Temporal

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact_manual, FloatSlider
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

# ===== Quasi-steady dimer functions =====
def plot_stream(alpha,beta,n,Kd):
  def compute_A2_B2(Atot, Btot, Kd=Kd):
      S = Atot + Btot
      denom_common = Kd + 4.0*S + np.sqrt(Kd**2 + 8.0*S*Kd)
      A2 = 2.0 * Atot**2 / denom_common
      B2 = 2.0 * Btot**2 / denom_common
      return A2, B2

  # ===== Vector field definition =====
  def dAt_dt(Atot, Btot):
      A2, _ = compute_A2_B2(Atot, Btot)
      actA = beta * (A2**n) / (1.0 + A2**n)
      return alpha + actA - Atot

  def dBt_dt(Atot, Btot):
      _, B2 = compute_A2_B2(Atot, Btot)
      actB = beta * (B2**n) / (1.0 + B2**n)
      return alpha + actB - Btot

  # ===== Step 1: Define the grid =====
  start_x, end_x, num_points_x = 0, 50, 1000
  start_y, end_y, num_points_y = 0, 50, 1000
  x, y = np.meshgrid(np.linspace(start_x, end_x, num_points_x),
                     np.linspace(start_y, end_y, num_points_y))

  # ===== Step 2: Compute vector components =====
  u = dAt_dt(x, y)  # x-component
  v = dBt_dt(x, y)  # y-component

  # ===== Step 3: Compute magnitude for coloring =====
  magnitude = np.sqrt(u**2 + v**2)

  # ===== Step 4: Create the streamline plot =====
  plt.figure(figsize=(8, 6))
  plt.streamplot(x, y, u, v, color=magnitude, cmap='plasma',
                 linewidth=1.5, density=2.5, arrowstyle='->')
  plt.colorbar(label="Vector Magnitude")
  plt.contour(x, y, u, levels=[0], colors='red', linewidths=2, linestyles='--')
  plt.contour(x, y, v, levels=[0], colors='black', linewidths=2, linestyles='--')

  # ===== Step 5: Labels and styling =====
  plt.title("Phase Portrait with Magnitude Coloring")
  plt.xlabel(r"$[A_{tot}]$")
  plt.ylabel(r"$[B_{tot}]$")
  plt.axis("scaled")
  plt.grid(True)

  plt.show()

# ===== Parameters =====
interact_manual(plot_stream,
          alpha = FloatSlider(min=0.1, max=3.0, step=0.1, value=1.0),
          beta  = FloatSlider(min=0.5, max=15, step=0.5, value=1.0),
          n = FloatSlider(min=0.5, max=5.0, step=0.5, value=2.0),
          Kd = FloatSlider(min=0.1, max=3.0, step=0.1, value=1.0))
```

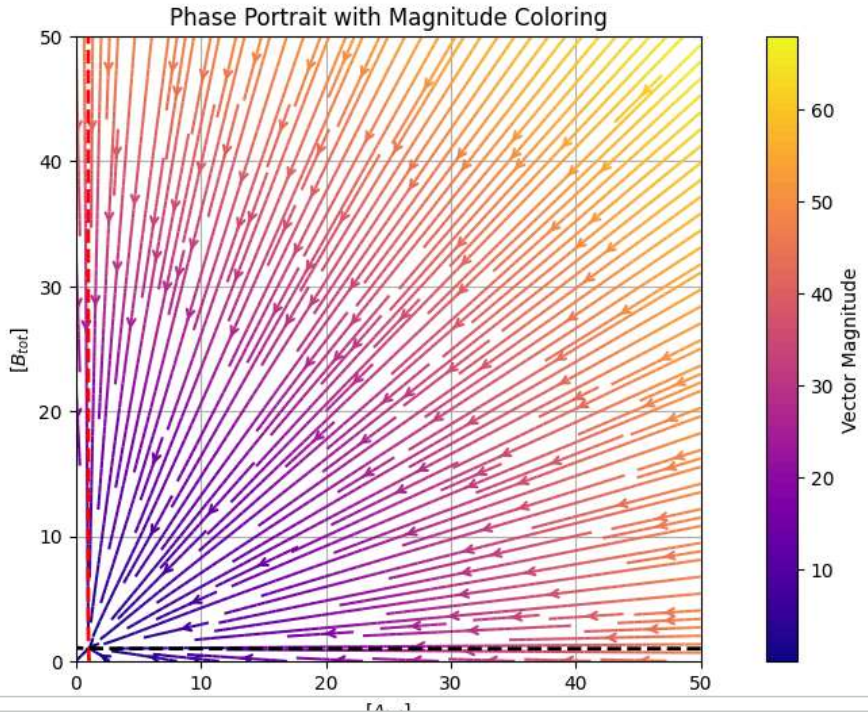
alpha 1.00

beta 1.00

n 2.00

Kd 1.00

Run Interact




```
import sympy as sp

# ===== Symbols =====
A, B = sp.symbols('A B', real=True, nonnegative=True)
alpha, beta, n, Kd = sp.symbols('alpha beta n Kd', positive=True)

# ===== Define A2, B2 symbolically =====
S = A + B
denom_common = Kd + 4*S + sp.sqrt(Kd**2 + 8*S*Kd)
A2 = 2*A**2 / denom_common
B2 = 2*B**2 / denom_common

# ===== Dynamics =====
dA = alpha + beta * (A2**n) / (1 + A2**n) - A
dB = alpha + beta * (B2**n) / (1 + B2**n) - B

# ===== Numerical solve =====
# Substitute parameter values
params = {alpha: 1.0, beta: 5.0, n: 2.0, Kd: 1.0}

# Try solving with an initial guess
sol = sp.nsolve([dA.subs(params), dB.subs(params)], (A, B), (5, 5))
print("Fixed point:", sol)
```

```
Fixed point: Matrix([[1.16312977452925], [1.16312977452925]])
```

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact_manual, FloatSlider
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

# ===== Numba acceleration =====
from numba import njit, prange

@njit(fastmath=True)
```

```
def _compute_A2_B2_elem(Atot, Btot, Kd):
    S = Atot + Btot
    # Denominator shared by both A2 and B2
    denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
    A2 = 2.0 * Atot * Atot / denom_common
    B2 = 2.0 * Btot * Btot / denom_common
    return A2, B2

@njit(parallel=True, fastmath=True)
def _compute_field(x, y, alpha, beta, n, Kd):
    nx, ny = x.shape
    u = np.empty((nx, ny), dtype=np.float64)
    v = np.empty((nx, ny), dtype=np.float64)
    for i in prange(nx):
        for j in range(ny):
            At = x[i, j]
            Bt = y[i, j]
            A2, B2 = _compute_A2_B2_elem(At, Bt, Kd)

            # actA = beta * (A2**n) / (1 + A2**n)
            tA = A2 ** n
            actA = beta * (tA / (1.0 + tA))
            u[i, j] = alpha + actA - At

            # actB = beta * (B2**n) / (1 + B2**n)
            tB = B2 ** n
            actB = beta * (tB / (1.0 + tB))
            v[i, j] = alpha + actB - Bt
    return u, v

@njit(parallel=True, fastmath=True)
def _magnitude(u, v):
    nx, ny = u.shape
    mag = np.empty((nx, ny), dtype=np.float64)
    for i in prange(nx):
        for j in range(ny):
            # hypot for numerical stability
            mag[i, j] = np.hypot(u[i, j], v[i, j])
    return mag

# ===== Quasi-steady dimer functions (Numba-backed) =====
def plot_stream(alpha, beta, n, Kd):
    # ----- Define the grid -----
    start_x, end_x, num_points_x = 0.0, 30.0, 1000
    start_y, end_y, num_points_y = 0.0, 30.0, 1000

    # Note: meshgrid returns (ny, nx) shaped arrays with default indexing='xy'
    x_vals = np.linspace(start_x, end_x, num_points_x, dtype=np.float64)
    y_vals = np.linspace(start_y, end_y, num_points_y, dtype=np.float64)
    x, y = np.meshgrid(x_vals, y_vals)  # shape = (ny, nx)

    # ----- Compute vector field with Numba -----
    u, v = _compute_field(x, y, float(alpha), float(beta), float(n), float(Kd))

    # ----- Magnitude for coloring (Numba) -----
    magnitude = _magnitude(u, v)

    # ----- Plot -----
    plt.figure(figsize=(8, 6))
    plt.streamplot(x, y, u, v, color=magnitude, cmap='plasma',
                   linewidth=1.5, density=2.5, arrowstyle='->')
    plt.colorbar(label="Vector Magnitude")

    # Nullclines: u=0 (red), v=0 (black)
    plt.contour(x, y, u, levels=[0], colors='red', linewidths=2, linestyles='--')
    plt.contour(x, y, v, levels=[0], colors='black', linewidths=2, linestyles='--')

    plt.title("Phase Portrait with Magnitude Coloring")
    plt.xlabel(r"$[A_{tot}]$")
    plt.ylabel(r"$[B_{tot}]$")
    plt.axis("scaled")
    plt.grid(True)
    plt.show()

# ===== Parameters =====
interact_manual(
```

```
    plot_stream,
    alpha = FloatSlider(min=0.1, max=3.0, step=0.1, value=1.0, description='alpha'),
    beta  = FloatSlider(min=0.5, max=20,  step=0.5, value=1.0, description='beta'),
    n     = FloatSlider(min=0.5, max=5.0, step=0.5, value=2.0, description='n'),
    Kd    = FloatSlider(min=0.1, max=3.0, step=0.1, value=1.0, description='Kd')
)
```

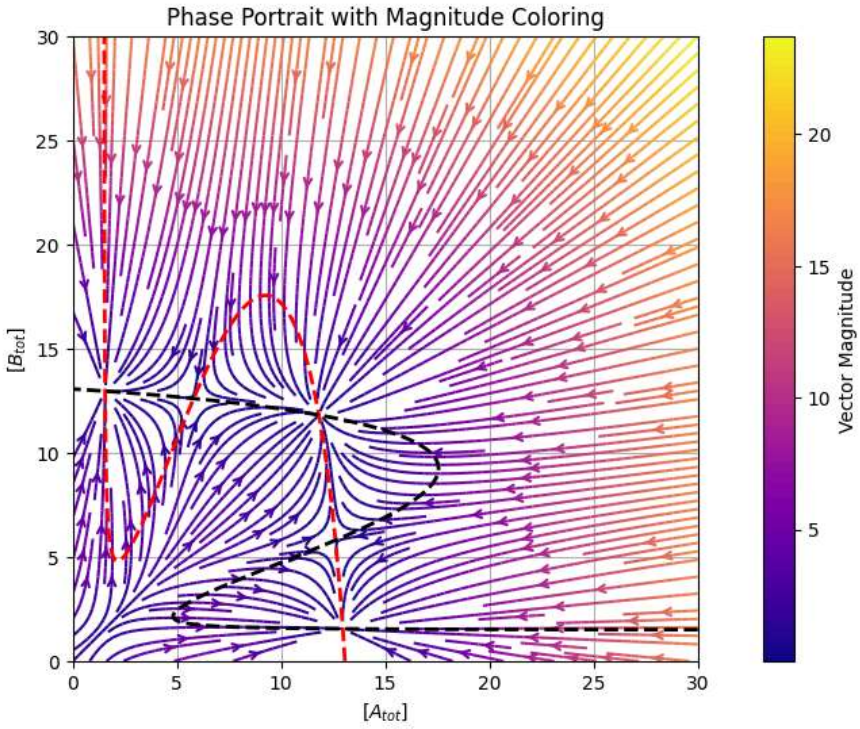
alpha 1.50

beta 12.00

n 2.00

Kd 1.50

Run Interact



plot_stream
def plot_stream(alpha, beta, n, Kd)

