Simulation of well-mixed stochastic chemical kinetics models

Notebook Setup

Note, this notebook was developed on Julia 1.10.4 and may not work on other versions.

Let's start by loading an environment for our notebook that will have all the needed libraries we'll use today.

In [ ]:
```
using Pkg

# this should be the path to the folder you downloaded from Github
Pkg.activate("./ibs_workshop_code")
```

```
  Activating project at `~/Dropbox/Documents/Presentations/2024-06-11 - IBS Works
hop in Korea/ibs_workshop_code`
```

The **first** time you want to run this code, uncomment and run the following code to install needed packages. Note, after this first time you should not need to run this again and can re-comment it out. *Note*, the code below downloads, installs, and compiles the needed libraries. It can take a while the first time you run it so allow time...

In [ ]:
```
# Pkg.instantiate()
# Pkg.precompile()
```

Let's now load all the libraries and setup some plot defaults:

In [ ]:
```
using Random, LinearAlgebra, BenchmarkTools, Plots, PlotThemes, Plots.PlotMeasur

# plotting defaults
theme(:dracula)
default(; lw = 2, size = (800, 400))
```

Finally, let's load some Latex macros to make equation typing easier:

Simulating a Simple Poisson Process

Let's look at two simple codes for simulating a Poisson counting process, that differ only in their choice of random number generator.

Suppose we have a Poisson counting process, $N(t)$, that fires with rate λ. Assume $N(0) = 0$. We can think of this as the reaction

$$\varnothing \xrightarrow{\lambda} N \tag{1}$$

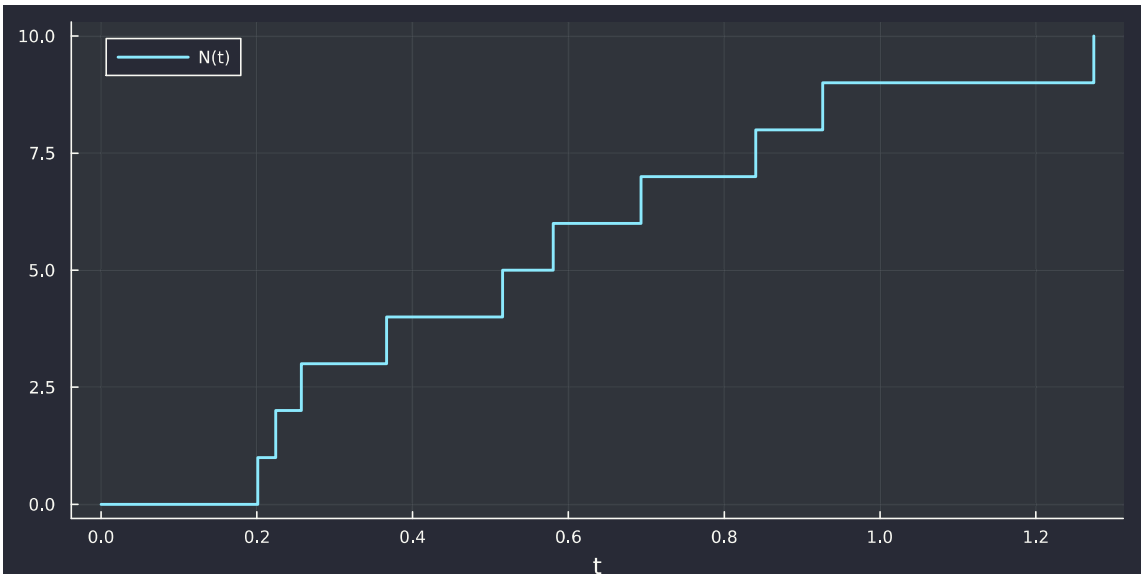
Our first code uses Inverse Transform Sampling to calculate the time of the next reaction, τ, given by

$$\tau = -\frac{\ln(1-u)}{\lambda}, \quad u \in \mathcal{U}([0,1)) \tag{2}$$

In [ ]:
```
function poisproc(p)
    (; Nmax, λ) = p

    N = zeros(Int, Nmax + 1)
    t = zeros(Nmax + 1)

    for i = 1:Nmax
        τ = - log(1 - rand()) / λ
        t[i+1] = t[i] + τ
        N[i+1] = i
    end

    (N, t)
end
```

```
poisproc (generic function with 1 method)
```

In [ ]:
```
p = (; Nmax = 10, λ = 10.0)
N, t = poisproc(p)
p1 = plot(t, N, xlabel = "t", label = "N(t)", linetype = :steppost)
```



