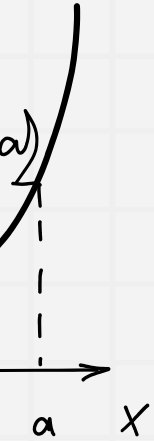
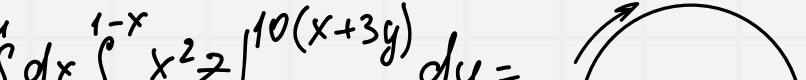
Influenza with Vaccination and Diffusion

Numerical Methods in Biomedical Applications

Ashar Seif Al-Nasr
Mariam Ashraf Mohamed
Mohamed Hisham
Mariam hussein khalil
Peter Tharwat Ibrahem





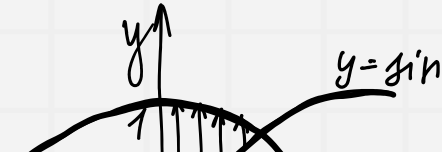
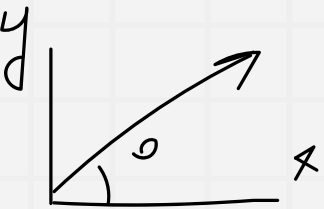
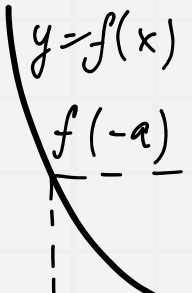


Table of Contents

01

Chapter Replication





Predicted Variables

- **What is Predicted?**
  - The model predicts the **spatiotemporal dynamics** of the influenza epidemic:
    - The number of **Susceptible**, **Vaccinated**, **Exposed**, **Infected**, and **Recovered** individuals over time and space.
  - Specifically, it predicts:
    - How the disease spreads across the spatial domain.
    - The impact of vaccination on reducing the spread of the disease.
- **Output**:
  - The values of $S(x,t)$, $V(x,t)$, $E(x,t)$, $I(x,t)$, and $R(x,t)$ at each spatial point x and time t.

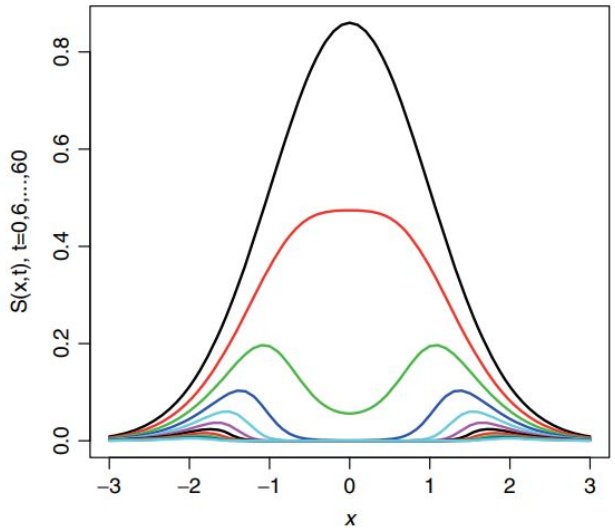


Figure 6.2 $S(x,t)$ versus x, $t = 0, 6, \ldots, 60$.

Numerical Method - Method of Lines (MOL)

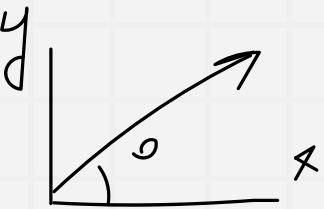
- **What is the Method of Lines?**
  - The **Method of Lines (MOL)** is a numerical technique for solving PDEs by discretizing the spatial derivatives and converting the PDEs into a system of ODEs.
- **Steps in MOL**:
  - **Spatial Discretization**:
    - The spatial domain is divided into a grid, and the spatial derivatives are approximated using finite difference methods (e.g., central differences).
    - In the chapter, the spatial derivatives are discretized using fourth-order finite difference approximations (e.g., dss004 and dss044).
  - **ODE System**:
    - The PDEs are converted into a system of ODEs in time.
  - **Time Integration**:
    - The resulting ODE system is solved using an ODE solver (e.g., `lsoda` in R or `odeint` in Python).

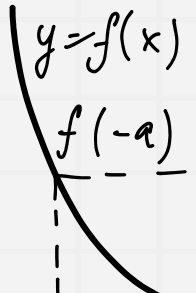
- For example, the PDE $\frac{\partial S}{\partial t} = D\frac{\partial^2 S}{\partial x^2}$ becomes:

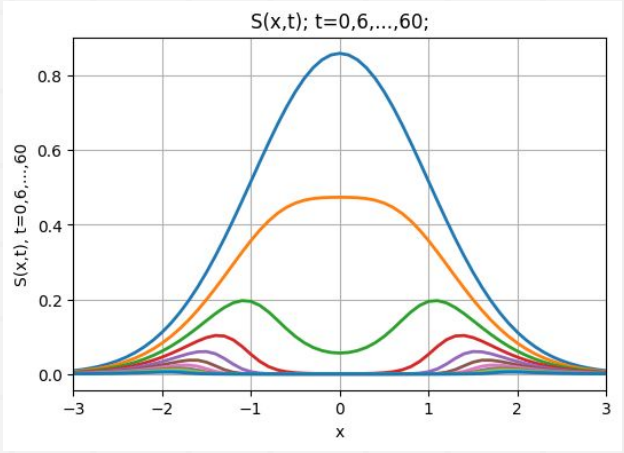
$$\frac{dS_i}{dt} = D\frac{S_{i-1} - 2S_i + S_{i+1}}{\Delta x^2}$$

