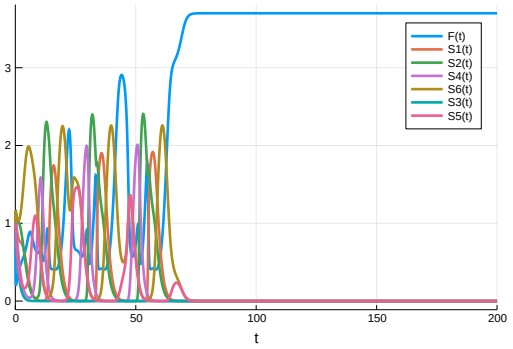


Figure 5: **Hypercycle for 6 species.** For $k_f < 3.8$ the dilution rate of the system is to high and the 6 replicators are washed out of the system.

The basic structure of file `6hypercyc.jl` is the same as in the ODE case (compare with `brusselator.jl`). The only major difference is the specification of the chemical reactions in lines 27-40 instead of the ODEs.

11. Load and execute `6hypercyc.jl` in `julia` (You should observe stable limit cycle oscillations see Figure 4).

```
1 julia> include("6hypercyc.jl")
```

12. Change the feeding rate k_f to $k_f = 3.7$ (Do you have an idea why the oscillation stops and the replicators die out see Figure 5 ?).

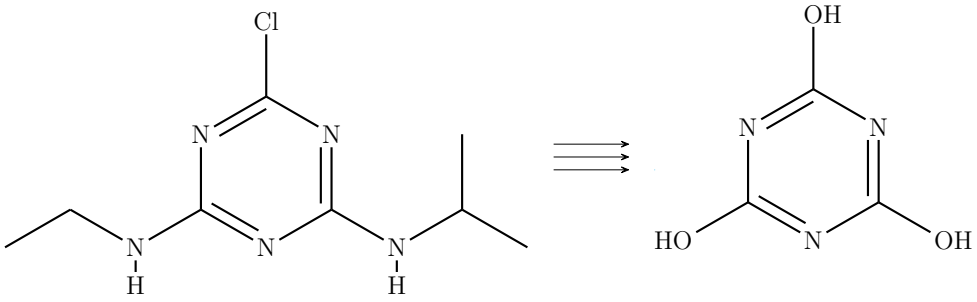
13. **Degradation Dynamics of Herbicides in Soil:**

    The degradation of s-triazine herbicides, such as `atrazine`, under anaerobic conditions in soil can be described by only two major reaction types[1]

    (a) Hydrolysis ($\mathrm{R-X} \xrightarrow{\mathrm{H_2O}} \mathrm{R-OH} + \mathrm{HX}$ where $\mathrm{X = Cl, NH_2}$)

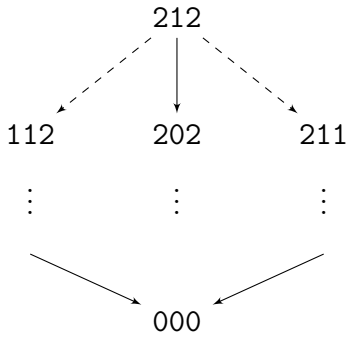
    (b) Reductive dealkylation ($\mathrm{R-NH-R'} \xrightarrow{\mathrm{H_2}} \mathrm{R-NH_2} + \mathrm{HR'}$ where $\mathrm{R'}$ is a $\mathrm{C_1}$-$\mathrm{C_4}$ alkyl-moiety such as ethyl, iso-propyl or tert-butyl).

[1] Erickson LE, Lee KH & Sumner DD (1989), Degradation of atrazine and related s–triazines, *Crit Rev Env Control* **19**:1-14 | doi:10.1080/10643388909388356



atrazine cyanuric acid

14. Draw the reaction network induced by the hydrolysis and reductive dealkylaton reactions transforming `atrazine` to `cyanuric acid`. (Hint: encode the hydrolysation state of the `triazine` side chains as 3 digit string (`atrazine` ≡ `212`, `cyanuric` acid ≡ `000`); the digit indicates how many hydrolysis steps the respective side chain has to run through to become an OH group (**Keep in mind, that different numerical encodings can be isomorphic on the level of the molecular graph and must therefore be represented by only one point in the lattice!**). The resulting reaction network is a 6-layer lattice with 1, 3, 5, 4, 2, 1 molecule(s) per layer



15. Use your reaction network drawing to setup a reaction network file named `atrazine.jl` for the degradation of `atrazine` to `cyanuric acid`. Name the molecules consecutively from top to bottom and left to right with single characters (A–P). Keep in mind that the three types of hydrolysis reactions use different reaction constants.

16. Simulate the degradation dynamics using the following rate constants:

$$\begin{aligned} k_{\text{hydrolysis}} &= 5.00 \times 10^{-9}\ s^{-1} \\ k_{\text{ethyl}} &= 3.32 \times 10^{-8}\ s^{-1} \\ k_{\text{i-propyl}} &= 2.65 \times 10^{-8}\ s^{-1} \\ k_{\text{t-butyl}} &= 2.21 \times 10^{-8}\ s^{-1} \end{aligned}$$

17. Analyse the kinetics of the individual reaction steps of the network. Which chemical species get populated at which time and to what extent?

18. Since the model is formulated in reaction equations, the simulation model can easily be switched from deterministic ODE to stochasic Gillespie based. In order to try that out replace the "setup ODE problem" and "solve ODES problem" lines in your `atrazine.jl` file with the following code snippet, and use `julia` so perform a stochastic simulation of the `atrazine` model.