<no docstring>

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact_manual, FloatSlider
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

# ===== Numba acceleration =====
from numba import njit, prange

@njit(parallel=True, fastmath=True)
def _compute_field(x, y, alpha_A, alpha_B, gamma_A, gamma_B):
    nx, ny = x.shape
    u = np.empty((nx, ny), dtype=np.float64)  # dA/dt
    v = np.empty((nx, ny), dtype=np.float64)  # dB/dt
    for i in prange(nx):
        for j in range(ny):
            A = x[i, j]
            B = y[i, j]
            # Toggle switch ODEs
            dA = alpha_A / (1.0 + B*B) - gamma_A * A
            dB = alpha_B / (1.0 + A*A) - gamma_B * B
            u[i, j] = dA
            v[i, j] = dB
    return u, v
```

```
@njit(parallel=True, fastmath=True)
def _magnitude(u, v):
    nx, ny = u.shape
    mag = np.empty((nx, ny), dtype=np.float64)
    for i in prange(nx):
        for j in range(ny):
            mag[i, j] = np.hypot(u[i, j], v[i, j])
    return mag

# ===== Toggle switch field plot (Numba-backed) =====
def plot_stream(alpha_A, alpha_B, gamma_A, gamma_B):
    # ----- Define the grid -----
    start_x, end_x, num_points_x = 0.0, 5.0, 500
    start_y, end_y, num_points_y = 0.0, 5.0, 500

    x_vals = np.linspace(start_x, end_x, num_points_x, dtype=np.float64)
    y_vals = np.linspace(start_y, end_y, num_points_y, dtype=np.float64)
    x, y = np.meshgrid(x_vals, y_vals)

    # ----- Compute vector field -----
    u, v = _compute_field(x, y, float(alpha_A), float(alpha_B), float(gamma_A), float(gamma_B))

    # ----- Magnitude for coloring -----
    magnitude = _magnitude(u, v)

    # ----- Plot -----
    plt.figure(figsize=(8, 6))
    plt.streamplot(x, y, u, v, color=magnitude, cmap='plasma',
                   linewidth=1.2, density=2.0, arrowstyle='->')
    plt.colorbar(label="Vector Magnitude")

    # Nullclines: dA/dt=0 (red), dB/dt=0 (black)
    plt.contour(x, y, u, levels=[0], colors='red', linewidths=2, linestyles='--')
    plt.contour(x, y, v, levels=[0], colors='black', linewidths=2, linestyles='--')

    plt.title("Phase Portrait of Toggle Switch System")
    plt.xlabel("A")
    plt.ylabel("B")
    plt.axis("scaled")
    plt.grid(True)
    plt.show()

# ===== Interactive sliders =====
interact_manual(
    plot_stream,
    alpha_A = FloatSlider(min=0.1, max=5.0, step=0.1, value=2.5, description='alpha_A'),
    alpha_B = FloatSlider(min=0.1, max=5.0, step=0.1, value=1.5, description='alpha_B'),
    gamma_A = FloatSlider(min=0.1, max=3.0, step=0.1, value=1.0, description='gamma_A'),
    gamma_B = FloatSlider(min=0.1, max=3.0, step=0.1, value=1.0, description='gamma_B')
)
```

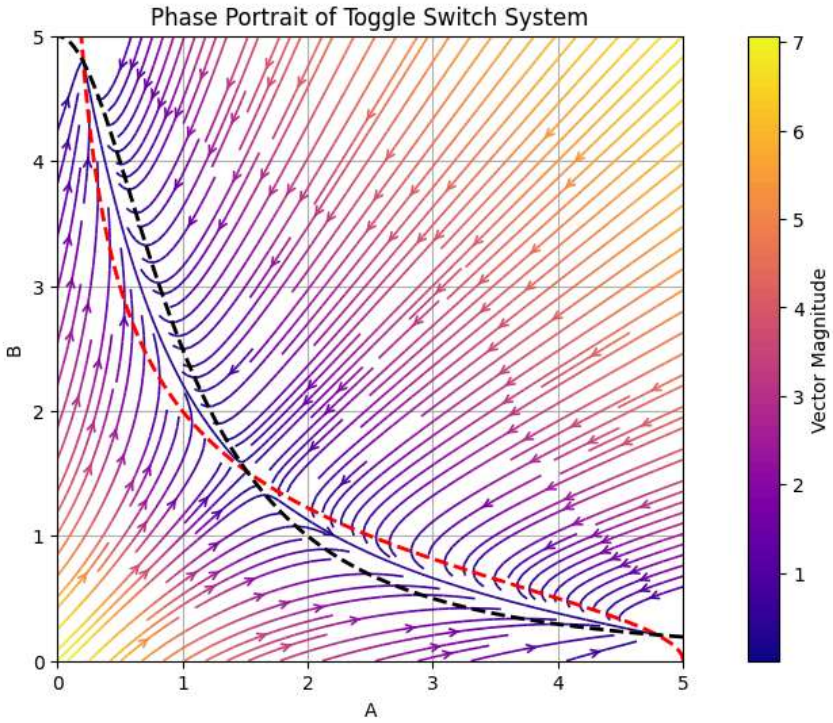
alpha_A 5.00

alpha_B 5.00

gamma_A 1.00

gamma_B 1.00

Run Interact



plot_stream
def plot_stream(alpha_A, alpha_B, gamma_A, gamma_B)

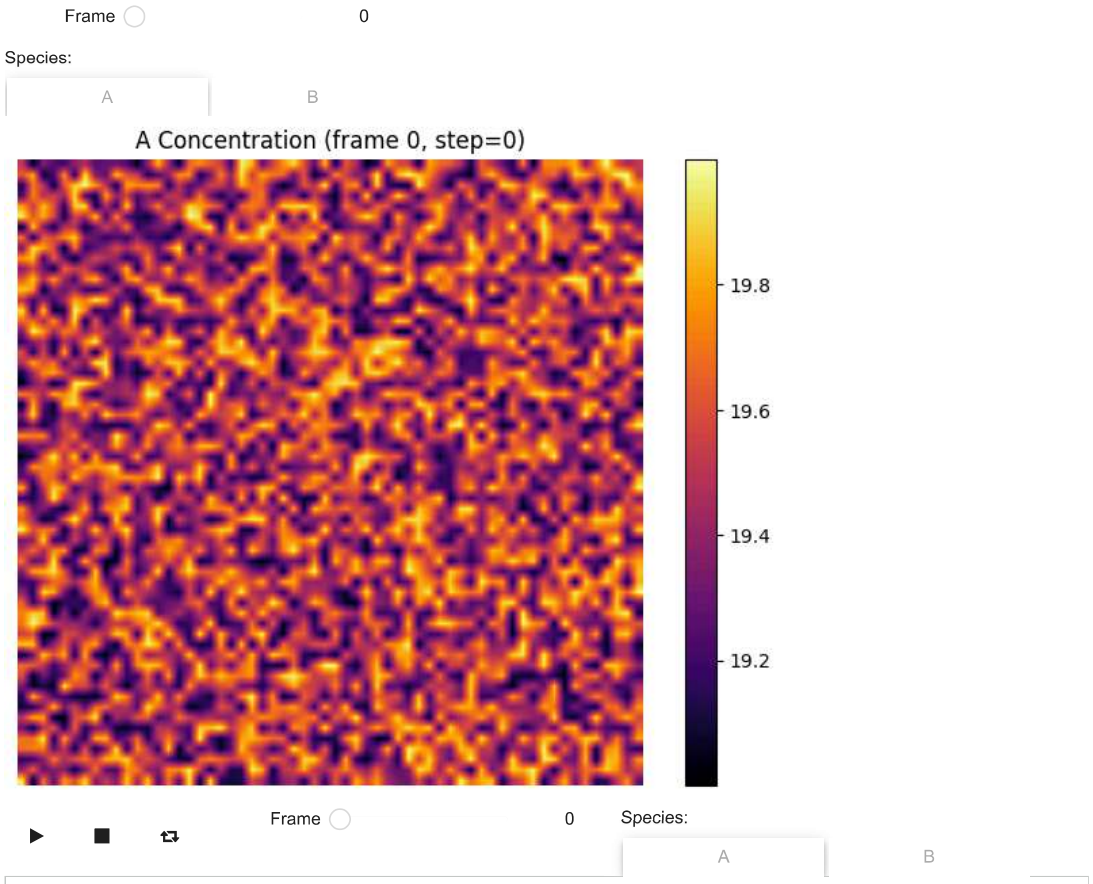
<no docstring>

Initial Conditions

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Gray-Scott-style UI) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

def Square_Initial():
  # -------------------- Grid & Initial Condition --------------------
  N = 60

  # Initialize desired fields
  A0 = np.random.uniform(19.0, 20.0, (N, N))
  B0 = np.random.uniform(19.0, 20.0, (N, N))

  # -------------------- Parameters --------------------
  Du, Dv = 0.10, 0.05
  alpha, beta, n, Kd = 0.4, 10.0, 1.5, 1.0
  dt = 0.2
  steps = 10000
  save_every = 1   # store frame every 100 steps

  # -------------------- Numba kernels --------------------
  from numba import njit, prange

  @njit(fastmath=True)
  def _compute_A2_B2_elem(Atot, Btot, Kd):
```

```
    S = Atot + Btot
    denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
    A2 = 2.0 * Atot * Atot / denom_common
    B2 = 2.0 * Btot * Btot / denom_common
    return A2, B2

@njit(parallel=True, fastmath=True)
def _step_periodic(A, B, Du, Dv, alpha, beta, n, Kd, dt):
    nx, ny = A.shape
    Anew = np.empty_like(A)
    Bnew = np.empty_like(B)

    for i in prange(nx):
        im = (i - 1) % nx
        ip = (i + 1) % nx
        for j in range(ny):
            jm = (j - 1) % ny
            jp = (j + 1) % ny

            LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
            LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

            A2, B2 = _compute_A2_B2_elem(A[i, j], B[i, j], Kd)

            tA = A2 ** n
            actA = beta * (tA / (1.0 + tA))
            fA = alpha + actA - A[i, j]

            tB = B2 ** n
            actB = beta * (tB / (1.0 + tB))
            fB = alpha + actB - B[i, j]

            Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
            Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

    # Clamp negatives only
    for i in prange(nx):
        for j in range(ny):
            if Anew[i, j] < 0.0:
                Anew[i, j] = 0.0
            if Bnew[i, j] < 0.0:
                Bnew[i, j] = 0.0

    return Anew, Bnew

# -------------------- Simulation --------------------
A_temp, B_temp = A0.copy(), B0.copy()
A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)  # JIT warmup

A_list = [A0.copy()]
B_list = [B0.copy()]
for s in range(1, steps + 1):
    A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)
    if s % save_every == 0:
        A_list.append(A_temp.copy())
        B_list.append(B_temp.copy())

# -------------------- Viewer --------------------
from ipywidgets import ToggleButtons

def plot_at_time(idx, species):
    plt.figure(figsize=(8, 6))

    if species == "A":
        data = A_list[idx]
        title = f"A Concentration (frame {idx}, step={idx*save_every})"
    else:
        data = B_list[idx]
        title = f"B Concentration (frame {idx}, step={idx*save_every})"

    img = plt.imshow(data, cmap='inferno', interpolation='bilinear')
    plt.colorbar(img, ax=plt.gca())
    plt.title(title)
    plt.axis('off')
    plt.show()
```

```
  # Species toggle
  species_toggle = ToggleButtons(
      options=["A", "B"],
      value="A",
      description="Species:",
      button_style="info"
  )

  # Frame controls
  play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100, description="Play")
  slider = IntSlider(value=0, min=0, max=len(A_list) - 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)

  from ipywidgets import jslink
  jslink((play, 'value'), (slider, 'value'))

  display(widgets, interact_ui)

Square_Initial()
```