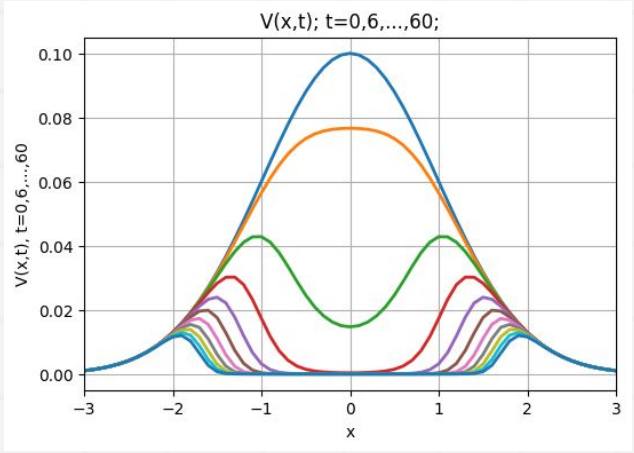
1.1

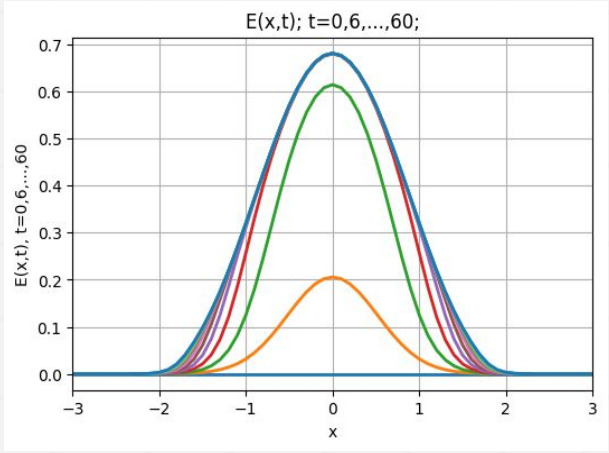
Chapter Replication
R Version

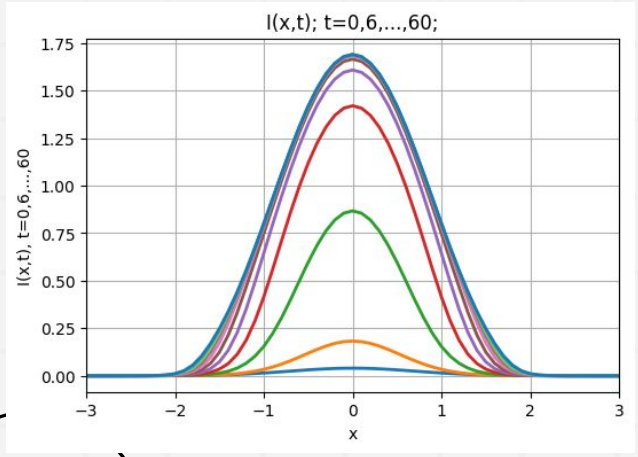


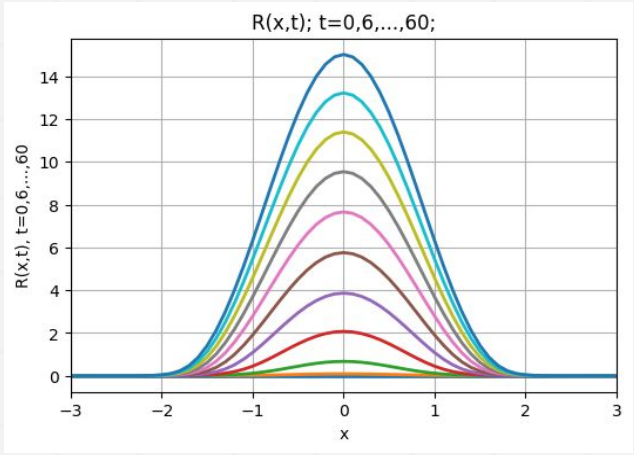


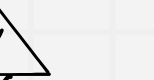

S(x,t); t=0,6,...,60;
S(x,t), t=0,6,...,60
x


V(x,t); t=0,6,...,60;
V(x,t), t=0,6,...,60
x

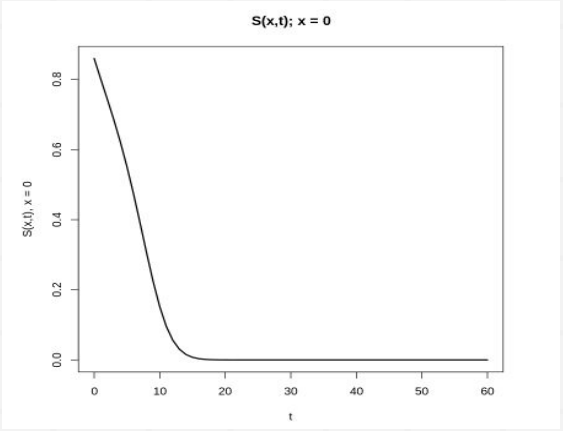

E(x,t); t=0,6,...,60;
E(x,t), t=0,6,...,60
x

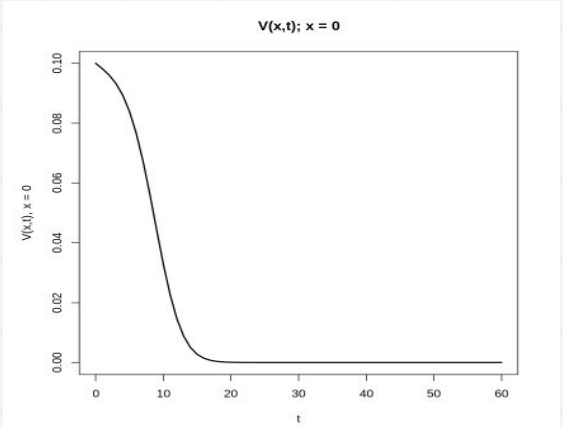

I(x,t); t=0,6,...,60;
I(x,t), t=0,6,...,60
x

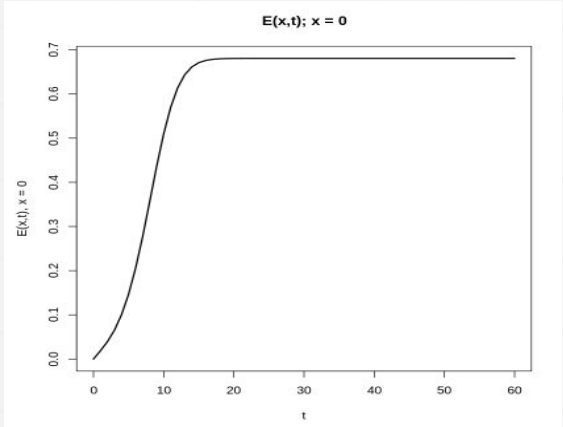

R(x,t); t=0,6,...,60;
R(x,t), t=0,6,...,60
x

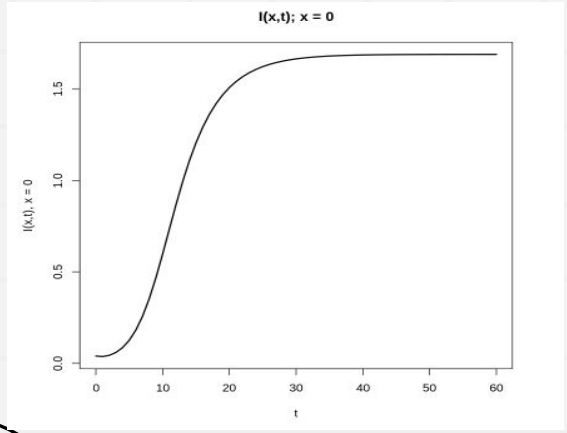


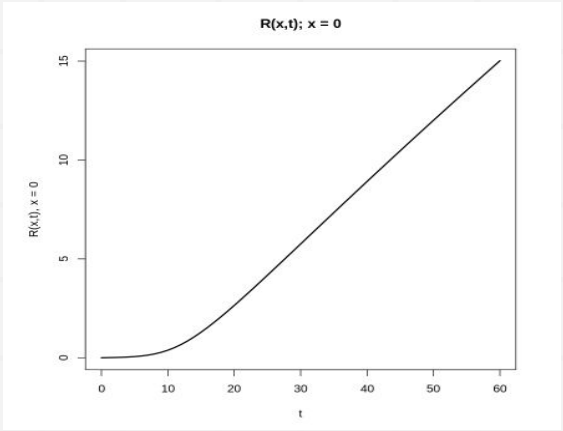
fdy


S(x,t); x = 0
S(x,t), x = 0
t


V(x,t); x = 0
V(x,t), x = 0
t

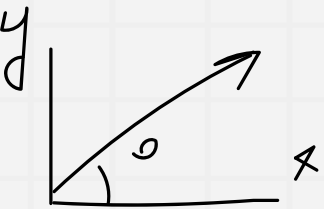

E(x,t); x = 0
E(x,t), x = 0
t

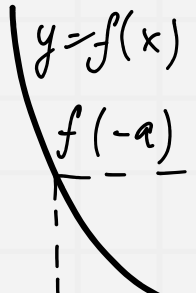

I(x,t); x = 0
I(x,t), x = 0
t

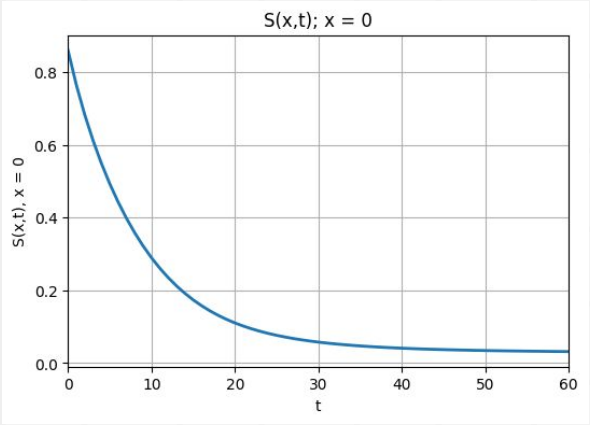

R(x,t); x = 0
R(x,t), x = 0
t

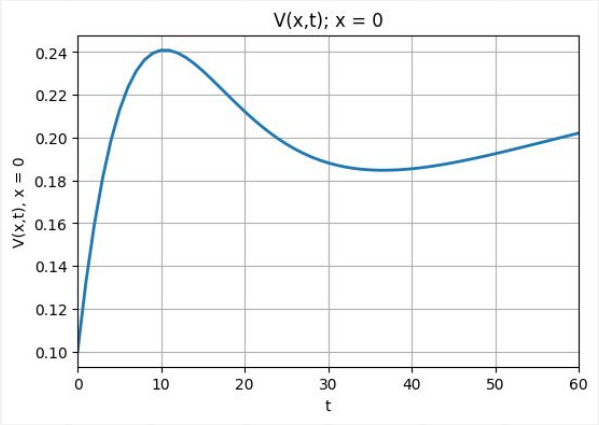
fdy

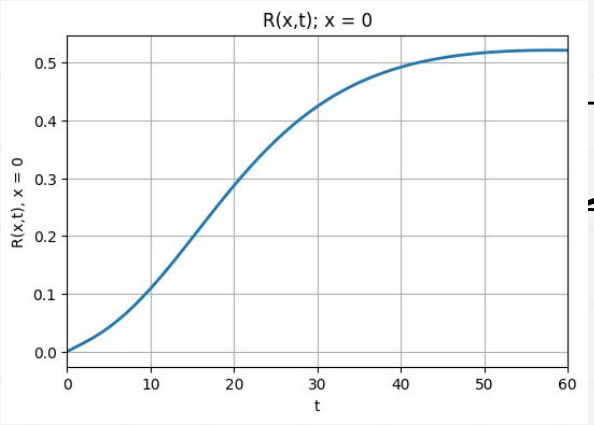
1.2
Chapter Replication Python Version

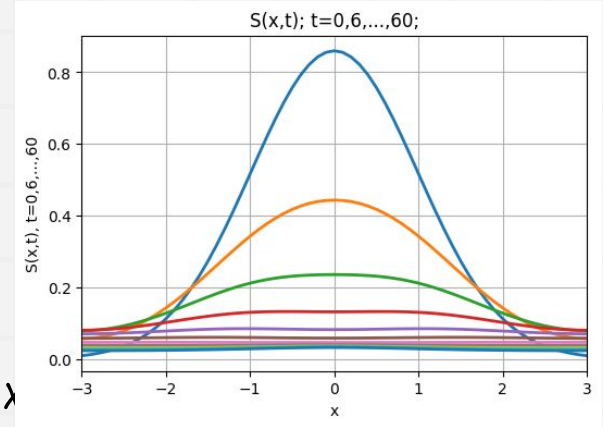


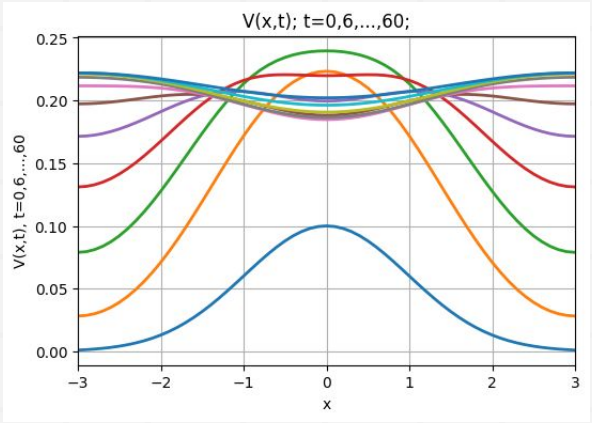


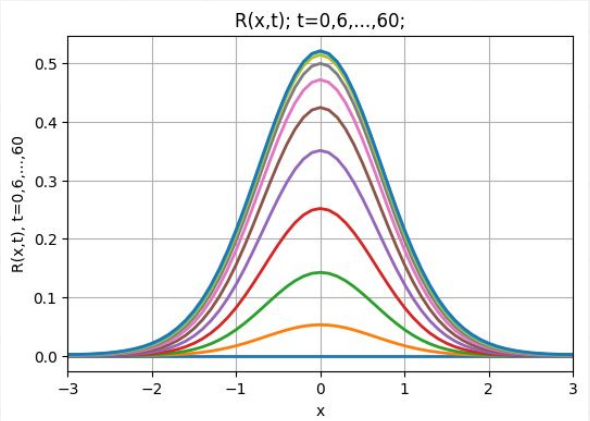

S(x,t); x = 0
S(x,t), x = 0
t


V(x,t); x = 0
V(x,t), x = 0
t


R(x,t); x = 0
R(x,t), x = 0
t

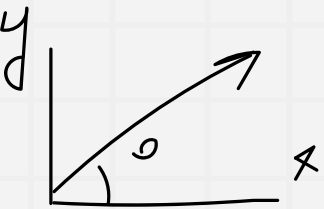

S(x,t); t=0,6,...,60;
S(x,t), t=0,6,...,60
x

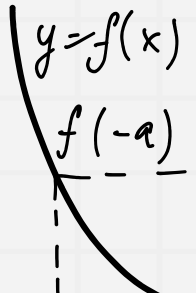

V(x,t); t=0,6,...,60;
V(x,t), t=0,6,...,60
x


R(x,t); t=0,6,...,60;
R(x,t), t=0,6,...,60
x

02

ML / DL Models





ML / DL Models Problem Definition

Objective :

The **goal** is to map the **spatial-temporal evolution of disease** (computed using **MOL on PDEs**) into a **Machine Learning (ML) or Deep Learning (DL) model** that can **predict future outbreaks without solving PDEs explicitly**.

Input Features

X , t , S0, V0,E0,I0,R0
(Initial Conditions)

Target Output

S,V,E,I,R at time t

$$(x,t) \rightarrow [S(x,t), V(x,t), E(x,t), I(x,t), R(x,t)]$$

ML as an Approximate Solver for Numerical Methods

- ML can **approximate** solutions to ODEs/PDEs based on **previously computed numerical solutions**.
- **Random Forests (RFs) or Gradient Boosting** trained on numerical solutions can **predict future states without running expensive numerical solvers**.

Pros

Faster than numerical solvers.

Cons

May not generalize to unseen conditions.

