SBEG108 NUMERICAL METHODS IN BME

Stem Cell Differentiation

Numerical and Machine Learning Methods for Differential Equations in Biomedical Engineering

Team 4 Course Project

SBEG108 NUMERICAL METHODS IN BME

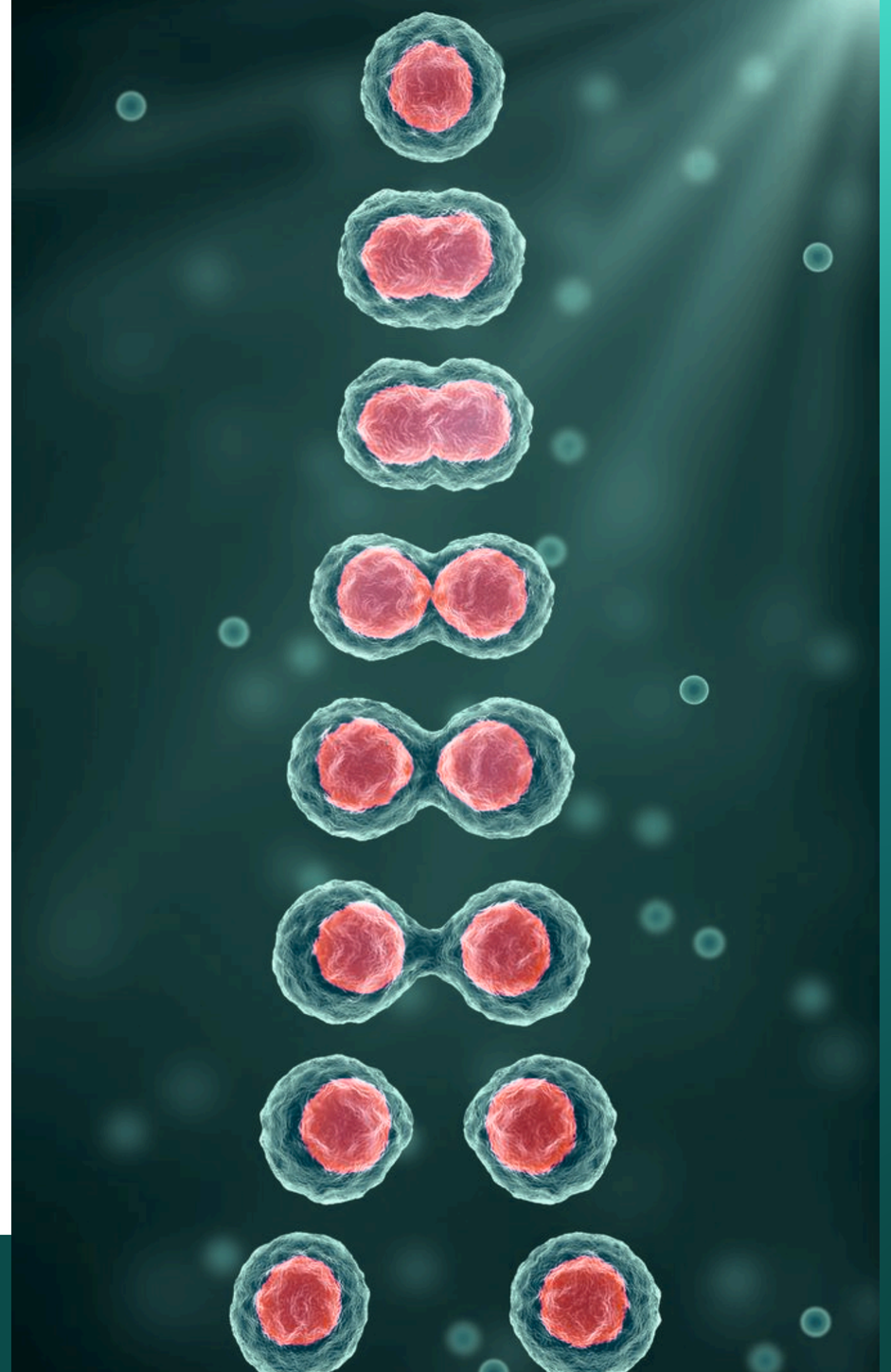
TABLE OF CONTENT

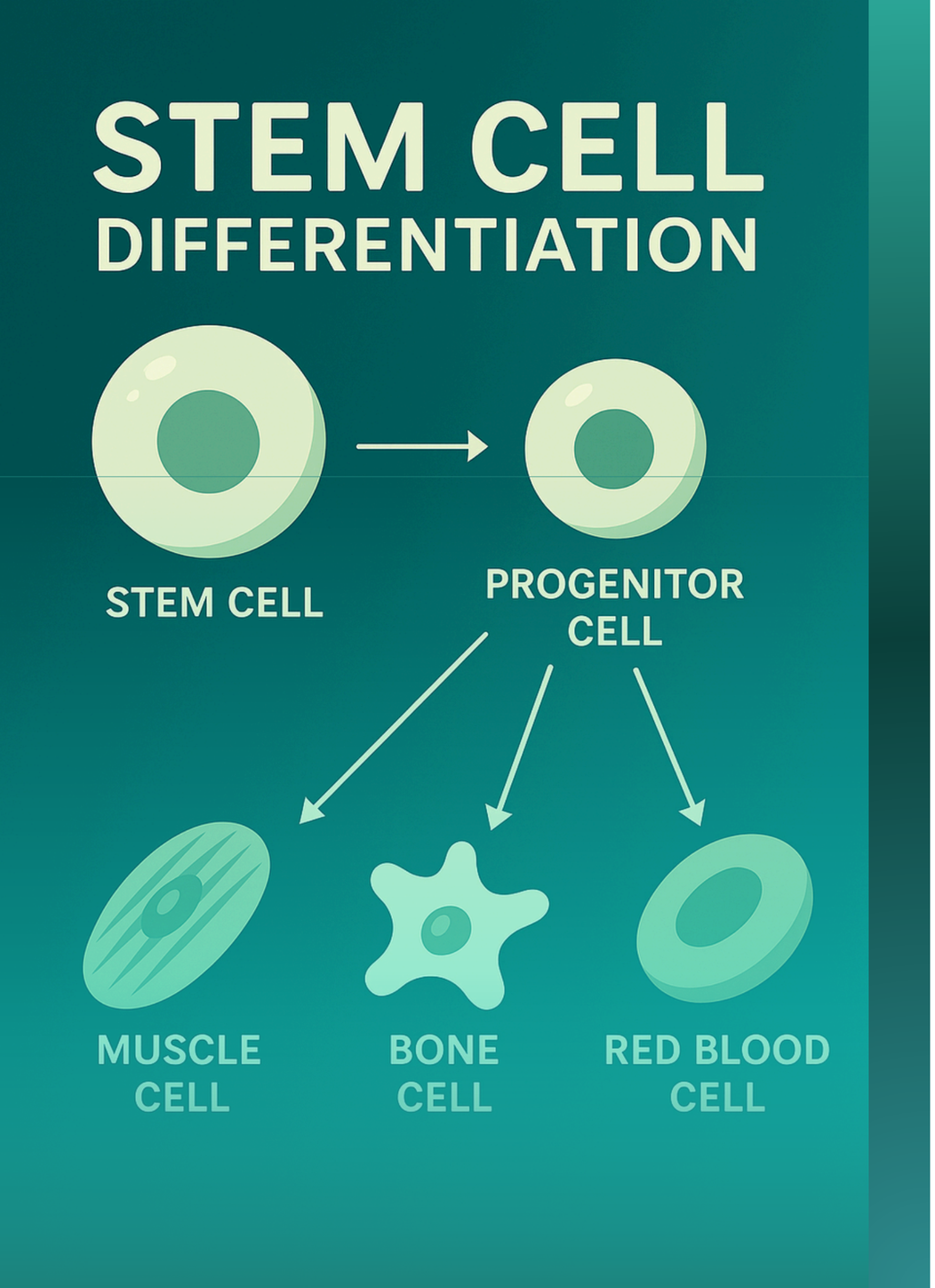
Overview of the Stem Cells Differentiation

What is Stem Cell Differentiation?