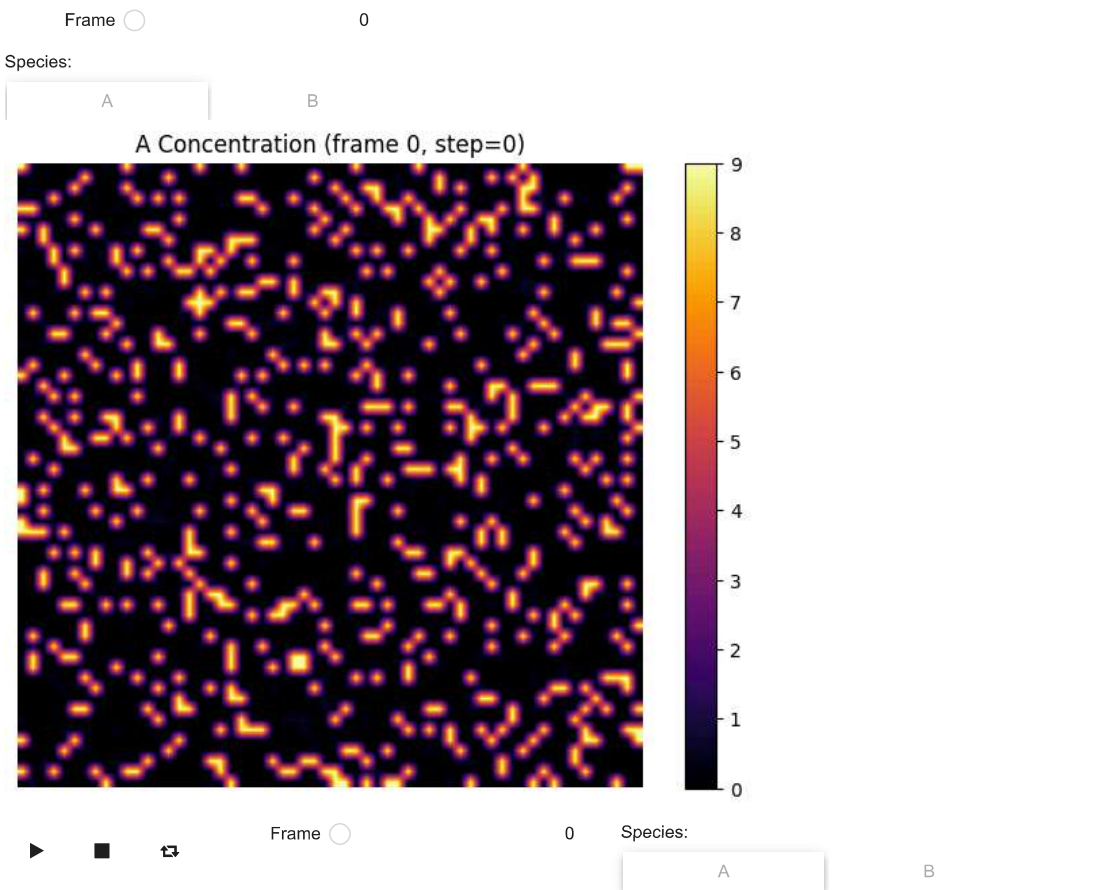
Square_Initial.<locals>.plot_at_time
def plot_at_time(idx, species)

<no docstring>

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Gray-Scott-style UI) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

def Random_Initial():
  # -------------------- Grid & Initial Condition --------------------
  N = 60
```

```
# Initialize desired fields
# ---- Grid setup ----
nx, ny = 60, 60
A0 = np.zeros((nx, ny))
B0 = np.zeros((nx, ny))

num_total = 900
num_set = 300
total_points = nx * ny

# ---- Species A ----
all_indices_A = np.random.choice(total_points, size=num_total, replace=False)
set1_A = all_indices_A[:num_set]
set2_A = all_indices_A[num_set:2*num_set]
set3_A = all_indices_A[2*num_set:3*num_set]

coords1_A = np.array(np.unravel_index(set1_A, (nx, ny))).T
coords2_A = np.array(np.unravel_index(set2_A, (nx, ny))).T
coords3_A = np.array(np.unravel_index(set3_A, (nx, ny))).T

A0[coords1_A[:,0], coords1_A[:,1]] = 9.0
A0[coords2_A[:,0], coords2_A[:,1]] = 0.5
A0[coords3_A[:,0], coords3_A[:,1]] = 9.0

# ---- Species B ----
all_indices_B = np.random.choice(total_points, size=num_total, replace=False)
set1_B = all_indices_B[:num_set]
set2_B = all_indices_B[num_set:2*num_set]
set3_B = all_indices_B[2*num_set:3*num_set]

coords1_B = np.array(np.unravel_index(set1_B, (nx, ny))).T
coords2_B = np.array(np.unravel_index(set2_B, (nx, ny))).T
coords3_B = np.array(np.unravel_index(set3_B, (nx, ny))).T

B0[coords1_B[:,0], coords1_B[:,1]] = 9.0
B0[coords2_B[:,0], coords2_B[:,1]] = 9.0
B0[coords3_B[:,0], coords3_B[:,1]] = 0.5

# -------------------- Parameters --------------------
Du, Dv = 0.10, 0.05
alpha, beta, n, Kd = 0.4, 10.0, 1.5, 1.0
dt = 0.2
steps = 10000
save_every = 1   # store frame every 100 steps

# -------------------- Numba kernels --------------------
from numba import njit, prange

@njit(fastmath=True)
def _compute_A2_B2_elem(Atot, Btot, Kd):
    S = Atot + Btot
    denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
    A2 = 2.0 * Atot * Atot / denom_common
    B2 = 2.0 * Btot * Btot / denom_common
    return A2, B2

@njit(parallel=True, fastmath=True)
def _step_periodic(A, B, Du, Dv, alpha, beta, n, Kd, dt):
    nx, ny = A.shape
    Anew = np.empty_like(A)
    Bnew = np.empty_like(B)

    for i in prange(nx):
        im = (i - 1) % nx
        ip = (i + 1) % nx
        for j in range(ny):
            jm = (j - 1) % ny
            jp = (j + 1) % ny

            LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
            LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

            A2, B2 = _compute_A2_B2_elem(A[i, j], B[i, j], Kd)

            tA = A2 ** n
            actA = beta * (tA / (1.0 + tA))
```

```
                fA = alpha + actA - A[i, j]

                tB = B2 ** n
                actB = beta * (tB / (1.0 + tB))
                fB = alpha + actB - B[i, j]

                Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
                Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

        # Clamp negatives only
        for i in prange(nx):
            for j in range(ny):
                if Anew[i, j] < 0.0:
                    Anew[i, j] = 0.0
                if Bnew[i, j] < 0.0:
                    Bnew[i, j] = 0.0

        return Anew, Bnew

    # -------------------- Simulation --------------------
    A_temp, B_temp = A0.copy(), B0.copy()
    A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)  # JIT warmup

    A_list = [A0.copy()]
    B_list = [B0.copy()]
    for s in range(1, steps + 1):
        A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)
        if s % save_every == 0:
            A_list.append(A_temp.copy())
            B_list.append(B_temp.copy())

    # -------------------- Viewer --------------------
    from ipywidgets import ToggleButtons

    def plot_at_time(idx, species):
        plt.figure(figsize=(8, 6))

        if species == "A":
            data = A_list[idx]
            title = f"A Concentration (frame {idx}, step={idx*save_every})"
        else:
            data = B_list[idx]
            title = f"B Concentration (frame {idx}, step={idx*save_every})"

        img = plt.imshow(data, cmap='inferno', interpolation='bilinear')
        plt.colorbar(img, ax=plt.gca())
        plt.title(title)
        plt.axis('off')
        plt.show()

    # Species toggle
    species_toggle = ToggleButtons(
        options=["A", "B"],
        value="A",
        description="Species:",
        button_style="info"
    )

    # Frame controls
    play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100, description="Play")
    slider = IntSlider(value=0, min=0, max=len(A_list) - 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)

    from ipywidgets import jslink
    jslink((play, 'value'), (slider, 'value'))

    display(widgets, interact_ui)

