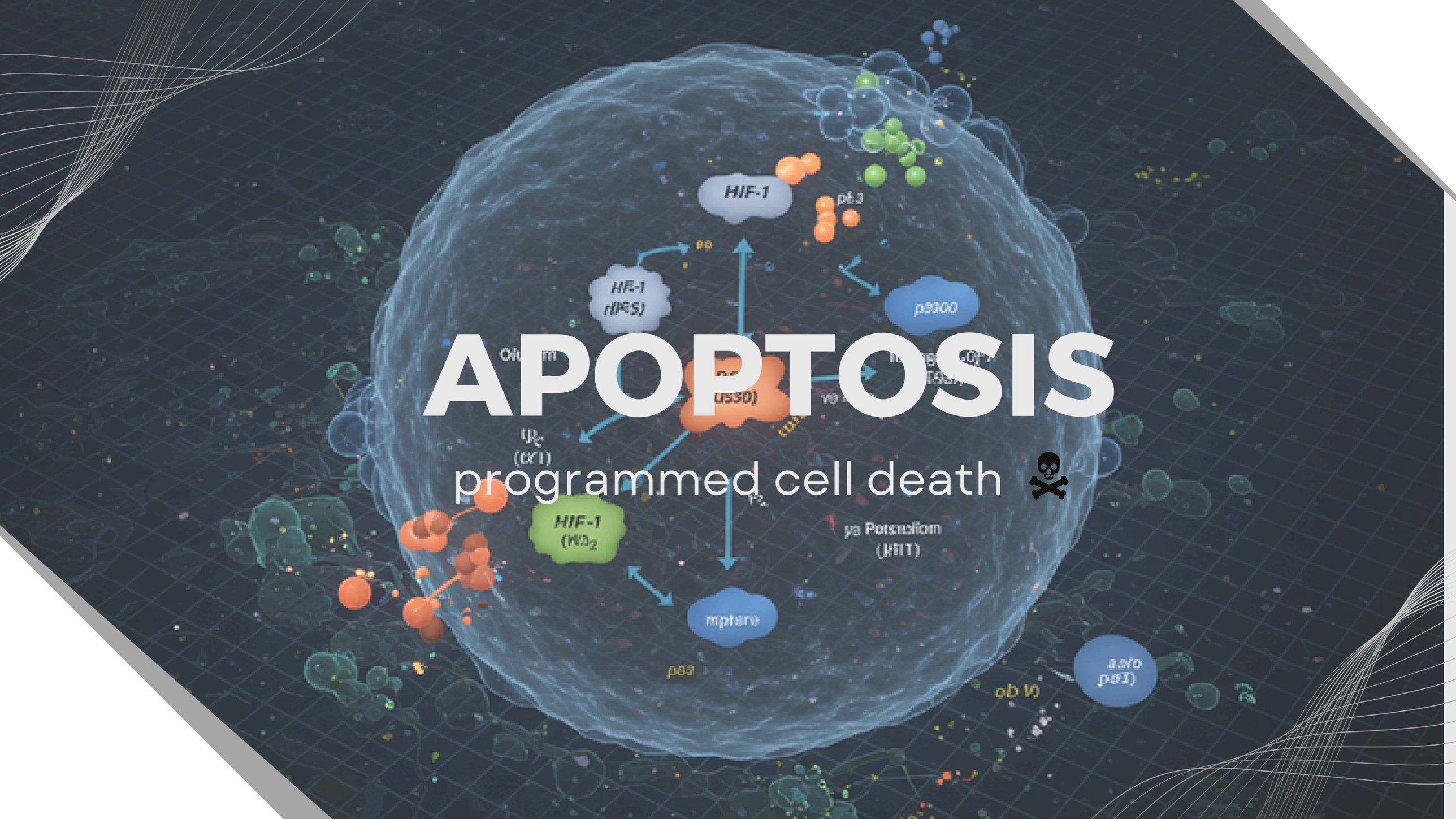
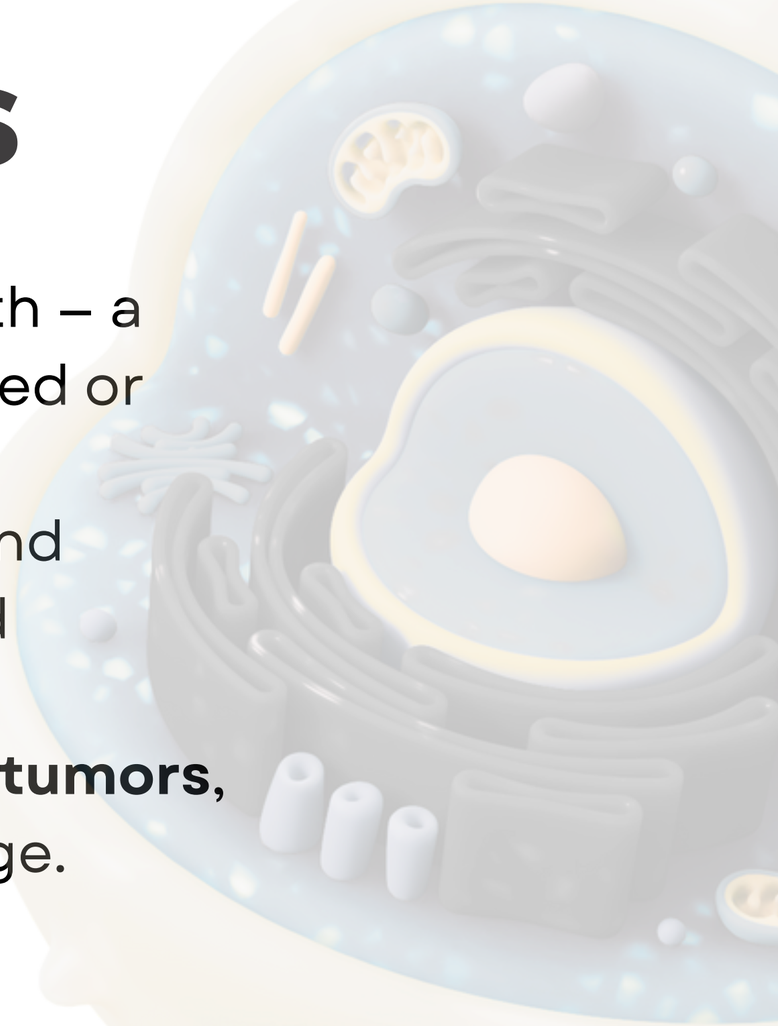