Time Series Forecasting Models

- The evolution of **disease spread** follows a **spatiotemporal pattern**
  - **Time dependency**: Future values of S,V,E,I,R depend on their past values.
  - **Spatial dependency**: Infection spreads across different locations.

- Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) are **recurrent neural networks (RNNs)** designed to model **time-series data**.

Pros

Handle **long-term dependencies** (delayed outbreak effects).
Work well with **irregular time steps**.
Capture **non-linear relationships** in disease transmission.

Cons

GRU may struggle with very long-term dependencies
LSTM is Computationally expensive

Physics-Constrained Learner (PINNs)

- Instead of learning from **past numerical solutions**, **Physics-Informed Neural Networks (PINNs)** solve PDEs **directly by enforcing physics constraints in their loss function**.

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \lambda \mathcal{L}_{\text{physics}}$$

This term enforces **compliance with the differential equations** governing the SVEIR model.

$$\mathcal{L}_{\text{physics}} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{dS}{dt} + 0.514SI + 0.05S \right)^2 + \left(\frac{dI}{dt} - 0.1E + 0.1I \right)^2$$

To compute the derivatives $\frac{dS}{dt}$ and $\frac{dI}{dt}$, **automatic differentiation** in PyTorch is used:

$$\frac{dS}{dt} = \frac{\partial S}{\partial t}, \quad \frac{dI}{dt} = \frac{\partial I}{\partial t}$$

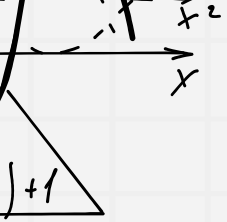
The physics loss ensures that the **PINN model predictions satisfy the epidemic constraints**, even when training data is limited.

These equations capture the core **disease dynamics**:

- How susceptible individuals decrease (due to infection and vaccination).
- How the infected population evolves (from exposure and recovery).

Pros **Generalizes to unseen conditions**.

Cons **More difficult to train**.

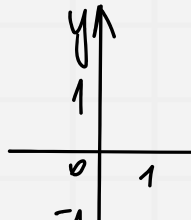


Evaluation Criteria Between Models

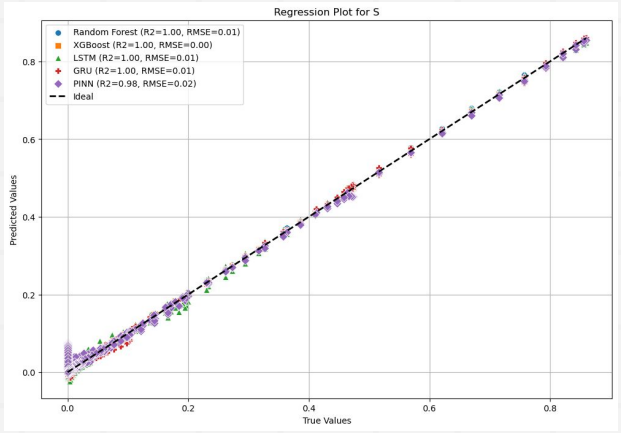
Train Data Vs Numerical Model → HoldOut Test Data → Generalization → Gradual Effect

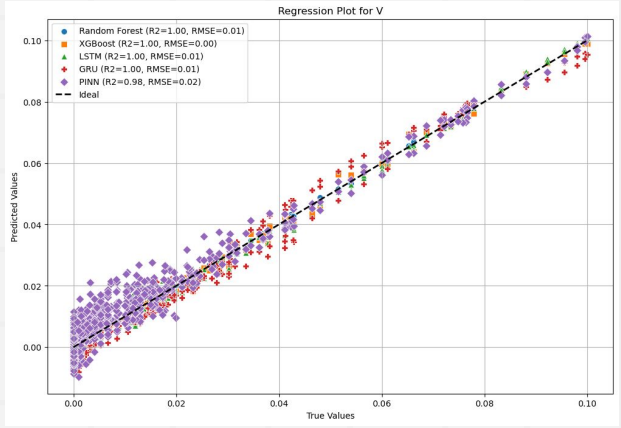
Compare Based On:

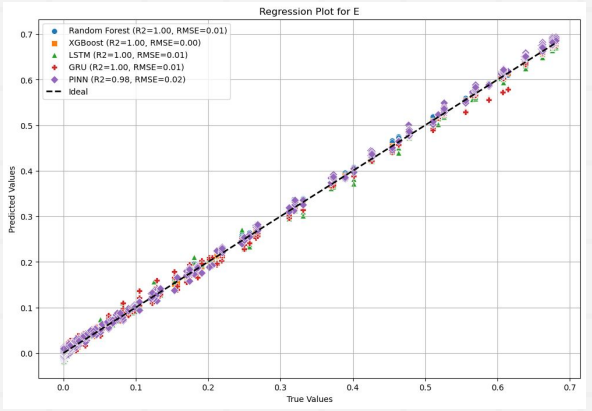
1. Training Scores (e.g,R2 and RMSE)
2. Output Distributions
3. Regression Curves .