Random_Initial()
```



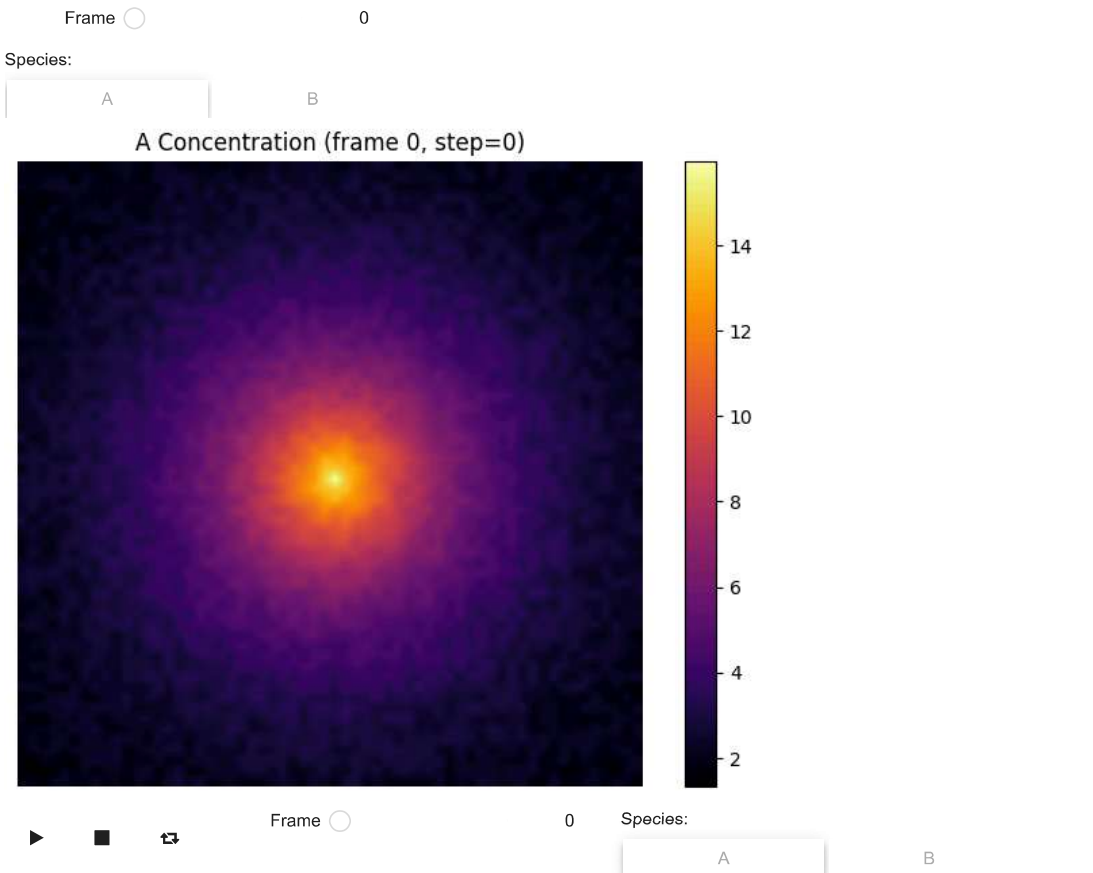
Random_Initial.<locals>.plot_at_time
def plot_at_time(idx, species)

<no docstring>

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Gray-Scott-style UI) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

def Radial_Initial():
  # -------------------- Grid & Initial Condition --------------------
  N = 60

  def radial_gradient_both(N, center=None,
                           A_inner=10.0, A_outer=0.0,
                           B_inner=10.0, B_outer=0.0):
      """Radial gradient initial condition for both A and B."""
      if center is None:
          center = (N//2, N//2)
      ii, jj = np.indices((N, N))
      r = np.sqrt((ii-center[0])**2 + (jj-center[1])**2)
      r = r/r.max()  # normalize radius [0,1]

      # Exponential radial interpolation
      A = A_outer + (A_inner - A_outer) * np.exp(-4 * r)
      B = B_outer + (B_inner - B_outer) * np.exp(-4 * r)
      return A, B

  # Use radial gradient instead of random init
  A, B = radial_gradient_both(N,
                              A_inner=15.0, A_outer=1.0,
                              B_inner=15.0, B_outer=1.0)

  # Adding some noise
  A0 = A + np.random.uniform(0.0, 1.0, (N, N))
  B0 = B + np.random.uniform(0.0, 1.0, (N, N))
```

```
# -------------------- Parameters --------------------
Du, Dv = 0.10, 0.05
alpha, beta, n, Kd = 0.4, 10.0, 1.5, 1.0
dt = 0.2
steps = 10000
save_every = 1   # store frame every step

# -------------------- Numba kernels --------------------
from numba import njit, prange

@njit(fastmath=True)
def _compute_A2_B2_elem(Atot, Btot, Kd):
    S = Atot + Btot
    denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
    A2 = 2.0 * Atot * Atot / denom_common
    B2 = 2.0 * Btot * Btot / denom_common
    return A2, B2

@njit(parallel=True, fastmath=True)
def _step_periodic(A, B, Du, Dv, alpha, beta, n, Kd, dt):
    nx, ny = A.shape
    Anew = np.empty_like(A)
    Bnew = np.empty_like(B)

    for i in prange(nx):
        im = (i - 1) % nx
        ip = (i + 1) % nx
        for j in range(ny):
            jm = (j - 1) % ny
            jp = (j + 1) % ny

            LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
            LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

            A2, B2 = _compute_A2_B2_elem(A[i, j], B[i, j], Kd)

            tA = A2 ** n
            actA = beta * (tA / (1.0 + tA))
            fA = alpha + actA - A[i, j]

            tB = B2 ** n
            actB = beta * (tB / (1.0 + tB))
            fB = alpha + actB - B[i, j]

            Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
            Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

    # Clamp negatives only
    for i in prange(nx):
        for j in range(ny):
            if Anew[i, j] < 0.0:
                Anew[i, j] = 0.0
            if Bnew[i, j] < 0.0:
                Bnew[i, j] = 0.0

    return Anew, Bnew

# -------------------- Simulation --------------------
A_temp, B_temp = A0.copy(), B0.copy()
A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)  # JIT warmup

A_list = [A0.copy()]
B_list = [B0.copy()]
for s in range(1, steps + 1):
    A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)
    if s % save_every == 0:
        A_list.append(A_temp.copy())
        B_list.append(B_temp.copy())

# -------------------- Viewer --------------------
from ipywidgets import ToggleButtons

def plot_at_time(idx, species):
    plt.figure(figsize=(8, 6))
```

```
        if species == "A":
            data = A_list[idx]
            title = f"A Concentration (frame {idx}, step={idx*save_every})"
        else:
            data = B_list[idx]
            title = f"B Concentration (frame {idx}, step={idx*save_every})"

        img = plt.imshow(data, cmap='inferno', interpolation='bilinear')
        plt.colorbar(img, ax=plt.gca())
        plt.title(title)
        plt.axis('off')
        plt.show()

    # Species toggle
    species_toggle = ToggleButtons(
        options=["A", "B"],
        value="A",
        description="Species:",
        button_style="info"
    )

    # Frame controls
    play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100, description="Play")
    slider = IntSlider(value=0, min=0, max=len(A_list) - 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)

    from ipywidgets import jslink
    jslink((play, 'value'), (slider, 'value'))

    display(widgets, interact_ui)

Radial_Initial()
```

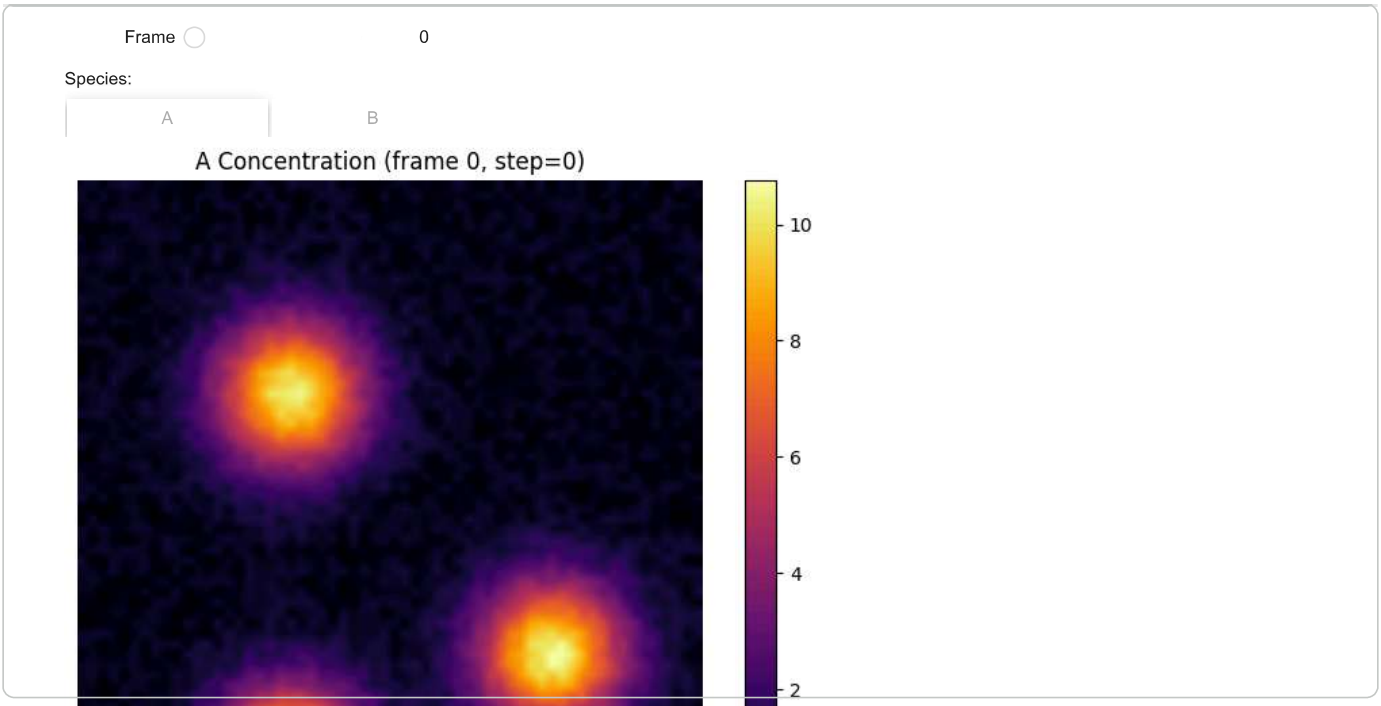


Radial_Initial.<locals>.plot_at_time
def plot_at_time(idx, species)

<no docstring>

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Gray-Scott-style UI) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display

def MultiGaussian_Initial():
  # -------------------- Grid & Initial Condition --------------------
  N = 60

  def multi_gaussian(N, centers, A_inner=10, B_inner=10):
    A = np.zeros((N, N))
    B = np.zeros((N, N))
    for cx, cy in centers:
        rr, cc = np.indices((N, N))
        dist2 = (rr-cx)**2 + (cc-cy)**2
        A += A_inner * np.exp(-dist2/(2*5.0**2))
        B += B_inner * np.exp(-dist2/(2*5.0**2))
    return A, B

  A, B = multi_gaussian(N, [(20, 20), (45, 45), (55,20)])

  # Adding some noise
  A0 = A + np.random.uniform(0.0, 1.0, (N, N))
  B0 = B + np.random.uniform(0.0, 1.0, (N, N))

  # -------------------- Parameters --------------------
  Du, Dv = 0.10, 0.05
  alpha, beta, n, Kd = 0.4, 10.0, 1.5, 1.0
  dt = 0.2
  steps = 10000
  save_every = 1   # store frame every step

