```
# -*- coding: utf-8 -*-
"""
Interactive 2D multicellular simulation of SNAIL-ZEB-miR200-miR34 gene regulatory network
with intercellular diffusion and video-like visualization using ipywidgets.
Neumann boundary conditions are applied.
"""

import numpy as np
import math
from numba import njit, prange
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, jslink, VBox, Output
import time
from typing import Tuple
from IPython.display import display
from google.colab import output
output.enable_custom_widget_manager()


# ------------------------------
# Grid and simulation parameters
# ------------------------------
GRID_SIZE = 50             # 10x10 multicellular system
STATE_SIZE = 7              # 7 molecules per cell
T_END = 500.0              # Total simulation time
DT = 0.1/15                    # Time step
STEPS = int(T_END / DT)     # Number of simulation steps
D_COEFF = np.array([0.5, 0.8, 0.6, 0.4, 0.4, 0.5, 0.0])  # diffusion per molecule
SAVE_EVERY = 10          # save every nth frame for visualization

# ------------------------------
# Molecular parameters
# ------------------------------
L = np.array([1.0, 0.6, 0.3, 0.1, 0.05, 0.05, 0.05], dtype=np.float64)
GAMMA_MRNA = np.array([0.0, 0.04, 0.2, 1.0, 1.0, 1.0, 1.0], dtype=np.float64)
GAMMA_MIRNA = np.array([0.0, 0.005, 0.05, 0.5, 0.5, 0.5, 0.5], dtype=np.float64)
COMB_6 = np.array([math.comb(6, i) for i in range(7)], dtype=np.float64)
COMB_2 = np.array([math.comb(2, i) for i in range(3)], dtype=np.float64)
INDICES_7 = np.arange(7, dtype=np.float64)
INDICES_3 = np.arange(3, dtype=np.float64)

G_MIR34, G_MSNAIL = 1.35e3, 90.0
G_SNAIL, G_MIR200 = 0.1e3, 2.1e3
G_MZEB, G_ZEB = 11.0, 0.1e3


K_MIR34, K_MSNAIL = 0.05, 0.5
K_SNAIL, K_MIR200 = 0.125, 0.05
K_MZEB, K_ZEB = 0.5, 0.1

T_MIR34_SNAIL, T_MSNAIL_SNAIL = 300e3, 200e3
T_MIR34_ZEB, T_MIR34, T_MSNAIL_I = 600e3, 10e3, 50e3
T_MIR200_ZEB, T_MIR200_SNAIL = 220e3, 180e3
T_MZEB_ZEB, T_MZEB_SNAIL, T_MIR200 = 25e3, 180e3, 10e3



N_MIR34_SNAIL, N_MIR34_ZEB = 1, 1
N_MSNAIL_SNAIL, N_MSNAIL_I = 1, 1
N_MIR200_ZEB, N_MIR200_SNAIL = 3, 2
N_MZEB_ZEB, N_MZEB_SNAIL = 2, 2

L_MIR34_SNAIL, L_MSNAIL_SNAIL = 0.1, 0.1
L_MIR34_ZEB, L_MSNAIL_I = 0.2, 10.0
L_MIR200_ZEB, L_MIR200_SNAIL = 0.1, 0.1
L_MZEB_ZEB, L_MZEB_SNAIL = 7.5, 10.0

# ------------------------------
# Numba-optimized single-cell dynamics
# ------------------------------
@njit(fastmath=True)
def hill(x, threshold, n, leak):
    base = 1.0 / (1.0 + (x / threshold) ** n)
    return base + leak * (1.0 - base)
```

```
@njit(fastmath=True)
def mir200_terms(state):
    fac_num = state[0] / T_MIR200
    fac = np.power(fac_num, INDICES_7) / np.power(1.0 + fac_num, 6)
    degrad_mir200 = np.sum(GAMMA_MIRNA * COMB_6 * INDICES_7 * fac)
    degrad_mzeb = np.sum(GAMMA_MRNA * COMB_6 * fac)
    trans_mzeb = np.sum(L * COMB_6 * fac)
    return degrad_mir200, degrad_mzeb, trans_mzeb

@njit(fastmath=True)
def mir34_terms(state):
    fac_num = state[5] / T_MIR34
    fac = np.power(fac_num, INDICES_3) / np.power(1.0 + fac_num, 2)
    degrad_mir34 = np.sum(GAMMA_MIRNA[:3] * COMB_2 * INDICES_3 * fac)