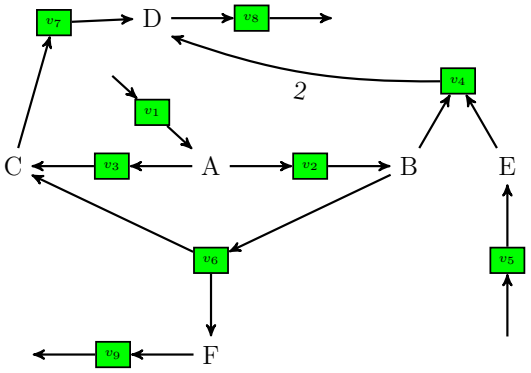
```
# setup discrete problem
disc = DiscreteProblem(rn, u0, tspan, p)

# set up Gillespie's direct method
jump = JumpProblem(disc, Direct(), rn)

# solve using the stochastic simulation algrithm (SSA)
sol = solve(jump, SSAStepper())
```

2 Flux Balance Analysis (FBA)

The basic idea of FBA is straightforward: at steady state all the mass that goes into the network must either come out again or is converted into biomass.

1. **Step-by-Step Example:**

   Formulate for the following reaction network the corresponding *linear programming (LP) problem* using the objective function $z = 0.75 \cdot v_8 + 0.5 \cdot v_9$.




```
   -> A      ; v1 = 10
   -> E      ; v5 = 6
A -> C       ; v3 >= 1
A -> B       ; v2
E + B -> 2 * D ; v4
B -> C + F   ; v6
C -> D       ; v7
D ->         ; v8
F ->         ; v9
```

   The FBA model consists of an **objective function** z, which shall be maximized, three **flux constraints** for v_1, v_3 and v_5 and six **flux balance equations** for each of the chemical species A – F.

$$\max z = 0.75 \cdot v_8 + 0.5 \cdot v_9$$

subject to:

$$\begin{aligned} v_1 &= 10 \\ v_3 &\geq 1 \\ v_5 &= 6 \end{aligned}$$

$$\begin{aligned} v_1 - v_2 - v_3 \\ v_2 - v_4 - v_6 \\ v_3 + v_6 - v_7 \\ v_7 + 2 \cdot v_4 - v_8 \\ v_5 - v_4 \\ v_6 - v_9 \end{aligned}$$

The corresponding `julia` input file (`simpleFBA.jl`) looks as follows:

```
$ wget -v http://www.tbi.univie.ac.at/~xtof/Leere/269020/simpleFBA.jl
$ cat simpleFBA.jl
```

```
# simple FBA example from Book ISBN 13: 978-0-98247739-7
# Sauro, HM 1st ed, (2014-2015)
# "Systems Biology: Linear Algebra for Pathway Modeling"
# 9.3 Example, pp 164-166
#
#       -> A     ; v1 = 10
#       -> E     ; v5 = 6
#     A -> C     ; v3 >= 1
#     A -> B     ; v2
# E + B -> 2 * D ; v4
#     B -> C + F ; v6
#     C -> D     ; v7
#     D ->       ; v8
#     F ->       ; v9
#
# v1, v5 measured input fluxes
# v8, v9 contribute to biomass
# objective function: z = 0.5 * v9 + 0.75 * v8
#
# For latest documentation on JuMP - Julia for Mathematical Optimization
# see https://jump.readthedocs.io/en/latest/
#
# syntax works for
#        julia v1.7.2
#         JuMP v1.0.0
#         GLPK v1.0.1

using JuMP
using GLPK
```

```
30
31 m = Model(GLPK.Optimizer)
32
33 # define variables v1 to v9
34 @variable(m, v[1:9])
35
36 # objective function
37 @objective(m, Max, 0.5 * v[9] + 0.75 * v[8])
38
39 ## flux constraints
40 @constraint(m, v[1] == 10)
41 @constraint(m, v[5] == 6)
42 @constraint(m, v[3] >= 1)
43
44 ## flux balances
45 @constraint(m, v[1] - v[2] - v[3] == 0)     # A
46 @constraint(m, v[2] - v[4] - v[6] == 0)     # B
47 @constraint(m, v[3] + v[6] - v[7] == 0)     # C
48 @constraint(m, 2 * v[4] - v[8] + v[7] == 0) # D
49 @constraint(m, v[5] - v[4] == 0)            # E
50 @constraint(m, v[6] - v[9] == 0)            # F
51
52 # output model
53 JuMP.print(m)
54
55 # do calculation
56 JuMP.optimize!(m)
57
58 # output results
59 println("Objective value: ", JuMP.objective_value(m))
60 for i in 1:9
61     println("v$i = ", JuMP.value(v[i]))
62 end
```

2. Solve the FBA problem using `julia`.

```
$ julia simpleFBA.jl
Max 0.5 v[9] + 0.75 v[8]
Subject to
 v[1] = 10
 v[5] = 6
 v[3] ≥ 1
 v[1] - v[2] - v[3] = 0
 v[2] - v[4] - v[6] = 0
 v[3] + v[6] - v[7] = 0
 v[6] - v[9] = 0
 v[5] - v[4] = 0
 2 v[4] - v[8] + v[7] = 0
 v[i] free ∀ i ∈ 1,2,...,8,9
Tried aggregator 1 time.
```

```
LP Presolve eliminated 9 rows and 9 columns.
All rows and columns eliminated.
Presolve time = 0.00 sec. (0.00 ticks)
Objective value: 13.5
v1 = 10.0
v2 = 9.0
v3 = 1.0
v4 = 6.0
v5 = 6.0
v6 = 3.0
v7 = 4.0
v8 = 16.0
v9 = 3.0
```

3. What are the values of the internal fluxes ($v_2 - v_4, v_6, v_7$) and the value of the objective function z? Convince yourself, that the calculated flux values are consistent with each other and the reaction network structure!

