Computational methods

Assignment 4

Rasmus Hammar

```python
# Imports
import numpy as np
import matplotlib.pyplot as plt
import sys

sys.path.append(
    "c:\\Programming\\Projects\\BioInf-Master\\Computational_methods\\Lab_5\
\Assignment_4"
)
import gillespy
from scipy.integrate import solve_ivp
```

Stochastic model

Use/encode this stochastic model.

$$
\left.\begin{array}{rcl}
D'_a & \xrightarrow{\theta_a D'_a} & D_a \\
D_a + A & \xrightarrow{\gamma_a D_a A} & D'_a \\
D'_r & \xrightarrow{\theta_r D'_r} & D_r \\
D_r + A & \xrightarrow{\gamma_r D_r A} & D'_r
\end{array}\right\}
\quad
\left.\begin{array}{rcl}
\emptyset & \xrightarrow{\alpha'_a D'_a} & M_a \\
\emptyset & \xrightarrow{\alpha_a D_a} & M_a \\
M_a & \xrightarrow{\delta_{ma} M_a} & \emptyset
\end{array}\right\}
\quad
\left.\begin{array}{rcl}
\emptyset & \xrightarrow{\beta_a M_a} & A \\
\emptyset & \xrightarrow{\theta_a D'_a} & A \\
\emptyset & \xrightarrow{\theta_r D'_r} & A \\
A & \xrightarrow{\delta_a A} & \emptyset \\
A + R & \xrightarrow{\gamma_c AR} & C
\end{array}\right\}
$$

$$
\left.\begin{array}{rcl}
\emptyset & \xrightarrow{\alpha'_r D'_r} & M_r \\
\emptyset & \xrightarrow{\alpha_r D_r} & M_r \\
M_r & \xrightarrow{\delta_{mr} M_r} & \emptyset
\end{array}\right\}
\quad
\left.\begin{array}{rcl}
\emptyset & \xrightarrow{\beta_r M_r} & R \\
R & \xrightarrow{\delta_r R} & \emptyset \\
C & \xrightarrow{\delta_a C} & R
\end{array}\right\}
$$

Figure 1: Stochastic model

a)

Reproduce Figure 2, c & d, from the article.

Constants and initial values

```
# Constants
alpha = [
    50,  # a_A
    0.01,  # a_R
    500,  # a_prim_A
    50,  # a_prim_R
]
beta = [
    50,  # b_A
    5,  # b_R
]
gamma = [
    1,  # g_A
    1,  # g_R
    2,  # g_C
]
delta = [
    1,  # d_A
    0.2,  # d_R
    10,  # d_MA
    0.5,  # d_MR
]
theta = [
    50,  # t_A
    100,  # t_R
]
coeff = {"alpha": alpha, "beta": beta, "gamma": gamma, "delta": delta,
"theta": theta}

# Initial values
y_0 = [
    1,  # D_A
    0,  # Dprim_A
    1,  # D_R
    0,  # Dprim_R
    0,  # M_A
    0,  # M_R
    0,  # A
    0,  # R
    0,  # C
]
```

Propensity function

```
def prop(Y, coeff):
    # Unpacking Y
    D_A = Y[0]  # dD_A/dt
    Dprim_A = Y[1]  # dDprim_A/dt
    D_R = Y[2]  # dD_R/dt
    Dprim_R = Y[3]  # dDprim_R/dt
    M_A = Y[4]  # dM_A/dt
    M_R = Y[5]  # dM_R/dt
    A = Y[6]  # dA/dt
    R = Y[7]  # dR/dt
    C = Y[8]  # dC/dt

    # Unpacking constants
    a_A, a_R, a_prim_A, a_prim_R = coeff["alpha"]
    b_A, b_R = coeff["beta"]
    g_A, g_R, g_C = coeff["gamma"]
    d_A, d_R, d_MA, d_MR = coeff["delta"]
    t_A, t_R = coeff["theta"]

    # Propensity vector
    prop = [
        t_A * Dprim_A,
        g_A * D_A * A,
        t_R * Dprim_R,
        g_R * D_R * A,
        #
        a_prim_A * Dprim_A,
        a_A * D_A,
        d_MA * M_A,
        #
        b_A * M_A,
        t_A * Dprim_A,
        t_R * Dprim_R,
        d_A * A,
        g_C * A * R,
        #
        a_prim_R * Dprim_R,
        a_R * D_R,
        d_MR * M_R,
        #
        b_R * M_R,
        d_R * R,
        d_A * C,
    ]

    return prop
```