Why PU.1 and GATA-1?

How to use ODE to study this?





What is Stem Cell Differentiation?

- Stem cells can divide and transform into specialized cells like muscle or bone.

The process:

Stem cell → ***Progenitor state*** → ***Specialized cell***

Why PU.1 and GATA-1?

- These two key transcription factors control blood cell development.
- Both are low in early progenitors, but their interaction decides the final cell type.
- Understanding them helps in:

→ ***Directing cell fate*** → ***Advancing therapies***

How to use ODE to study this?

- **To model this interaction, we use a 2×2 nonlinear ODE system:**

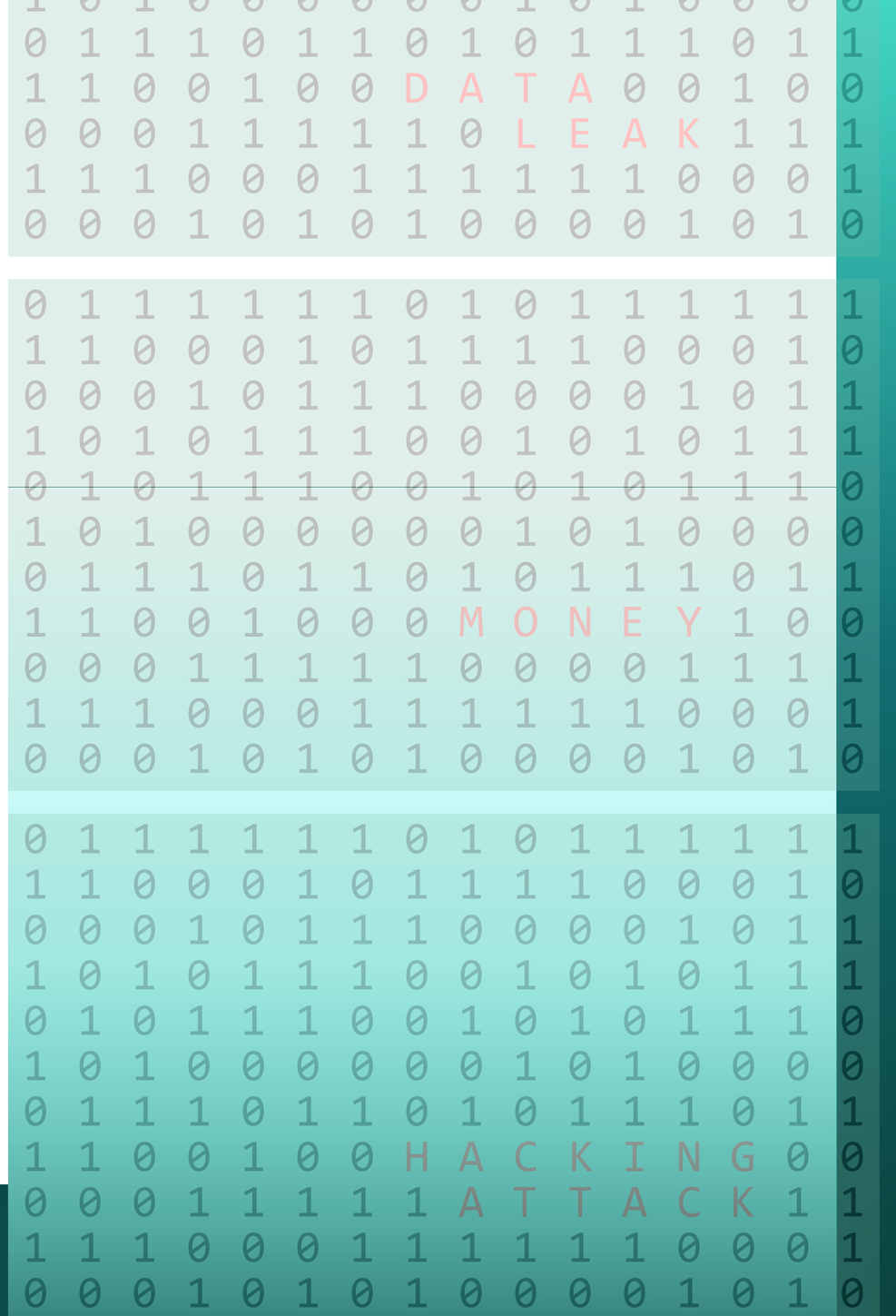
→ *Two equations → Two unknowns (PU.1 and GATA-1 concentrations)*

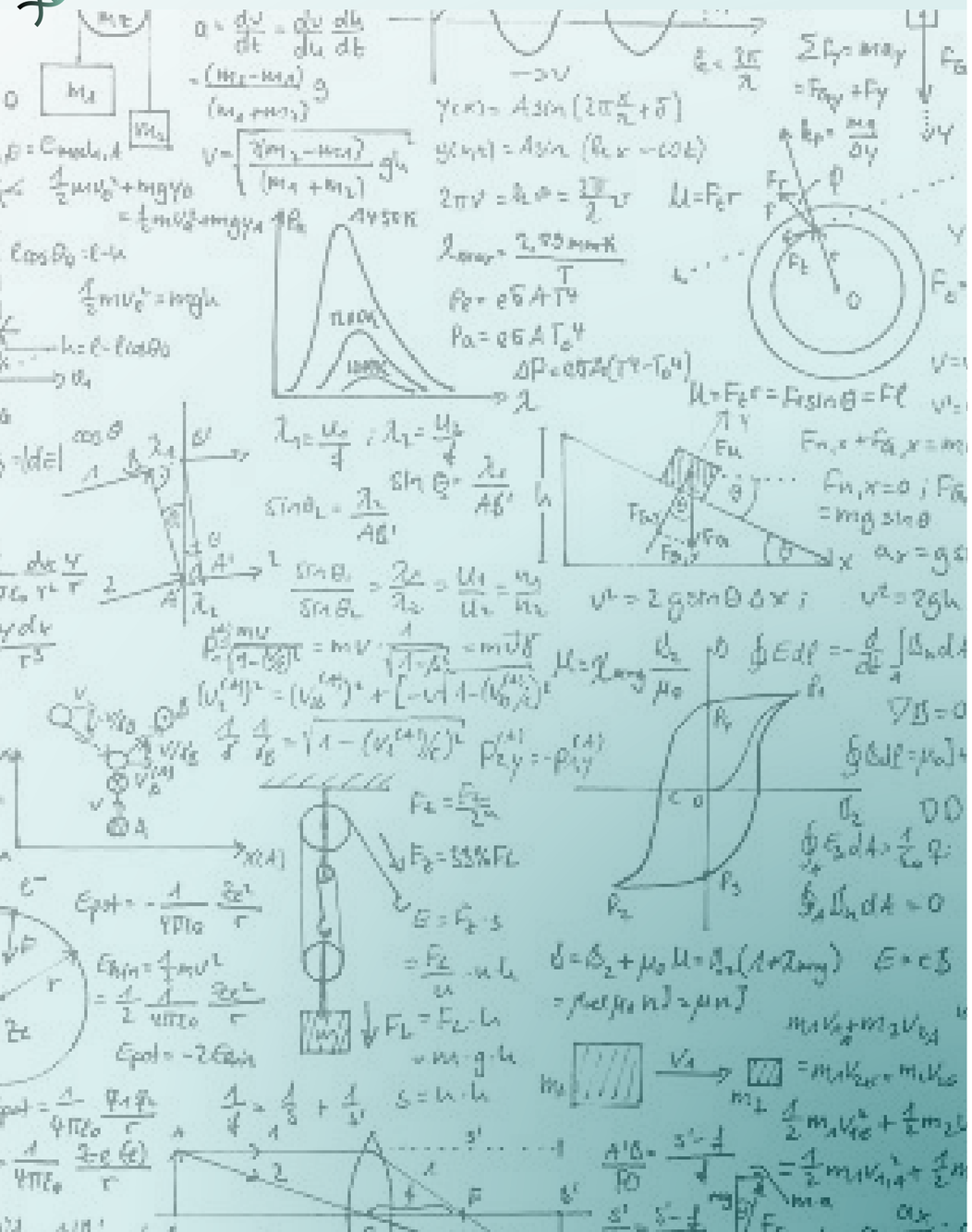
- The system tracks how these two transcription factors change over time.