4. Change the linear combination coefficients in the objective function. How do the values of z and the internal fluxes v_i change?

5. **Rate of Synthesis of a Polymer with Sequence `DCDE`:**

   Figure 6 depicts a small reaction network, which produces three buiding

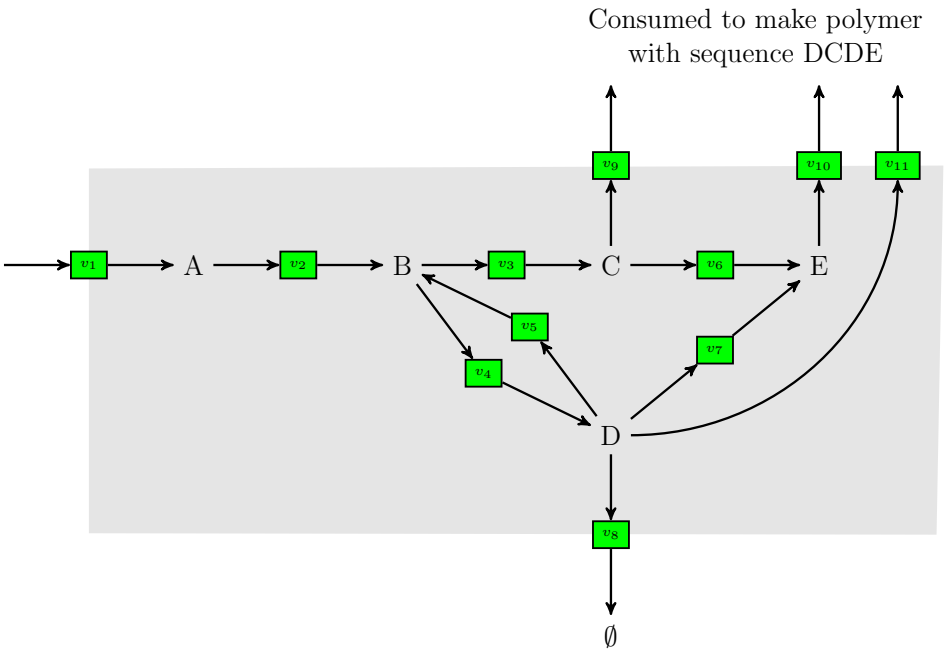


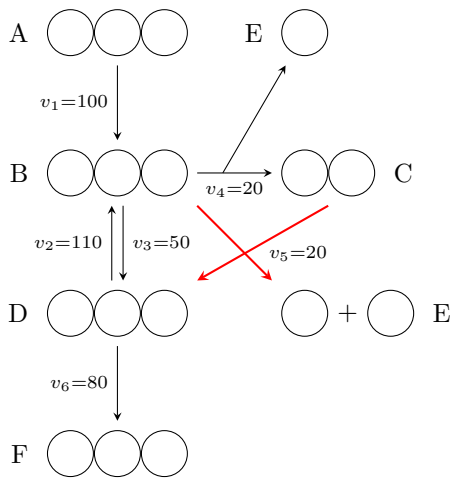
Figure 6: **Reaction Network for Tetramere Synthesis:** From the resource `A` three building blocks `C`, `D` and `E` are produced. The building blocks are exported to the outside via reactions $v_9 - v_{11}$, where they are asseambled into a tetramer with the sequence `DCDE`.

blocks `C`, `D` and `E`. Four of the building blocks ($1 \times$ `C`, $2 \times$ `D` and $1 \times$ `E`) assemble into a polymer with the sequence `DCDE`. What is the maximum possible synthesis rate for the tetramer `DCDE`?

To answer this question setup and solve the FBA problem. Maximize the fluxes $v_9 - v_{11}$ to the polymer synthesis. Constrain the influx v_1 to 100 and the degradation flux v_8 to 10 (Hint: the outflux v_{11} from the `D` monomere to the polymer synthesis must be 2 times larger than outflux v_9 or v_{10}, since the tetramer contains two units of `D`).

3 Simulation of Isotope Labeling Experiments

1. Translate the following reaction network[2]




```
R1:      A -> B
R2:      B -> D
R3:      D -> B
R4:      B -> C + E
R5: C + B -> D + E + E
R6:      D -> F

R1:        #abc -> #abc
R2:        #abc -> #abc
R3:        #abc -> #abc
R4:        #abc -> #bc  + #a
R5: #AB + #abc -> #bcA + #a + #B
R6:        #abc -> #abc
```

   into ODEs and simulate the time course of the system. Scale the reaction velocities relative to v_1 and use $A = 100$ as initial condition.

2. Try to construct the permutation tables for reaction `R4` and `R5` from the string representation of the carbon atom maps (lower schema on the right).

3. Draw the atom transfer network. Can fully labeld B be produced from A labeled only on the 1st position?

4. Expand the ODE system to the full isotopomere system. How many equations do we get (label the ODE variables with a binary number e.g. A_{101} for A labeld in the 1st and 3rd position)?