APOPTOSIS

programmed cell death ☠

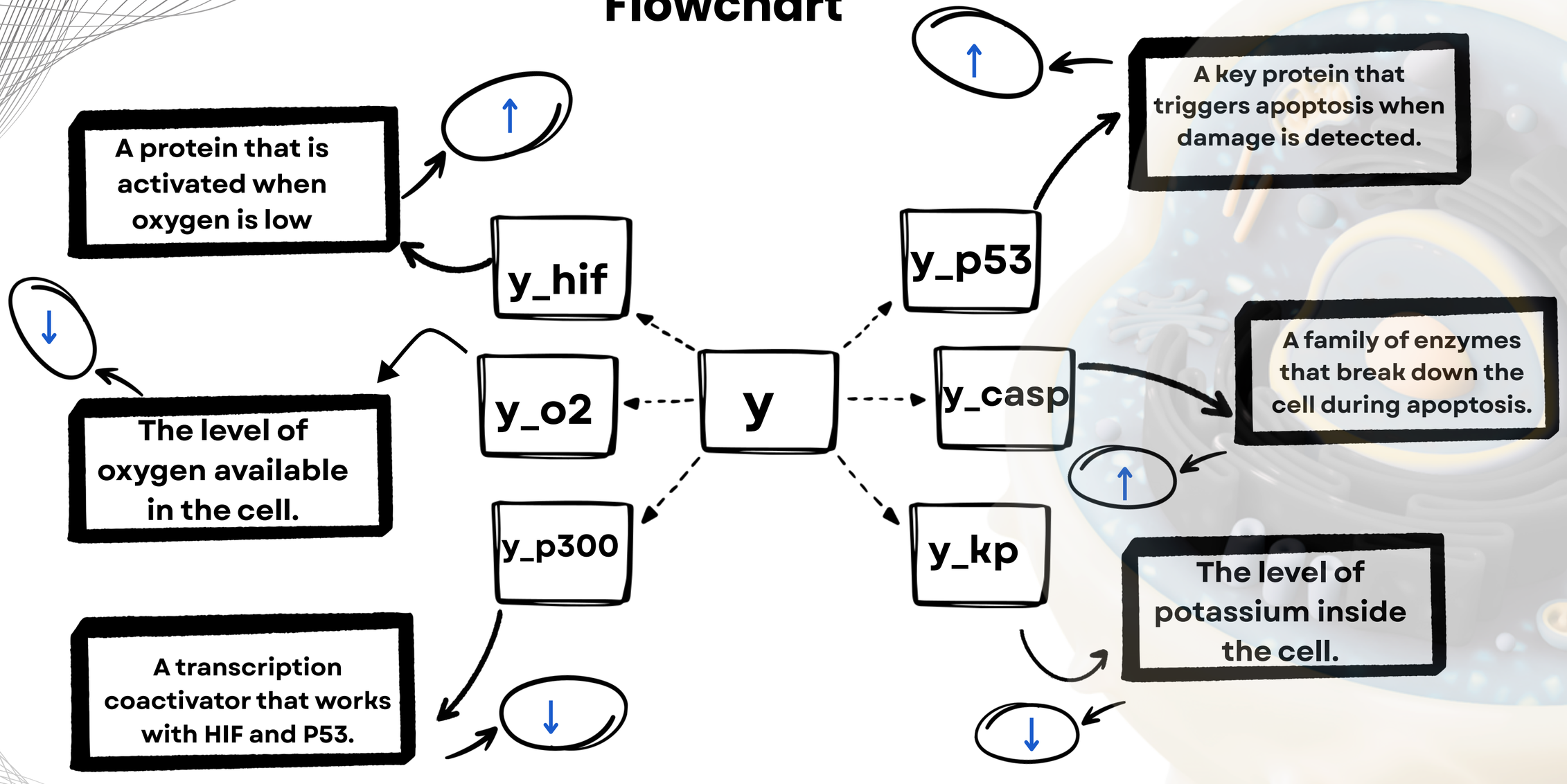
ABOUT APOPTOSIS



- **Apoptosis** is programmed cell death – a self-destruction system for damaged or unwanted cells.
- This process is **controlled**, **clean**, and necessary for growth, defense, and maintaining balance.
- It plays a crucial role in **preventing tumors**, infections, and autoimmune damage.

"Apoptosis is like a biological delete button – it removes harmful cells without damaging surrounding tissue."

Flowchart


A protein that is activated when oxygen is low
↑
y_hif
↓
The level of oxygen available in the cell.
y_o2
y
y_p300
A transcription coactivator that works with HIF and P53.
↓
↑
A key protein that triggers apoptosis when damage is detected.
y_p53
y_casp
A family of enzymes that break down the cell during apoptosis.
↑
y_kp
The level of potassium inside the cell.
↓

ODE Model of Apoptosis

What is it?

We use a system of Ordinary Differential Equations (ODEs) to describe how key molecules change over time during apoptosis.

Each equation shows:

- How the concentration of each variable (y) increases or decreases
- Based on biological interactions

```
# Define parameters
params = {
 'ahif': 1.52, 'ao2': 1.8, 'ap53': 0.05, 'a3': 0.9, 'a4': 0.2,
 'a5': 0.001, 'a7': 0.7, 'a8': 0.06, 'a9': 0.1, 'a10': 0.7,
 'a11': 0.2, 'a12': 0.1, 'a13': 0.1, 'a14': 0.05 }

# System of ODEs
def apoptosis_ode_system(t, y, p):
 """Defines the system of ODEs for apoptosis."""
 yhif, yo2, yp300, yp53, ycasp, ykp = y
 dyhif_dt = p['ahif'] - p['a3']*yo2*yhif - p['a4']*yhif*yp300 - p['a7']*yp53*yhif
 dyo2_dt = p['ao2'] - p['a3']*yo2*yhif + p['a4']*yhif*yp300 - p['a11']*yo2
 dyp300_dt = p['a8'] - p['a4']*yhif*yp300 - p['a5']*yp300*yp53
 dyp53_dt = p['ap53'] - p['a5']*yp300*yp53 - p['a9']*yp53
 dycasp_dt = p['a12'] + p['a9']*yp53 - p['a13']*ycasp
 dykp_dt = -p['a10']*ycasp*ykp + p['a11']*yo2 - p['a14']*ykp
 return np.array([dyhif_dt, dyo2_dt, dyp300_dt, dyp53_dt, dycasp_dt, dykp_dt])
```

PROJECT GOALS

Simulate apoptosis modeling

Use ODEs to capture how HIF, P53, caspases and other factors evolve over time under low oxygen.