  # -------------------- Numba kernels --------------------
  from numba import njit, prange

  @njit(fastmath=True)
  def _compute_A2_B2_elem(Atot, Btot, Kd):
      S = Atot + Btot
      denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
      A2 = 2.0 * Atot * Atot / denom_common
      B2 = 2.0 * Btot * Btot / denom_common
      return A2, B2

  @njit(parallel=True, fastmath=True)
  def _step_periodic(A, B, Du, Dv, alpha, beta, n, Kd, dt):
      nx, ny = A.shape
      Anew = np.empty_like(A)
      Bnew = np.empty_like(B)

      for i in prange(nx):
          im = (i - 1) % nx
          ip = (i + 1) % nx
          for j in range(ny):
              jm = (j - 1) % ny
              jp = (j + 1) % ny

              LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
              LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

              A2, B2 = _compute_A2_B2_elem(A[i, j], B[i, j], Kd)

              tA = A2 ** n
              actA = beta * (tA / (1.0 + tA))
              fA = alpha + actA - A[i, j]

              tB = B2 ** n
              actB = beta * (tB / (1.0 + tB))
              fB = alpha + actB - B[i, j]

              Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
              Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

      # Clamp negatives only
```

```
        # Clamp negatives only
        for i in prange(nx):
            for j in range(ny):
                if Anew[i, j] < 0.0:
                    Anew[i, j] = 0.0
                if Bnew[i, j] < 0.0:
                    Bnew[i, j] = 0.0

        return Anew, Bnew

    # -------------------- Simulation --------------------
    A_temp, B_temp = A0.copy(), B0.copy()
    A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)  # JIT warmup

    A_list = [A0.copy()]
    B_list = [B0.copy()]
    for s in range(1, steps + 1):
        A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)
        if s % save_every == 0:
            A_list.append(A_temp.copy())
            B_list.append(B_temp.copy())

    # -------------------- Viewer --------------------
    from ipywidgets import ToggleButtons

    def plot_at_time(idx, species):
        plt.figure(figsize=(8, 6))

        if species == "A":
            data = A_list[idx]
            title = f"A Concentration (frame {idx}, step={idx*save_every})"
        else:
            data = B_list[idx]
            title = f"B Concentration (frame {idx}, step={idx*save_every})"

        img = plt.imshow(data, cmap='inferno', interpolation='bilinear')
        plt.colorbar(img, ax=plt.gca())
        plt.title(title)
        plt.axis('off')
        plt.show()

    # Species toggle
    species_toggle = ToggleButtons(
        options=["A", "B"],
        value="A",
        description="Species:",
        button_style="info"
    )

    # Frame controls
    play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100, description="Play")
    slider = IntSlider(value=0, min=0, max=len(A_list) - 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)

    from ipywidgets import jslink
    jslink((play, 'value'), (slider, 'value'))

    display(widgets, interact_ui)

MultiGaussian_Initial()
```



MultiFate 2

PBC

Frame 0 Species:

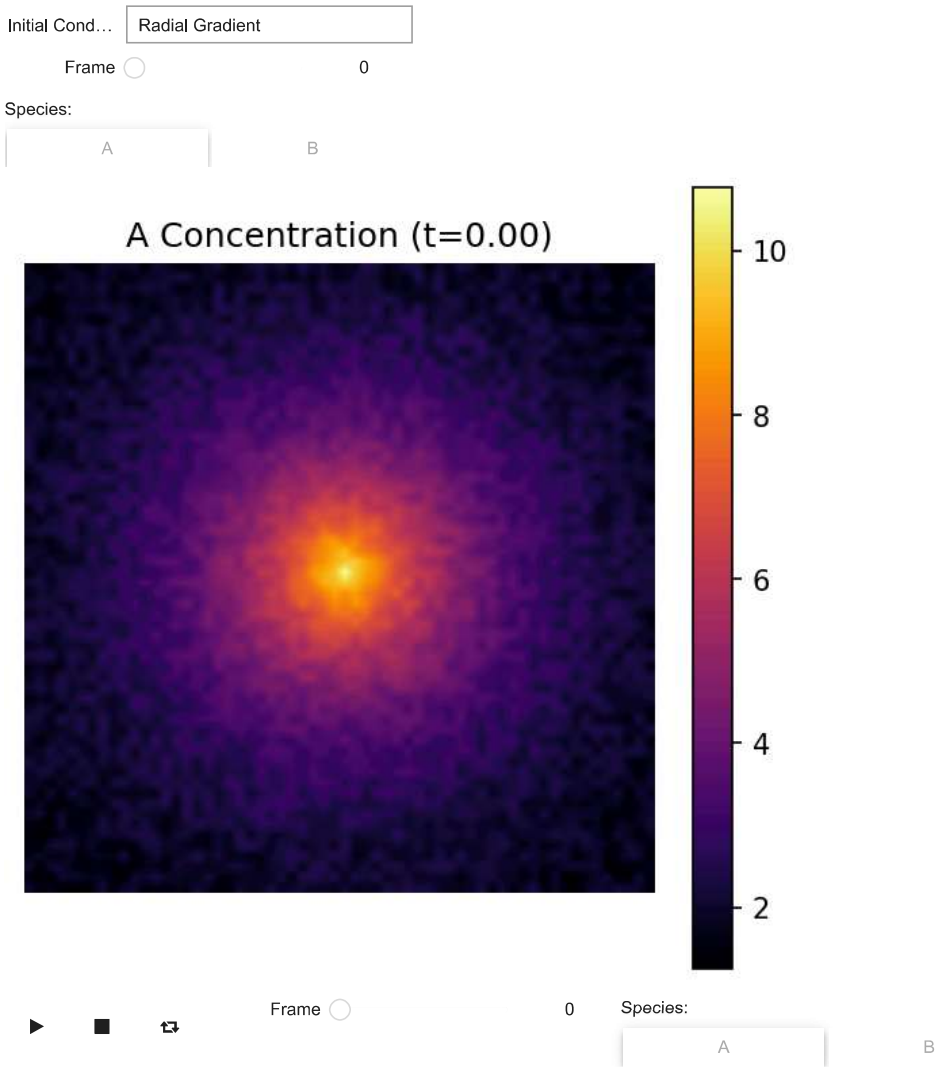
A B

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Gray-Scott-style UI) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox, Dropdown, ToggleButtons, jslink
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display
from numba import njit, prange

def Initial():
    # -------------------- Grid --------------------
    N = 60

    # -------------------- Initial Conditions --------------------
    def square(N):
        """Uniform random high values for both species."""
        A0 = np.random.uniform(19.0, 20.0, (N, N))
        B0 = np.random.uniform(19.0, 20.0, (N, N))

        return A0, B0

    def random_300(N):
        """Place three sets (300 each) of special values for A and B at random coordinates."""
        nx, ny = N, N
        A0 = np.zeros((nx, ny))
        B0 = np.zeros((nx, ny))

        num_total = 900
        num_set = 300
        total_points = nx * ny

        # Species A
        all_indices_A = np.random.choice(total_points, size=num_total, replace=False)
        for k, val in enumerate([9.0, 0.5, 9.0]):
            part = all_indices_A[k * num_set:(k + 1) * num_set]
            coords = np.array(np.unravel_index(part, (nx, ny))).T
            A0[coords[:, 0], coords[:, 1]] = val

        # Species B
        all_indices_B = np.random.choice(total_points, size=num_total, replace=False)
        for k, val in enumerate([9.0, 9.0, 0.5]):
```

```
            part = all_indices_B[k * num_set:(k + 1) * num_set]
            coords = np.array(np.unravel_index(part, (nx, ny))).T
            B0[coords[:, 0], coords[:, 1]] = val

        return A0, B0

    def radial_gradient_both(N, center=None,
                             A_inner=10.0, A_outer=1.0,
                             B_inner=10.0, B_outer=1.0):
        if center is None:
            center = (N // 2, N // 2)
        ii, jj = np.indices((N, N))
        r = np.sqrt((ii - center[0]) ** 2 + (jj - center[1]) ** 2)
        r = r / r.max()

        A = A_outer + (A_inner - A_outer) * np.exp(-4 * r)
        B = B_outer + (B_inner - B_outer) * np.exp(-4 * r)

        A0 = A + np.random.uniform(0.0, 1.0, (N, N))
        B0 = B + np.random.uniform(0.0, 1.0, (N, N))

        return A0, B0

    def multi_gaussian(N, centers=[(30, 30), (15, 45), (45, 15)], A_inner=10, B_inner=10):
        A = np.zeros((N, N))
        B = np.zeros((N, N))
        rr, cc = np.indices((N, N))
        for cx, cy in centers:
            dist2 = (rr - cx) ** 2 + (cc - cy) ** 2
            A += A_inner * np.exp(-dist2 / (2 * 5.0 ** 2))
            B += B_inner * np.exp(-dist2 / (2 * 5.0 ** 2))

        A0 = A + np.random.uniform(0.0, 1.0, (N, N))
        B0 = B + np.random.uniform(0.0, 1.0, (N, N))

        return A0, B0

    def sinusoidal_initial(N=60, noise_strength=0.1):
        """
        Sinusoidal checkerboard initial condition with small noise.
        Returns A0, B0 fields on an N×N grid.
        """
        # Grid in [0, 2π]
        x = np.linspace(0, 2*np.pi, N)
        y = np.linspace(0, 2*np.pi, N)
        X, Y = np.meshgrid(x, y)

        # Sinusoidal pattern
        pattern = np.sin(X) * np.sin(Y)

        # Scale into ~[19,20] for consistency with your code
        A = 19.0 + (pattern + 1) / 2
        B = 19.0 + (pattern + 1) / 2

        # Add small uniform noise
        A0 = A + noise_strength * (np.random.rand(N, N) - 0.5)
        B0 = B + noise_strength * (np.random.rand(N, N) - 0.5)

        return A0, B0

    # Dictionary to access IC functions
    ic_options = {
        "Radial Gradient": radial_gradient_both,
        "Random 300": random_300,
        "Multi Gaussian Peaks": multi_gaussian,
        "Square": square,
        "Sinusoidal": sinusoidal_initial
    }

    # -------------------- Parameters --------------------
    Du, Dv = 0.10, 0.05
    alpha, beta, n, Kd = 0.40, 10.0, 1.5, 1.0
    dt = 0.2
    steps = 10000  # reduce for faster demo
    save_every = 1
```

```
# -------------------- Numba kernels --------------------
@njit(fastmath=True)
def _compute_A2_B2_elem(Atot, Btot, Kd):
    S = Atot + Btot
    denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
    A2 = 2.0 * Atot * Atot / denom_common
    B2 = 2.0 * Btot * Btot / denom_common
    return A2, B2

@njit(parallel=True, fastmath=True)
def _step_periodic(A, B, Du, Dv, alpha, beta, n, Kd, dt):
    nx, ny = A.shape
    Anew = np.empty_like(A)
    Bnew = np.empty_like(B)
    for i in prange(nx):
        im = (i - 1) % nx
        ip = (i + 1) % nx
        for j in range(ny):
            jm = (j - 1) % ny
            jp = (j + 1) % ny

            LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
            LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

            A2, B2 = _compute_A2_B2_elem(A[i, j], B[i, j], Kd)

            tA = A2 ** n
            actA = beta * (tA / (1.0 + tA))
            fA = alpha + actA - A[i, j]

            tB = B2 ** n
            actB = beta * (tB / (1.0 + tB))
            fB = alpha + actB - B[i, j]

            Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
            Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

    for i in prange(nx):
        for j in range(ny):
            if Anew[i, j] < 0.0:
                Anew[i, j] = 0.0
            if Bnew[i, j] < 0.0:
                Bnew[i, j] = 0.0

    return Anew, Bnew

# -------------------- Simulation Runner --------------------
def run_sim(initial_condition):
    # Get initial condition by passing N to the selected IC function
    A0, B0 = ic_options[initial_condition](N)
    A_temp, B_temp = A0.copy(), B0.copy()
    A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)  # warmup

    # Run simulation
    A_list = [A0.copy()]
    B_list = [B0.copy()]
    for s in range(1, steps + 1):
        A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)
        if s % save_every == 0:
            A_list.append(A_temp.copy())
            B_list.append(B_temp.copy())

    # Plotting function
    def plot_at_time(idx, species):
        plt.figure(figsize=(5, 5), dpi=150)
        if species == "A":
            data = A_list[idx]
            title = f"A Concentration (t={idx * save_every * dt:.2f})"
        else:
            data = B_list[idx]
            title = f"B Concentration (t={idx * save_every * dt:.2f})"
        img = plt.imshow(data, cmap='inferno', origin='lower', interpolation='bilinear')
        plt.colorbar(img)
        plt.title(title)
        plt.axis('off')
        plt.show()
```

```
        # Controls
        species_toggle = ToggleButtons(options=["A", "B"], value="A", description="Species:", button_style="info")
        play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100)
        slider = IntSlider(value=0, min=0, max=len(A_list) - 1, step=1, description="Frame")
        jslink((play, 'value'), (slider, 'value'))

        widgets = HBox([play, slider, species_toggle])
        interact_ui = interact(plot_at_time, idx=slider, species=species_toggle)
        display(widgets, interact_ui)

    # -------------------- Dropdown for IC --------------------
    ic_dropdown = Dropdown(options=list(ic_options.keys()), value="Radial Gradient", description="Initial Condition:")
    interact(run_sim, initial_condition=ic_dropdown)

Initial()
```