    degrad_msna = np.sum(GAMMA_MRNA[:3] * COMB_2 * fac)
    trans_msna = np.sum(L[:3] * COMB_2 * fac)
    return degrad_mir34, degrad_msna, trans_msna

@njit(fastmath=True)
def cell_rhs(state : np.ndarray) -> np.ndarray:
    deriv = np.zeros(STATE_SIZE, dtype=np.float64)
    mir200 = mir200_terms(state)
    mir34 = mir34_terms(state)

    h_mir200_zeb = hill(state[2], T_MIR200_ZEB, N_MIR200_ZEB, L_MIR200_ZEB)
    h_mir200_sna = hill(state[3], T_MIR200_SNAIL, N_MIR200_SNAIL, L_MIR200_SNAIL)
    h_mzeb_zeb = hill(state[2], T_MZEB_ZEB, N_MZEB_ZEB, L_MZEB_ZEB)
    h_mzeb_sna = hill(state[3], T_MZEB_SNAIL, N_MZEB_SNAIL, L_MZEB_SNAIL)
    h_mir34_sna = hill(state[3], T_MIR34_SNAIL, N_MIR34_SNAIL, L_MIR34_SNAIL)
    h_mir34_zeb = hill(state[2], T_MIR34_ZEB, N_MIR34_ZEB, L_MIR34_ZEB)
    h_msna_sna = hill(state[3], T_MSNAIL_SNAIL, N_MSNAIL_SNAIL, L_MSNAIL_SNAIL)
    h_msna_i = hill(state[6], T_MSNAIL_I, N_MSNAIL_I, L_MSNAIL_I)

    deriv[0] = G_MIR200 * h_mir200_zeb * h_mir200_sna - state[1] * mir200[0] - K_MIR200 * state[0]
    deriv[1] = G_MZEB * h_mzeb_zeb * h_mzeb_sna - state[1] * mir200[1] - K_MZEB * state[1]
    deriv[2] = G_ZEB * state[1] * mir200[2] - K_ZEB * state[2]

    deriv[3] = G_SNAIL * state[4] * mir34[2] - K_SNAIL * state[3]
    deriv[4] = G_MSNAIL * h_msna_i * h_msna_sna - state[4] * mir34[1] - K_MSNAIL * state[4]
    deriv[5] = G_MIR34 * h_mir34_zeb * h_mir34_sna - state[4] * mir34[0] - K_MIR34 * state[5]
    deriv[6] = 0.0
    return deriv

# ----------------------------
# Optimized 2D Laplacian
# ----------------------------
@njit(parallel=True, fastmath=True)
def laplacian_2d_optimized(grid: np.ndarray, lap: np.ndarray) -> np.ndarray:
    """
    Computes the 2D Laplacian using 5-point stencil with Neumann (zero-flux) B.C.
    The boundary conditions are implemented by using 2*interior_neighbor
    instead of (interior_neighbor + exterior_fictitious_neighbor).
    """
    nx, ny, ns = grid.shape

    # 1. Interior Cells (i, j in [1, N-2])
    for i in prange(1, nx - 1):
        for j in range(1, ny - 1):
            for k in range(ns):
                lap[i, j, k] = (
                    grid[i + 1, j, k] + grid[i - 1, j, k] +
                    grid[i, j + 1, k] + grid[i, j - 1, k] -
                    4.0 * grid[i, j, k]
                )

    # Use a separate loop for boundaries (parallelized over chemical species k)
    for k in prange(ns):
        # 2. Edges (excluding corners)

        # Left (i=0, j in [1, N-2]): 2 * grid[1] replaces (grid[1] + grid[-1_fict])
        for j in range(1, ny - 1):
            lap[0, j, k] = (
                2.0 * grid[1, j, k] +
                grid[0, j + 1, k] + grid[0, j - 1, k] -
                4.0 * grid[0, j, k]
            )

        # Right (i=N-1, j in [1, N-2]): 2 * grid[N-2]
```

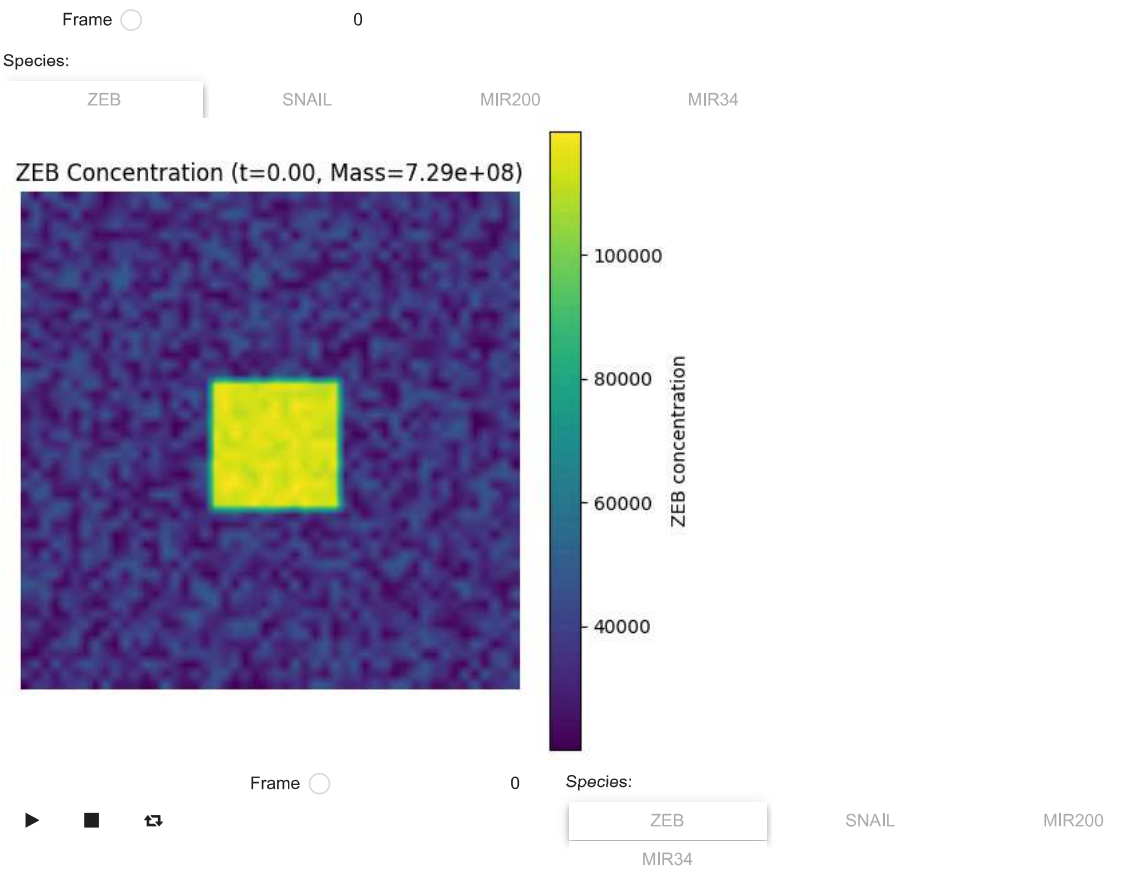
```
            for j in range(1, ny - 1):
                lap[nx - 1, j, k] = (
                    2.0 * grid[nx - 2, j, k] +
                    grid[nx - 1, j + 1, k] + grid[nx - 1, j - 1, k] -
                    4.0 * grid[nx - 1, j, k]