Compare numerical solvers

Evaluate Euler, RK4 and RKF45 in terms of accuracy, stability and computational cost.

Apply PINNs

Implement physics-informed neural networks as an alternative method for solving the ODE system.

Visualize molecular dynamics

Plot time courses of key concentrations to reveal trigger points and response patterns.

Bridge biology and engineering

Translate a complex cellular process into a mathematical framework for biomedical insights.

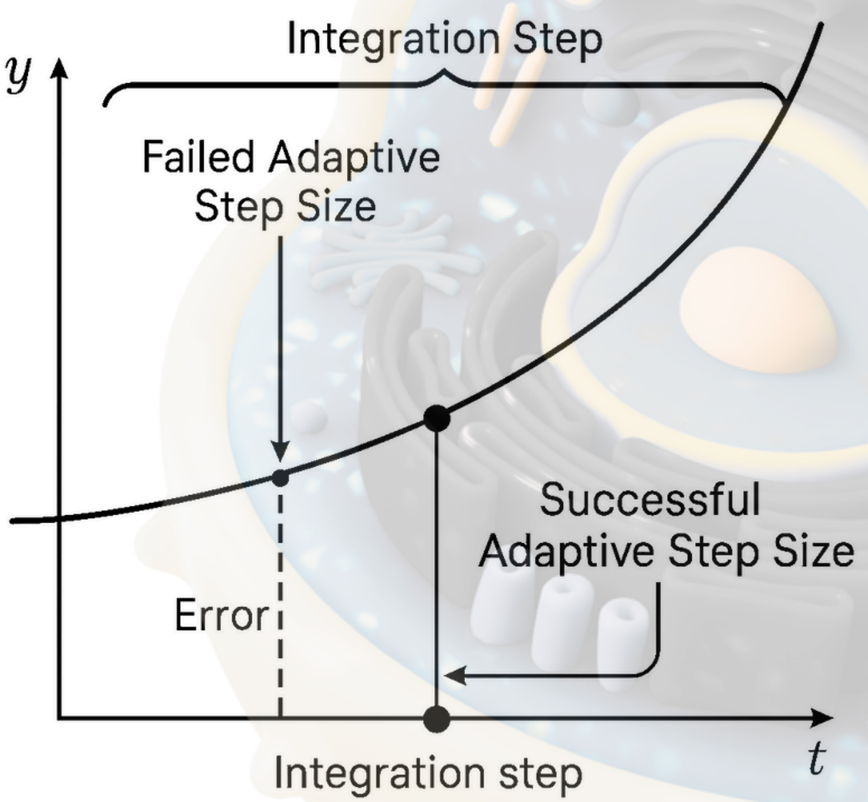
Investigate applications

Relate our findings to cancer therapy, drug screening and tissue engineering scenarios.

RKF45

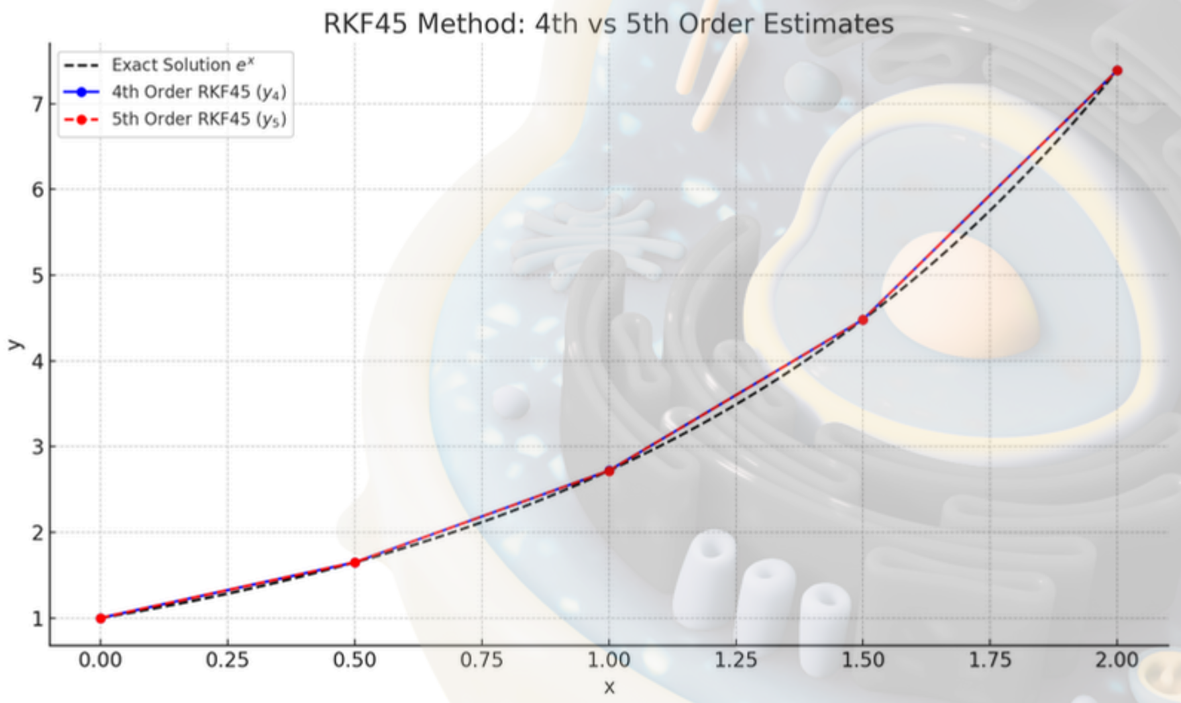
What is RKF45

- Runge-Kutta-Fehlberg (RKF45)
- Combines 4th and 5th order methods
- Provides adaptive step-size control
- Used to solve ODEs efficiently with local error estimation