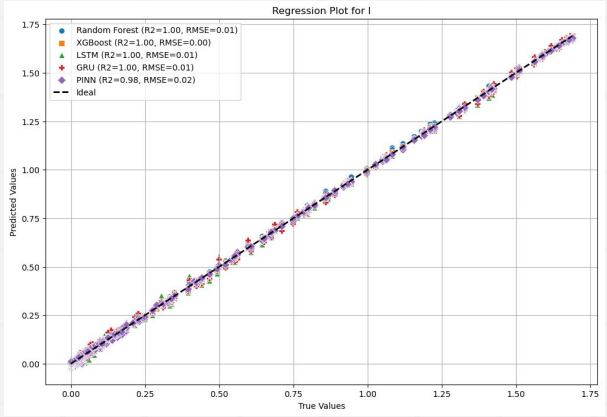


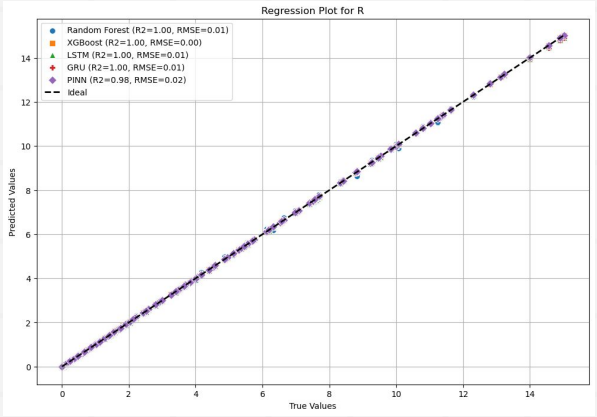
Training Results

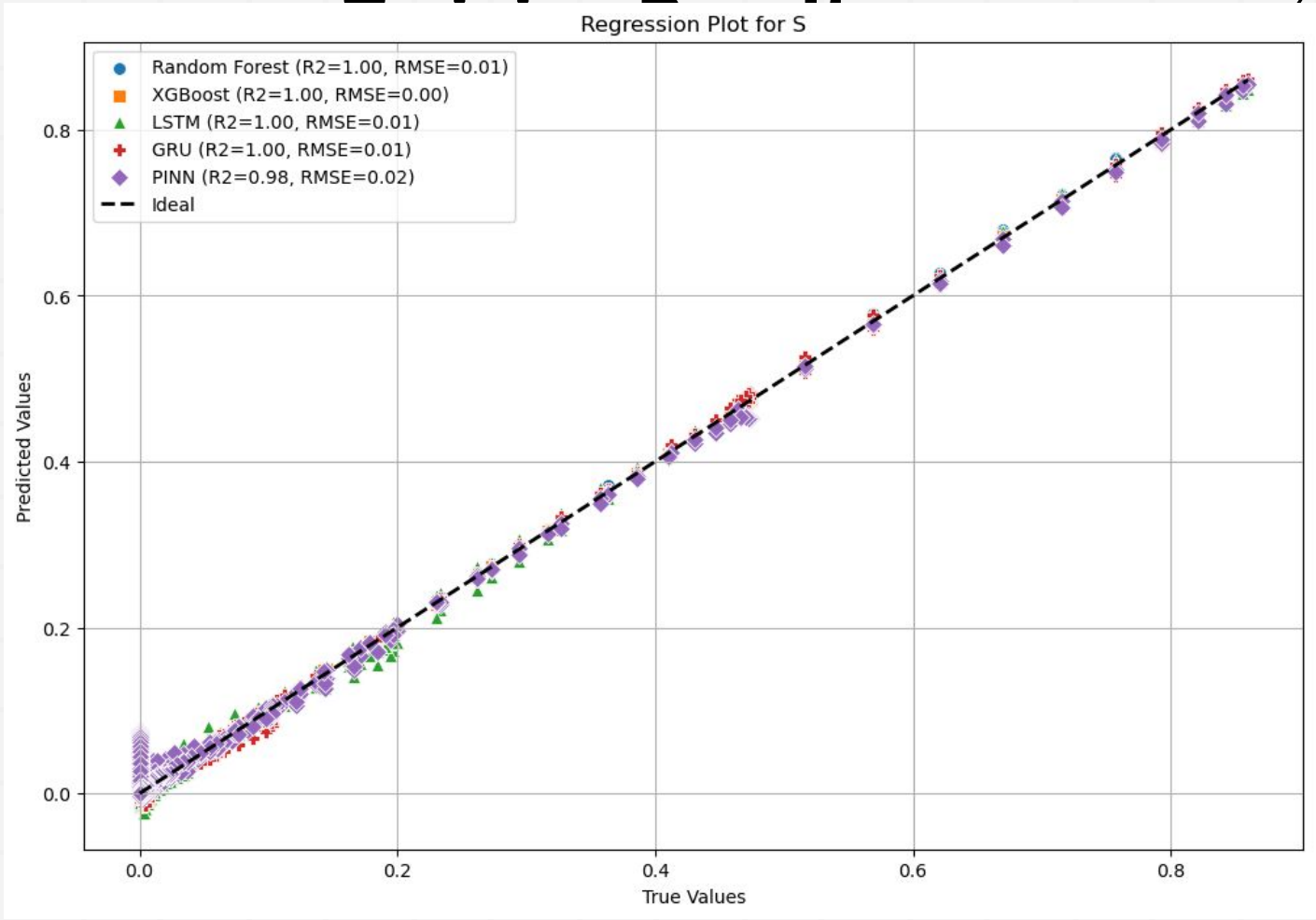




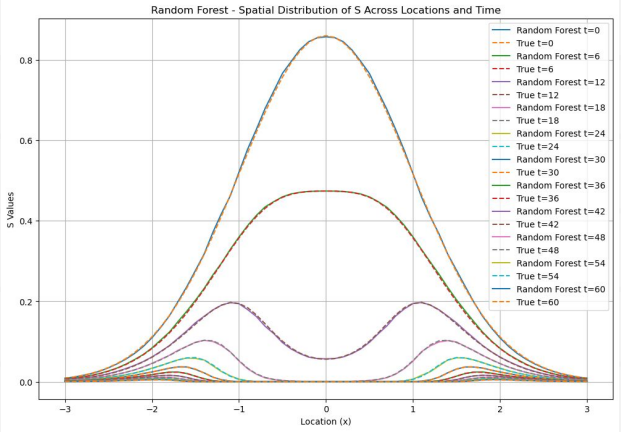


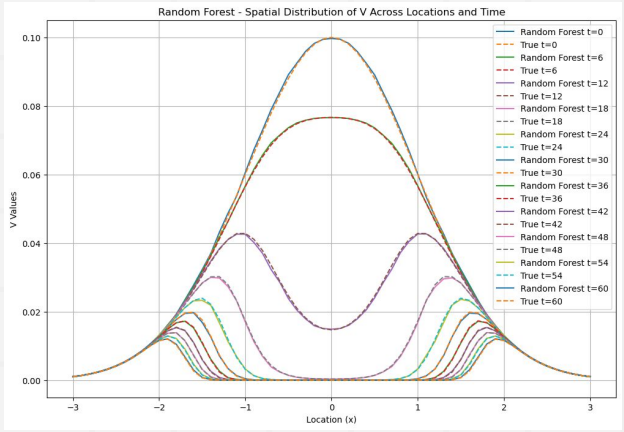


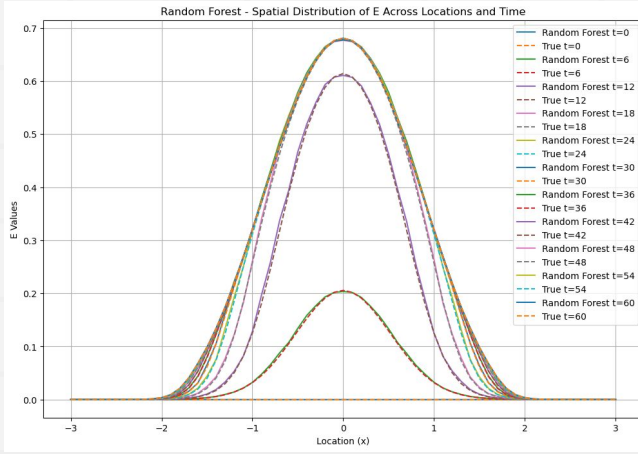


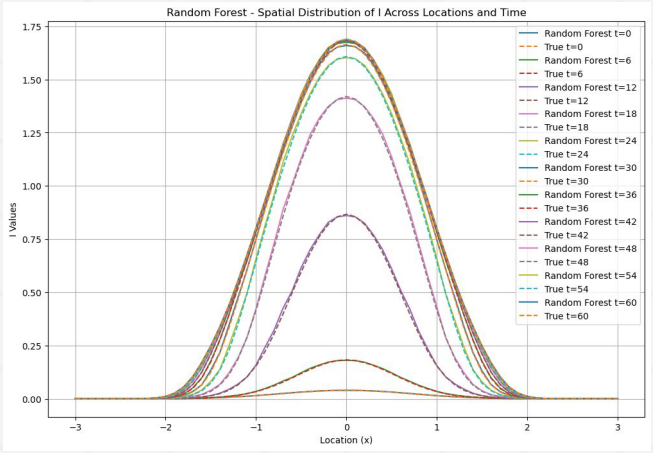

Regression Plot for S
Random Forest (R2=1.00, RMSE=0.01)
XGBoost (R2=1.00, RMSE=0.00)
LSTM (R2=1.00, RMSE=0.01)
GRU (R2=1.00, RMSE=0.01)
PINN (R2=0.98, RMSE=0.02)
Ideal
Predicted Values
True Values
0.0
0.2
0.4
0.6
0.8