                )

            # Bottom (j=0, i in [1, N-2]): 2 * grid[1]
            for i in range(1, nx - 1):
                lap[i, 0, k] = (
                    grid[i + 1, 0, k] + grid[i - 1, 0, k] +
                    2.0 * grid[i, 1, k] -
                    4.0 * grid[i, 0, k]
                )

            # Top (j=N-1, i in [1, N-2]): 2 * grid[N-2]
            for i in range(1, nx - 1):
                lap[i, ny - 1, k] = (
                    grid[i + 1, ny - 1, k] + grid[i - 1, ny - 1, k] +
                    2.0 * grid[i, ny - 2, k] -
                    4.0 * grid[i, ny - 1, k]
                )

            # 3. Corners (Double Flux-Free Boundary)

            # (0, 0) - Left-Bottom: 2*grid[1, 0] + 2*grid[0, 1]
            lap[0, 0, k] = (2.0 * grid[1, 0, k] + 2.0 * grid[0, 1, k] - 4.0 * grid[0, 0, k])

            # (N-1, 0) - Right-Bottom: 2*grid[N-2, 0] + 2*grid[N-1, 1]
            lap[nx - 1, 0, k] = (2.0 * grid[nx - 2, 0, k] + 2.0 * grid[nx - 1, 1, k] - 4.0 * grid[nx - 1, 0, k])

            # (0, N-1) - Left-Top: 2*grid[1, N-1] + 2*grid[0, N-2]
            lap[0, ny - 1, k] = (2.0 * grid[1, ny - 1, k] + 2.0 * grid[0, ny - 2, k] - 4.0 * grid[0, ny - 1, k])

            # (N-1, N-1) - Right-Top: 2*grid[N-2, N-1] + 2*grid[N-1, N-2]
            lap[nx - 1, ny - 1, k] = (2.0 * grid[nx - 2, ny - 1, k] + 2.0 * grid[nx - 1, ny - 2, k] - 4.0 * grid[nx - 1, ny - 1, k])

        return lap

    # ----------------------------
    # Optimized simulation with frame saving
    # ----------------------------
    @njit(fastmath=True)
    def simulate_multicell_frames(grid : np.ndarray, dt : float, steps : int, D : np.ndarray, save_every : int) -> Tuple[np.ndarray,
        nx, ny, ns = grid.shape
        tmp = np.zeros_like(grid)
        lap = np.zeros_like(grid)
        n_frames = steps // save_every + 1
        frames = np.zeros((n_frames, nx, ny, ns), dtype=np.float64)
        mass = np.zeros(n_frames, dtype=np.float64)

        frame_idx = 0
        frames[frame_idx] = grid.copy()
        mass[frame_idx] = np.sum(grid)
        frame_idx += 1

        for t in range(1, steps+1):
            laplacian_2d_optimized(grid, lap)
            for i in range(nx):
                for j in range(ny):
                    rhs = cell_rhs(grid[i,j,:])
                    for k in range(ns):
                        # Include both diffusion and reaction terms here
                        tmp[i,j,k] = grid[i,j,k] + dt*(D[k]*lap[i,j,k] + rhs[k])
            grid[:,:,:] = tmp[:,:,:]

            if t % save_every == 0:
                frames[frame_idx] = grid.copy()
                mass[frame_idx] = np.sum(grid)
                frame_idx += 1

        return frames, mass

    # -------------------
    # NUMBA WARM-UP BLOCK
    # -------------------
    grid_w = np.zeros((5,5,STATE_SIZE), dtype=np.float64)
```

```
tmp_w  = np.zeros_like(grid_w)
lap_w  = np.zeros_like(grid_w)

laplacian_2d_optimized(grid_w, lap_w)
_ = cell_rhs(grid_w[0,0,:])
simulate_multicell_frames(grid_w, 0.01, 3, D_COEFF, 3)