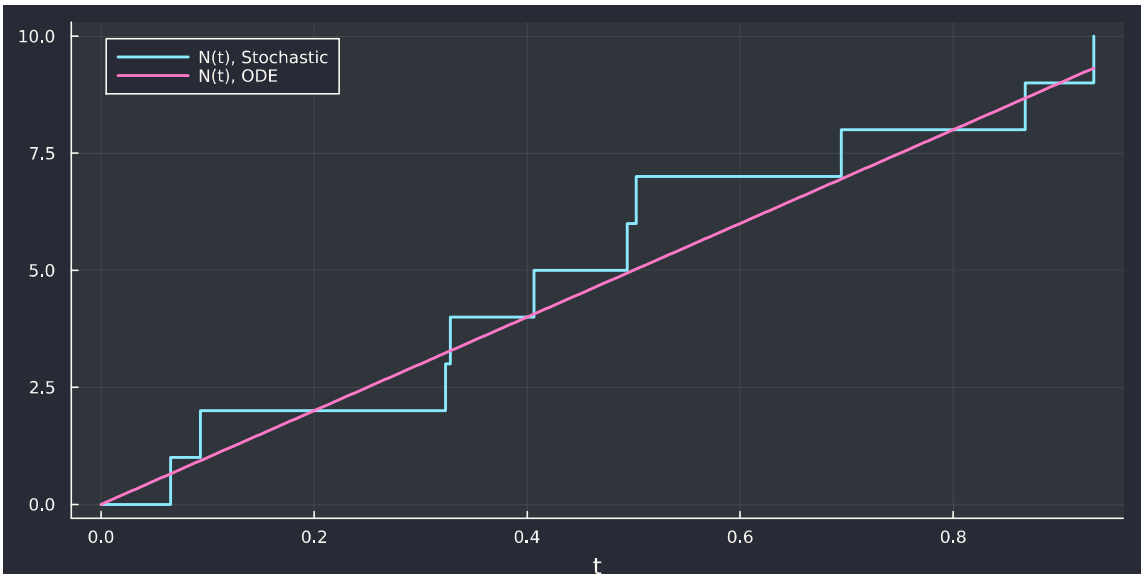
For comparison, the corresponding ODE model for $N(t)$ would be

$$\frac{dN}{dt} = \lambda, \quad N(0) = 0 \tag{3}$$

so that $N(t) = \lambda t$.

Plotting these together we can compare:

In [ ]:
```
N, t = poisproc(p)
p1 = plot(t, N, xlabel = "t", label = "N(t), Stochastic", linetype = :steppost)

times = range(0.0, t[end]; length = 200)
plot!(p1, times, p.λ .* times, label = "N(t), ODE")
```



An alternative method

Sampling $\tau := -\ln(1-u)/\lambda$ can be interpreted as

1. Sampling an exponential random variable with parameter **one** $(-\ln(1-u))$.
2. Converting it to an exponential random variable with parameter λ (dividing by λ).

- Julia provides a builtin random number generator that can do the first step for us, the `randexp` function.
- This uses a Ziggurat method approach instead of inverse transform sampling.

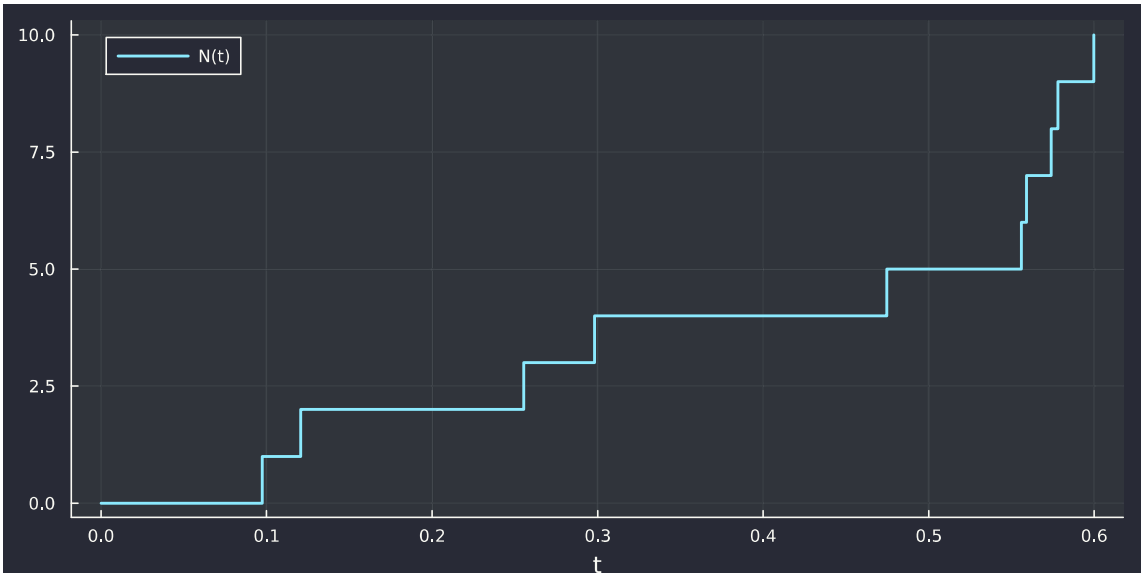
Let's look at how this alternative approach performs:

In [ ]:
```
function randexp_poisproc(p)
    (; Nmax, λ) = p

    N = zeros(Int, Nmax + 1)
    t = zeros(Nmax + 1)

    for i = 1:Nmax
        τ = randexp() / λ
        t[i+1] = t[i] + τ
        N[i+1] = i
    end

    (N, t)
end
```

```
randexp_poisproc (generic function with 1 method)
```

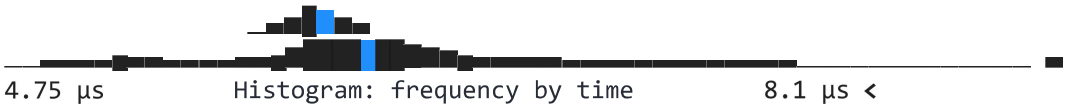
In [ ]:
```
p = (; Nmax = 10, λ = 10.0)
N, t = randexp_poisproc(p)
```

```
plot(t, N, xlabel = "t", label = "N(t)", linetype = :steppost)
```