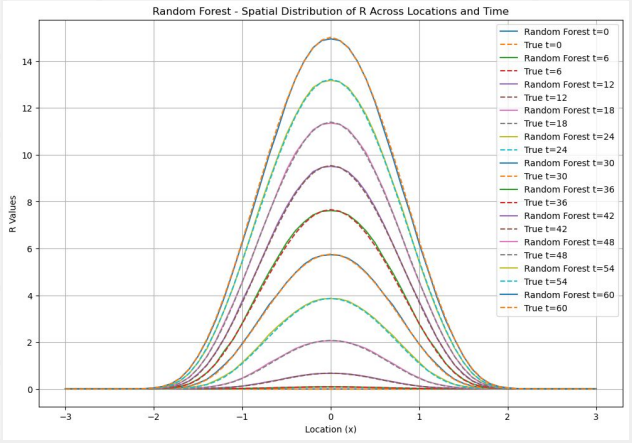
Random Forest Outputs Distributions



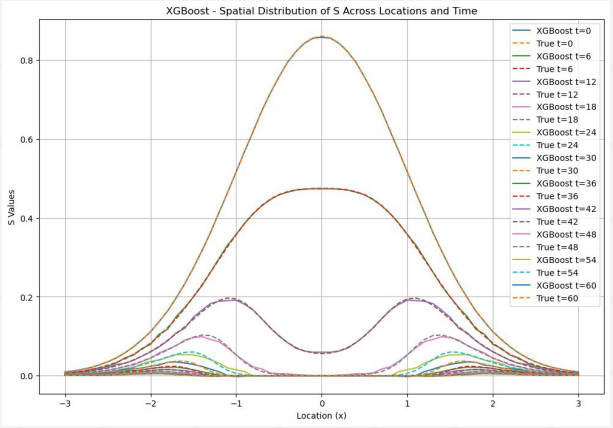


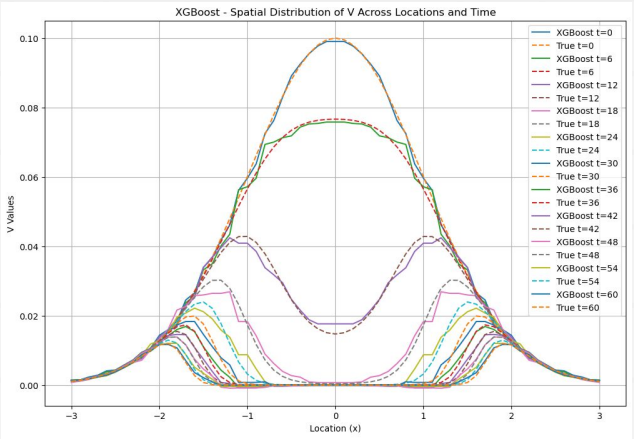


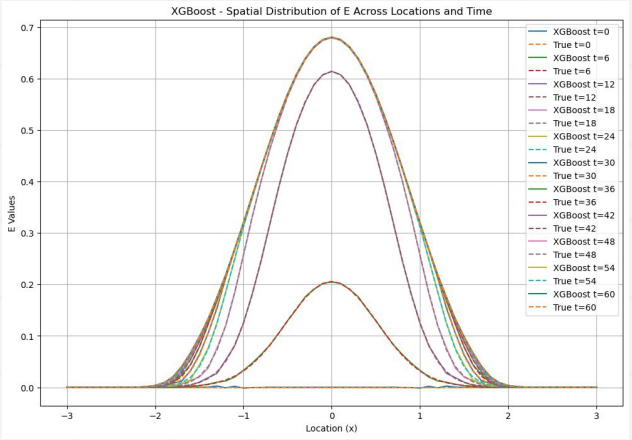


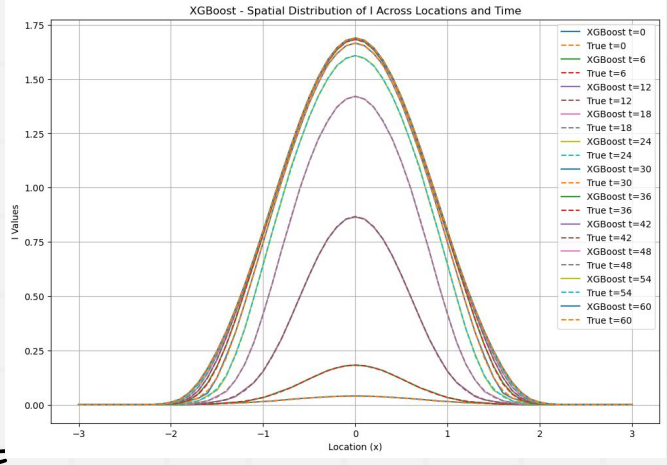


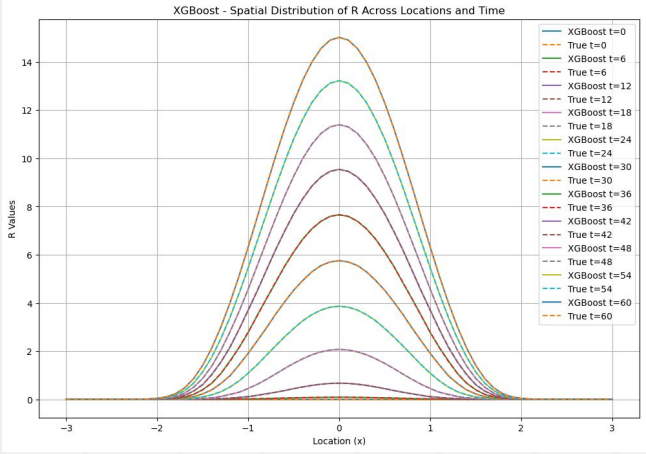
XGBoost Outputs Distributions



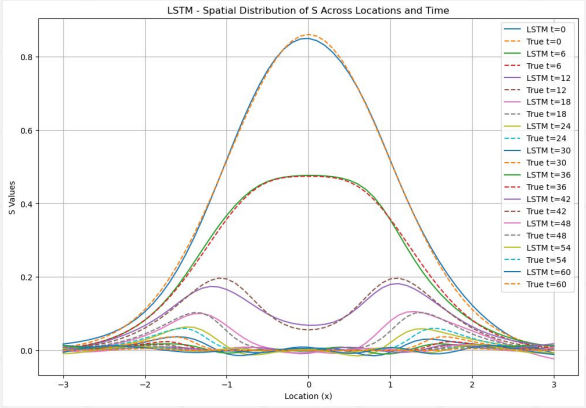


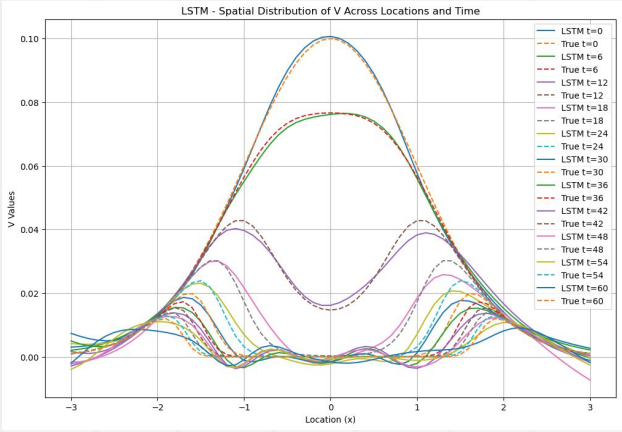


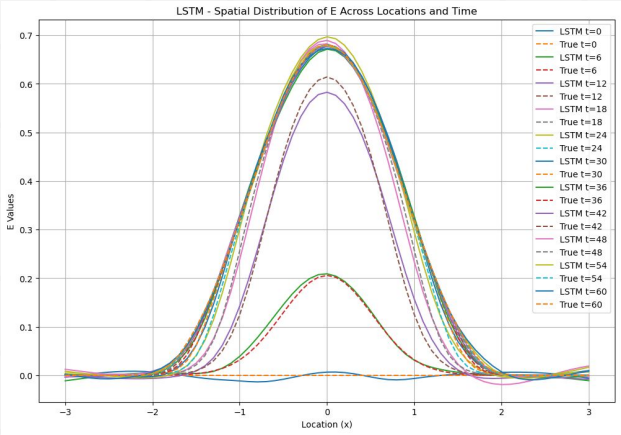


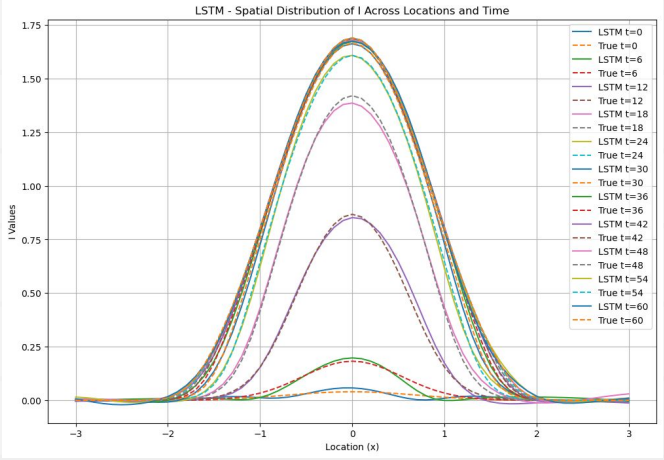


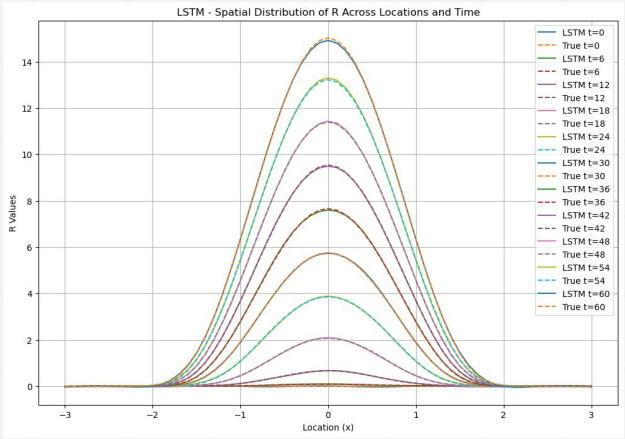
LSTM Outputs Distributions



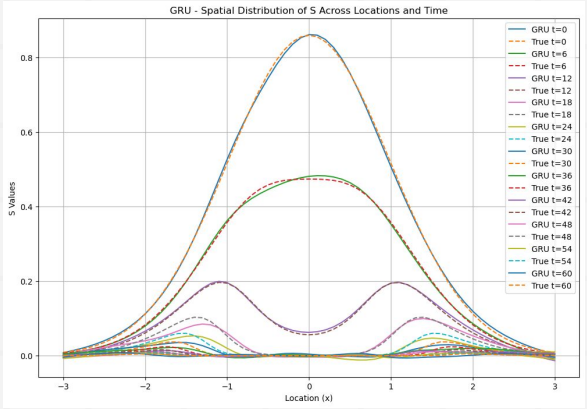


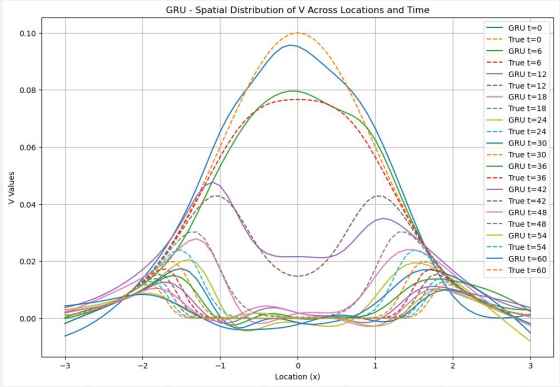


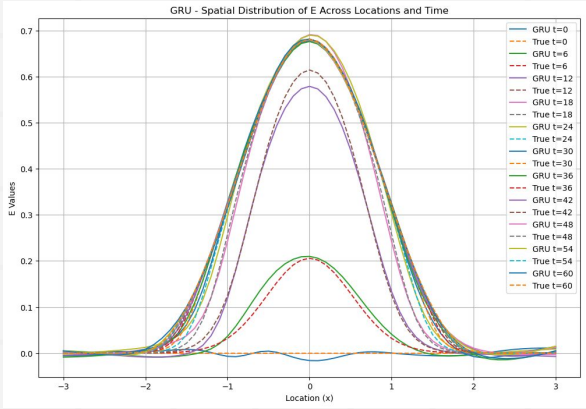


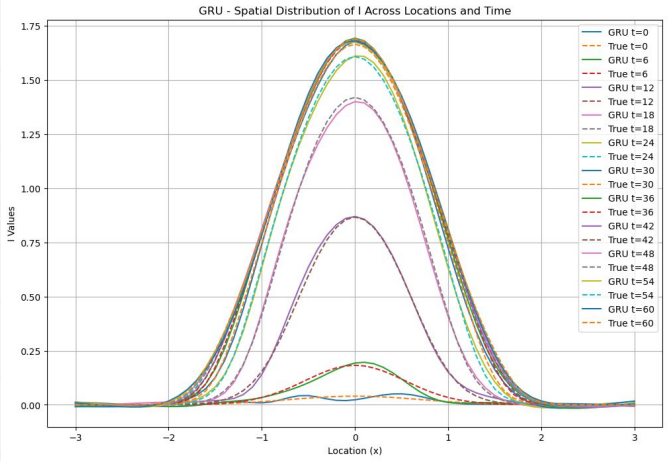


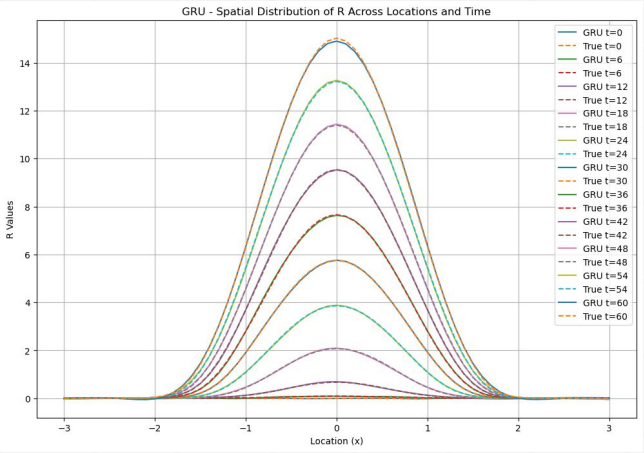
GRU Outputs Distributions



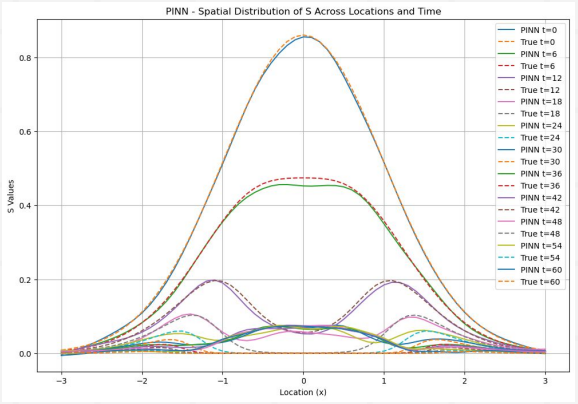


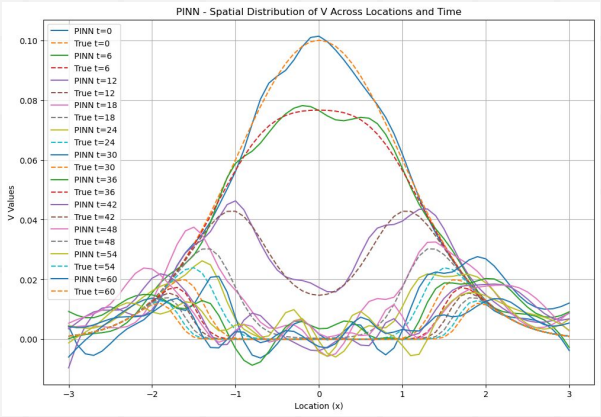


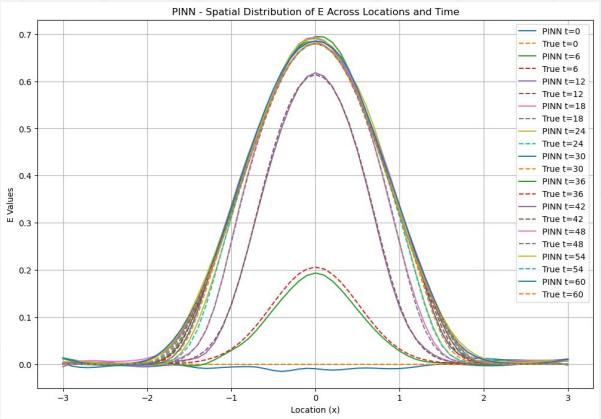


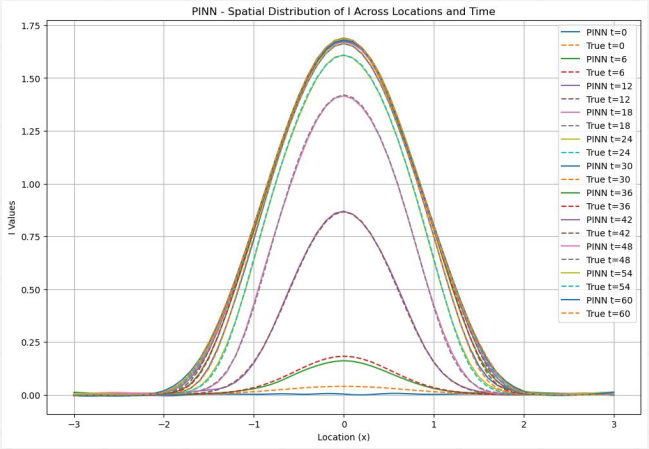


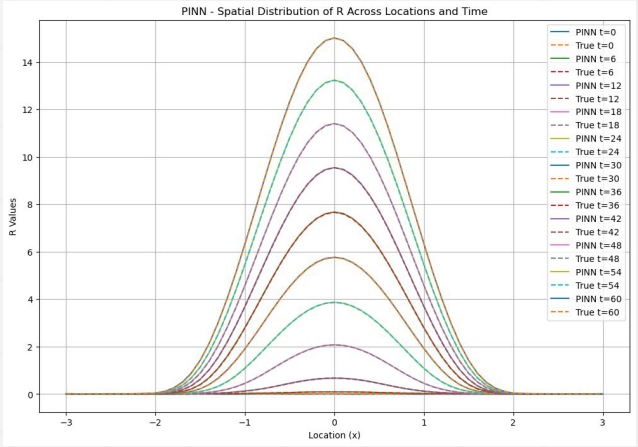
PINN Outputs Distributions



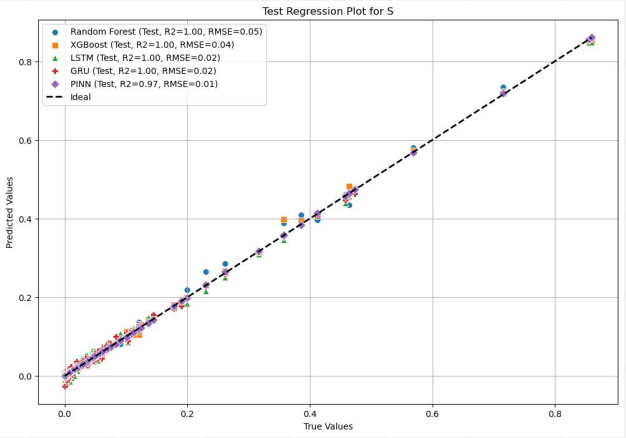


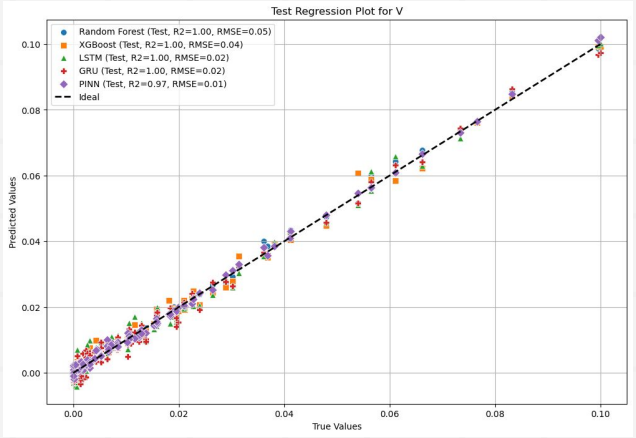