How RKF45 Works

- At each step:
  a. Calculate six intermediate slopes (k_1 to k_6).
  b. Compute two solution estimates:
     i. 4th order: $y^{(4)}$
     ii. 5th order: $y^{(5)}$
- Estimate local error:
  - $e = | y^{(5)} - y^{(4)} |$
- Compare error to tolerance:
  - If e>tolerance, reduce step size and recalculate.
  - If e≤tolerance, accept step and possibly increase step size.
  - Repeat until the solution reaches the final time.



Solving Apoptosis ODEs Using RKF45 in Python

- MODEL: 6 ODES SIMULATE KEY PROTEINS IN APOPTOSIS (HIF-1, O_2, P300, P53, CASPASE, K^+)
- METHOD: HAND-CODED RKF45 (RUNGE-KUTTA-FEHLBERG) WITH ADAPTIVE STEP SIZE AND ERROR CONTROL
- SIMULATION: COMPARED TWO INITIAL CONDITIONS:
  - CASE 1: HIF-1 ACTIVE → Y_0 = [1, 0, 0, 0, 0, 0]
  - CASE 2: ALL INACTIVE → Y_0 = [0, 0, 0, 0, 0, 0]
- OUTPUT:
  - TRACKED CONCENTRATIONS OVER TIME
  - EVALUATED TRUNCATION ERROR ($Y_5 - Y_4$)
  - HIGHLIGHTED SENSITIVITY TO INITIAL CONDITIONS VIA PLOTS

RKF45 INTEGRATOR LOOP (CORE IDEA)

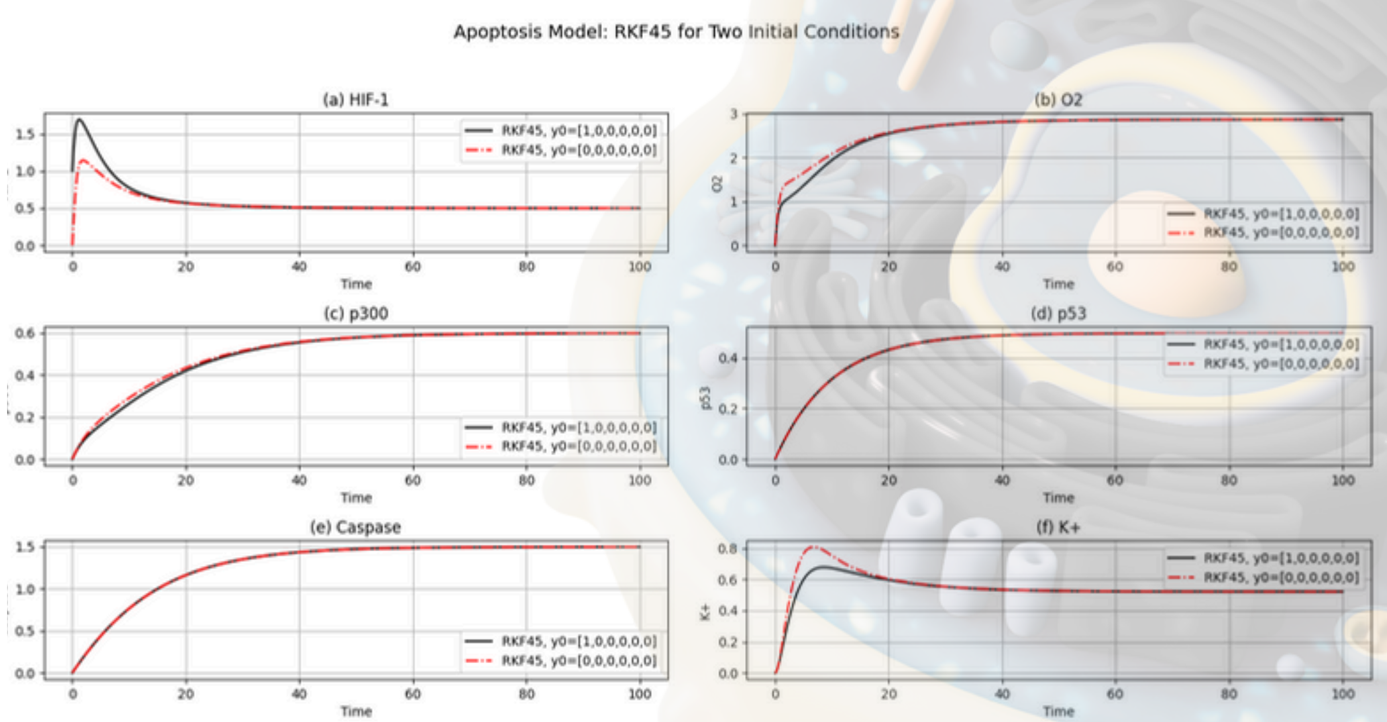
```
def rkf45_with_error(f, t0, y0, t_end, h0=0.1, tol=1e-6):

  while t < t_end:
    if t + h > t_end:
      h = t_end - t

    # coefficients for Fehlberg RKF45
    c = np.array([0, 1/4, 3/8, 12/13, 1, 1/2])
    a2 = np.array([1/4])
    a3 = np.array([3/32, 9/32])
    a4 = np.array([1932/2197, -7200/2197, 7296/2197])
    a5 = np.array([439/216, -8, 3680/513, -845/4104])
    a6 = np.array([-8/27, 2, -3544/2565, 1859/4104, -11/40])
    b4 = np.array([25/216, 0, 1408/2565, 2197/4104, -1/5, 0])
    b5 = np.array([16/135, 0, 6656/12825, 28561/56430, -9/50, 2/55])

    k1 = f(t + c[0]*h, y) * h
    k2 = f(t + c[1]*h, y + a2[0]*k1) * h
    k3 = f(t + c[2]*h, y + a3[0]*k1 + a3[1]*k2) * h
    k4 = f(t + c[3]*h, y + a4[0]*k1 + a4[1]*k2 + a4[2]*k3) * h
    k5 = f(t + c[4]*h, y + a5[0]*k1 + a5[1]*k2 + a5[2]*k3 + a5[3]*k4) * h
    k6 = f(t + c[5]*h, y + a6[0]*k1 + a6[1]*k2 + a6[2]*k3 + a6[3]*k4 + a6[4]*k5) * h

    y4 = y + b4[0]*k1 + b4[1]*k2 + b4[2]*k3 + b4[3]*k4 + b4[4]*k5
    y5 = y + b5[0]*k1 + b5[1]*k2 + b5[2]*k3 + b5[3]*k4 + b5[4]*k5 + b5[5]*k6

    ee = y5 - y4 # Truncation error estimate at this step

    err = np.linalg.norm(ee, ord=np.inf)
    if err <= tol:
      t += h
      y = y4
      ts.append(t)
      ys.append(y.copy())
      ees.append(ee.copy())

    # adapt step size
    delta = 0.84 * (tol / (err + 1e-16))**0.25
    h = max(min(delta * h, 5*h), 0.1*h)
```

