```julia
begin
    # numerical libraries
    using Expokit, PROPACK, Arpack, SparseArrays
    # output and plotting
    using ProgressLogging, JLD, CairoMakie
    # modelling and statistics
    using Catalyst, JumpProcesses, StatsBase, DifferentialEquations
    using Interpolations
    # importing local fsp package
    using Revise
    local_mod = include("../src/DiscStochSim.jl")
    using .local_mod.DiscStochSim
end
```

```
Replacing docs for `Main.var"workspace#2".DiscStochSim.FindLowestValuesPercent :
: Union{Tuple{T}, Tuple{Vector{T}, Number}} where T` in module `Main.var"workspa
ce#2".DiscStochSim`
```

rn =

$$
\begin{aligned}
\mathrm{X} &\xrightarrow{k_1} 2\mathrm{X} \\
\mathrm{X} + \mathrm{Y} &\xrightarrow{k_2} 2\mathrm{Y} \\
\mathrm{Y} &\xrightarrow{k_3} \varnothing
\end{aligned}
$$

```julia
rn = @reaction_network begin
    k₁, X --> 2X
    k₂, X + Y --> 2Y
    k₃, Y --> 0
end
```

```julia
model = DiscreteStochasticSystem(rn);
```

def_params = 0.0:0.005:30.0

```julia
def_params = begin
    # reaction rates
    rates = [1.0, 0.005, 0.6]
    # boundary
    bounds = (0, 500) #(lower limit, upper limit)
    boundary_condition(x) = RectLatticeBoundaryCondition(x, bounds);
    # time interval and initial values
    δt = 0.005
    T = 0:δt:30
end
```

Initialize Finite State Space

```julia
init_fsp_vars = begin
    global U₀ = CartesianIndex(50, 100)
    global 𝒮₀ = Set([U₀])
    global 𝒮₀ = expand!(𝒮₀, model, rates, 0.0, boundary_condition, 1);
    # initial probability vector (only for active states)
    global p₀ = zeros(𝒮₀ |> length)
    global p₀[FindElement(U₀, 𝒮₀)] = 1
end;
```

fsp_sim =

```julia
fsp_sim = begin
    # copy initial values
    pₜ = copy(p₀)
    𝒮ₜ = copy(𝒮₀)
    # time stepping loop
    iter = 1
    pₜ = p₀
    # variables to store simulation observables
    size_𝒮ₜ = Int.(zeros(length(T))) # system sizes
    ϵₜ = zeros(length(T)) #local truncation err
    sol = Array{SparseMatrixCSC,1}(undef, length(T))
    @progress for (iter, t) ∈ enumerate(T)
        # expand state space
        global 𝒮ₜ, pₜ = expand!(𝒮ₜ, pₜ, model, rates , t, boundary_condition, 3)
        global size_𝒮ₜ[iter] = length(𝒮ₜ)
        A = MasterEquation(𝒮ₜ, model, rates, boundary_condition, t)
        # solve system and normalize (using expokit)
        global pₜ = expmv(δt, A, pₜ)
        # add add states in sparse arrays
        I = [a[1] for a in 𝒮ₜ]
        J = [a[2] for a in 𝒮ₜ]
        global sol[iter] = sparse(I, J, pₜ)
        # purge state space
        𝒮ₜ, pₜ = purge!(𝒮ₜ, pₜ, 3.0)
        ϵₜ[iter] = 1.0 - sum(pₜ)
        pₜ ./= sum(pₜ)
    end
end
```

100%

FSP Mean Trajectory

Since the solution computed by the FSP algorithm is

$$X_{mean} = \sum_{i=0}^{N} X_i \cdot p(X_i, t)$$

```
compute_fsp_mean = begin
    cart2vec(x) = [Tuple(x)...]
    sol_mean = map(1:length(T)) do i
        I, J, V = findnz(sol[i])
        sum(hcat(I, J) .* V, dims=1)
    end

    x = @inbounds [sol_mean[i][1] for i ∈ 1:length(sol_mean)]
    y = @inbounds [sol_mean[i][2] for i ∈ 1:length(sol_mean)]
    fsp_mean = hcat(x, y);
end;
```

Generate SSA Trajectories

We use jump processes

generate_ssa_samples =

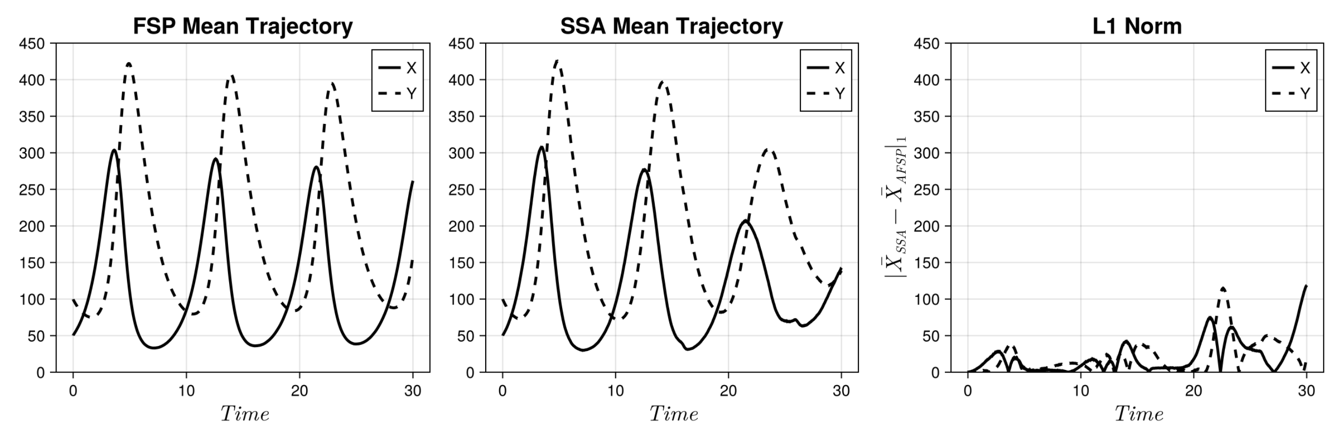
```
generate_ssa_samples = begin
    # Create an ODE that can be simulated.
    u0_integers = [:X => 50, :Y => 100]
    ps = [:k₁ => 1.0, :k₂ => 0.005, :k₃ => 0.6]
    tspan = (0., 30.)

    jinput = JumpInputs(rn, u0_integers, tspan, ps)
    jprob = JumpProblem(jinput)
    jump_sol = solve(jprob)

    n_trajs = 300 # increase for a more accurate ssa mean
    ssa_trajs1=[]
    @progress for i in 1:n_trajs
        push!(ssa_trajs1, solve(jprob, SSAStepper()))
    end;
end
```

100%

```
uniform_time, ssa_mean = mean_trajectory(ssa_trajs1, T[1], T[end], length(T));
```




```
@save "lv_data.jld" sol, ϵₜ size_𝒮ₜ
```