- To analyze the model, we use:

→ *Numerical solvers → Machine learning methods*

Explanation of the ODE Model





Overview of the ODE Model

$$\frac{d[G]}{dt} = \frac{a_1[G]^n}{\theta_{a1}^n + [G]^n} + \frac{b_1\theta_{b1}^m}{\theta_{b1}^m + [G]^m[P]^m} - k_1[G]$$

$$\frac{d[P]}{dt} = \frac{a_2[P]^n}{\theta_{a2}^n + [P]^n} + \frac{b_2\theta_{b2}^m}{\theta_{b2}^m + [G]^m[P]^m} - k_2[P]$$

Variables:

- **[G], [P] :** Normalized expression levels of GATA-1 and PU.1
- **t:** Time
- **a_1, a_2:** Self-activation rates
- **b_1, b_2:** Cross-inhibition influence on basal activation
- **θa_1, θa_2, θb_1, θb_2:** Thresholds of activation/inhibition
- **k_1, k_2:** Degradation rates
- **n, m:** Hill coefficients (control response sharpness)

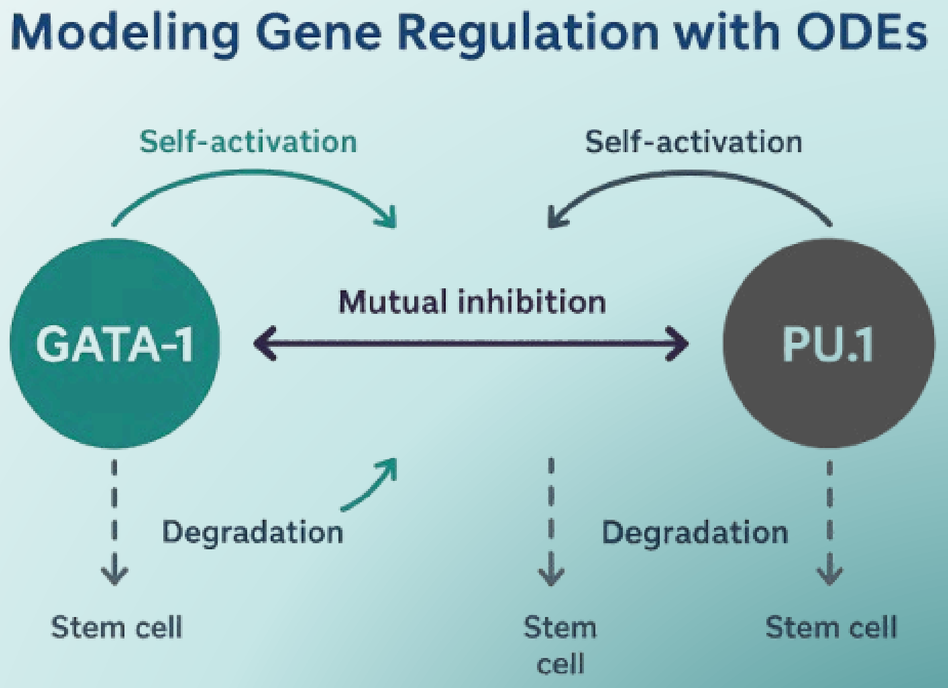
Each ODE consists of three biologically significant components:

- **Self-activation:** Each gene boosts its own expression.
- **Mutual inhibition:** GATA-1 and PU.1 suppress each other.
- **Degradation:** Natural decay of gene expression over time.

The system can settle into one of two stable gene expression states.