RESULTS - PLOTS

- **Simulation** of six apoptosis-related components using RKF45.
- **Black line:** Initial condition y_0 = [1, 0, 0,0,0 0] resent at start
- **Red dashed line**: Initial condition y_0 = [0,0,0,0 0,] components start at zero

Limitations:

- Can be slower on very stiff systems.
- Requires more computations per step than simple methods.
- Complexity in implementation compared to fixed-step solvers.

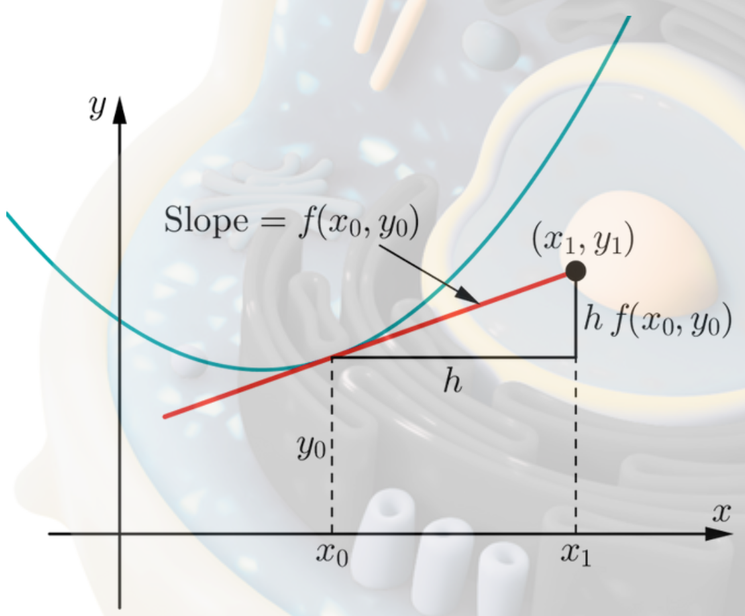


EULER

What is Euler's method?

$$y_{n+1} = y_n + h\frac{dy}{dx}$$

- A first-order numerical procedure for solving ordinary differential equations (ODEs).
- Approximates the solution by stepping forward in small increments using the derivative (slope) at the current point.



Solving Apoptosis ODEs Using Euler Method in Python:

- **SIMULATION: COMPARED TWO INITIAL CONDITIONS:**
  - CASE 1: HIF-1 ACTIVE → Y_0 = [1, 0, 0, 0, 0, 0]
  - CASE 2: ALL INACTIVE → Y_0 = [0, 0, 0, 0, 0, 0]

- **OUTPUT**:
  - TRACKED PROTEIN CONCENTRATIONS OVER TIME
  - DEMONSTRATED SENSITIVITY TO INITIAL CONDITIONS THROUGH PLOTTED TRAJECTORIES
  - COMPARED STABILITY AND ACCURACY LIMITATIONS

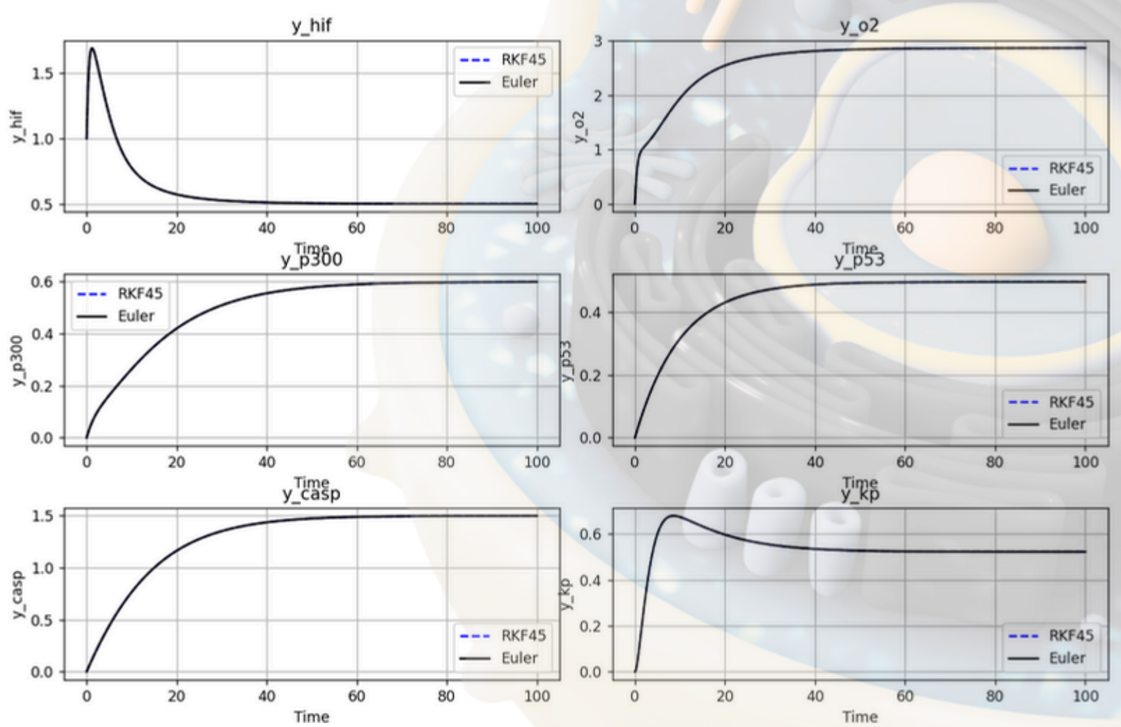
- **LIMITATIONS OF EULER METHOD:**
  - LOW ACCURACY: FIRST-ORDER, ACCUMULATES SIGNIFICANT ERROR
  - FIXED STEP SIZE: CAN MISS RAPID BIOLOGICAL CHANGES
  - COMPUTATIONALLY INEFFICIENT: NEEDS VERY SMALL STEPS FOR PRECISION
  - NO ERROR CONTROL: LACKS ADAPTIVE CORRECTION

```
# Euler method solver
def euler_solver(ode_func, y0, t_span, dt, p):
    """An implementation of the Explicit Euler method."""
    t_start, t_end = t_span
    t_vals = np.arange(t_start, t_end + dt, dt)
    y_vals = np.zeros((len(t_vals), len(y0)))
    y_vals[0, :] = y0
    for i in range(len(t_vals) - 1):
        y_current = y_vals[i, :]
        slope = ode_func(t_vals[i], y_current, p)
        y_vals[i+1, :] = y_current + dt * slope
    return t_vals, y_vals
```