# -------------------
# END WARM-UP BLOCK
# -------------------

# -----------------------------
# Initialize and run simulation
# -----------------------------
if __name__ == "__main__":

    np.random.seed(42)

    GRID_SIZE = 50      # example grid size
    STATE_SIZE = 7      # number of molecules per cell

    # Initialize the full grid with very low concentrations
    grid_init = np.zeros((GRID_SIZE, GRID_SIZE, STATE_SIZE))

    # Define central square coordinates
    center = GRID_SIZE // 2
    half_patch = 6  # 13x13 central square (since 6 on each side + center)

    # High concentration values for the central patch
    high_conc_min = 110e3
    high_conc_max = 120e3

    # Low concentration values for the rest of the grid
    low_conc_min = 20e3
    low_conc_max = 50e3

    # Constant value for the 7th molecule (k = 6)
    constant_value = 50e3   # <-- set this as desired

    for k in range(STATE_SIZE):
        if k < 6:
            # Fill entire grid with low concentrations
            grid_init[:, :, k] = np.random.uniform(
                low_conc_min, low_conc_max, (GRID_SIZE, GRID_SIZE)
            )

            # Overwrite central square with high concentrations
            grid_init[
                center - half_patch : center + half_patch + 1,
                center - half_patch : center + half_patch + 1,
                k
            ] = np.random.uniform(
                high_conc_min, high_conc_max, (2 * half_patch + 1, 2 * half_patch + 1)
            )

        else:
            # For k = 6 → constant value everywhere (including central patch)
            grid_init[:, :, k] = constant_value

    # Optional: tiny random noise to break symmetry
    grid_init += np.random.rand(GRID_SIZE, GRID_SIZE, STATE_SIZE) * 0.5

    start_time = time.time()
    frames, mass = simulate_multicell_frames(grid_init, DT, STEPS, D_COEFF, SAVE_EVERY)
    print(f"Simulation done in {time.time() - start_time:.2f}s")
    n_frames = frames.shape[0]
```

```
Simulation done in 139.44s
```

```
# -----------------------------
# Interactive visualization: species toggle + frame slider
# -----------------------------
from ipywidgets import Play, IntSlider, jslink, HBox, ToggleButtons, interact
from IPython.display import display
import matplotlib.pyplot as plt

# Let's define species mapping
species_map = {
```

```
    "ZEB": 2,
    "SNAIL": 3,
    "MIR200": 0,
    "MIR34": 5
}

# Function to plot a given species at a given frame
def plot_at_time(idx, species):
    plt.figure(figsize=(6,6))

    # select the species index
    k = species_map[species]

    # select data for that frame
    data = frames[idx, :, :, k]

    title = f"{species} Concentration (t={idx * SAVE_EVERY * DT:.2f}, Mass={mass[idx]:.2e})"

    img = plt.imshow(data, cmap='viridis', interpolation='bilinear')
    plt.colorbar(img, ax=plt.gca(), label=f"{species} concentration")
    plt.title(title)
    plt.axis('off')
    plt.show()

# Species toggle
species_toggle = ToggleButtons(
    options=list(species_map.keys()),
    value="ZEB",
    description="Species:",
    button_style="info"
)