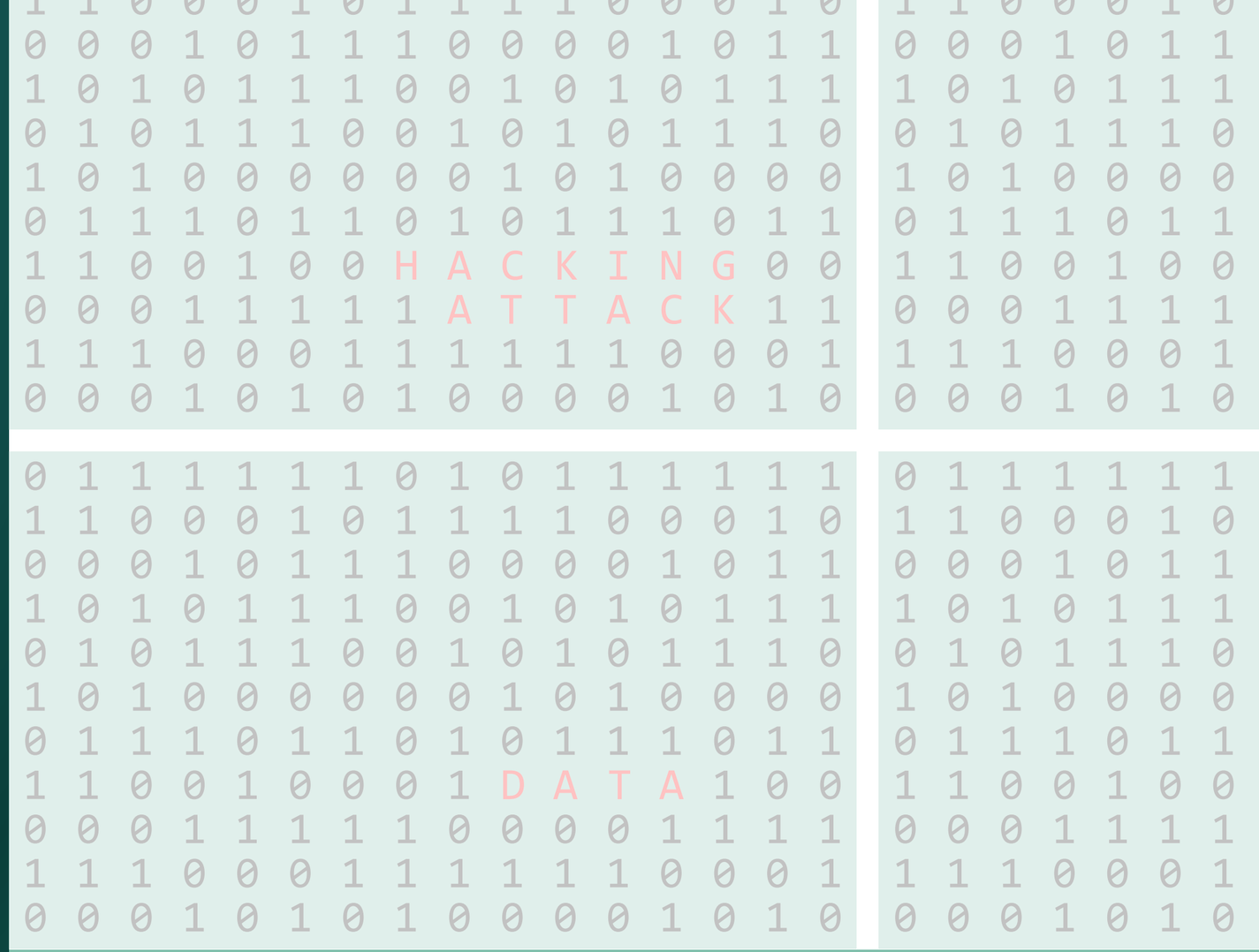
GATA-1 dominance → Erythroid cells (e.g., red blood cells)

PU.1 dominance → Myeloid cells (e.g., macrophages)



SBEG108 NUMERICAL METHODS IN BME

Numerical and Learning-based solution steps



LSODES

Numerical Solution

What is LSODES?

LSODES (Livermore Solver for ODEs with Sparse Matrices) is an implicit, adaptive step size solver based on Backward Differentiation Formulas (BDF) an implicit method for (stiff systems) , and Adams-Bashforth an explicit method for (non-stiff systems), offering strong stability for stiff problems. The simplest BDF is the Backward Euler method which is also called BDF1 (1st-order BDF , and it's formula :

$$\frac{y_n - y_{n-1}}{\Delta t} = f(t_n, y_n)$$

Work Flow

1.DEFINE THE ODE SYSTEM

Create a custom function like (stem_1) to describe the system of differential equations governing the behavior of variables like G(t) and P(t).

2.INITIALIZE PARAMETERS AND INITIAL CONDITIONS

Set model parameters and initial values:

- G(0)=1 , P(0)=1
- Example parameters: n=4 , m=1

3.RUN THE SOLVER (LSODES)

- Use the lsodes() function from the deSolve package in R to integrate the system over the desired time interval.
- This solver is particularly suited for stiff systems with sparse Jacobians.

TRAPEZOIDAL

Numerical Solution

What is TRAPEZOIDAL ?

To improve accuracy over the basic Euler method, we implemented the trapezoidal rule for solving the nonlinear ODE system. The Euler approximation:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$

is replaced by the more accurate trapezoidal rule:

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$$

which is implicit in yn+1 and requires fixed-point iteration at
each time step.

Work Flow

1.INITIALIZE

- Set convergence tolerance 10^{-6}
- Set number of steps 200
- Set maximum iterations per step (100)

2.INITIAL GUESS

predict yn+1 using Euler's method :

$$y_{\text{guess}} = y_n + h \cdot f(t_n, y_n)$$

3.APPLY FIXED-POINT ITERATION

1-Evaluate:

$$y_{\text{guess}} = y_n + h \cdot f(t_n, y_n) f_{\text{guess}} = f(t_n + h, y_{\text{guess}})$$

2-Compute:

$$y_{\text{guess}} = y_n + h \cdot f(t_n, y_n)$$

3- Check:

IF : $\|y_{\text{next}} - y_{\text{guess}}\| < 10^{-6}$

- stop iteration.
- Else, set yguess=ynext and repeat.

Work Flow

4.UPDATE SOLUTION AND TIME

Update: Set yn+1 = ynext, increment t, and continue to Final time.

FINALLY

5.RECORD PERFORMANCE METRICS

- Measure excution time.
- Calculate accuracy.

RADAU

Numerical Solution

What is RADAU?

The Radau method is a type of implicit Runge-Kutta method designed specifically for stiff ODE systems.
It is particularly effective when
The system has nonlinear terms like G^n , G^m , P^m , and Parameters such as k1,k2,... operating on very different time scales.

Radau allows:

- Stable integration with relatively large time steps,
- High accuracy, and
- Built-in error control and step-size adaptation.

Work Flow

1.INITIALIZE

- Choose the initial time step h:

If no t_eval $= \dfrac{t_{\text{end}} - t_{\text{start}}}{100}$

If t_eval is used $= \dfrac{t_{\text{end}} - t_{\text{start}}}{100}$

- Set tolerance to 10 ($^{-8}$)
- Max iterations per step: 50

COMPUTE RESIDUALS

$$R_i = Y_i - y - h\sum_j A_{ij}F_j$$

2.INITIAL GUESS

For 3-stage Radau method:

Y1=Y2=Y3=yn

3.PERFORM NEWTON ITERATION AT EACH STEP

1-Evaluate :

$$f(t + c_i h, Y_i)$$

2-Compute residuals and the Jacobian matrix

3-Solve the linear system to update Yi , and iterate until convergence $J\delta = -R$

Work Flow

4. UPDATE THE SOLUTION

$$Y \leftarrow Y + \delta$$

$$y_{n+1} = y + h \sum b_i f(t + c_i h, Y_i)$$

5.ESTIMATE ERROR AND CONTROL STEP SIZE

$$error = h \sum (b_i - \hat{b}_i) \cdot f(Y_i)$$

$$\text{error\_norm} = \frac{\|\text{error}\|}{\max(1, \|y\|)}$$

FINALLY

6.REPEAT!

- Continue integration until reaching final time t=5.

7.RECORD PERFORMANCE METRICS

- Count number of function calls (ncalls).
- Measure total computation time.

PINNS Learning-Based Solution

Physics-Informed Neural Networks

are deep learning models that integrate physical laws (like ODEs or PDEs) into their training. Instead of learning only from data, PINNs also minimize how much they violate known physics, ensuring the model fits both the data and the governing equations.

Key Features

- **Physics-based Loss:** The loss function includes terms that penalize violations of the governing equations (e.g., ODE or PDE residuals) and initial/boundary conditions.
- **Generalization:** By enforcing physical laws, PINNs can generalize better, even with limited data.
- **Automatic Differentiation:** PINNs leverage frameworks like PyTorch or TensorFlow to compute derivatives needed for the physics loss using automatic differentiation.