EULER VS RKF45

- **Error:** Euler builds up error quickly (Global: O(h)), while RKF45 maintains high accuracy with adaptive steps (Global: $O(h^4)$).

- **Stability:** Euler can become unstable for stiff equations, while RKF45 remains robust and reliable across varying step sizes.

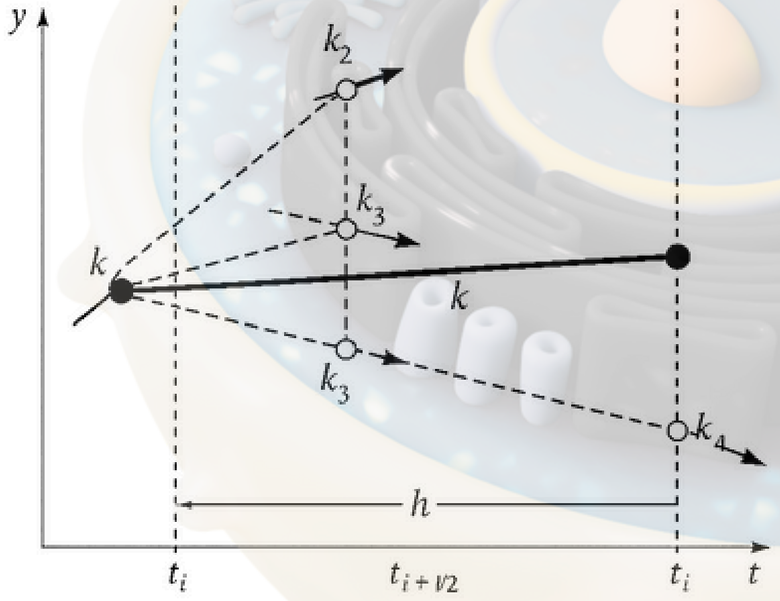


RK4

4TH ORDER RUNGE-KUTTA METHOD

What is RK4?

- **RK4**: fixed-step method for solving ODEs
- Ideal for modeling biological systems like apoptosis
- More accurate than Euler → uses 4 slope estimates
- Reliable for predicting system behavior over time



HOW RK4 WORKS?

RK4 estimates the solution using a weighted average of 4 slopes (k_1 to k_4):

- k_1 → Slope at the beginning of the interval
- k_2 → Slope at the midpoint
- k_3 → Another midpoint slope
- k_4 → Slope at the end of the interval

Final result is a weighted average of these slopes:

$$y_{n+1} = y_n + (h/6)\cdot(k_1 + 2k_2 + 2k_3 + k_4)$$

```
def rk4_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h/2, y + h * k1 / 2)
    k3 = f(t + h/2, y + h * k2 / 2)
    k4 = f(t + h, y + h * k3)
    return y + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4)

# Time integration loop
y = y0.copy()
for n, t in enumerate(times[::-1]):
    y = rk4_step(dydt, t, y, h)
    results[n + 1] = y
```

This approach captures the curvature of the solution more accurately

EFFECT OF INITIAL CONDITIONS

RK4 Results for Two Scenarios

Case A (y(0) = [1,0,0,0,0,0])