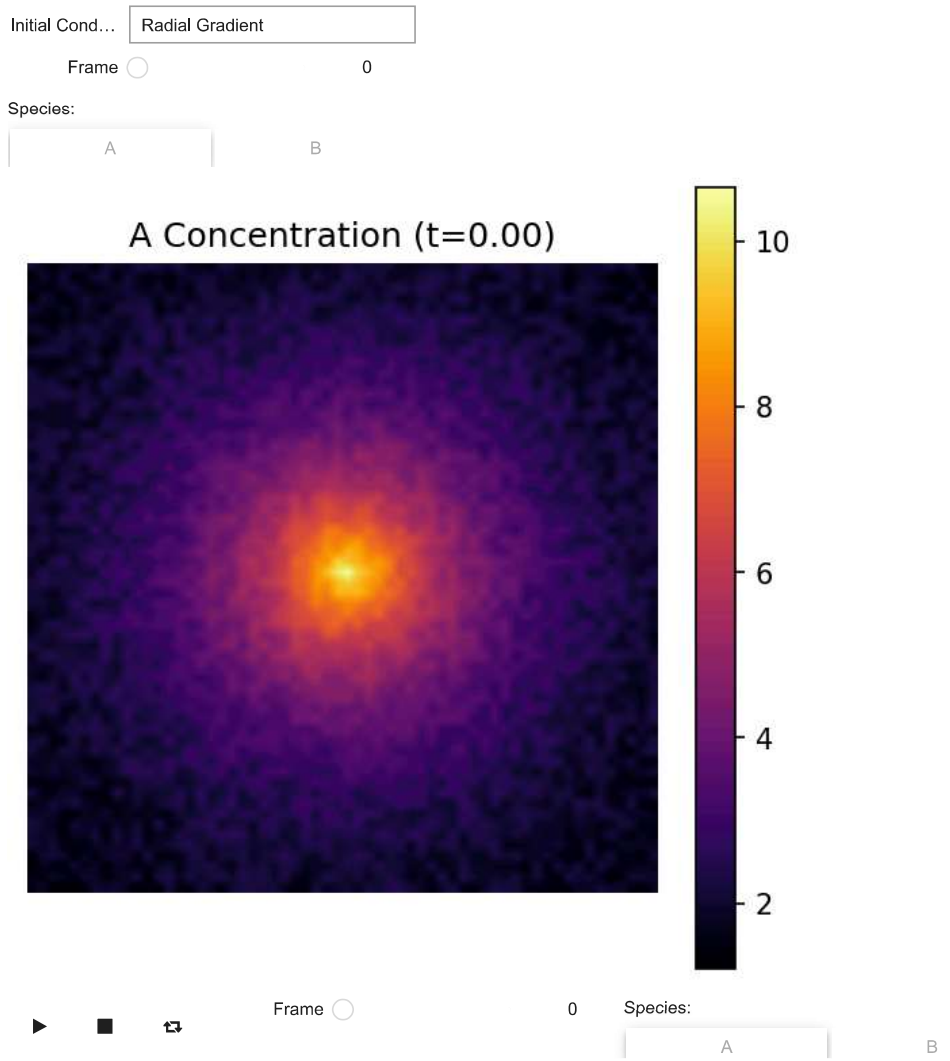
NBC

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Gray-Scott-style UI) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox, Dropdown, ToggleButtons, jslink
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display
from numba import njit, prange

def Initial_N():
    # -------------------- Grid --------------------
    N = 60
```

```
# -------------------- Initial Conditions --------------------
def square(N):
    """Uniform random high values for both species."""
    A0 = np.random.uniform(19.0, 20.0, (N, N))
    B0 = np.random.uniform(19.0, 20.0, (N, N))

    return A0, B0

def random_300(N):
    """Place three sets (300 each) of special values for A and B at random coordinates."""
    nx, ny = N, N
    A0 = np.zeros((nx, ny))
    B0 = np.zeros((nx, ny))

    num_total = 900
    num_set = 300
    total_points = nx * ny

    # Species A
    all_indices_A = np.random.choice(total_points, size=num_total, replace=False)
    for k, val in enumerate([9.0, 0.5, 9.0]):
        part = all_indices_A[k * num_set:(k + 1) * num_set]
        coords = np.array(np.unravel_index(part, (nx, ny))).T
        A0[coords[:, 0], coords[:, 1]] = val

    # Species B
    all_indices_B = np.random.choice(total_points, size=num_total, replace=False)
    for k, val in enumerate([9.0, 9.0, 0.5]):
        part = all_indices_B[k * num_set:(k + 1) * num_set]
        coords = np.array(np.unravel_index(part, (nx, ny))).T
        B0[coords[:, 0], coords[:, 1]] = val

    return A0, B0

def radial_gradient_both(N, center=None,
                         A_inner=10.0, A_outer=1.0,
                         B_inner=10.0, B_outer=1.0):
    if center is None:
        center = (N // 2, N // 2)
    ii, jj = np.indices((N, N))
    r = np.sqrt((ii - center[0]) ** 2 + (jj - center[1]) ** 2)
    r = r / r.max()

    A = A_outer + (A_inner - A_outer) * np.exp(-4 * r)
    B = B_outer + (B_inner - B_outer) * np.exp(-4 * r)

    A0 = A + np.random.uniform(0.0, 1.0, (N, N))
    B0 = B + np.random.uniform(0.0, 1.0, (N, N))

    return A0, B0

def multi_gaussian(N, centers=[(30, 30), (15, 45), (45, 15)], A_inner=10, B_inner=10):
    A = np.zeros((N, N))
    B = np.zeros((N, N))
    rr, cc = np.indices((N, N))
    for cx, cy in centers:
        dist2 = (rr - cx) ** 2 + (cc - cy) ** 2
        A += A_inner * np.exp(-dist2 / (2 * 5.0 ** 2))
        B += B_inner * np.exp(-dist2 / (2 * 5.0 ** 2))

    A0 = A + np.random.uniform(0.0, 1.0, (N, N))
    B0 = B + np.random.uniform(0.0, 1.0, (N, N))

    return A0, B0

def sinusoidal_initial(N=60, noise_strength=0.1):
    """
    Sinusoidal checkerboard initial condition with small noise.
    Returns A0, B0 fields on an N×N grid.
    """
    # Grid in [0, 2π]
    x = np.linspace(0, 2*np.pi, N)
    y = np.linspace(0, 2*np.pi, N)
    X, Y = np.meshgrid(x, y)

    # Sinusoidal pattern
```

```
        pattern = np.sin(X) * np.sin(Y)

        # Scale into ~[19,20] for consistency with your code
        A = 19.0 + (pattern + 1) / 2
        B = 19.0 + (pattern + 1) / 2

        # Add small uniform noise
        A0 = A + noise_strength * (np.random.rand(N, N) - 0.5)
        B0 = B + noise_strength * (np.random.rand(N, N) - 0.5)

        return A0, B0

    # Dictionary to access IC functions
    ic_options = {
        "Radial Gradient": radial_gradient_both,
        "Random 300": random_300,
        "Multi Gaussian Peaks": multi_gaussian,
        "Square": square,
        "Sinusoidal": sinusoidal_initial
    }

    # -------------------- Parameters --------------------
    Du, Dv = 0.01, 0.01
    alpha, beta, n, Kd = 0.40, 10.0, 1.50, 1.0
    dt = 0.2
    steps = 10000  # reduce for faster demo
    save_every = 1

    # -------------------- Numba kernels --------------------
    @njit(fastmath=True)
    def _compute_A2_B2_elem(Atot, Btot, Kd):
        S = Atot + Btot
        denom_common = Kd + 4.0 * S + np.sqrt(Kd * Kd + 8.0 * S * Kd)
        A2 = 2.0 * Atot * Atot / denom_common
        B2 = 2.0 * Btot * Btot / denom_common
        return A2, B2

    @njit(parallel=True, fastmath=True)
    def _step_neumann(A, B, Du, Dv, alpha, beta, n, Kd, dt):
        nx, ny = A.shape
        Anew = np.empty_like(A)
        Bnew = np.empty_like(B)
        for i in prange(nx):
            im = max(i - 1, 0)
            ip = min(i + 1, nx - 1)
            for j in range(ny):
                jm = max(j - 1, 0)
                jp = min(j + 1, ny - 1)

                LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
                LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