[2] from Antoniewicz MR et al (2006), Determination of confidence intervals of metabolic fluxes estimated from stable isotope measurements, *Metab Eng* **8**:324-337 | doi:10.1016/j.ymben.2006.01.004

5. Simulate the full isotopomere system with different initial labelings of A (e.g. $A_{100} = 100$, $A_{010} = 100$ and $A_{001} = 100$). What is the difference? (For input $A100$ the species $A_{010}, B_{010}, D_{010}, E_0, F_{010}$ and F_{101} possess population densities above 5% plot this subset of species with the folloing command)

```
julia> plot(sol, vars=[(0,5),(0,6),(0,19),(0,26),(0,33),(0,36)])
```

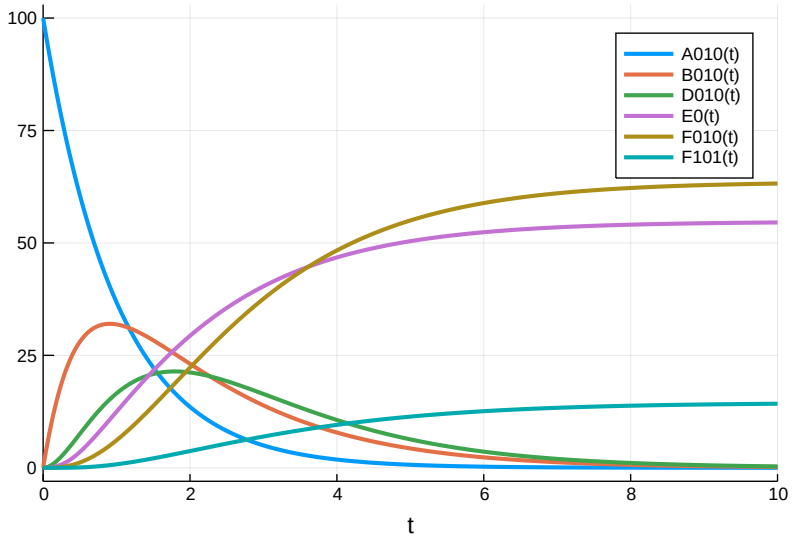


Figure 7: **Isotopomere simulation.** The time course when starting in isotopomere $A_{010} = 100$ is shown. Note that that no labeled E is generated.

Appendix

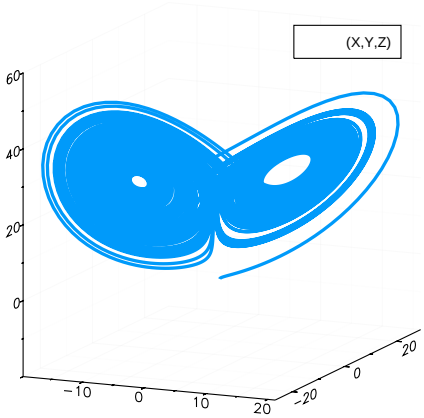


Figure 8: **State space of the Lorenz system.**

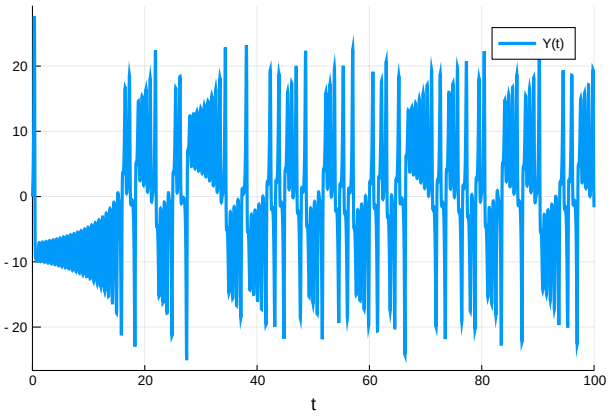


Figure 9: **Chaotic behaviour of variable Y in the Lorenz system.**

```
# Lorenz system
#
# dX = s * (Y - X)      : s = 10.0
# dY = X * (r - Z) - Y  : r = 28.0
# dZ = X * Y - b * Z    : b = 8 / 3
#
# X(0) = 1.0
# Y(0) = Z(0) = 0.0
#

# loading some usefull julia addon packages
using DifferentialEquations
using ParameterizedFunctions
using Plots

# define ODEs
f = @ode_def Lorenz begin
    dX = s*(Y - X)
    dY = X*(r - Z) - Y
    dZ = X*Y - b*Z
end s r b

# initial value array [X, Y, Z]
u0 = [1.0, 0.0, 0.0]

# parameter array [s, r, b]
p = [10.0, 28.0, 8/3]

# simulation time (start, stop)
tspan = (0.0, 100.0)
```

```
31
32  # setup ODE problem
33  odes = ODEProblem(f, u0, tspan, p)
34
35  # solve ODES problem
36  sol = solve(odes)
37
38  # plot state space
39  plot(sol, vars=(1,2,3))
```