In [ ]:
```
p = (; Nmax = 1000, λ = 1.0)
bsimple = @benchmark poisproc($p)
```

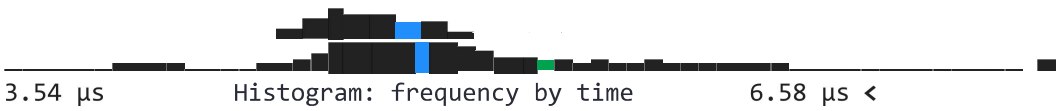
```
BenchmarkTools.Trial: 10000 samples with 6 evaluations.
 Range (min … max):  4.750 μs …  29.612 ms  ┊ GC (min … max):  0.00% … 99.97%
 Time  (median):     5.958 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   9.302 μs ± 296.239 μs  ┊ GC (mean ± σ):  35.10% ±  3.10%
```




```
 Memory estimate: 16.25 KiB, allocs estimate: 2.
```

In [ ]:
```
bfaster = @benchmark randexp_poisproc($p)
```

```
BenchmarkTools.Trial: 10000 samples with 8 evaluations.
 Range (min … max):  3.542 μs … 288.188 μs  ┊ GC (min … max): 0.00% … 98.13%
 Time  (median):     4.760 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   5.192 μs ±   9.102 μs  ┊ GC (mean ± σ):  6.48% ±  3.65%
```




```
 Memory estimate: 16.25 KiB, allocs estimate: 2.
```

The relative difference in the median time of the inverse transform sampling method from the Ziggurat method is:

In [ ]:
```
abs(mean(bsimple).time - mean(bfaster).time) / mean(bsimple).time
```

```
0.44187178792053405
```

We can test this difference more directly by just comparing sampling the unit rate exponential random variables by each approach:

In [ ]:
```
randexp_slow() = -log(1-rand())
```

```
randexp_slow (generic function with 1 method)
```

In [ ]:
```
b_randexp_slow = @benchmark randexp_slow()
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  4.125 ns … 20.417 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     4.709 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   4.806 ns ±  0.463 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  4.12 ns        Histogram: frequency by time        6.21 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [ ]:
```
b_randexp = @benchmark randexp()
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  3.084 ns … 16.500 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     3.500 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   3.575 ns ±  0.401 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  3.08 ns        Histogram: frequency by time        4.71 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [ ]:
```
abs(mean(b_randexp).time - mean(b_randexp_slow).time) / mean(b_randexp_slow).tim
```

```
0.2560774287915302
```

Point: Inverse transform sampling is generally not what we should use for sampling exponential random variables, and should be avoided in codes.

This is particularly relevant for Stochastic Chemical Kinetics simulations, which will typically involve *very* large numbers of samples of exponential random variables!

Basic Direct Method

Let's implement the Direct method for the following autoregulatory negative feedback reaction model.

Our states will be

- $DNA =$ state where gene is not repressed and free to produce mRNA.
- $DNAR =$ state where gene is repressed and unable to transcribe.
- $mRNA =$ number of mRNA
- $P =$ number of proteins.

with the reaction model

$$
\begin{aligned}
DNA &\xrightarrow{\alpha} DNA + mRNA \\
mRNA &\xrightarrow{\beta} mRNA + P \\
mRNA &\xrightarrow{\delta} \varnothing \\
P &\xrightarrow{\mu} \varnothing \\
DNA + P &\xrightarrow{k_+} DNAR \\
DNAR &\xrightarrow{k_-} DNA + P
\end{aligned}
$$

In the notation we just discussed we'll let

- $m =$ the current number of mRNA.
- $p =$ the current number of proteins.
- $d = 1$ when in state DNA and 0 when in state $DNAR$.

The state vector for the system will then be $\boldsymbol{x} = (m, p, d)$.

Given $\boldsymbol{x}$, the propensity vector for the system will be

$$\boldsymbol{a}(\boldsymbol{x}) = (\alpha d, \beta m, \delta m, \mu p, k_+ dp, k_-(1-d))$$

Finally, we for the stoichiometry matrix where the ith column is the change in $\boldsymbol{x}$ when the ith reaction occurs:

$$\boldsymbol{\nu} = [\nu_1, \ldots, \nu_6] = \begin{pmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

In [ ]:
```
function negfeedback_propens!(a, x, ps)
        # rate constants
        (; α, β, δ, μ, k₊, k₋) = ps

        # species
        m = x[1]; p = x[2]; d = x[3]

        # DNA --> DNA + mRNA at rate α
        a[1] = α * d

        # mRNA --> mRNA + P at rate β
        a[2] = β * m

        # mRNA --> 0 at rate δ
        a[3] = δ * m

        # P --> 0 at rate μ
        a[4] = μ * p

        # DNA + P --> DNAR at rate k₊
        a[5] = k₊ * d * p

        # DNAR --> DNA + P at rate k₋
        a[6] = k₋ * (1-d)
```

```
        nothing
end
```

```
negfeedback_propens! (generic function with 1 method)
```