LOSS COMPUTATION:

The PINN computes its loss as the sum of two main parts:

Physics loss: Measures how well the model satisfies the ODE by penalizing residuals using automatic differentiation and MSE.

```python
# Physics Loss
def ode_residuals(model, t, a1, a2):
    t.requires_grad = True
    G_pred, P_pred = model(t).split(1, dim=1)

    dG_dt = torch.autograd.grad(G_pred, t, torch.ones_like(G_pred), create_graph=True)[0]
    dP_dt = torch.autograd.grad(P_pred, t, torch.ones_like(P_pred), create_graph=True)[0]

    f1 = (a1 * G_pred**n / (tha1**n + G_pred**n)) + (b1 * thb1**m / (thb1**m + G_pred**m * P_pred**n
    f2 = (a2 * P_pred**n / (tha2**n + P_pred**n)) + (b2 * thb2**m / (thb2**m + G_pred**m * P_pred**n

    res1 = dG_dt - f1
    res2 = dP_dt - f2

    return res1, res2, G_pred, P_pred
```

Initial condition loss:

Initial condition loss: Ensures the model's predictions match known initial values using mean squared error.

```python
# Case-specific loss computation
loss_phys = w_phys * (loss_fn(resG, torch.zeros_like(resG)) + loss_fn(resP, torch.zeros_like(resP)))

# Initial condition loss
t0 = torch.tensor([[0.0]], dtype=torch.float32, requires_grad=True).to(device)
G0_pred, P0_pred = model(t0).split(1, dim=1)
loss_ic = w_ic * (loss_fn(G0_pred, torch.tensor([[G0]], dtype=torch.float32, device=device)) +
                  loss_fn(P0_pred, torch.tensor([[P0]], dtype=torch.float32, device=device)))
```

LOSS COMPUTATION:

The model is trained by:

- **Forward Pass: Predict values across all time points using the neural network.**
- **Loss Calculation: Compute total loss = physics loss + initial condition loss.**
- Backward Pass: Call backward() to compute gradients for model optimization.

```
loss = loss_phys + loss_ic
loss.backward()
```

FEATURE 1: CASE-DEPENDENT NETWORK ARCHITECTURE

- **What I changed:**

For the simple case, kept the small network (3 hidden layers).

For the complex case, switched to a deeper, wider network (4 hidden layers, more neurons).

- **Benefit:**

Better accuracy and convergence for hard cases, without unnecessary complexity for easy ones.

Why?

Simple problems don’t need large networks—smaller models are faster and less prone to overfitting.

Complex, nonlinear problems require greater capacity to capture intricate dynamics.

The Model Architecture

```python
# PINN Model
class PINN(nn.Module):
    def __init__(self, case_num):
        super().__init__()
        # Case-specific architecture
        if case_num == 1:
            # Simpler architecture for Case 1
            self.net = nn.Sequential(
                nn.Linear(1, 128),
                nn.Tanh(),
                nn.Linear(128, 128),
                nn.Tanh(),
                nn.Linear(128, 64),
                nn.Tanh(),
                nn.Linear(64, 2)
            )
        else:
            # More complex architecture for Case 2
            self.net = nn.Sequential(
                nn.Linear(1, 256),
                nn.Tanh(),
                nn.Linear(256, 256),
                nn.Tanh(),
                nn.Linear(256, 256),
                nn.Tanh(),
                nn.Linear(256, 128),
                nn.Tanh(),
                nn.Linear(128, 2)
            )
```

FEATURE 2: XAVIER (GLOROT) INITIALIZATION

What I changed:

- Added Xavier initialization for all linear layers.

Benefit:

- The model starts training with more balanced weights, leading to faster and more stable convergence.

Why?

- Default initialization can cause vanishing or exploding gradients, especially in deep networks.

- Xavier initialization keeps the scale of gradients roughly the same across all layers, improving training stability.

FEATURE 3: GRADIENT CLIPPING

What I changed:

- Added gradient clipping
  (with different thresholds for each case).

Benefit:

- Training remains stable even for challenging cases, and the optimizer can use larger learning rates safely.

Why?

- In deep or stiff systems, gradients can become extremely large, causing unstable updates (exploding gradients).

- Gradient clipping limits the maximum allowed gradient norm, preventing instability.

FEATURE 4: CURRICULUM LEARNING (FOR COMPLEX CASE)

What I changed:

- For the hard case, trained the model on small time intervals first, then gradually increased the interval.

Benefit:

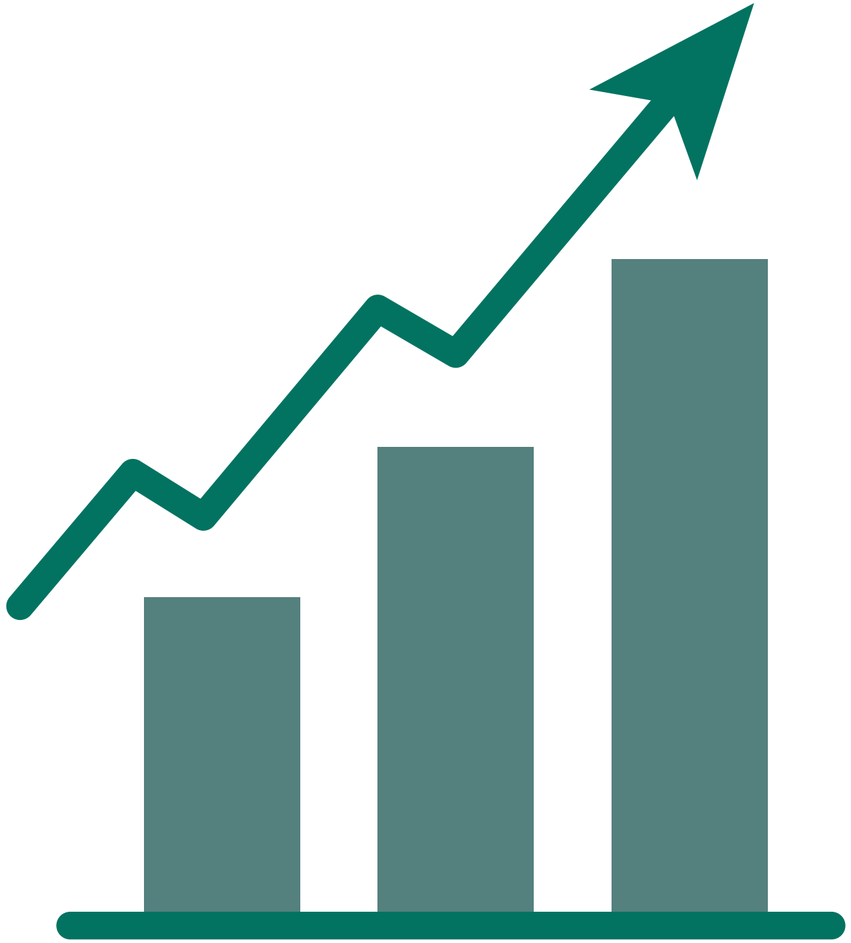
- Faster, more reliable convergence for the complex scenario; avoids poor local minima.

Why?

- Learning the full complex problem at once is hard; starting with easier subproblems helps the network learn basic patterns first.

- Mimics how humans learn complex tasks step by step.