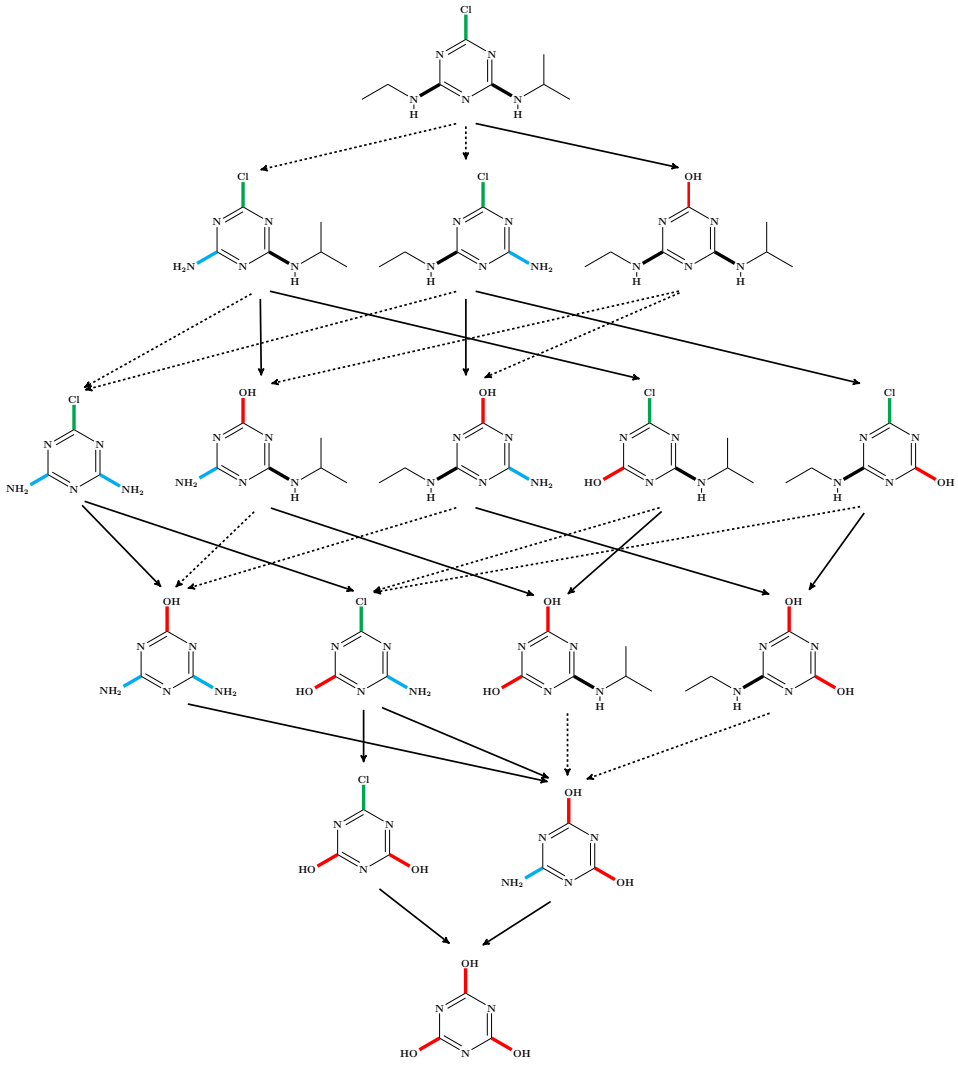


Figure 10: **Degradation network of atrazine in soil.** The chemical species were named by single characters (A-P) consecutively from top to down and left to right to set up the ordinary differential equation system. v_1 is the rate for the hydrolysis reaction (blunt arrows), and $v2$ and v_3 are the rates for the two different reductive dealcylation reactions (dashed arrows).

```
# atrazine degradation model
#
# A -> B : k2   # B -> E : k3   # C -> E : k2
# A -> C : k3   # B -> F : k1   # C -> G : k1
# A -> D : k1   # B -> H : k1   # C -> I : k1
#
# D -> F : k2   # E -> J : k1   # F -> J : k3
# D -> G : k3   # E -> K : k1   # F -> L : k1
#
# G -> J : k2   # H -> K : k3   # I -> K : k2
# G -> M : k1   # H -> L : k1   # I -> M : k1
#
# J -> O : k1   # K -> N : k1   # L -> O : k3
#               # K -> O : k1
#
# M -> O : k2   # N -> P : k1   # O -> P : k1
#

# loading some usefull julia addon packages
using DifferentialEquations
using Catalyst
using Plots

# define reaction network
rn = @reaction_network Atrazine begin
     (k2, k3, k1), A --> (B, C, D)
     (k3, k1, k1), B --> (E, F, H)
     (k2, k1, k1), C --> (E, G, I)
         (k2, k3), D --> (F, G)
         (k1, k1), E --> (J, K)
         (k3, k1), F --> (J, L)
         (k2, k1), G --> (J, M)
         (k2, k1), H --> (K, L)
         (k2, k1), I --> (K, M)
               k1, J --> O
         (k1, k1), K --> (N, O)
               k3, L --> O
               k2, M --> O
               k1, N --> P
               k1, O --> P
end k1 k2 k3

# initial value array [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P]
u0 = [100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ]

# parameter array [s, r, b]
p = [0.0005, 0.00332, 0.00265]

# simulation time (start, stop)
```

```
tspan = (0.1, 1000000.0)

# setup ODE problem
odes = ODEProblem(rn, u0, tspan, p)

# solve ODES problem
sol = solve(odes)

# plot time course
plot(sol, xscale = :log10)
```