In [ ]:

```
"""
# Runs one Direct Method simulation

## Inputs:
- x₀    = vector of the initial condition
- afun! = in-place function to evaluate ``(a_1(x),...,a_M(x))``
- v     = N by M matrix with k'th column giving v_k
- p     = parameters that should be passed to afun!
- T     = time to stop the simulation

## Optional Keyword Inputs:
- Δt    = if set to a number, state is only saved every Δt units of time

## Returns:
- xv    = vector of vectors storing the state after each jump
- tv    = vector of the times at which reactions occur
"""
function directmethod(x₀, afun!, p, v, T; Δt=nothing)
        x     = copy(x₀)  # current state
    xold  = copy(x)   # to save previous x value
        t     = 0.0       # current time
        xvals = [copy(x)] # saves the state after each jump
        tvals = [0.0]     # saves the times of each jump
        numrx = size(v,2) # number of reactions
        tsave = isnothing(Δt) ? 0.0 : Δt

        # stores the vector (a_1(x),...,a_M(x))
        avec  = zeros(numrx)

        # apply each function in afuns to x
        while t < T

                # eval (a_1(x),...,a_M(x))
                afun!(avec, x, p)

                # calculate the cumulative sum of the propensities
                avec = cumsum!(avec, avec)

                # sample next reaction time
                t += randexp() / avec[end]

                # sample which reaction occurs
                k = searchsortedfirst(avec, rand()*avec[end])

                # execute the reaction
        xold .= x
                x .+= @view v[:,k]

                # saving
                if t < T
                        if isnothing(Δt)
                                push!(xvals, copy(x)); push!(tvals, t)
                        else
                                while tsave <= t
                                        push!(xvals, copy(xold)); push!(tvals, t
```

```
                                end
                        end
                end
        end

        # We need to save the state at time T, it should be the same
        # as at the last time we saved (since the next reaction time t > T!)
        if isnothing(Δt)
                push!(xvals, copy(xvals[end])); push!(tvals, T)
        else
                while tsave <= T
                        push!(xvals, copy(xold)); push!(tvals, tsave); tsave +=
                end
        end
        return xvals,tvals
end
```

directmethod

In [ ]:
```
# helper function to convert the solution returned by directmethod to a matrix
# xmat[i,j] = value of species j at time tᵢ, Xⱼ(tᵢ)
function tomatrix(xv)
        numspecs = length(xv[1])
        numtimes = length(xv)

        xmat = zeros(Int, numtimes, numspecs)

        # store solution at time t as n'th row in xmat
        # hence each column is one species at all times
        for (n,x) in pairs(xv)
                xmat[n,:] .= x
        end

        xmat
end
```

tomatrix (generic function with 1 method)

Let's setup our simulation input. First we have our model parameters

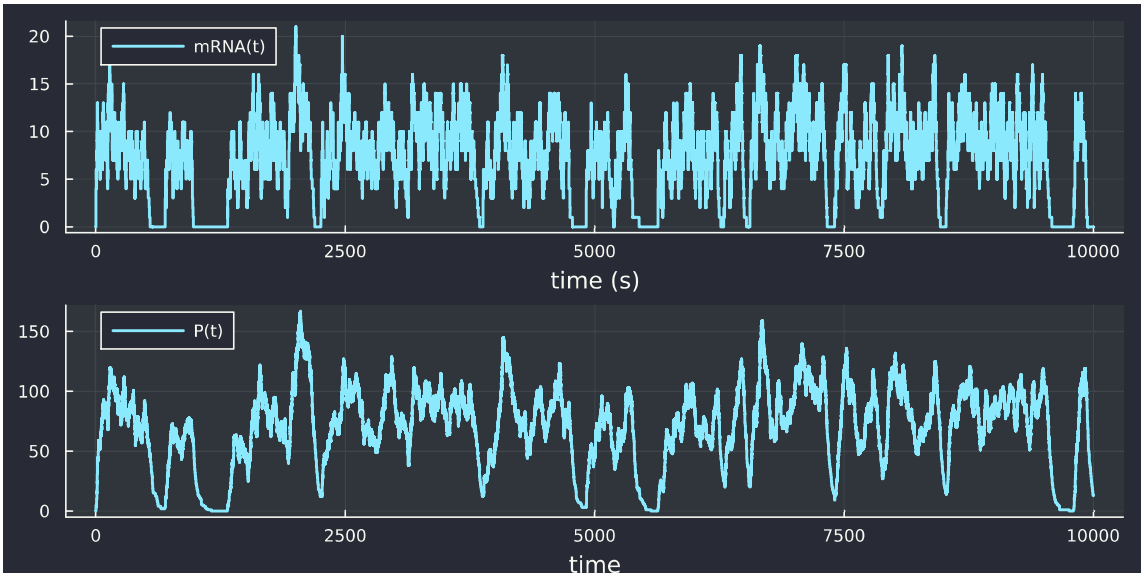
In [ ]:
```
p = (
    α = .5,               # transcription rate (molecules per second)
    β = 5 * log(2) / 12,  # translation rates (per second), 20 proteins on avera
    δ = log(2) / 12,      # mRNA degradation rate (per second)
    μ = log(2) / 24,      # protein degradation rate (per second)
        k₊ = .000025,         # protein-DNA binding rate (per second)
        k₋ = .005             # protein-DNA unbinding rate (per second) (SLOW CA
)

# stoichiometry matrix
# 1: dna -> m + dna
# 2: m -> m + p
# 3: m -> 0
# 4: p -> 0
# 5: dna + p -> dnar (1-dna here)
# 6: dnar -> dna + p
# species order  (changing rows)
# (m, p, d)
v = [1 0 -1  0 0  0;
        0 1  0 -1 -1 1;
```

```
         0 0  0  0 -1 1]

# initial condition, x₀[1] = m(0), x₀[2] = p(0), x₀[3] = d(0)
x₀ = [0,0,1]

# time to stop simulation at
T = 10000.0
```

```
10000.0
```

Finally we can run and plot a simulation:

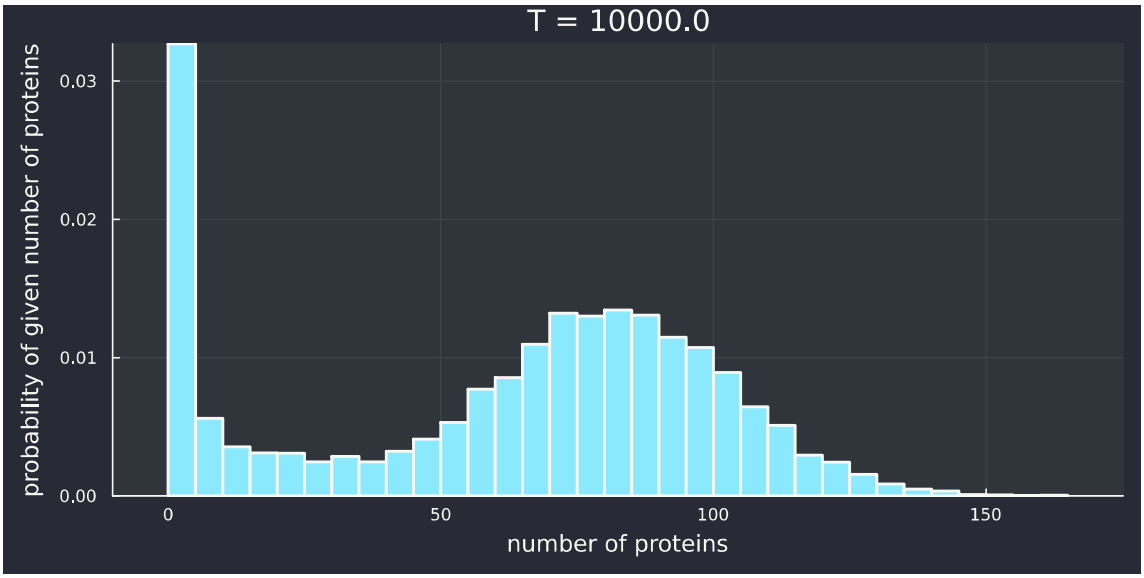
In [ ]:
```
xvals, tvals = directmethod(x₀, negfeedback_propens!, p, v, T)
xsol = tomatrix(xvals)

# plot the number of mRNA vs time
pm = plot(tvals, xsol[:,1], xlabel="time (s)", label="mRNA(t)", seriestype=:step
    legend = :topleft)

# plot the number of proteins vs time
pp = plot(tvals, xsol[:,2], xlabel="time", label="P(t)", seriestype=:steppost,
    legend = :topleft)

plot(pm, pp, layout = (2,1))
```



In [ ]:
```
T = 10000.
Nsims = 10000
Δt = T          # only save the solution values at times 0. and tf

# vector of SS values from each simulation
mRNA = Vector{Int}(undef, Nsims)
prot = Vector{Int}(undef, Nsims)

# loop over number of simulations to run
for i = 1:Nsims
    xv, tv = directmethod(x₀, negfeedback_propens!, p, v, T; Δt)
    mRNA[i] = xv[end][1]
    prot[i] = xv[end][2]
end
```

In [ ]:
```
# plot a histogram of the distribution of proteins at SS
histogram(prot, normalize=true, xlabel="number of proteins",
```

```
    ylabel="probability of given number of proteins",
    title = "T = $T",
    bins=40, legend = false, left_margin = 5mm, bottom_margin = 5mm)
```



One simple performance tweak

Users often forget that such simulations can have **many** individual reaction occurrences.

- While saving the state at the time of each reaction allows one to exactly reconstruct the path, it can mean **each simulation** is allocating memory to save large vectors for the state.
- If the exact path is not needed, it is much better to just save simuulations results at the needed times.

Let's see how this simple tweak can speed up our simulation:

In [ ]:
```
T = 100000.0
Δt = T

# saving all times
b_save_all_state = @benchmark directmethod($x₀, $negfeedback_propens!, $p, $v, $

# saving just the final time
b_save_at_final_time = @benchmark directmethod($x₀, $negfeedback_propens!, $p, $


# percent difference of latter from former
abs(median(b_save_all_state).time - median(b_save_at_final_time).time) / median(
```

```
0.34577424367122256
```

Representing reactions in different ways; impact on code performance

We've now seen two simple code changes that can give noticeable speedups. Before we discuss *algorithmic improvements*, one last code optimization that can impact speed is **how** we represent the propensity functions.

Above I just filled in a vector of them directly, with a function that knows their explicit formulas. But general simulation libraries like JumpProcesses (in Julia), PySB (Python), StochSim (Python), etc. will usually have a per-reaction mechanism for evaluating rates.

- How propensities and reactions are represented can also have a significant performance impact.

Let's rebuild our last example in Julia's Catalyst.jl library, and then run a Direct method simulation using JumpProcesses.jl.

We start by loading Catalyst to create our model and JumpProcesses to simulate it:

In [ ]:
```
using Catalyst, JumpProcesses
```

```
WARNING: using Catalyst.mm in module Main conflicts with an existing identifier.
```

We then create our symbolic reaction model in Catalyst

In [ ]:
```
rn = @reaction_network repressed_gene begin
    α, DNA --> DNA + mRNA
    β, mRNA --> mRNA + P
    δ, mRNA --> 0
    μ, P --> 0
    k₊, DNA + P --> DNAR
    k₋, DNAR --> DNA + P
end
rn = complete(rn)
```

$$\begin{aligned}
\mathrm{DNA} &\xrightarrow{\alpha} \mathrm{DNA} + \mathrm{mRNA} \\
\mathrm{mRNA} &\xrightarrow{\beta} \mathrm{mRNA} + \mathrm{P} \\
\mathrm{mRNA} &\xrightarrow{\delta} \varnothing \\
\mathrm{P} &\xrightarrow{\mu} \varnothing \\
\mathrm{DNA} + \mathrm{P} &\xrightleftharpoons[k_-]{k_+} \mathrm{DNAR}
\end{aligned}$$

We next specify a mapping from the symbolic species to their initial values, and the symbolic parameters to this initial values.