                A2, B2 = _compute_A2_B2_elem(A[i, j], B[i, j], Kd)

                tA = A2 ** n
                actA = beta * (tA / (1.0 + tA))
                fA = alpha + actA - A[i, j]

                tB = B2 ** n
                actB = beta * (tB / (1.0 + tB))
                fB = alpha + actB - B[i, j]

                Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
                Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

        # Clamp negative concentrations
        for i in prange(nx):
            for j in range(ny):
                if Anew[i, j] < 0.0:
                    Anew[i, j] = 0.0
                if Bnew[i, j] < 0.0:
                    Bnew[i, j] = 0.0

        return Anew, Bnew
```

```
        # -------------------- Simulation Runner --------------------
        def run_sim(initial_condition):
            # Get initial condition by passing N to the selected IC function
            A0, B0 = ic_options[initial_condition](N)
            A_temp, B_temp = A0.copy(), B0.copy()
            A_temp, B_temp = _step_neumann(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)  # warmup

            # Run simulation
            A_list = [A0.copy()]
            B_list = [B0.copy()]
            for s in range(1, steps + 1):
                A_temp, B_temp = _step_neumann(A_temp, B_temp, Du, Dv, alpha, beta, n, Kd, dt)
                if s % save_every == 0:
                    A_list.append(A_temp.copy())
                    B_list.append(B_temp.copy())

            # Plotting function
            def plot_at_time(idx, species):
                plt.figure(figsize=(5, 5), dpi=150)
                if species == "A":
                    data = A_list[idx]
                    title = f"A Concentration (t={idx * save_every * dt:.2f})"
                else:
                    data = B_list[idx]
                    title = f"B Concentration (t={idx * save_every * dt:.2f})"
                img = plt.imshow(data, cmap='inferno', origin='lower', interpolation='bilinear')
                plt.colorbar(img)
                plt.title(title)
                plt.axis('off')
                plt.show()

            # Controls
            species_toggle = ToggleButtons(options=["A", "B"], value="A", description="Species:", button_style="info")
            play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100)
            slider = IntSlider(value=0, min=0, max=len(A_list) - 1, step=1, description="Frame")
            jslink((play, 'value'), (slider, 'value'))

            widgets = HBox([play, slider, species_toggle])
            interact_ui = interact(plot_at_time, idx=slider, species=species_toggle)
            display(widgets, interact_ui)

        # -------------------- Dropdown for IC --------------------
        ic_dropdown = Dropdown(options=list(ic_options.keys()), value="Radial Gradient", description="Initial Condition:")
        interact(run_sim, initial_condition=ic_dropdown)

    Initial_N()
```



Toggle Switch

PBC

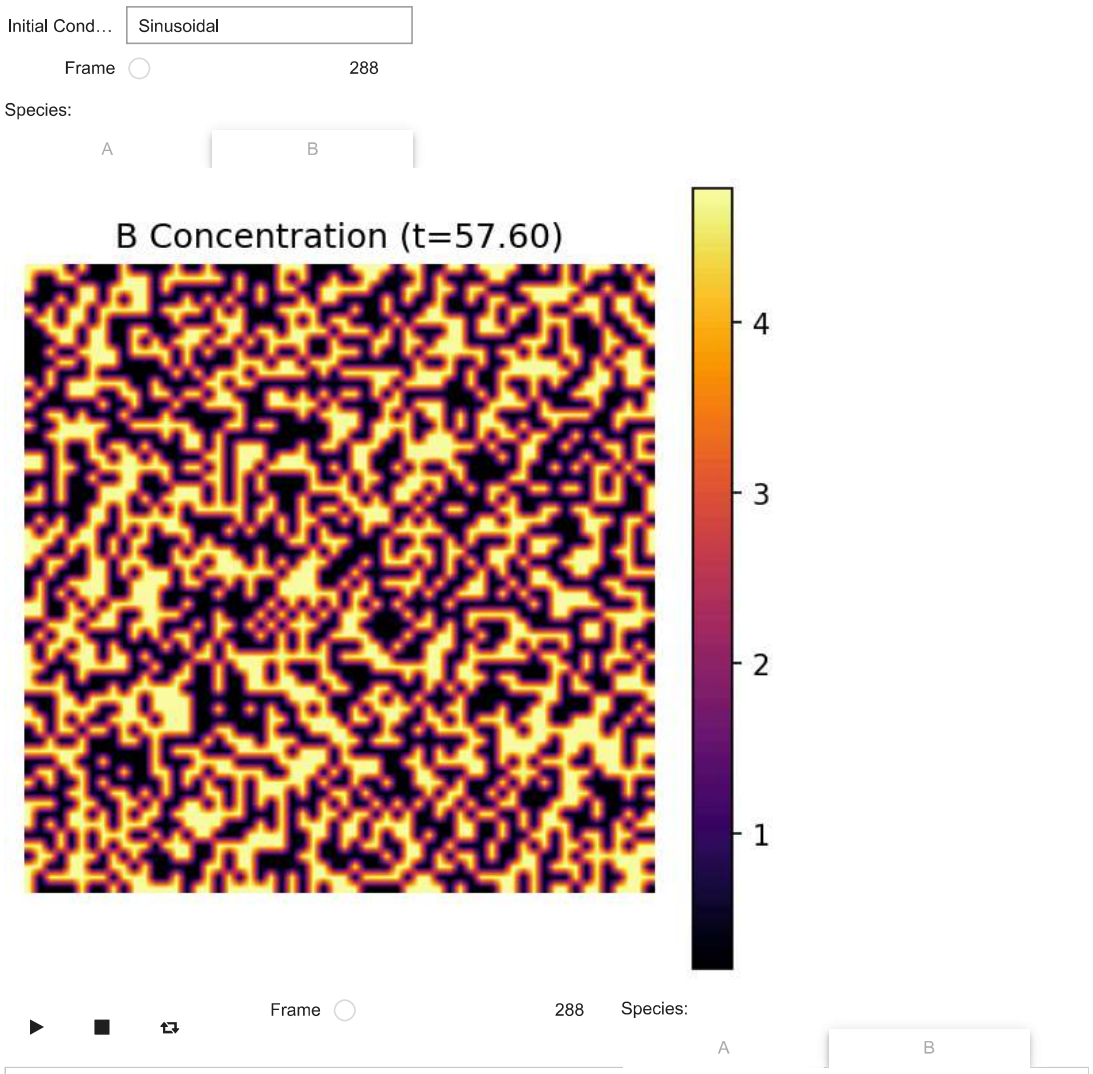
Double-click (or enter) to edit

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Model from figure) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox, Dropdown, ToggleButtons, jslink
from google.colab import output
output.enable_custom_widget_manager()
from IPython.display import display
from numba import njit, prange

def Initial_1():
    # -------------------- Grid --------------------
    N = 60

    # -------------------- Initial Conditions --------------------
    def square(N):
        """Uniform random high values for both species."""
        A0 = np.random.uniform(19.0, 20.0, (N, N))
        B0 = np.random.uniform(19.0, 20.0, (N, N))
        return A0, B0

    def random_300(N):
        """Place three sets (300 each) of special values for A and B at random coordinates."""
```

```
        nx, ny = N, N
        A0 = np.zeros((nx, ny))
        B0 = np.zeros((nx, ny))

        num_total = 900
        num_set = 300
        total_points = nx * ny

        # Species A
        all_indices_A = np.random.choice(total_points, size=num_total, replace=False)
        for k, val in enumerate([9.0, 0.5, 9.0]):
            part = all_indices_A[k * num_set:(k + 1) * num_set]
            coords = np.array(np.unravel_index(part, (nx, ny))).T
            A0[coords[:, 0], coords[:, 1]] = val

        # Species B
        all_indices_B = np.random.choice(total_points, size=num_total, replace=False)
        for k, val in enumerate([9.0, 9.0, 0.5]):
            part = all_indices_B[k * num_set:(k + 1) * num_set]
            coords = np.array(np.unravel_index(part, (nx, ny))).T
            B0[coords[:, 0], coords[:, 1]] = val

        return A0, B0

    def radial_gradient_both(N, center=None,
                             A_inner=10.0, A_outer=1.0,
                             B_inner=10.0, B_outer=1.0):
        if center is None:
            center = (N // 2, N // 2)
        ii, jj = np.indices((N, N))
        r = np.sqrt((ii - center[0]) ** 2 + (jj - center[1]) ** 2)
        r = r / r.max()

        A = A_outer + (A_inner - A_outer) * np.exp(-4 * r)
        B = B_outer + (B_inner - B_outer) * np.exp(-4 * r)

        A0 = A + np.random.uniform(0.0, 1.0, (N, N))
        B0 = B + np.random.uniform(0.0, 1.0, (N, N))

        return A0, B0

    def multi_gaussian(N, centers=[(30, 30), (15, 45), (45, 15)], A_inner=10, B_inner=10):
        A = np.zeros((N, N))
        B = np.zeros((N, N))
        rr, cc = np.indices((N, N))
        for cx, cy in centers:
            dist2 = (rr - cx) ** 2 + (cc - cy) ** 2
            A += A_inner * np.exp(-dist2 / (2 * 5.0 ** 2))
            B += B_inner * np.exp(-dist2 / (2 * 5.0 ** 2))

        A0 = A + np.random.uniform(0.0, 1.0, (N, N))
        B0 = B + np.random.uniform(0.0, 1.0, (N, N))

        return A0, B0

    def sinusoidal_initial(N=60, noise_strength=0.1):
        """
        Sinusoidal checkerboard initial condition with small noise.
        Returns A0, B0 fields on an N×N grid.
        """
        x = np.linspace(0, 2*np.pi, N)
        y = np.linspace(0, 2*np.pi, N)
        X, Y = np.meshgrid(x, y)
        pattern = np.sin(X) * np.sin(Y)
        A = 19.0 + (pattern + 1) / 2
        B = 19.0 + (pattern + 1) / 2
        A0 = A + noise_strength * (np.random.rand(N, N) - 0.5)
        B0 = B + noise_strength * (np.random.rand(N, N) - 0.5)
        return A0, B0

    # Dictionary to access IC functions
    ic_options = {
        "Radial Gradient": radial_gradient_both,
        "Random 300": random_300,
        "Multi Gaussian Peaks": multi_gaussian,
        "Square": square,
```

```
    "Sinusoidal": sinusoidal_initial
}

# -------------------- Parameters --------------------
Du, Dv = 0.0001, 0.0001          # diffusion coefficients (increase if patterns diffuse too slowly)
alpha_u, alpha_v = 5.0, 5.0
gamma_u, gamma_v = 1.0, 1.0
dt = 0.2
steps = 10000              # lower by default so interactive demo isn't too slow
save_every = 1

# -------------------- Numba kernels --------------------
@njit(fastmath=True)
def reaction_u(u_ij, v_ij, alpha_u, gamma_u):
    return alpha_u / (1.0 + v_ij * v_ij) - gamma_u * u_ij

@njit(fastmath=True)
def reaction_v(u_ij, v_ij, alpha_v, gamma_v):
    return alpha_v / (1.0 + u_ij * u_ij) - gamma_v * v_ij

@njit(parallel=True, fastmath=True)
def _step_periodic(A, B, Du, Dv, alpha_u, alpha_v, gamma_u, gamma_v, dt):
    nx, ny = A.shape
    Anew = np.empty_like(A)
    Bnew = np.empty_like(B)
    for i in prange(nx):
        im = (i - 1) % nx
        ip = (i + 1) % nx
        for j in range(ny):
            jm = (j - 1) % ny
            jp = (j + 1) % ny

            # 5-point Laplacian (no dx factor; Du/Dv absorb grid spacing)
            LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
            LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

            fA = reaction_u(A[i, j], B[i, j], alpha_u, gamma_u)
            fB = reaction_v(A[i, j], B[i, j], alpha_v, gamma_v)

            Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
            Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

    # enforce non-negativity
    for i in prange(nx):
        for j in range(ny):
            if Anew[i, j] < 0.0:
                Anew[i, j] = 0.0
            if Bnew[i, j] < 0.0:
                Bnew[i, j] = 0.0

    return Anew, Bnew

# -------------------- Simulation Runner --------------------
def run_sim(initial_condition):
    # Get initial condition by passing N to the selected IC function
    A0, B0 = ic_options[initial_condition](N)
    A_temp, B_temp = A0.copy(), B0.copy()
    # warmup
    A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha_u, alpha_v, gamma_u, gamma_v, dt)

    # Run simulation
    A_list = [A0.copy()]
    B_list = [B0.copy()]
    for s in range(1, steps + 1):
        A_temp, B_temp = _step_periodic(A_temp, B_temp, Du, Dv, alpha_u, alpha_v, gamma_u, gamma_v, dt)
        if s % save_every == 0:
            A_list.append(A_temp.copy())
            B_list.append(B_temp.copy())

    # Plotting function
    def plot_at_time(idx, species):
        plt.figure(figsize=(5, 5), dpi=150)
        if species == "A":
            data = A_list[idx]
            title = f"A Concentration (t={idx * save_every * dt:.2f})"
        else:
```

```
                data = B_list[idx]
                title = f"B Concentration (t={idx * save_every * dt:.2f})"
            img = plt.imshow(data, cmap='inferno', origin='lower', interpolation='bilinear')
            plt.colorbar(img)
            plt.title(title)
            plt.axis('off')
            plt.show()

        # Controls
        species_toggle = ToggleButtons(options=["A", "B"], value="A", description="Species:", button_style="info")
        play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100)
        slider = IntSlider(value=0, min=0, max=len(A_list) - 1, step=1, description="Frame")
        jslink((play, 'value'), (slider, 'value'))

        widgets = HBox([play, slider, species_toggle])
        interact_ui = interact(plot_at_time, idx=slider, species=species_toggle)
        display(widgets, interact_ui)

    # -------------------- Dropdown for IC --------------------
    ic_dropdown = Dropdown(options=list(ic_options.keys()), value="Radial Gradient", description="Initial Condition:")
    interact(run_sim, initial_condition=ic_dropdown)

Initial_1()
```