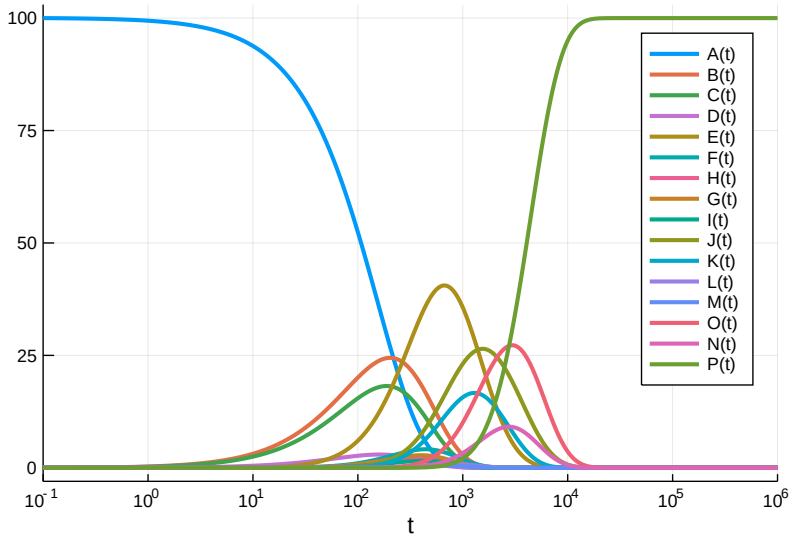


Figure 11: **ODE-based degradation dynamics of atrazine in soil.**

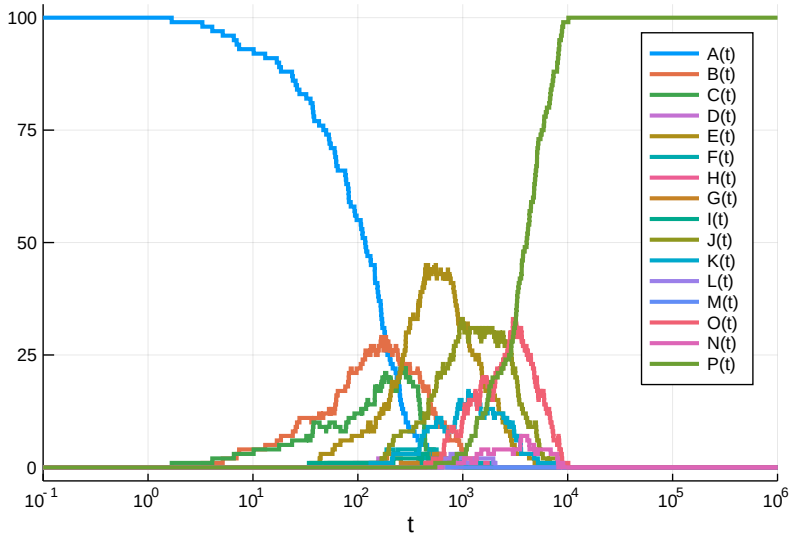


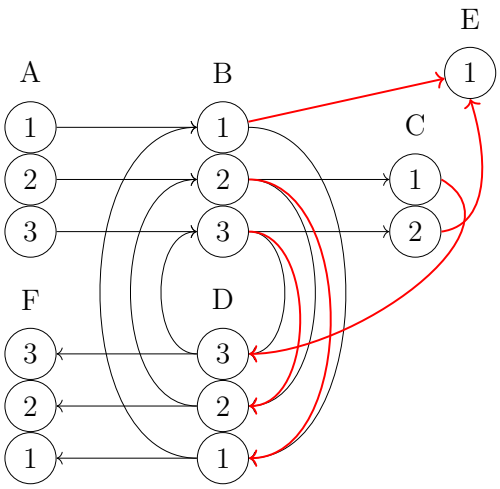
Figure 12: **Gillespie-based degradation dynamics of atrazine in soil.**

```
# polymer.jl
# inspired from
# Joshua D Rabinowitz & Livia Vastag
# Nature Chemical Biology 8, 497{501 (2012)
# doi:10.1038/nchembio.969
#
#   -> A     ; v1 = 100
# D ->       ; v8 = 10
# A -> B     ; v2
# B -> C     ; v3
# B -> D     ; v4
# D -> B     ; v5
# C -> E     ; v6
# D -> E     ; v7
# C -> X     ; v9
# E -> X     ; v10
# D -> X     ; v11
#
# A .. E are monomeres
# X is a polymer with the sequence DCDE
#
# formulate the flux balance equations such that the monomeres
# are produced in the correct proportions to form the polymere
#
# syntax works for
#        julia v1.7.2
#         JuMP v1.0.0
#         GLPK v1.0.1

using JuMP
using GLPK

m = Model(GLPK.Optimizer)

# define variables v1 to v11
@variable(m, v[1:11])

# objective function
@objective(m, Max, v[9] + v[10] + v[11])

## flux constraints
@constraint(m, v[1] == 100)
@constraint(m, v[8] == 10)
# constrain flux C to equal flux E
@constraint(m, v[9] == v[10])
# constrain flux D to be at least 2 times larger then flux C or E
@constraint(m, v[11] >= v[9] + v[10])
# disallow negative flux values
@constraint(m, v[1:11] .>= 0)