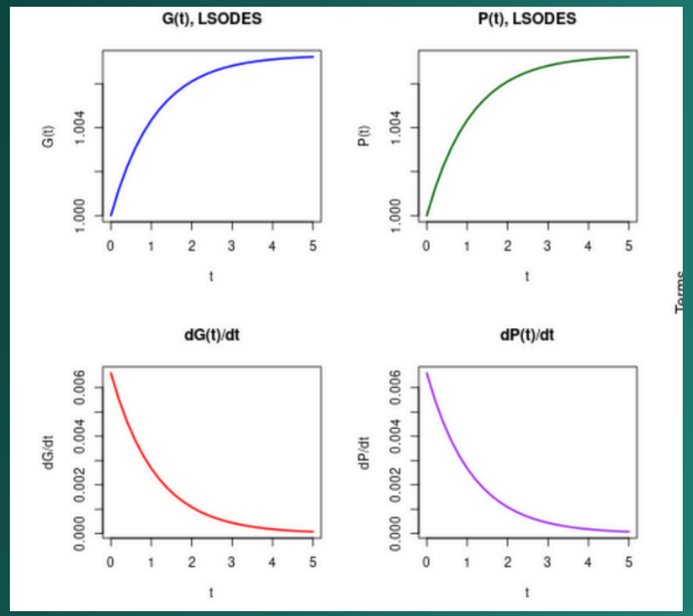
Results



Using *deSolve* Package (Base Case) *Chapter Method*

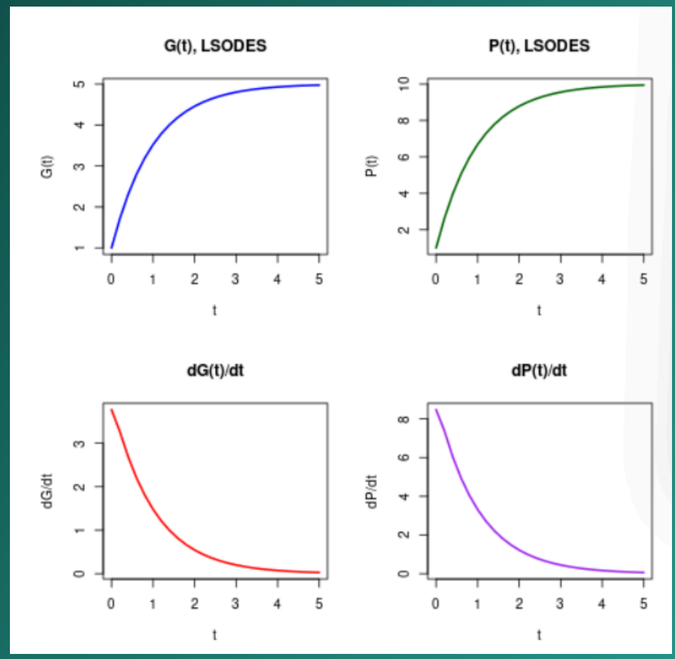
- **Case 1 (Symmetric) Results:**

Time	G(t)	P(t)	dG/dt	dP/dt
0.00	1.000	1.000	0.007	0.007
1.00	1.004	1.004	0.003	0.003
3.00	1.007	1.007	0.000	0.000
5.00	1.007	1.007	0.000	0.000



- **Case 2 (Asymmetric) Results:**

Time	G(t)	P(t)	dG/dt	dP/dt
0.00	1.000	1.000	3.771	8.477
1.00	3.521	6.685	1.480	3.318
3.00	4.801	9.553	0.200	0.449
5.00	4.974	9.941	0.027	0.061



Demonstrations and results of solutions

Case 1 – Symmetric Results

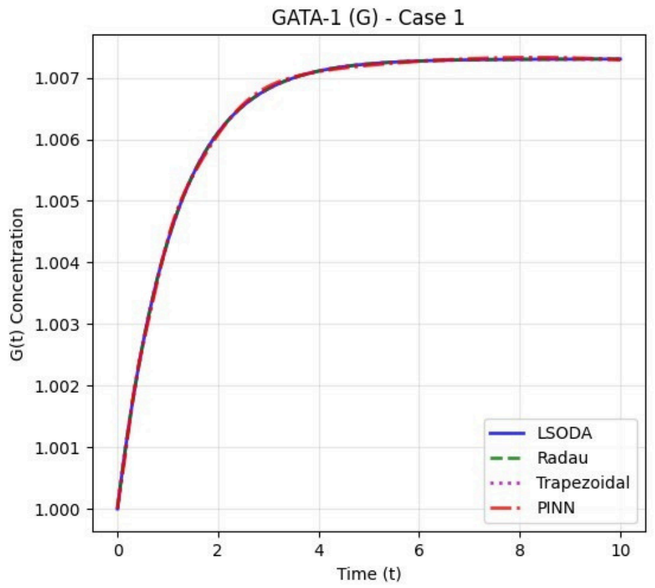
Numerical Methods:
- All methods converge quickly to steady state.
- G(t), P(t) remain nearly constant.

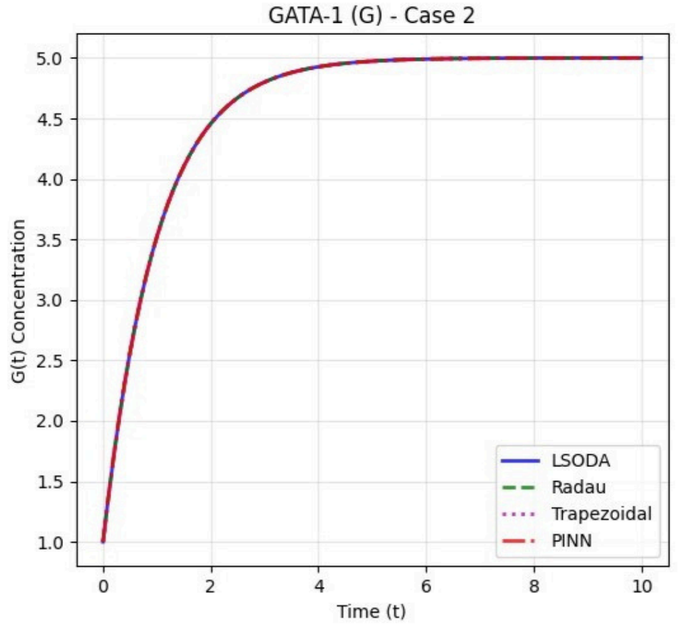
PINNs:
- Learns flat solution with very low loss.
- Interpretation: Cells remain undifferentiated – minimal dynamic activity.