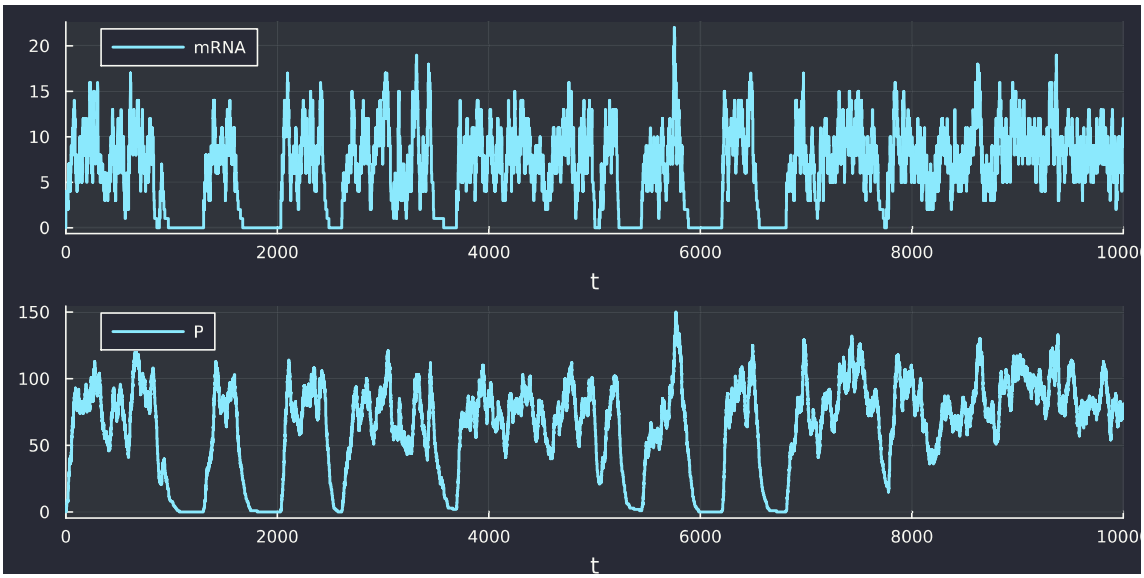
In [ ]:
```
# parameter mapping
pmap = [rn.α => .5, rn.β => 5 * log(2) / 12, rn.δ => log(2) / 12,
    rn.μ => log(2) / 24, rn.k₊ => .000025, rn.k₋ => .005]

# initial condition mapping
u0map = [rn.mRNA => 0, rn.P => 0, rn.DNA => 1, rn.DNAR => 0]
```

```
# time intervale to solve over
tspan = (0.0, 10000.0)
```

```
(0.0, 10000.0)
```

We then convert our symbolic problem into concrete input for the JumpProcesses' Direct method, simulate the model, and plot the amount of mRNA and proteins we find:

In [ ]:
```
# this stores out input parameter values and initial conditions
dprob = DiscreteProblem(rn, u0map, tspan, pmap)

# this creates the input to JumpProcesses and selects the Direct method
# as our simulation approach
jprob = JumpProblem(rn, dprob, Direct())

# Solve and then plot our model
sol = solve(jprob, SSAStepper())
p1 = plot(sol, idxs = rn.mRNA)
p2 = plot(sol, idxs = rn.P)
plot(p1, p2; layout = (2,1))
```



So we see a similar type of trajectory as before.

One last performance optimization

JumpProcesses offers several ways to represents jumps (i.e. our reactions).

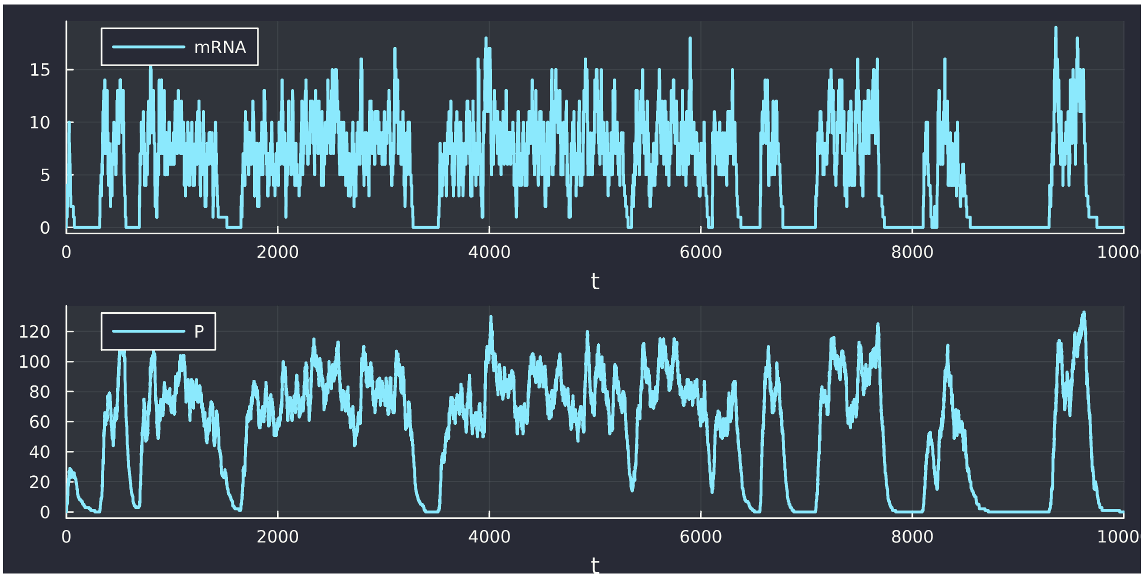
1. For mass action reactions you (i.e. Catalyst) provides the reaction rate constant, the net stoichiometry vector and the substrate stoichiometry vector.
   - These are called `MassActionJump`s in JumpProcesses.
2. To support more general reactions users can also pass general `rate` functions to evaluate the current propensity given the state and time, and a general `affect!` function that updates the state when a given reaction occurs.
   - This enables non-mass action rates to be simulated.
   - These are called `ConstantRateJump`s in JumpProcesses.