```

```
## flux balances
@constraint(m, v[1] - v[2] == 0)                            # A
@constraint(m, v[2] - v[3] - v[4] + v[5] == 0)              # B
@constraint(m, v[3] - v[6] - v[9] == 0)                     # C
@constraint(m, v[4] - v[5] - v[7] - v[8] - 2 * v[11] == 0) # D
@constraint(m, v[6] + v[7] - v[10] == 0)                    # E

# output model
JuMP.print(m)

# do calculation
JuMP.optimize!(m)

# output results
println("Objective value: ", JuMP.objective_value(m))
for i in 1:11
    println("v$i  = ", JuMP.value(v[i]))
end
```

```
$ julia polymer.jl

Max v[9] + v[10] + v[11]
Subject to
 v[1] = 100
 v[8] = 10
 v[9] - v[10] = 0
 v[11] - v[9] - v[10] ≥ 0
 v[1] ≥ 0
 v[2] ≥ 0
 v[3] ≥ 0
 v[4] ≥ 0
 v[5] ≥ 0
 v[6] ≥ 0
 v[7] ≥ 0
 v[8] ≥ 0
 v[9] ≥ 0
 v[10] ≥ 0
 v[11] ≥ 0
 v[1] - v[2] = 0
 v[2] - v[3] - v[4] + v[5] = 0
 v[3] - v[6] - v[9] = 0
 v[4] - v[5] - v[7] - v[8] - 2 v[11] = 0
 v[6] + v[7] - v[10] = 0
 v[i] ∀ i ∈ 1,2,...,10,11
Objective value: 60.0
v1  = 100.0
v2  = 100.0
v3  = 30.0
v4  = 70.0
v5  = 0.0
v6  = 15.0
```

```
v7  = 0.0
v8  = 10.0
v9  = 15.0
v10  = 15.0
v11  = 30.0
```



Permutation table for reaktions 4 and 5

R4: [3, 1, 2]

R5: [4, 1, 2, 3, 5]

Figure 13: **Atom transfer network.** Reacton 5 (highlighted in red) is responsible for mixing of initial label from A. Note fully labeled B can only be produce if A is labeled in positions 2 or 3

```
# example network fig 1 of doi:10.1016/j.ymben.2006.01.004
# antoniewicz ME 2006
#
#     A -> B          ; v1 #abc -> #abc
#     B -> D          ; v2 #abc -> #abc
#     D -> B          ; v3 #abc -> #abc
#     B -> C + E      ; v4 #abc -> #bc + #a
# B + C -> D + E + E ; v5 #abc + #AB -> #bcA + #a + #B
#     D -> F          ; v6 #abc -> #abc
#

# loading some usefull julia addon packages
using DifferentialEquations
using Catalyst
using Plots

# define reaction network
rn = @reaction_network Antoniewicz begin
    v1, A --> B
    v2, B --> D
    v3, D --> B
    v4, B --> C + E
```

```
    v5, A + C --> D + 2 * E
    v6, D --> F
end v1 v2 v3 v4 v5 v6

# initial value array [A, B, C, D, E, F]
u0 = [100.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# parameter array [v1, v2, v3, v4, v5, v6]
p = [1.0, 1.1, 0.5, 0.2, 0.2, 0.8]

# simulation time (start, stop)
tspan = (0.0, 10.0)

# setup ODE problem
odes = ODEProblem(rn, u0, tspan, p)

# solve ODES problem
sol = solve(odes)

# plot time course
plot(sol)
```