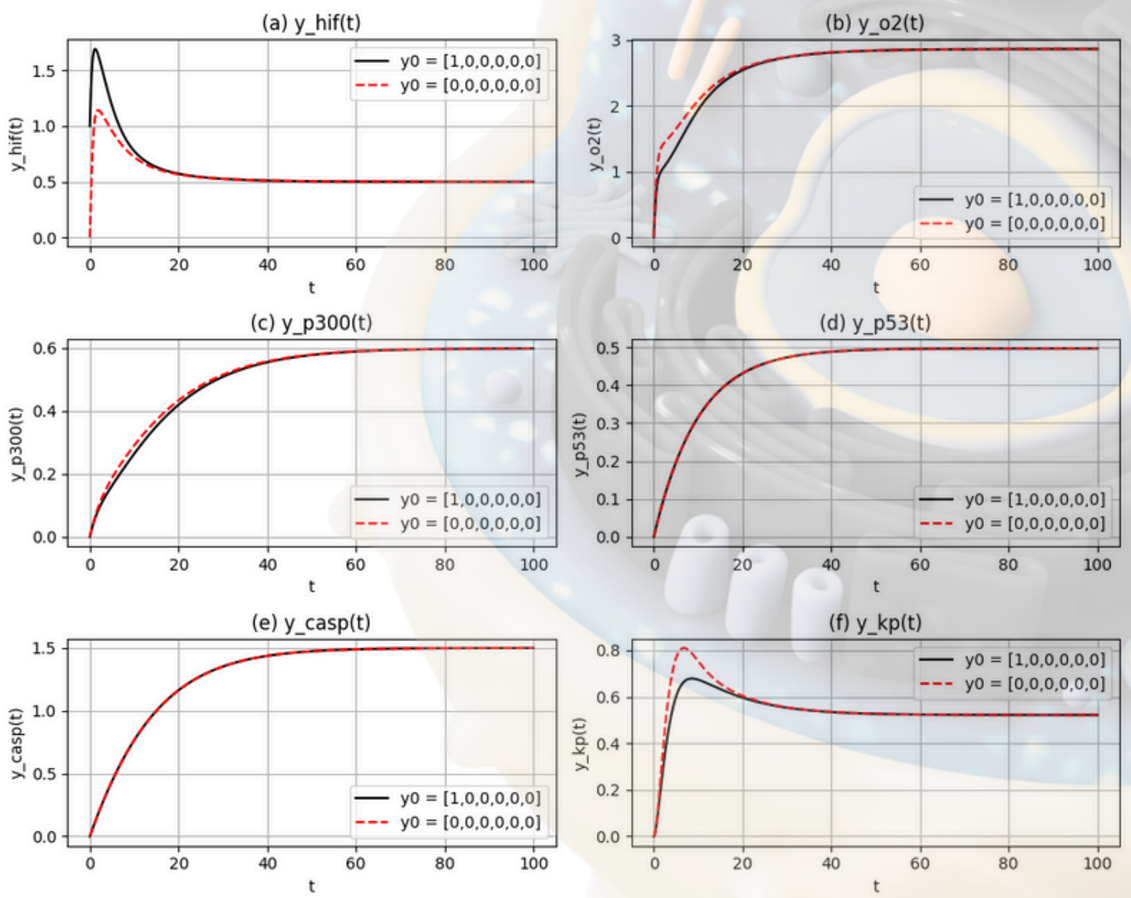
- HIF increases rapidly, activating p53 and caspase cascades.
- Oxygen, p300, and K^+ exhibit anticipated dynamics.

Case B (y(0) = [0,0,0,0,0,0])

- By changing the initial values the system still staple.

Visual:

- Two overlaid time-course plots (t ∈ [0,100]) for each variable:
  - Solid lines: Case A
  - Dashed lines: Case B



LIMITATIONS OF RK4

Limitations:

- Higher per-step cost with four function evaluations, 3–4 times more than Euler
- Requires manual tuning of step size; larger values may miss fast transients..
- Less adaptive, lacking automatic error control like RKF45 for optimal precision.

PINNS OVERVIEW

Environment & Data

- **Framework**: PyTorch
- **Training Domain**: $t \in [0, 100]$, 200 collocation points
- **Initial Condition**: $y(0) = [1, 0, 0, 0, 0, 0]$
- **Baseline Solver for Comparison**: SciPy's solve_ivp (RKF45)

NETWORK ARCHITECTURE & TRAINING LOOP

PINN Design & Optimization

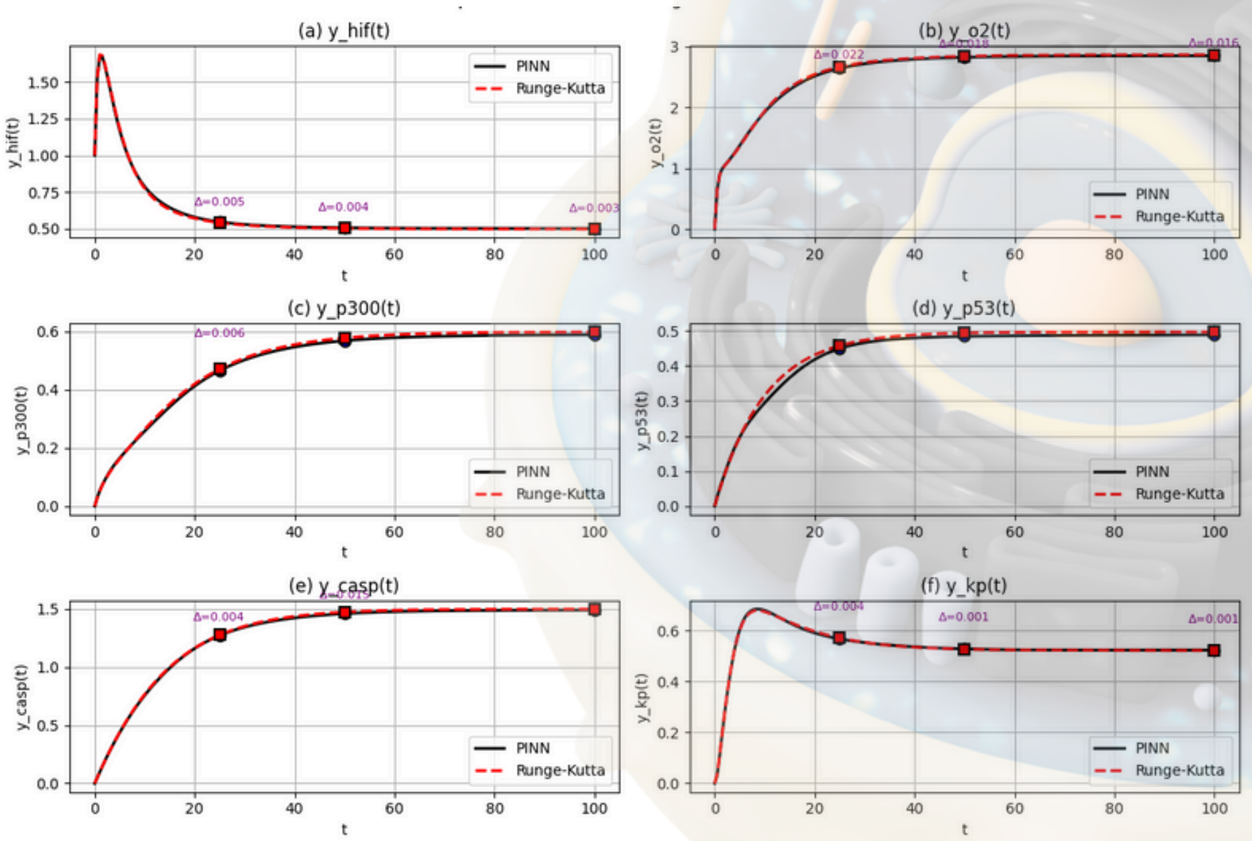
- **Model**: **3-layer** MLP (1→64→64→6) with **tanh** activations
- **Inputs/Outputs:**
  - **Input**: scalar t
  - **Output**: [ŷ_hif, ŷ_o2, ŷ_p300, ŷ_p53, ŷ_casp, ŷ_kp]
- **Loss**:
  - **Physics Loss**: MSE of ODE residuals $f_1 \ldots f_6$
  - **IC Loss**: MSE at t=0
  - **Total** = Physics + IC
- **Optimizer**:
  - **Adam** (lr=1e-3), 5 000 epochs
  - Print **loss** every 500 steps

```
model = PINN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
mse_loss = nn.MSELoss()

# Training data
t_train = torch.linspace(0, 100, 200).view(-1, 1).requires_grad_(True).to(device)
t0 = torch.tensor([[0.]], requires_grad=True, device=device)
y0 = torch.tensor([[1., 0., 0., 0., 0., 0.]], device=device)

# Derivative helper
def gradients(y, x):
    return torch.autograd.grad(y, x, grad_outputs=torch.ones_like(y),
create_graph=True)[0]

print("Starting training...")
for epoch in range(5000):
    optimizer.zero_grad()

    # Forward pass
    y_pred = model(t_train)
    y_hif, y_o2, y_p300, y_p53, y_casp, y_kp = [y_pred[:, i:i+1] for i in range(6)]

    # Losses
    loss_f = (f1**2 + f2**2 + f3**2 + f4**2 + f5**2 + f6**2).mean()
    y0_pred = model(t0)
    loss_ic = mse_loss(y0_pred, y0)
    loss = loss_f + loss_ic

    # Backward pass
    loss.backward()
    optimizer.step()

    if epoch % 500 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.5f}")
```