Our model is purely mass action, so Catalyst generated `MassActionJump`s automatically. Let's see the performance difference if we instead manually create `ConstantRateJump`s for each reaction:

In [ ]:
```
# x = (m, p, d)

# DNA --> DNA + mRNA
r1(x, p, t) = p.α * x[3]
af1!(integ) = (integ.u[1] += 1)
crj1 = ConstantRateJump(r1, af1!)

# mRNA --> mRNA + P
r2(x, p, t) = p.β * x[1]
af2!(integ) = (integ.u[2] += 1)
crj2 = ConstantRateJump(r2, af2!)

# mRNA --> ∅
r3(x, p, t) = p.δ * x[1]
af3!(integ) = (integ.u[1] -= 1)
crj3 = ConstantRateJump(r3, af3!)

# P --> 0
r4(x, p, t) = p.μ * x[2]
af4!(integ) = (integ.u[2] -= 1)
crj4 = ConstantRateJump(r4, af4!)

# DNA + P --> DNAR
r5(x, p, t) = p.k₊ * x[3] * x[2]
af5!(integ) = (integ.u[2] -= 1; integ.u[3] -= 1)
crj5 = ConstantRateJump(r5, af5!)

# DNAR --> DNA + P
r6(x, p, t) = p.k₋ * (1 - x[3])
af6!(integ) = (integ.u[2] += 1; integ.u[3] += 1)
crj6 = ConstantRateJump(r6, af6!)

# collect the jumps together
jset = JumpSet(crj1, crj2, crj3, crj4, crj5, crj6)
```

```
JumpSet{Tuple{}, Tuple{ConstantRateJump{typeof(r1), typeof(af1!)}, ConstantRateJu
mp{typeof(r2), typeof(af2!)}, ConstantRateJump{typeof(r3), typeof(af3!)}, Constan
tRateJump{typeof(r4), typeof(af4!)}, ConstantRateJump{typeof(r5), typeof(af5!)},
ConstantRateJump{typeof(r6), typeof(af6!)}}, Nothing, Nothing}((), (ConstantRateJ
ump{typeof(r1), typeof(af1!)}(r1, af1!), ConstantRateJump{typeof(r2), typeof(af
2!)}(r2, af2!), ConstantRateJump{typeof(r3), typeof(af3!)}(r3, af3!), ConstantRat
eJump{typeof(r4), typeof(af4!)}(r4, af4!), ConstantRateJump{typeof(r5), typeof(af
5!)}(r5, af5!), ConstantRateJump{typeof(r6), typeof(af6!)}(r6, af6!)), nothing, n
othing)
```

In [ ]:
```
T = 10000.0
tspan = (0.0, T)
u0 = [0, 0, 1]
dprob = DiscreteProblem(u0, tspan, p)
jprob = JumpProblem(dprob, Direct(), jset)
sol = solve(jprob, SSAStepper())
p1 = plot(sol, idxs = 1, label = "mRNA", xlabel = "t")
p2 = plot(sol, idxs = 2, label = "P", xlabel = "t")
plot(p1, p2; layout = (2,1))
```



In [ ]:
```
T = 100000.0
tspan = (0.0, T)
dprob = DiscreteProblem(rn, u0map, tspan, pmap)
jprob = JumpProblem(rn, dprob, Direct(); save_positions = (false, false))
b_massact = @benchmark solve($(jprob), $(SSAStepper()))
```

```
BenchmarkTools.Trial: 252 samples with 1 evaluation.
 Range (min … max):  18.117 ms …  21.761 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     19.853 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   19.865 ms ± 661.592 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 18.1 ms         Histogram: frequency by time        21.2 ms <

 Memory estimate: 2.05 KiB, allocs estimate: 11.
```

In [ ]:
```
dprob = DiscreteProblem(u0, tspan, p)
jprob = JumpProblem(dprob, DirectFW(), jset; save_positions = (false, false))
b_crj = @benchmark solve($(jprob), $(SSAStepper()))
```

```
BenchmarkTools.Trial: 42 samples with 1 evaluation.
 Range (min … max):  108.247 ms … 292.095 ms  ┊ GC (min … max):  6.04% … 62.21%
 Time  (median):     116.502 ms               ┊ GC (median):     7.46%
 Time  (mean ± σ):   120.379 ms ±  27.406 ms  ┊ GC (mean ± σ):  10.04% ±  8.65%

 108 ms          Histogram: frequency by time         292 ms <

 Memory estimate: 222.60 MiB, allocs estimate: 7294277.
```

In [ ]:
```
median(b_crj).time / median(b_massact).time
```

```
5.868157369560453
```

We see that `ConstantRateJump` s are **much** slower than `MassActionJump` s!

Why?