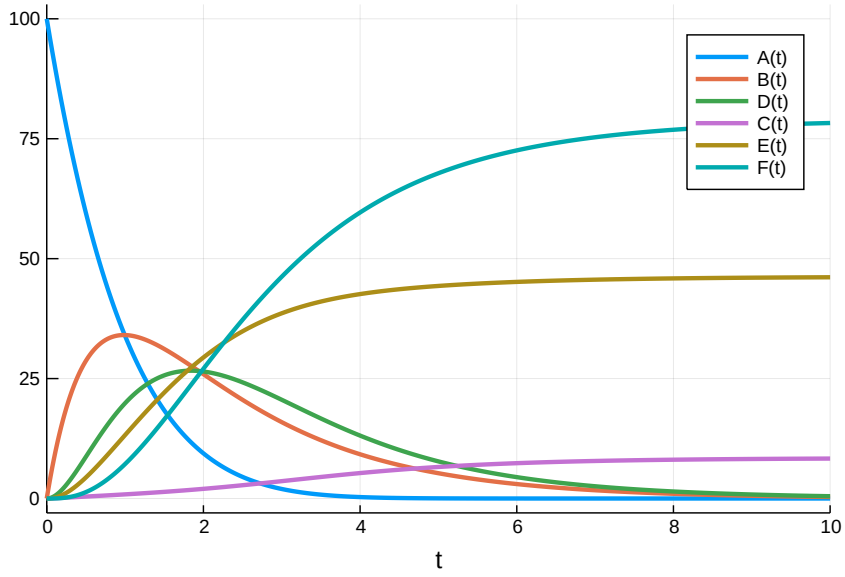


Figure 14: **Time course of the Antoniewicz model**

```
# antoniewicz-c13.jl
# example network fig 1 of doi:10.1016/j.ymben.2006.01.004
# antoniewicz ME 2006
# full isotopeomere model

#
#       A000 -> B000            ; #abc -> #abc
#       A001 -> B001            ; #abc -> #abc
#       A010 -> B010            ; #abc -> #abc
```

```
#       A011 -> B011            ; #abc -> #abc
#       A100 -> B100            ; #abc -> #abc
#       A101 -> B101            ; #abc -> #abc
#       A110 -> B110            ; #abc -> #abc
#       A111 -> B111            ; #abc -> #abc
#       B000 -> D000            ; #abc -> #abc
#       B001 -> D001            ; #abc -> #abc
#       B010 -> D010            ; #abc -> #abc
#       B011 -> D011            ; #abc -> #abc
#       B100 -> D100            ; #abc -> #abc
#       B101 -> D101            ; #abc -> #abc
#       B110 -> D110            ; #abc -> #abc
#       B111 -> D111            ; #abc -> #abc
#       D000 -> B000            ; #abc -> #abc
#       D001 -> B001            ; #abc -> #abc
#       D010 -> B010            ; #abc -> #abc
#       D011 -> B011            ; #abc -> #abc
#       D100 -> B100            ; #abc -> #abc
#       D101 -> B101            ; #abc -> #abc
#       D110 -> B110            ; #abc -> #abc
#       D111 -> B111            ; #abc -> #abc
#       B000 -> C00 + E0        ; #abc -> #bc + #a
#       B001 -> C01 + E0        ; #abc -> #bc + #a
#       B010 -> C10 + E0        ; #abc -> #bc + #a
#       B011 -> C11 + E0        ; #abc -> #bc + #a
#       B100 -> C00 + E1        ; #abc -> #bc + #a
#       B101 -> C01 + E1        ; #abc -> #bc + #a
#       B110 -> C10 + E1        ; #abc -> #bc + #a
#       B111 -> C11 + E1        ; #abc -> #bc + #a
# B000 + C00 -> D000 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B000 + C01 -> D000 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B000 + C10 -> D001 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B000 + C11 -> D001 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B001 + C00 -> D010 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B001 + C01 -> D010 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B001 + C10 -> D011 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B001 + C11 -> D011 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B010 + C00 -> D100 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B010 + C01 -> D100 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B010 + C10 -> D101 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B010 + C11 -> D101 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B011 + C00 -> D110 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B011 + C01 -> D110 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B011 + C10 -> D111 + E0 + E0  ; #abc + #AB -> #bcA + #a + #B
# B011 + C11 -> D111 + E0 + E1  ; #abc + #AB -> #bcA + #a + #B
# B100 + C00 -> D000 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B100 + C01 -> D000 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B100 + C10 -> D001 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B100 + C11 -> D001 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B101 + C00 -> D010 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B101 + C01 -> D010 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B101 + C10 -> D011 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B101 + C11 -> D011 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B110 + C00 -> D100 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B110 + C01 -> D100 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B110 + C10 -> D101 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B110 + C11 -> D101 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B111 + C00 -> D110 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B111 + C01 -> D110 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
# B111 + C10 -> D111 + E1 + E0  ; #abc + #AB -> #bcA + #a + #B
# B111 + C11 -> D111 + E1 + E1  ; #abc + #AB -> #bcA + #a + #B
#       D000 -> F000            ; #abc -> #abc
#       D001 -> F001            ; #abc -> #abc
#       D010 -> F010            ; #abc -> #abc
#       D011 -> F011            ; #abc -> #abc
#       D100 -> F100            ; #abc -> #abc
#       D101 -> F101            ; #abc -> #abc
#       D110 -> F110            ; #abc -> #abc
#       D111 -> F111            ; #abc -> #abc
#

# loading some usefull julia addon packages
using DifferentialEquations
using Catalyst
using Plots

# define reaction network
rn = @reaction_network Antoniewicz begin
    v1, (A000,A001,A010,A011,A100,A101,A110,A111) --> (B000,B001,B010,B011,B100,B101,B110,B111)
    v2, (B000,B001,B010,B011,B100,B101,B110,B111) --> (D000,D001,D010,D011,D100,D101,D110,D111)
    v3, (D000,D001,D010,D011,D100,D101,D110,D111) --> (B000,B001,B010,B011,B100,B101,B110,B111)
    v4, (B000,B001,B010,B011,B100,B101,B110,B111) --> (C00+E0,C01+E0,C10+E0,C11+E0,C00+E1,C01+E1,C10+E1,C11+E1)
    v5, (B000+C00,B000+C01,B000+C10,B000+C11,B001+C00,B001+C01,B001+C10,B001+C11,
         B010+C00,B010+C01,B010+C10,B010+C11,B011+C00,B011+C01,B011+C10,B011+C11,
         B100+C00,B100+C01,B100+C10,B100+C11,B101+C00,B101+C01,B101+C10,B101+C11,
         B110+C00,B110+C01,B110+C10,B110+C11,B111+C00,B111+C01,B111+C10,B111+C11) -->
             (D000+E0+E0,D000+E0+E1,D001+E0+E0,D001+E0+E1,D010+E0+E0,D010+E0+E1,D011+E0+E0,D011+E0+E1,
              D100+E0+E0,D100+E0+E1,D101+E0+E0,D101+E0+E1,D110+E0+E0,D110+E0+E1,D111+E0+E0,D111+E0+E1,
```

```
            D000+E1+E0,D000+E1+E1,D001+E1+E0,D001+E1+E1,D010+E1+E0,D010+E1+E1,D011+E1+E0,D011+E1+E1,
            D100+E1+E0,D100+E1+E1,D101+E1+E0,D101+E1+E1,D110+E1+E0,D110+E1+E1,D111+E1+E0,D111+E1+E1)
    v6, (D000,D001,D010,D011,D100,D101,D110,D111) --> (F000,F001,F010,F011,F100,F101,F110,F111)
end v1 v2 v3 v4 v5 v6

# initial value array of 38 species
u0 = zeros(size(species(rn)))

# initialize A000 to 100
u0[1] = 100.0
# parameter array [v1, v2, v3, v4, v5, v6]
p = [1.0, 1.1, 0.5, 0.2, 0.2, 0.8]

# simulation time (start, stop)
tspan = (0.0, 10.0)

# setup ODE problem
odes = ODEProblem(rn, u0, tspan, p)

# solve ODES problem
sol = solve(odes)

# plot time course
plot(sol)
```