Stochiometry matrix

```
stoch = np.array(
    [
        # D_A, Dprim_A, D_R, Dprim_R, M_A, M_R, A, R, C
        [   1,      -1,   0,       0,   0,   0, 0, 0, 0],  # t_A * Dprim_A
        [  -1,       1,   0,       0,   0,   0,-1, 0, 0],  # g_A * D_A * A
        [   0,       0,   1,      -1,   0,   0, 0, 0, 0],  # t_R * Dprim_R
        [   0,       0,  -1,       1,   0,   0,-1, 0, 0],  # g_R * D_R * A
        #
        [   0,       0,   0,       0,   1,   0, 0, 0, 0],  # a_prim_A *
Dprim_A
        [   0,       0,   0,       0,   1,   0, 0, 0, 0],  # a_A * D_A
        [   0,       0,   0,       0,  -1,   0, 0, 0, 0],  # d_MA * M_A
        #
        [   0,       0,   0,       0,   0,   0, 1, 0, 0],  # b_A * M_A
        [   0,       0,   0,       0,   0,   0, 1, 0, 0],  # t_A * Dprim_A
        [   0,       0,   0,       0,   0,   0, 1, 0, 0],  # t_R * Dprim_R
        [   0,       0,   0,       0,   0,   0,-1, 0, 0],  # d_A * A
        [   0,       0,   0,       0,   0,   0,-1,-1, 1],  # g_C * A * R
        #
        [   0,       0,   0,       0,   0,   1, 0, 0, 0],  # a_prim_R *
Dprim_R
        [   0,       0,   0,       0,   0,   1, 0, 0, 0],  # a_R * D_R
        [   0,       0,   0,       0,   0,  -1, 0, 0, 0],  # d_MR * M_R
        #
        [   0,       0,   0,       0,   0,   0, 0, 1, 0],  # b_R * M_R
        [   0,       0,   0,       0,   0,   0, 0,-1, 0],  # d_R * R
        [   0,       0,   0,       0,   0,   0, 0, 1,-1],  # d_A * C
    ]
)
```

Time span

```
# Time span
t_0 = 0
t_stop = 400
times = np.arange(t_0, t_stop, 0.1)
```

Run SSA

```
t, Y = gillespy.SSA(
    prop = prop,
    stoch = stoch,
    X0 = y_0,
    tspan = (t_0, t_stop),
    coeff = coeff
)
```

Run the ODE for comparison

```python
def ode_rhs(t, y, coeff):
    # Unpacking y
    D_A = y[0]  # dD_A/dt
    Dprim_A = y[1]  # dDprim_A/dt
    D_R = y[2]  # dD_R/dt
    Dprim_R = y[3]  # dDprim_R/dt
    M_A = y[4]  # dM_A/dt
    M_R = y[5]  # dM_R/dt
    A = y[6]  # dA/dt
    R = y[7]  # dR/dt
    C = y[8]  # dC/dt

    # Unpacking constants
    a_A, a_R, a_prim_A, a_prim_R = coeff["alpha"]
    b_A, b_R = coeff["beta"]
    g_A, g_R, g_C = coeff["gamma"]
    d_A, d_R, d_MA, d_MR = coeff["delta"]
    t_A, t_R = coeff["theta"]

    # Equation system
    yt = [
        t_A * Dprim_A - g_A * D_A * A,  # dD_A/dt
        g_A * D_A * A - t_A * Dprim_A,  # dDprim_A/dt
        t_R * Dprim_R - g_R * D_R * A,  # dD_R/dt
        g_R * D_R * A - t_R * Dprim_R,  # dDprim_R/dt
        a_prim_A * Dprim_A + a_A * D_A - d_MA * M_A,  # dM_A/dt
        a_prim_R * Dprim_R + a_R * D_R - d_MR * M_R,  # dM_R/dt
        b_A * M_A
        + t_A * Dprim_A
        + t_R * Dprim_R
        - A * (g_A * D_A + g_R * D_R + g_C * R + d_A),  # dA/dt
        b_R * M_R - g_C * A * R + d_A * C - d_R * R,  # dR/dt
        g_C * A * R - d_A * C,  # dC/dt
    ]

    return yt
```

```python
sol = solve_ivp(ode_rhs, (t_0, t_stop), y_0, args=(coeff, ), t_eval=times,
method="BDF")
```

Results

The results from the SSA is largely the same as the ODE. The SSA is a less smooth due to the introduced randomnes.

```python
plt.figure(1)
plt.subplot(4, 1, 1)
plt.plot(sol.t, sol.y[5], color="blue")
plt.ylabel("A (ODE)")

plt.subplot(4, 1, 2)
plt.plot(sol.t, sol.y[7], color="orange")
plt.ylabel("R (ODE)")

plt.subplot(4, 1, 3)
plt.plot(t, Y[:, 6], color="green")
plt.ylabel("A (SSA)")

plt.subplot(4, 1, 4)
plt.plot(t, Y[:, 7], color="purple")
plt.ylabel("R (SSA)")
plt.xlabel("time [hr]")
plt.show()
```