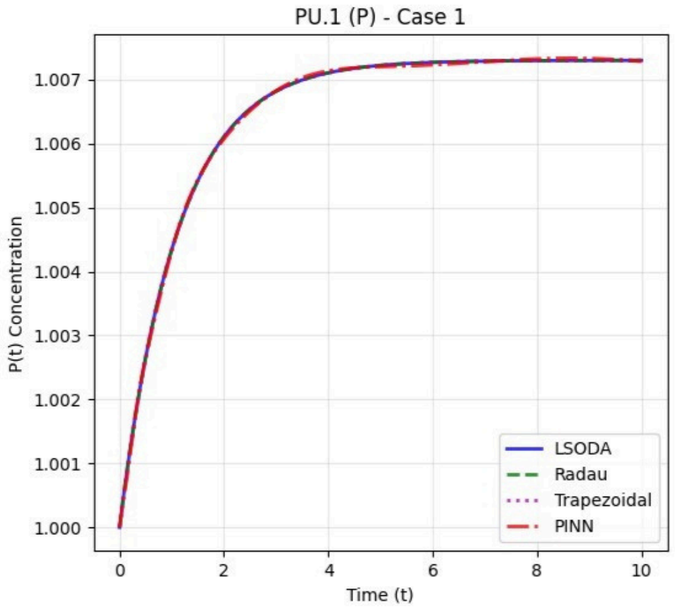
Case 2 – Asymmetric Results

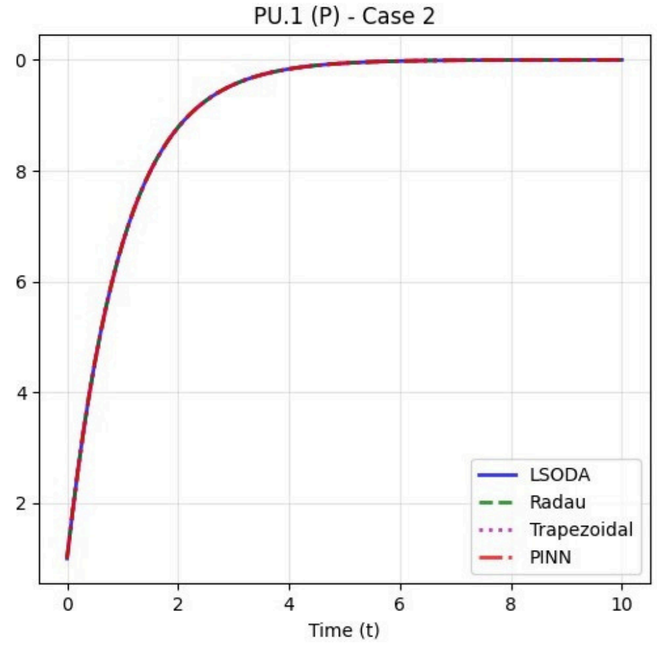
Numerical Methods:
- Accurate capture of stiff dynamics.
- Handles well but less precise.

PINNs:
- Slower convergence, but captured overall trends.
- Grows rapidly early on, then stabilizes.









RESULTS

To evaluate the reliability and effectiveness of different solution technique, we performed a comparative analysis between the three approaches.

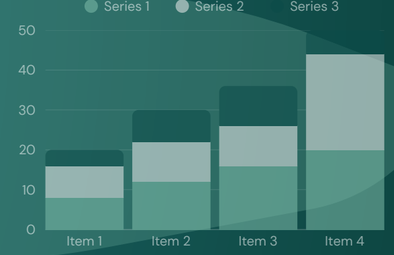
For each method, we computed several error metrics for the variables G(t) and P(t) across the time domain, including:

MSE
(MeanSquared Error)

MAE
(Mean Absolute Error)

R^2score
indicating goodness of fit

RESULTS



- **Accuracy of G(t) – Case 1 (Symmetric):**

Method	MSE	MAE	R^2
Radau	2.74×10^{-14}	0.0000	1.0000
Trapzoidal	8.00×10^{-14}	0.0000	1.0000
PINN	7.26×10^{-10}	0.0000	0.9996

- **Accuracy of P(t) – Case 1 (Symmetric):**

Method	MSE	MAE	R^2
Radau	2.74×10^{-14}	0.0000	1.0000
Trapzoidal	8.00×10^{-14}	0.0000	1.0000
PINN	9.24×10^{-10}	0.0000	0.9996

Accuracy of P(t) – Case 1 : As with G(t), the predictions for P(t) are highly accurate across all methods.

- **Accuracy of G(t) – Case 2 (Asymmetric):**

Method	MSE	MAE	R^2
Radau	2.32×10^{-14}	0.0000	1.0000
Trapzoidal	1.14×10^{-8}	0.0001	1.0000
PINN	4.70×10^{-8}	0.0001	0.9996

- **Accuracy of P(t) – Case 2 (Asymmetric):**

Method	MSE	MAE	R^2
Radau	1.29×10^{-13}	0.0000	1.0000
Trapzoidal	4.10×10^{-7}	0.0002	1.0000
PINN	5.90×10^{-7}	0.0004	0.9996

Accuracy of P(t) – Case 2 : This table confirms the trend seen with G(t) in Case 2. All methods remain stable, but slight error differences emerge.

COMPUTATION TIME

Computational Performance:

As expected, traditional numerical solvers complete in milliseconds to seconds. PINNs, which require intensive training and optimization, are significantly slower, limiting their use in real-time applications.

Training Time

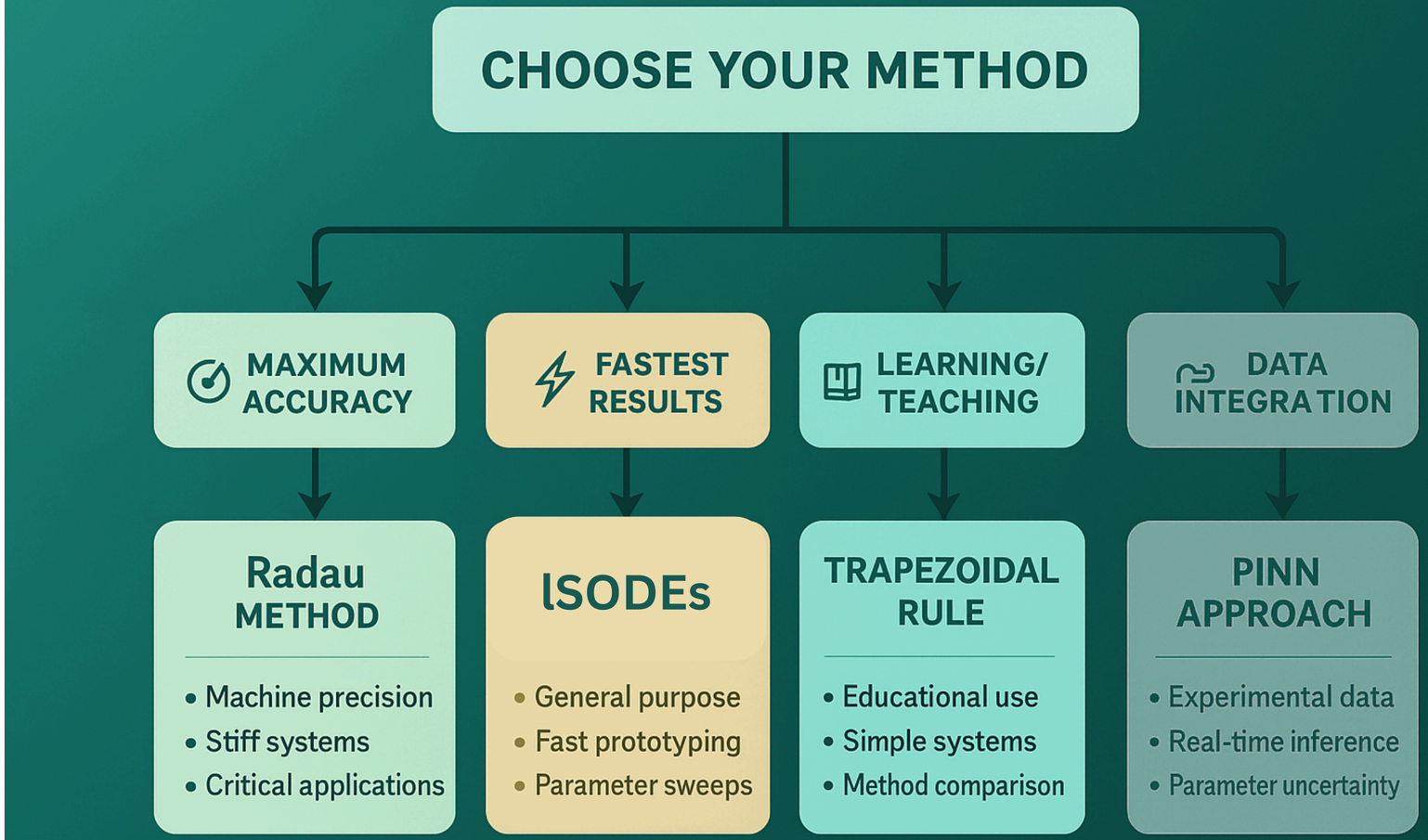
CASE	METHOD	TIME
1	LSODES	0.0022
	RADAU	0.0519
	TRAPEZOIDAL	0.0037
	PINN	0.00056(197.73)
2	LSODES	0.0033
	RADAU	0.0519
	TRAPEZOIDAL	0.0062
	PINN	0.0006(370.63)