# Frame controls
play = Play(value=0, min=0, max=frames.shape[0]-1, step=1, interval=100, description="Play")
slider = IntSlider(value=0, min=0, max=frames.shape[0]-1, step=1, description='Frame')

# Layout with toggle button
widgets = HBox([play, slider, species_toggle])
interact_ui = interact(plot_at_time, idx=slider, species=species_toggle)

# Link play and slider
jslink((play, 'value'), (slider, 'value'))

# Display everything
display(widgets, interact_ui)
```



plot_at_time
def plot_at_time(idx, species)

```
import numpy as np

# Compute global percentiles for each species to avoid spikes ruining contrast
species_vmin = {}
species_vmax = {}

for sp, k in species_map.items():
    data = frames[:, :, :, k]        # shape = (T,50,50)
    species_vmin[sp] = np.percentile(data, 1)   # 1st percentile
    species_vmax[sp] = np.percentile(data, 99)  # 99th percentile

print("vmin values:", species_vmin)
print("vmax values:", species_vmax)
```

```
vmin values: {'ZEB': np.float64(714091.5482744654), 'SNAIL': np.float64(222588.94793660604), 'MIR200': np.float64(680.6425094611
vmax values: {'ZEB': np.float64(1332397.936031316), 'SNAIL': np.float64(1458966.5944064031), 'MIR200': np.float64(1328.661300253
```

```
# -*- coding: utf-8 -*-
"""
Interactive 2D multicellular simulation of SNAIL-ZEB-miR200-miR34 gene regulatory network
with intercellular diffusion and video-like visualization using ipywidgets.
Neumann boundary conditions are applied.
"""

import numpy as np
import math
from numba import njit, prange
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, jslink, VBox, Output
import time
from typing import Tuple
from IPython.display import display
from google.colab import output
output.enable_custom_widget_manager()
```

```
# ------------------------------
# Grid and simulation parameters
# ------------------------------
GRID_SIZE = 50             # 10x10 multicellular system
STATE_SIZE = 7              # 7 molecules per cell
T_END = 500.0              # Total simulation time
DT = 0.1/15                    # Time step
STEPS = int(T_END / DT)     # Number of simulation steps
D_COEFF = np.array([0.5, 0.8, 0.6, 0.4, 0.4, 0.5, 0.0])  # diffusion per molecule
SAVE_EVERY = 10          # save every nth frame for visualization

# ------------------------------
# Molecular parameters
# ------------------------------
L = np.array([1.0, 0.6, 0.3, 0.1, 0.05, 0.05, 0.05], dtype=np.float64)
GAMMA_MRNA = np.array([0.0, 0.04, 0.2, 1.0, 1.0, 1.0, 1.0], dtype=np.float64)
GAMMA_MIRNA = np.array([0.0, 0.005, 0.05, 0.5, 0.5, 0.5, 0.5], dtype=np.float64)
COMB_6 = np.array([math.comb(6, i) for i in range(7)], dtype=np.float64)
COMB_2 = np.array([math.comb(2, i) for i in range(3)], dtype=np.float64)
INDICES_7 = np.arange(7, dtype=np.float64)
INDICES_3 = np.arange(3, dtype=np.float64)

G_MIR34, G_MSNAIL = 1.35e3, 90.0
G_SNAIL, G_MIR200 = 0.1e3, 2.1e3
G_MZEB, G_ZEB = 11.0, 0.1e3


K_MIR34, K_MSNAIL = 0.05, 0.5
K_SNAIL, K_MIR200 = 0.125, 0.05
K_MZEB, K_ZEB = 0.5, 0.1

T_MIR34_SNAIL, T_MSNAIL_SNAIL = 300e3, 200e3
T_MIR34_ZEB, T_MIR34, T_MSNAIL_I = 600e3, 10e3, 50e3