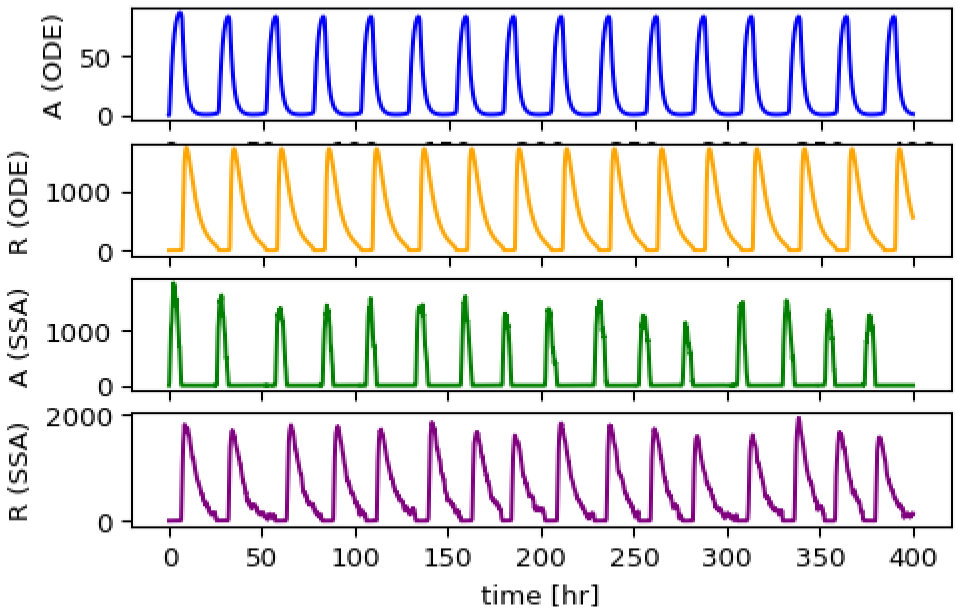

Figure 2: Recreation of Figure 2 from the article.

b)

We change the parameter $\delta_R = 0.2h^{-1}$ to $\delta_R = 0.05h^{-1}$ and compare the ODE to the SSA for R.

```
coeff["delta"][1] = 0.05  # d_R
```

```
sol = solve_ivp(
    ode_rhs,
    (t_0, t_stop),
    y_0,
    args=(coeff,),
    t_eval=times,
    method="BDF",
)

t, Y = gillespy.SSA(prop=prop, stoch=stoch, X0=y_0, tspan=(t_0, t_stop),
coeff=coeff)
```

Results

In the deterministic model, R sinks to near zero and never recovers while the SSA model retains the cyclic spikes. Showing that stochastic models are more robust when modeling biological processes.

```
plt.figure(2)
plt.subplot(2, 1, 1)
plt.plot(sol.t, sol.y[7], color="orange")
plt.ylabel("R (ODE)")

plt.subplot(2, 1, 2)
plt.plot(t, Y[:, 7], color="purple")
plt.ylabel("R (SSA)")
plt.xlabel("time [hr]")
plt.show()
```

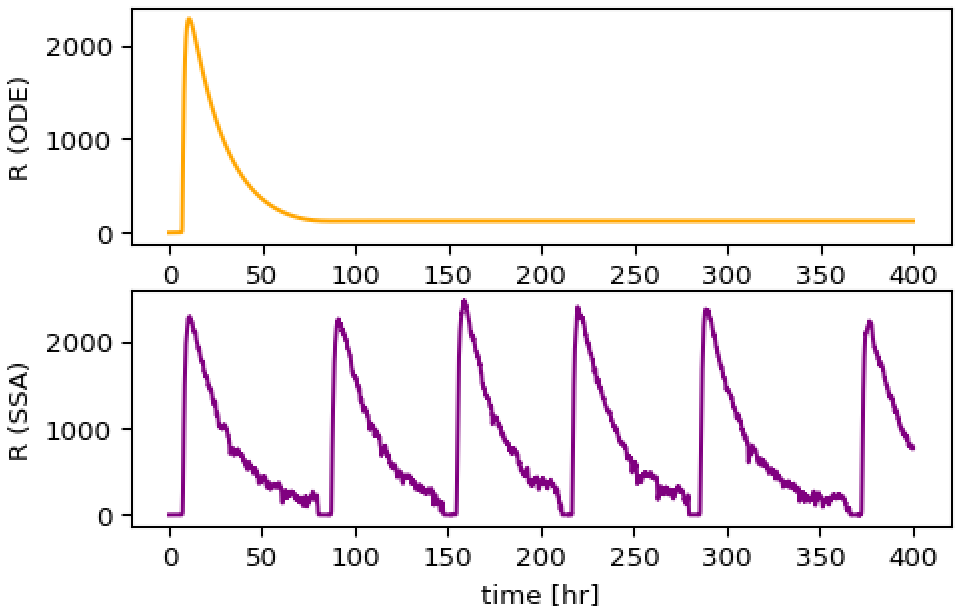

Figure 3: Comparison of ODE vs SSA showing that stochastic models sometimes handle biological processes better.