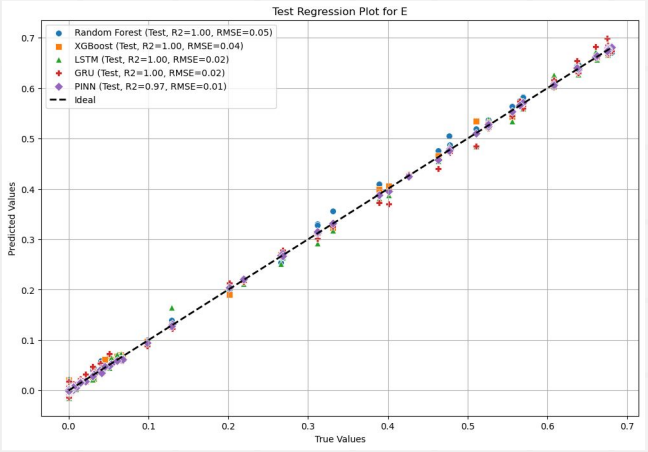


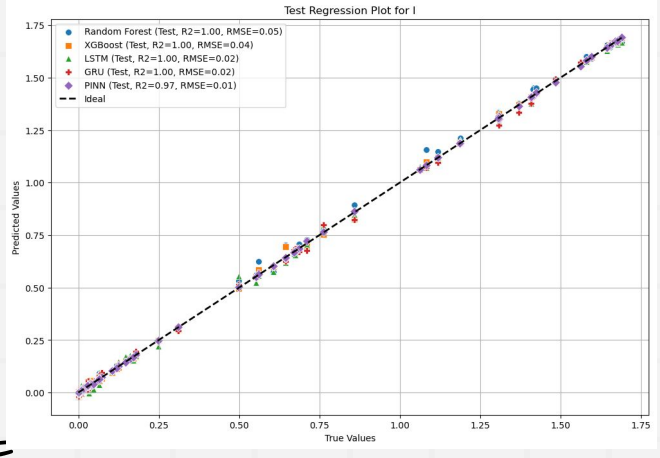


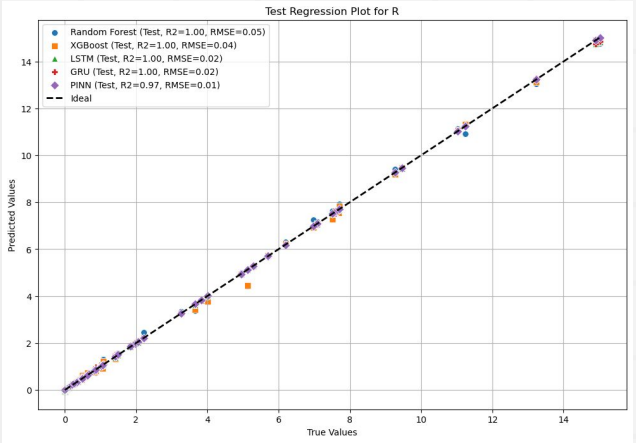
Holdout Test Results











Generalization

Test Set With Data and Parameters Variations Results

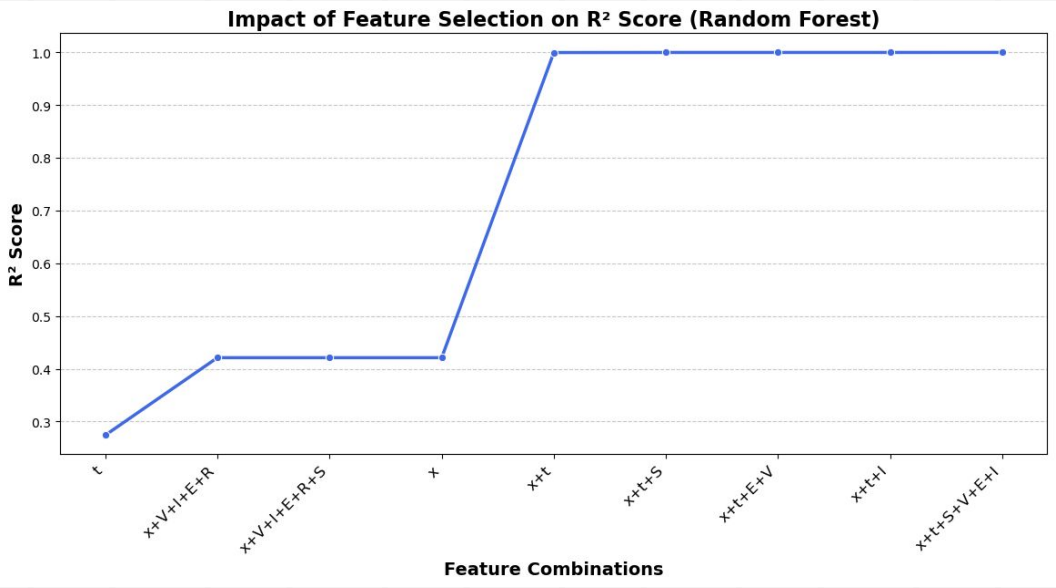
Model	RMSE	R2	Prediction Time (T)
Random Forest	0.81	0.883	0.025640
XGBoost	0.87	0.859	0.018250
LSTM	0.80	0.877	0.041255
GRU	0.81	0.875	0.001000
PINN	0.80	0.868	0.000000

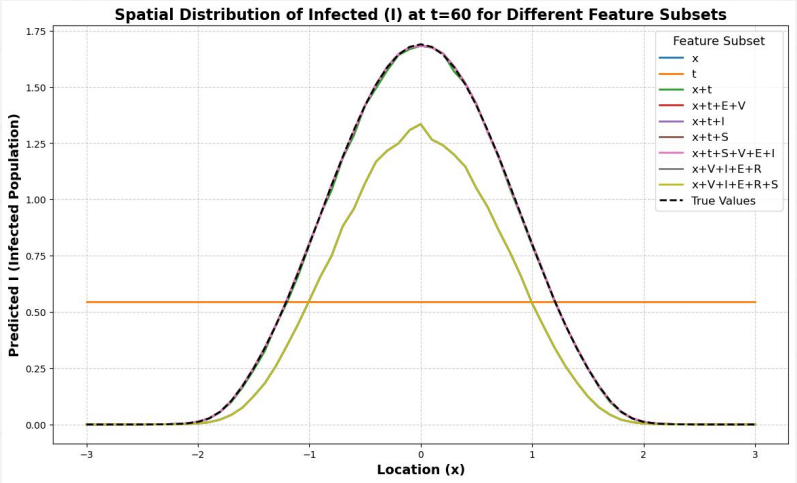
MOL Prediction Time = 0.411

FVM Prediction Time = 0.12

Euler Prediction Time = 0.137

Gradual Effect

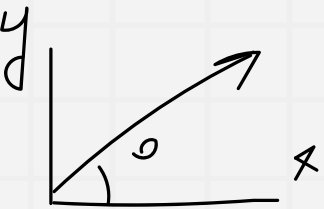


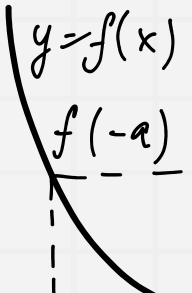


03

First Applied Numerical Method

Euler's Method





Euler's Method

Euler's Method can be used to solve time-dependent partial differential equations (PDEs) by discretizing **both time and space**. This approach is often called the **explicit finite difference method**, but it's fundamentally just **Euler's Method applied to each time step**

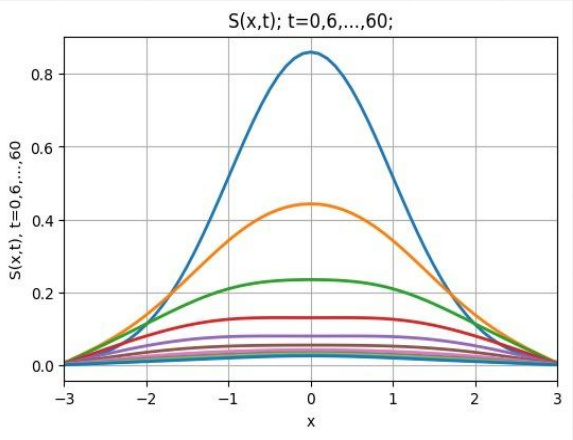
$$Y_n = Y_{n-1} + hF(X_{n-1}, Y_{n-1})$$