PINN VS RKF45 – TIME COURSES

Solution Trajectories

- **Plot Details:**
  - **Black solid:** PINN predictions
  - **Red dashed:** RK45 reference
  - Markers at **t=25,50,100** with Δ-value annotations
- **Observation:** PINN closely tracks RK45 across all six variables



ANALYSIS

- **All Calculations at T = 100**

Method	Yhif	Yo2	Yp300	Yp53	Ycasp	Ykp
Runge-Kutta-Fehlberg (RKF45)	0.4990	2.8646	0.5977	0.4970	1.4967	0.5219
Fourth order Runge-Kutta (RK4)	0.4990	2.8646	0.5977	0.4970	1.4967	0.5219
Error	0.144×10^{-4} %	0.470×10^{-5} %	0.488×10^{-4} %	0.579×10^{-4} %	0.903×10^{-5} %	0.477×10^{-4} %
Explicit Euler	0.4990	2.8646	0.5977	0.4970	1.4967	0.5219
Error	0.705×10^{-4} %	0.463×10^{-4} %	0.192×10^{-3} %	0.410×10^{-4} %	0.869×10^{-4} %	0.216×10^{-4} %
Physics-Informed Neural Network (PINN)	0.5027	2.8491	0.5871	0.4887	1.4878	0.5228
Error	0.7435%	0.5417%	1.7866%	1.6647%	0.5925%	0.1722%

RESULT

Method	Execution Time	Max Percentage Error	Min Percentage Error
Runge-Kutta-Fehlberg (RKF45)	0.042 sec	----	----
fourth order Runge-Kutta (RK4)	0.939 sec	0.579×10^{-4} %	0.470×10^{-5} %
Explicit Euler	0.189 sec	0.192×10^{-3} %	0.216×10^{-4} %
Physics-Informed Neural Network (PINN)	0.01092 sec	1.7866%	0.1722%

FUTURE WORK

Next Steps

- Refining parameters using experimental data
- Developing hybrid RKF45–PINN solvers
- Validating results in cell cultures
- Extending the work to spatial (PDE) models

CONCLUSION

Developed an ordinary differential equation (ODE) model to investigate hypoxia-induced apoptosis, focusing on the dynamics of HIF, O_2, p300, p53, caspases, and K^+.

Through comparative tests, we assessed:

- **Euler's** method for its straightforwardness
- **RK4** for its effective balance of cost and accuracy
- **RKF45** for its adaptive precision

Utilizing **Physics-Informed Neural Networks (PINNs)** yielded smooth solutions that closely aligned with the outcomes derived from RKF45.

This integrated strategy—combining classical solvers with machine learning—presents a flexible framework for applications in biomedical engineering.

REFERENCE

[1] L. Euler, Institutionum calculi integralis, 1768–1770. Introduced the explicit Euler method, the first-order ODE solver.

[2] C. Runge, “ ¨Uber die numerische Aufl¨osung von Differentialgleichun- gen,” Mathematische Annalen, vol. 46, pp. 167–178, 1895. M. Kutta, “Beitrag zur n¨aherungsweisen Integration totaler Differentialgleichun- gen,” Zeitschrift f¨ur Mathematik und Physik, vol. 46, pp. 435–453, 1901. Together, they developed the RK family, including the 4th-order RK4 method.

[3] E. Fehlberg, “Low-order classical Runge-Kutta formulas with step- size control and their application to some heat transfer problems,” NASA Technical Report R-315, 1969. This paper introduced the adaptive RKF45 solver with embedded error control.

[4] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations,” Journal of Computational Physics, vol. 378, pp. 686–707, 2019. Pro- posed PINNs—a modern ML-based solver that embeds physical laws into neural networks.

[5] W. E. Schiesser, Differential Equation Analysis in Biomedical Science and Engineering: Ordinary Differential Equation Applications with R. Wiley, 2014. Demonstrated biomedical modeling using ODEs and R, with case studies including apoptosis.

THANK YOU