T_MIR200_ZEB, T_MIR200_SNAIL = 220e3, 180e3
T_MZEB_ZEB, T_MZEB_SNAIL, T_MIR200 = 25e3, 180e3, 10e3



N_MIR34_SNAIL, N_MIR34_ZEB = 1, 1
N_MSNAIL_SNAIL, N_MSNAIL_I = 1, 1
N_MIR200_ZEB, N_MIR200_SNAIL = 3, 2
N_MZEB_ZEB, N_MZEB_SNAIL = 2, 2

L_MIR34_SNAIL, L_MSNAIL_SNAIL = 0.1, 0.1
L_MIR34_ZEB, L_MSNAIL_I = 0.2, 10.0
L_MIR200_ZEB, L_MIR200_SNAIL = 0.1, 0.1
L_MZEB_ZEB, L_MZEB_SNAIL = 7.5, 10.0

# ------------------------------
# Numba-optimized single-cell dynamics
# ------------------------------
@njit(fastmath=True)
def hill(x, threshold, n, leak):
    base = 1.0 / (1.0 + (x / threshold) ** n)
    return base + leak * (1.0 - base)

@njit(fastmath=True)
def mir200_terms(state):
    fac_num = state[0] / T_MIR200
    fac = np.power(fac_num, INDICES_7) / np.power(1.0 + fac_num, 6)
    degrad_mir200 = np.sum(GAMMA_MIRNA * COMB_6 * INDICES_7 * fac)
    degrad_mzeb = np.sum(GAMMA_MRNA * COMB_6 * fac)
    trans_mzeb = np.sum(L * COMB_6 * fac)
    return degrad_mir200, degrad_mzeb, trans_mzeb

@njit(fastmath=True)
def mir34_terms(state):
    fac_num = state[5] / T_MIR34
    fac = np.power(fac_num, INDICES_3) / np.power(1.0 + fac_num, 2)
    degrad_mir34 = np.sum(GAMMA_MIRNA[:3] * COMB_2 * INDICES_3 * fac)
    degrad_msna = np.sum(GAMMA_MRNA[:3] * COMB_2 * fac)
    trans_msna = np.sum(L[:3] * COMB_2 * fac)
    return degrad_mir34, degrad_msna, trans_msna

@njit(fastmath=True)
def cell_rhs(state : np.ndarray) -> np.ndarray:
```

```
        deriv = np.zeros(STATE_SIZE, dtype=np.float64)
        mir200 = mir200_terms(state)
        mir34 = mir34_terms(state)

        h_mir200_zeb = hill(state[2], T_MIR200_ZEB, N_MIR200_ZEB, L_MIR200_ZEB)
        h_mir200_sna = hill(state[3], T_MIR200_SNAIL, N_MIR200_SNAIL, L_MIR200_SNAIL)
        h_mzeb_zeb = hill(state[2], T_MZEB_ZEB, N_MZEB_ZEB, L_MZEB_ZEB)
        h_mzeb_sna = hill(state[3], T_MZEB_SNAIL, N_MZEB_SNAIL, L_MZEB_SNAIL)
        h_mir34_sna = hill(state[3], T_MIR34_SNAIL, N_MIR34_SNAIL, L_MIR34_SNAIL)
        h_mir34_zeb = hill(state[2], T_MIR34_ZEB, N_MIR34_ZEB, L_MIR34_ZEB)
        h_msna_sna = hill(state[3], T_MSNAIL_SNAIL, N_MSNAIL_SNAIL, L_MSNAIL_SNAIL)
        h_msna_i = hill(state[6], T_MSNAIL_I, N_MSNAIL_I, L_MSNAIL_I)

        deriv[0] = G_MIR200 * h_mir200_zeb * h_mir200_sna - state[1] * mir200[0] - K_MIR200 * state[0]
        deriv[1] = G_MZEB * h_mzeb_zeb * h_mzeb_sna - state[1] * mir200[1] - K_MZEB * state[1]
        deriv[2] = G_ZEB * state[1] * mir200[2] - K_ZEB * state[2]

        deriv[3] = G_SNAIL * state[4] * mir34[2] - K_SNAIL * state[3]
        deriv[4] = G_MSNAIL * h_msna_i * h_msna_sna - state[4] * mir34[1] - K_MSNAIL * state[4]
        deriv[5] = G_MIR34 * h_mir34_zeb * h_mir34_sna - state[4] * mir34[0] - K_MIR34 * state[5]
        deriv[6] = 0.0
        return deriv

    # ----------------------------
    # Optimized 2D Laplacian
    # ----------------------------
    @njit(parallel=True, fastmath=True)
    def laplacian_2d_optimized(grid: np.ndarray, lap: np.ndarray) -> np.ndarray:
        """
        Computes the 2D Laplacian using 5-point stencil with Neumann (zero-flux) B.C.