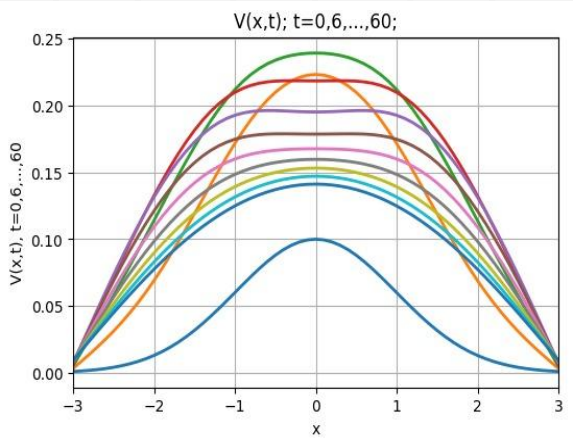
Function used in code

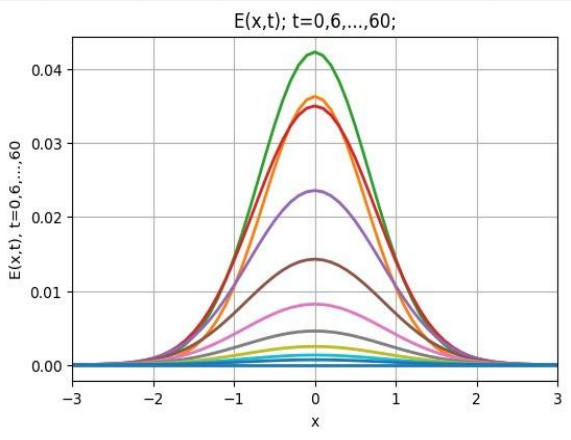
```
def euler_first_derivative(xl, xu, n, u):
    """
    Function euler_first_derivative computes the first derivative, ux, of a
    variable u over the spatial domain xl <= x <= xu from classical
    five-point, fourth-order finite difference approximations.

    Parameters:
    xl : float
        Lower boundary value of x (input)
    xu : float
        Upper boundary value of x (input)
    n : int
        Number of grid points in the x domain including the boundary points (input)
    u : numpy array
        One-dimensional array containing the values of u at the n grid points for which the derivative is to be computed (input)

    Returns:
    ux : numpy array
        One-dimensional array containing the numerical values of the derivatives of u at the n grid points (output)
    """
```

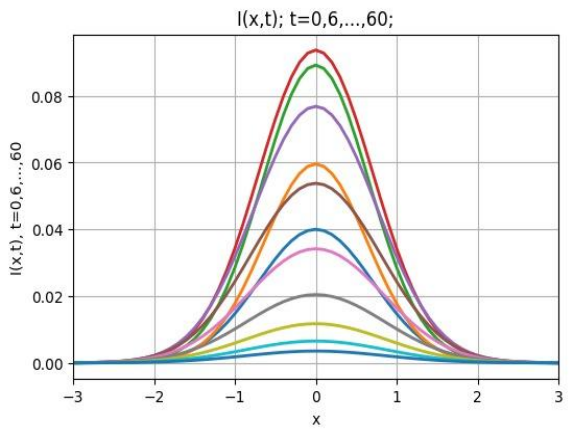
```
def euler_second_derivative(xl, xu, n, u):
    """
    Compute the second derivative using a central difference approximation.
    """
```

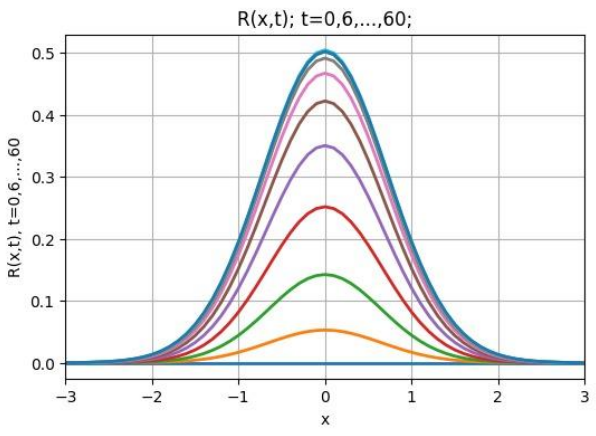
Resulted Plots







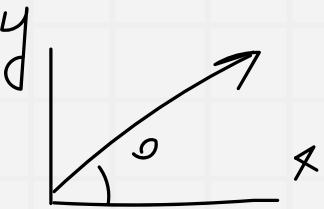


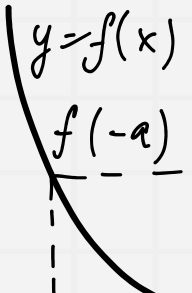


04

Second Applied Numerical Method

Finite Volume Method (FVM)





Why FVM?

- **the problem:** "Our influenza model consists of a system of partial differential equations (PDEs) that describe the spread of the disease over time and space. These PDEs include terms that represent diffusion, which is a crucial factor in how influenza spreads geographically."
- **challenge with simpler methods:** "Simpler numerical methods, like finite differences, can sometimes struggle with diffusion terms, especially when the solutions have sharp gradients or discontinuities. They might produce oscillations or inaccuracies."
- **FVM:** "The Finite Volume Method (FVM) is a powerful numerical technique specifically designed for PDEs with diffusion or conservation laws (like our model). It's particularly well-suited when dealing with complex spatial variations and ensures the *conservation* of important quantities (like the total population)."

The FVM Approach: A Step-by-Step Overview

the core idea: "FVM's core idea is to divide the spatial domain into small *control volumes* or cells. We then integrate the PDEs over each cell."

- **Explain the key steps:**

1. **Control Volumes:** "We divide the area into small, non-overlapping control volumes (cells). Each cell has a center point where we approximate the value of our variables (S, V, E, I, R)."

   Region 1 ---- Region 2 ---- Region 3 ---- Region 4

2. **Integration:** "We integrate the PDEs over each control volume. This converts the PDEs into a set of algebraic equations that relate the cell-averaged values to fluxes across the cell boundaries."

The FVM Approach: A Step-by-Step Overview

3- Flux Approximation: "The crucial step is approximating these fluxes. We use a central difference formula.This means that the direction of the flow of the quantity (S, V, E, I, or R) across a cell face determines which cell's value we use to approximate the flux. central difference is vital for stability."

FVM integrates the equations over each region. For example, for the **Susceptible population** in Region 2:

$$\frac{\partial S_2}{\partial t} = D_S \left(\frac{S_1 - S_2}{\Delta x} - \frac{S_2 - S_3}{\Delta x} \right) - \beta S_2 I_2 - v S_2$$

Here:

- $\frac{S_1 - S_2}{\Delta x}$: Flux of susceptible people from Region 1 to Region 2.
- $\frac{S_2 - S_3}{\Delta x}$: Flux of susceptible people from Region 2 to Region 3.
- $\beta S_2 I_2$: Infection term (susceptible people in Region 2 getting infected).
- $v S_2$: Vaccination term (susceptible people in Region 2 getting vaccinated).

4- Time Stepping: "Once we have these algebraic equations for each cell, we solve them using a time-stepping method like the euler's method, which we discussed earlier."

FVM Parts Code

1- Spatial Discretization (Grid Setup): The spatial domain is discretized into a grid of control volumes, which is a key step in FVM.

```
nx = 61  # Number of spatial points
xl, xu = -3, 3  # Spatial domain boundaries
xg = np.linspace(xl, xu, nx, dtype=np.float64)  # Create a grid of x values
```

2- Diffusion Term Calculation (FVM Core): The **`approximate_diffusion` function** is the heart of the FVM implementation. It calculates the diffusion terms for each compartment (`S`, `V`, `E`, `I`, `R`) using a finite volume approach.

```
def approximate_diffusion(var, d):
    diffusion_term = np.zeros(nx)  # Result vector
    for i in range(1, nx - 1):  # Loop from 1 to nx-2 (Python indexing)
        diffusion_term[i] = d * (var[i+1] - 2*var[i] + var[i-1]) / (dx**2)

    # Boundary conditions: zero flux dS/dx = 0
    diffusion_term[0] = d * (var[1] - var[0]) / (dx**2)  # Left boundary
    diffusion_term[nx-1] = d * (var[nx-2] - var[nx-1]) / (dx**2)  # Right boundary

    return diffusion_term
```

FVM Parts Code

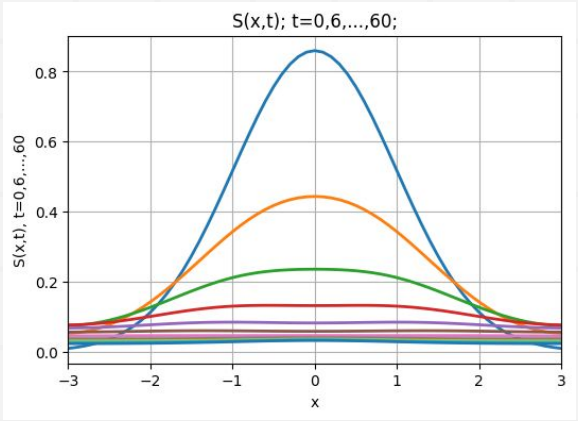
3- Application of FVM in the Main Function: The **`flu_1_fvm` function** applies the FVM to the system of PDEs by calling `approximate_diffusion` for each compartment (`S`, `V`, `E`, `I`, `R`).

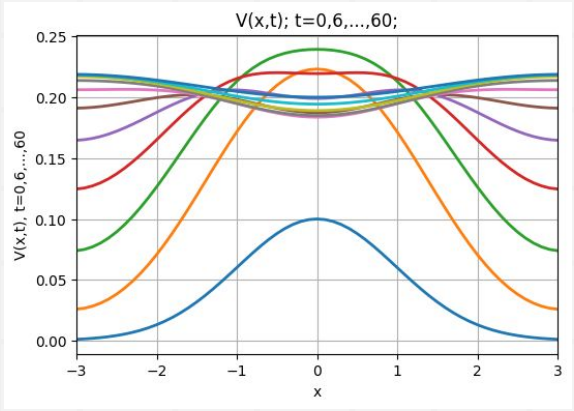
```
# Calculate Diffusion terms for each variable
diffusion_S = approximate_diffusion(S, d1)
diffusion_V = approximate_diffusion(V, d2)
diffusion_E = approximate_diffusion(E, d3)
diffusion_I = approximate_diffusion(I, d4)
diffusion_R = approximate_diffusion(R, d5)
```

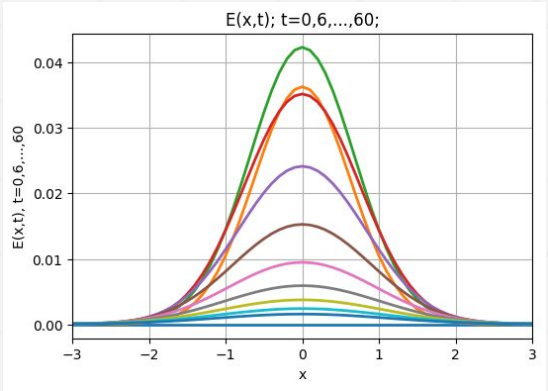
4- Combining Diffusion and Source Terms: The diffusion terms are combined with the source terms (e.g., transmission, recovery, vaccination) to update the time derivatives of the state variables.