Note:

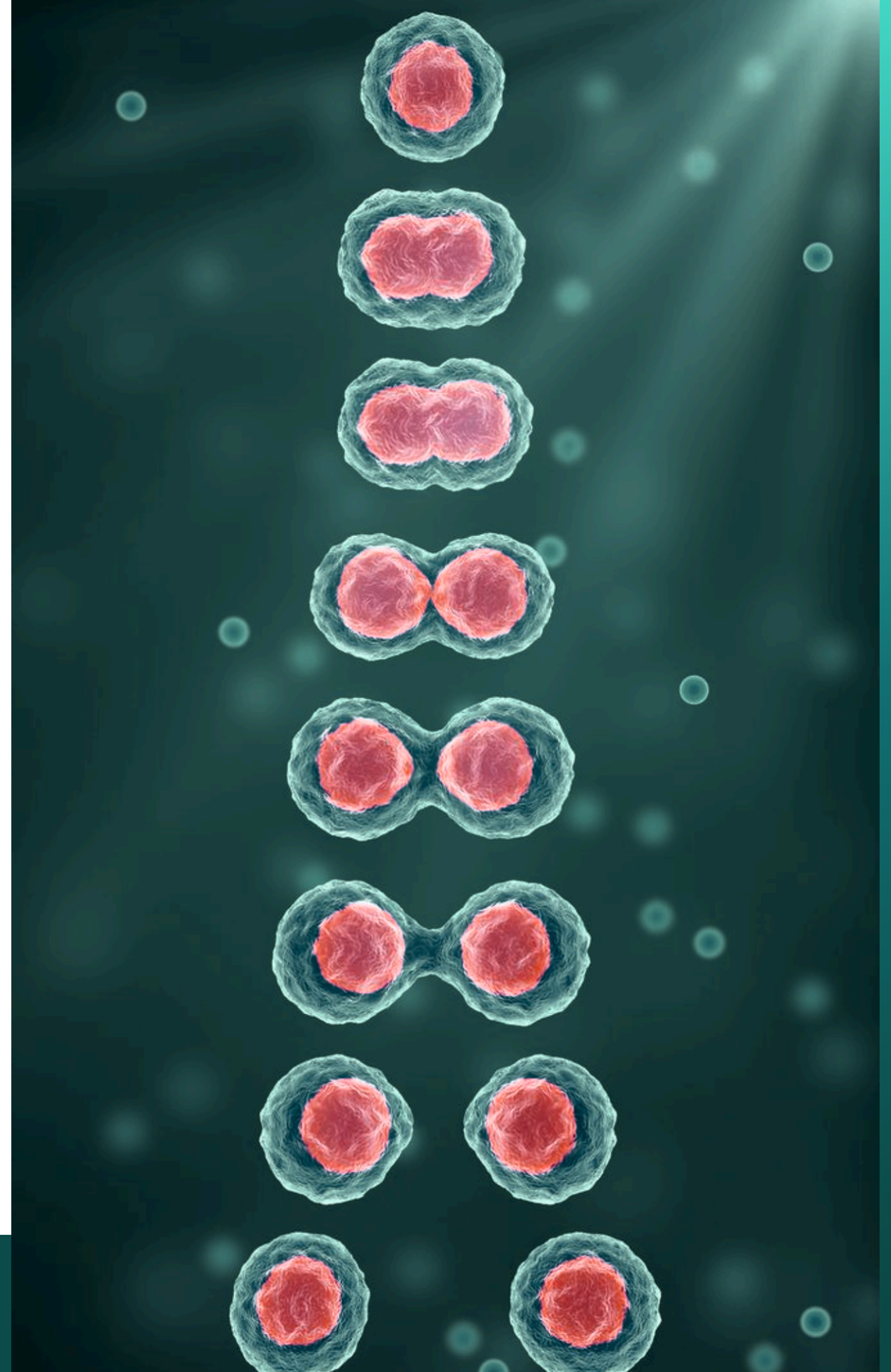
For the PINNs : 197.73 is the total of training and testing time

0.00056 is the testing time

To Conclude


CHOOSE YOUR METHOD
MAXIMUM ACCURACY
FASTEST RESULTS
LEARNING/ TEACHING
DATA INTEGRATION
Radau METHOD
• Machine precision
• Stiff systems
• Critical applications
lSODEs
• General purpose
• Fast prototyping
• Parameter sweeps
TRAPEZOIDAL RULE
• Educational use
• Simple systems
• Method comparison
PINN APPROACH
• Experimental data
• Real-time inference
• Parameter uncertainty

Future Work and Improvements



How Can we improve the the model, methods, and results

Using simpler architecture
that can reduce parameter sensitivity

Transfer Learning:
Leveraging pretrained models on similar systems

***Adaptive Time Stepping*:**
Incorporate dynamic step size control based on local error estimation to enhance efficiency and stability.

***Adaptive Sampling*:**
Selecting collocation points based on solution gradients

Survey of Work on Stem Cell Modeling

References:

- [1] S. Huang, Y.-P. Guo, G. May, and T. Enver,
"Bifurcation dynamics in lineage-commitment in bipotent progenitor cells,"
Exp. Hematol., vol. 35, no. 11, pp. 1707–1718, Nov. 2007.

- [2] C. Duff, K. Smith-Miles, L. Lopes, and T. Tian,
"Mathematical modelling of stem cell differentiation: The PU.1–GATA-1 interaction,"
BMC Systems Biology, vol. 6, no. 1, 2012.

- [3] T. Tian et al.,
"Mathematical modeling of GATA-switching for regulating the differentiation of hematopoietic stem cells,"
BMC Bioinformatics, vol. 9, Suppl 9, 2008.

- [4] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang,
"Physics-informed machine learning,"
Nature Reviews Physics, vol. 3, no. 6, pp. 422–440, Jun. 2021.

Tools:

- R – deSolve Package
- PyTorch – PINNs Implementation

THANKS FOR LISTENING