Initial_1.<locals>.run_sim.<locals>.plot_at_time
def plot_at_time(idx, species)

<no docstring>

NBC

```
# === MultiFate Reaction-Diffusion (Numba-accelerated, Model from figure, Neumann BC) ===
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, IntSlider, Play, HBox, Dropdown, ToggleButtons, jslink
from google.colab import output
```

```
output.enable_custom_widget_manager()
from IPython.display import display
from numba import njit, prange

def Initial_N1():
    # -------------------- Grid --------------------
    N = 60

    # -------------------- Initial Conditions --------------------
    def square(N):
        A0 = np.random.uniform(19.0, 20.0, (N, N))
        B0 = np.random.uniform(19.0, 20.0, (N, N))
        return A0, B0

    def random_300(N):
        nx, ny = N, N
        A0 = np.zeros((nx, ny))
        B0 = np.zeros((nx, ny))
        num_total = 900
        num_set = 300
        total_points = nx * ny

        all_indices_A = np.random.choice(total_points, size=num_total, replace=False)
        for k, val in enumerate([9.0, 0.5, 9.0]):
            part = all_indices_A[k * num_set:(k + 1) * num_set]
            coords = np.array(np.unravel_index(part, (nx, ny))).T
            A0[coords[:, 0], coords[:, 1]] = val

        all_indices_B = np.random.choice(total_points, size=num_total, replace=False)
        for k, val in enumerate([9.0, 9.0, 0.5]):
            part = all_indices_B[k * num_set:(k + 1) * num_set]
            coords = np.array(np.unravel_index(part, (nx, ny))).T
            B0[coords[:, 0], coords[:, 1]] = val

        return A0, B0

    def radial_gradient_both(N, center=None,
                             A_inner=10.0, A_outer=1.0,
                             B_inner=10.0, B_outer=1.0):
        if center is None:
            center = (N // 2, N // 2)
        ii, jj = np.indices((N, N))
        r = np.sqrt((ii - center[0]) ** 2 + (jj - center[1]) ** 2)
        r = r / r.max()
        A = A_outer + (A_inner - A_outer) * np.exp(-4 * r)
        B = B_outer + (B_inner - B_outer) * np.exp(-4 * r)
        A0 = A + np.random.uniform(0.0, 1.0, (N, N))
        B0 = B + np.random.uniform(0.0, 1.0, (N, N))
        return A0, B0

    def multi_gaussian(N, centers=[(30, 30), (15, 45), (45, 15)], A_inner=10, B_inner=10):
        A = np.zeros((N, N))
        B = np.zeros((N, N))
        rr, cc = np.indices((N, N))
        for cx, cy in centers:
            dist2 = (rr - cx) ** 2 + (cc - cy) ** 2
            A += A_inner * np.exp(-dist2 / (2 * 5.0 ** 2))
            B += B_inner * np.exp(-dist2 / (2 * 5.0 ** 2))
        A0 = A + np.random.uniform(0.0, 1.0, (N, N))
        B0 = B + np.random.uniform(0.0, 1.0, (N, N))
        return A0, B0

    def sinusoidal_initial(N=60, noise_strength=0.1):
        x = np.linspace(0, 2*np.pi, N)
        y = np.linspace(0, 2*np.pi, N)
        X, Y = np.meshgrid(x, y)
        pattern = np.sin(X) * np.sin(Y)
        A = 19.0 + (pattern + 1) / 2
        B = 19.0 + (pattern + 1) / 2
        A0 = A + noise_strength * (np.random.rand(N, N) - 0.5)
        B0 = B + noise_strength * (np.random.rand(N, N) - 0.5)
        return A0, B0

    ic_options = {
        "Radial Gradient": radial_gradient_both,
        "Random 300": random_300,
```

```
        "Multi Gaussian Peaks": multi_gaussian,
        "Square": square,
        "Sinusoidal": sinusoidal_initial
    }

    # -------------------- Parameters --------------------
    Du, Dv = 0.0001, 0.0001
    alpha_u, alpha_v = 5.0, 5.0
    gamma_u, gamma_v = 1.0, 1.0
    dt = 0.2
    steps = 10000
    save_every = 1

    # -------------------- Numba kernels --------------------
    @njit(fastmath=True)
    def reaction_u(u_ij, v_ij, alpha_u, gamma_u):
        return alpha_u / (1.0 + v_ij * v_ij) - gamma_u * u_ij

    @njit(fastmath=True)
    def reaction_v(u_ij, v_ij, alpha_v, gamma_v):
        return alpha_v / (1.0 + u_ij * u_ij) - gamma_v * v_ij

    @njit(parallel=True, fastmath=True)
    def _step_neumann(A, B, Du, Dv, alpha_u, alpha_v, gamma_u, gamma_v, dt):
        nx, ny = A.shape
        Anew = np.empty_like(A)
        Bnew = np.empty_like(B)

        for i in prange(nx):
            im = max(i - 1, 0)      # Neumann BC
            ip = min(i + 1, nx - 1)
            for j in range(ny):
                jm = max(j - 1, 0)
                jp = min(j + 1, ny - 1)

                LapA = -4.0 * A[i, j] + A[im, j] + A[ip, j] + A[i, jm] + A[i, jp]
                LapB = -4.0 * B[i, j] + B[im, j] + B[ip, j] + B[i, jm] + B[i, jp]

                fA = reaction_u(A[i, j], B[i, j], alpha_u, gamma_u)
                fB = reaction_v(A[i, j], B[i, j], alpha_v, gamma_v)

                Anew[i, j] = A[i, j] + dt * (Du * LapA + fA)
                Bnew[i, j] = B[i, j] + dt * (Dv * LapB + fB)

        # enforce non-negativity
        for i in prange(nx):
            for j in range(ny):
                if Anew[i, j] < 0.0:
                    Anew[i, j] = 0.0
                if Bnew[i, j] < 0.0:
                    Bnew[i, j] = 0.0

        return Anew, Bnew

    # -------------------- Simulation Runner --------------------
    def run_sim(initial_condition):
        A0, B0 = ic_options[initial_condition](N)
        A_temp, B_temp = A0.copy(), B0.copy()

        # warmup
        A_temp, B_temp = _step_neumann(A_temp, B_temp, Du, Dv, alpha_u, alpha_v, gamma_u, gamma_v, dt)

        A_list = [A0.copy()]
        B_list = [B0.copy()]
        for s in range(1, steps + 1):
            A_temp, B_temp = _step_neumann(A_temp, B_temp, Du, Dv, alpha_u, alpha_v, gamma_u, gamma_v, dt)
            if s % save_every == 0:
                A_list.append(A_temp.copy())
                B_list.append(B_temp.copy())

        def plot_at_time(idx, species):
            plt.figure(figsize=(5, 5), dpi=150)
            data = A_list[idx] if species == "A" else B_list[idx]
            title = f"{species} Concentration (t={idx * save_every * dt:.2f})"
            img = plt.imshow(data, cmap='inferno', origin='lower', interpolation='bilinear')
            plt.colorbar(img)
```

```
                plt.title(title)
                plt.axis('off')
                plt.show()

            species_toggle = ToggleButtons(options=["A", "B"], value="A", description="Species:", button_style="info")
            play = Play(value=0, min=0, max=len(A_list) - 1, step=1, interval=100)
            slider = IntSlider(value=0, min=0, max=len(A_list) - 1, step=1, description="Frame")
            jslink((play, 'value'), (slider, 'value'))

            widgets = HBox([play, slider, species_toggle])
            interact_ui = interact(plot_at_time, idx=slider, species=species_toggle)
            display(widgets, interact_ui)

        ic_dropdown = Dropdown(options=list(ic_options.keys()), value="Radial Gradient", description="Initial Condition:")
        interact(run_sim, initial_condition=ic_dropdown)

Initial N1()
```