        The boundary conditions are implemented by using 2*interior_neighbor
        instead of (interior_neighbor + exterior_fictitious_neighbor).
        """
        nx, ny, ns = grid.shape

        # 1. Interior Cells (i, j in [1, N-2])
        for i in prange(1, nx - 1):
            for j in range(1, ny - 1):
                for k in range(ns):
                    lap[i, j, k] = (
                        grid[i + 1, j, k] + grid[i - 1, j, k] +
                        grid[i, j + 1, k] + grid[i, j - 1, k] -
                        4.0 * grid[i, j, k]
                    )

        # Use a separate loop for boundaries (parallelized over chemical species k)
        for k in prange(ns):
            # 2. Edges (excluding corners)

            # Left (i=0, j in [1, N-2]): 2 * grid[1] replaces (grid[1] + grid[-1_fict])
            for j in range(1, ny - 1):
                lap[0, j, k] = (
                    2.0 * grid[1, j, k] +
                    grid[0, j + 1, k] + grid[0, j - 1, k] -
                    4.0 * grid[0, j, k]
                )

            # Right (i=N-1, j in [1, N-2]): 2 * grid[N-2]
            for j in range(1, ny - 1):
                lap[nx - 1, j, k] = (
                    2.0 * grid[nx - 2, j, k] +
                    grid[nx - 1, j + 1, k] + grid[nx - 1, j - 1, k] -
                    4.0 * grid[nx - 1, j, k]
                )

            # Bottom (j=0, i in [1, N-2]): 2 * grid[1]
            for i in range(1, nx - 1):
                lap[i, 0, k] = (
                    grid[i + 1, 0, k] + grid[i - 1, 0, k] +
                    2.0 * grid[i, 1, k] -
                    4.0 * grid[i, 0, k]
                )

            # Top (j=N-1, i in [1, N-2]): 2 * grid[N-2]
            for i in range(1, nx - 1):
                lap[i, ny - 1, k] = (
                    grid[i + 1, ny - 1, k] + grid[i - 1, ny - 1, k] +
```

```
                    2.0 * grid[i, ny - 2, k] -
                    4.0 * grid[i, ny - 1, k]
                )

            # 3. Corners (Double Flux-Free Boundary)

            # (0, 0) - Left-Bottom: 2*grid[1, 0] + 2*grid[0, 1]
            lap[0, 0, k] = (2.0 * grid[1, 0, k] + 2.0 * grid[0, 1, k] - 4.0 * grid[0, 0, k])

            # (N-1, 0) - Right-Bottom: 2*grid[N-2, 0] + 2*grid[N-1, 1]
            lap[nx - 1, 0, k] = (2.0 * grid[nx - 2, 0, k] + 2.0 * grid[nx - 1, 1, k] - 4.0 * grid[nx - 1, 0, k])

            # (0, N-1) - Left-Top: 2*grid[1, N-1] + 2*grid[0, N-2]
            lap[0, ny - 1, k] = (2.0 * grid[1, ny - 1, k] + 2.0 * grid[0, ny - 2, k] - 4.0 * grid[0, ny - 1, k])

            # (N-1, N-1) - Right-Top: 2*grid[N-2, N-1] + 2*grid[N-1, N-2]
            lap[nx - 1, ny - 1, k] = (2.0 * grid[nx - 2, ny - 1, k] + 2.0 * grid[nx - 1, ny - 2, k] - 4.0 * grid[nx - 1, ny - 1, k]

        return lap

    # ----------------------------
    # Optimized simulation with frame saving
    # ----------------------------
    @njit(fastmath=True)
    def simulate_multicell_frames(grid : np.ndarray, dt : float, steps : int, D : np.ndarray, save_every : int) -> Tuple[np.ndarray
        nx, ny, ns = grid.shape
        tmp = np.zeros_like(grid)
        lap = np.zeros_like(grid)
        n_frames = steps // save_every + 1
        frames = np.zeros((n_frames, nx, ny, ns), dtype=np.float64)
        mass = np.zeros(n_frames, dtype=np.float64)

        frame_idx = 0
        frames[frame_idx] = grid.copy()
        mass[frame_idx] = np.sum(grid)
        frame_idx += 1

        for t in range(1, steps+1):
            laplacian_2d_optimized(grid, lap)
            for i in range(nx):
                for j in range(ny):
                    rhs = cell_rhs(grid[i,j,:])
                    for k in range(ns):