1. For `MassActionJump` s, internally one function is available that takes the stoichiometry and rate constant information, the current state value (i.e. $\boldsymbol{x}$), and

returns the associated propensity value.

2. For `ConstantRateJump`s internally these are generally stored as a vector of function pointers.

The overhead of calling the functions via the pointers results in the slow down.

Conclusion: If you are writing a general stochastic chemical kinetics solver, how you even represent your propensity functions can have a big performance impact!

As an aside, for this specific network the fastest possible code actually uses the `ConstantRateJump`s, but uses a compiler trick to actually expand the vector of rate and affect functions inline within the `Direct` code:

In [ ]:
```
dprob = DiscreteProblem(u0, tspan, p)
jprob = JumpProblem(dprob, Direct(), jset; save_positions = (false, false))
b_crj_fast = @benchmark solve($(jprob), $(SSAStepper()))
```

```
BenchmarkTools.Trial: 337 samples with 1 evaluation.
 Range (min … max):  13.121 ms … 17.014 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     14.798 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   14.860 ms ± 661.791 μs ┊ GC (mean ± σ):  0.00% ± 0.00%

 13.1 ms         Histogram: frequency by time        16.5 ms <

 Memory estimate: 1.17 KiB, allocs estimate: 11.
```

In [ ]:
```
median(b_massact).time / median(b_crj_fast).time
```

```
1.3415892523272281
```

Showing it is actually 30% faster than using the `MassActionJump` representation. While this is a nice speedup, it only works with Direct for very small systems, so we don't usually exploit it via Catalyst.

Optimized Solvers Compared to Direct

Let's compare the various SSAs we've discussed on our gene expression example. Note the following code can take a while to run -- you can speed it up by reducing the final time, T.

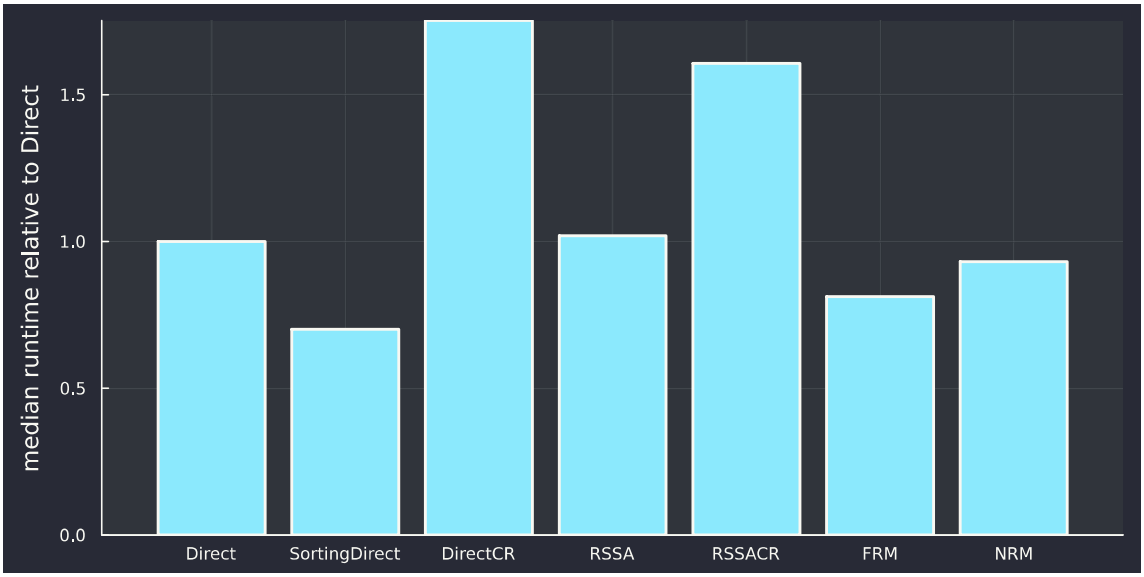
In [ ]:
```
function benchmark_ssas(ssas, rn, dprob)
    medtimes = zeros(length(ssas))
    for (i, ssa) in enumerate(ssas)
        jprob = JumpProblem(rn, dprob, ssa(); save_positions = (false, false))
        medtimes[i] = median(@benchmark solve($(jprob), $(SSAStepper())) samples
    end
    return medtimes
end
```

```
benchmark_ssas (generic function with 1 method)
```

In [ ]:
```
ssas = (Direct, SortingDirect, DirectCR, RSSA, RSSACR, FRM, NRM)
T = 100000.0
tspan = (0.0, T)
dprob = DiscreteProblem(rn, u0map, tspan, pmap)
medtimes = benchmark_ssas(ssas, rn, dprob)
```

```
7-element Vector{Float64}:
 1.9947354e7
 1.39907705e7
 3.49640205e7
 2.0340625e7
 3.2045417e7
 1.62134165e7
 1.8583042e7
```

In [ ]:
```
bar([string(ssa) for ssa in ssas], medtimes ./ medtimes[1], ylabel = "median run
```



Scratch code

In [ ]:
```
# p = (; Nmax = 10, λ = 10.0)
# times = range(0.0, 1.0, length = 200)
# p2 = plot(times, 1 .- exp.(-p.λ .* times), xlabel = "t", label = "F(τ) = CDF o
# plot!(p2, times, p.λ .* exp.(-p.λ .* times), xlabel = "t", label = "p(τ; n) =
# savefig(p2, "./cdf_pdf_zero_order.pdf")
```

```
"/Users/isaacsas/Dropbox/Documents/Presentations/2024-06-11 - IBS Workshop in Kor
ea/cdf_pdf_zero_order.pdf"
```