                        # Include both diffusion and reaction terms here
                        tmp[i,j,k] = grid[i,j,k] + dt*(D[k]*lap[i,j,k] + rhs[k])
            grid[:,:,:] = tmp[:,:,:]

            if t % save_every == 0:
                frames[frame_idx] = grid.copy()
                mass[frame_idx] = np.sum(grid)
                frame_idx += 1

        return frames, mass

    # -------------------
    # NUMBA WARM-UP BLOCK
    # -------------------
    grid_w = np.zeros((5,5,STATE_SIZE), dtype=np.float64)
    tmp_w  = np.zeros_like(grid_w)
    lap_w  = np.zeros_like(grid_w)

    laplacian_2d_optimized(grid_w, lap_w)
    _ = cell_rhs(grid_w[0,0,:])
    simulate_multicell_frames(grid_w, 0.01, 3, D_COEFF, 3)
    # -------------------
    # END WARM-UP BLOCK
    # -------------------

    # ----------------------------
    # Initialize and run simulation
    # ----------------------------
    if __name__ == "__main__":

        np.random.seed(42)

        GRID_SIZE = 50
        STATE_SIZE = 7
```

```
    grid_init = np.zeros((GRID_SIZE, GRID_SIZE, STATE_SIZE))

    # Coordinate grid
    x, y = np.meshgrid(np.arange(GRID_SIZE), np.arange(GRID_SIZE))
    center = GRID_SIZE / 2

    # Radial distance from center
    radius = np.sqrt((x - center)**2 + (y - center)**2)

    # Ring parameters
    ring_spacing = 4.0      # distance between rings
    base_level = 25e3
    amplitude = 90e3

    for k in range(STATE_SIZE - 1):
        # Alternating high/low rings
        grid_init[:, :, k] = base_level + amplitude * (
            0.5 * (1.0 + np.sin(radius / ring_spacing))
        )

        # Small biological noise
        grid_init[:, :, k] += np.random.rand(GRID_SIZE, GRID_SIZE) * 2e3

    # Constant external signal / inducer
    grid_init[:, :, 6] = 50e3

    # Run simulation
    start_time = time.time()
    frames, mass = simulate_multicell_frames(
        grid_init, DT, STEPS, D_COEFF, SAVE_EVERY
    )
    print(f"Simulation done in {time.time() - start_time:.2f}s")
    n_frames = frames.shape[0]
```

```
Simulation done in 153.74s
```

```
# ----------------------------
# Interactive visualization: species toggle + frame slider
# ----------------------------
from ipywidgets import Play, IntSlider, jslink, HBox, ToggleButtons, interact
from IPython.display import display
import matplotlib.pyplot as plt

# Let's define species mapping
species_map = {
    "ZEB": 2,
    "SNAIL": 3,
    "MIR200": 0,
    "MIR34": 5
}

# Function to plot a given species at a given frame
def plot_at_time(idx, species):
    plt.figure(figsize=(6,6))

    # select the species index
    k = species_map[species]

    # select data for that frame
    data = frames[idx, :, :, k]

    title = f"{species} Concentration (t={idx * SAVE_EVERY * DT:.2f}, Mass={mass[idx]:.2e})"

    img = plt.imshow(data, cmap='viridis', interpolation='bilinear')
    plt.colorbar(img, ax=plt.gca(), label=f"{species} concentration")
    plt.title(title)
    plt.axis('off')
    plt.show()

# Species toggle
species_toggle = ToggleButtons(
    options=list(species_map.keys()),
    value="ZEB",
    description="Species:",
    button_style="info"
```

```
)

# Frame controls
play = Play(value=0, min=0, max=frames.shape[0]-1, step=1, interval=100, description="Play")
slider = IntSlider(value=0, min=0, max=frames.shape[0]-1, step=1, description='Frame')

# Layout with toggle button
widgets = HBox([play, slider, species_toggle])
interact_ui = interact(plot_at_time, idx=slider, species=species_toggle)

# Link play and slider
jslink((play, 'value'), (slider, 'value'))

# Display everything
display(widgets, interact_ui)
```

Frame 0

Species:

ZEB SNAIL MIR200 MIR34