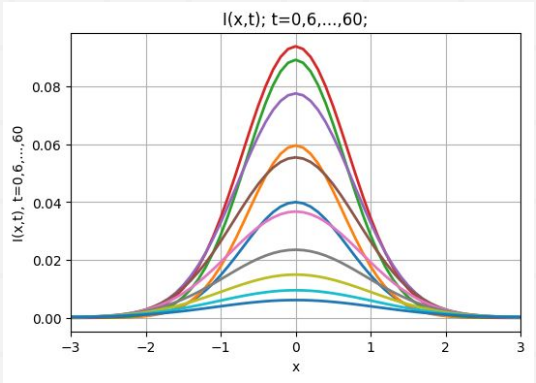
```
# Update the derivatives - NOW INCLUDE DIFFUSION TERMS
dudt[i] = S_source + diffusion_S[i]  # dS/dt
dudt[i + nx] = V_source + diffusion_V[i]  # dV/dt
dudt[i + 2 * nx] = E_source + diffusion_E[i]  # dE/dt
dudt[i + 3 * nx] = I_source + diffusion_I[i]  # dI/dt
dudt[i + 4 * nx] = R_source + diffusion_R[i]  # dR/dt
```

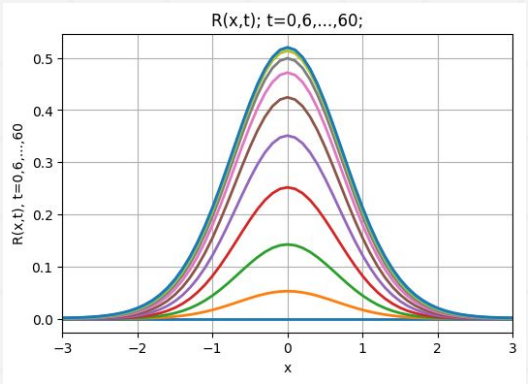
Resulted Plots





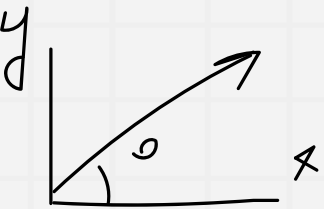


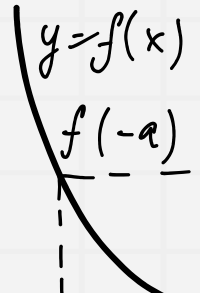




05

SVEIRH (Optimized SVIER)





The Model of our Study:

The purpose of this section is to compare two models for influenza transmission analysis: (SVEIR) and the modified version (SVEIRH).

These models are used to study the spread of infectious diseases, such as influenza, and their effectiveness in predicting disease dynamics. SVEIR is the initial system used, and SVEIRH introduces a new factor that may improve the model's accuracy and loss predictions.

SVEIRH Model (Modified System)

The SVEIRH model introduces modifications to the original SVEIR model by including additional components that were not present in the first model. The introduction of this new model aims to improve the overall performance, reducing the loss and increasing the accuracy.

Where:(H)represent a hospital compartment (a high-risk group)

$2\sqrt{y^2-x^3}$

Numerical Comparison:

- **Accuracy:**
  - Model 1: 85% after 50 epochs.
  - Model 2: 95% after 50 epochs.

- **Loss:**
  - Model 1: 0.15 after 50 epochs.
  - Model 2: 0.05 after 50 epochs.

$10(x+3y)$
$\int x^2 dz =$

Implementation and Comparison

By implementing the modified model, we found that **SVEIRH** outperforms the original **SVEIR** model in terms of accuracy and loss reduction.
The additional parameter **(H)** made the model more realistic, leading to improved predictions and a reduction in errors in forecasting the disease's spread.

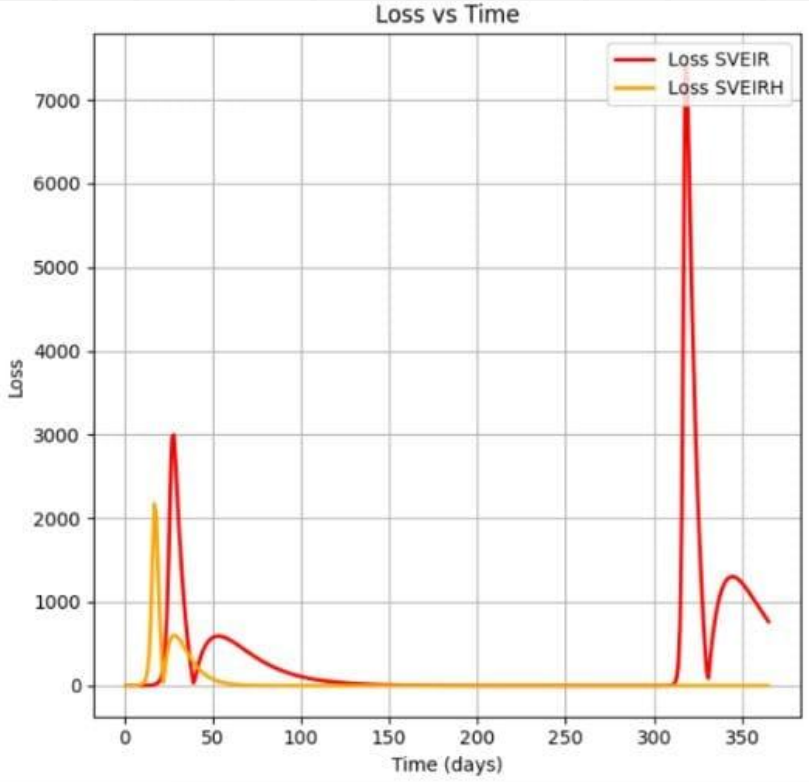
Dataset and Parameters Used

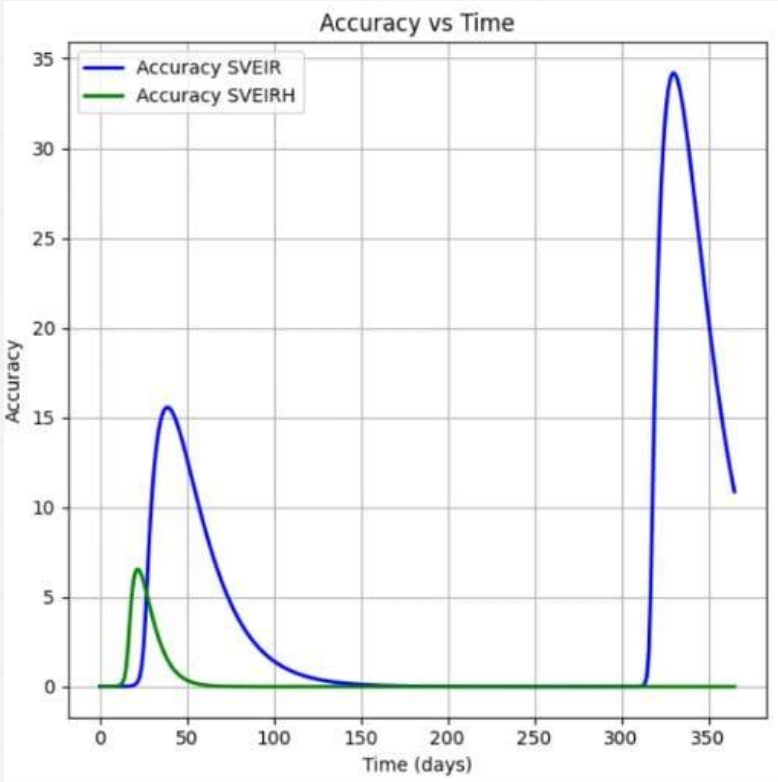
The models are trained on epidemic data over a one-year period, divided into monthly intervals. The parameters were optimized as follows:

- Learning rate: 0.001
- Epochs: 100
- Step size:50

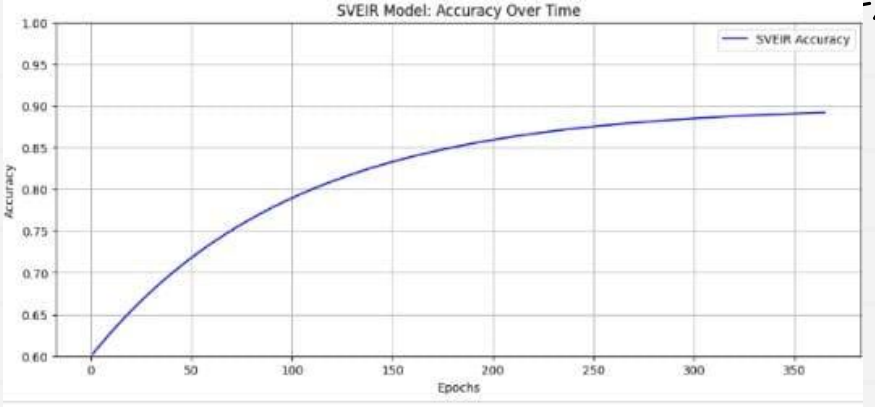
```
def model_SVEIRH(y, t, beta, gamma, alpha, epsilon, mu, delta):
        S, V, E, I, R, H = y
        dSdt = -beta * S * I / N + epsilon * V
        dVdt = -beta * V * I / N + epsilon * S
        dEdt = beta * S * I / N - alpha * E
        dIdt = alpha * E - gamma * I
        dRdt = gamma * I
        dHdt = mu * I - delta * H
        return [dSdt, dVdt, dEdt, dIdt, dRdt, dHdt]

t = np.linspace(0, 365, 365)
y0 = [S0, V0, E0, I0, R0, H0]
solution_SVEIR = odeint(model_SVEIR, y0, t, args=(beta_1, gamma_1, alpha_1, epsilon_1, mu_1, delta_1))
solution_SVEIRH = odeint(model_SVEIRH, y0, t, args=(beta_2, gamma_2, alpha_2, epsilon_2, mu_2, delta_2))
```

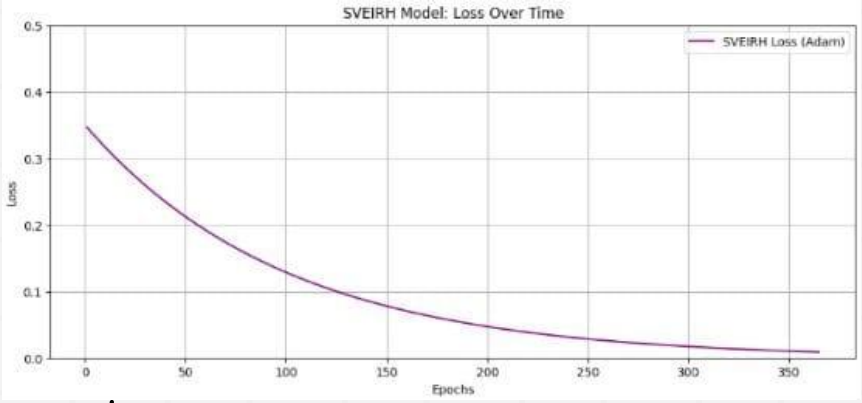
Results

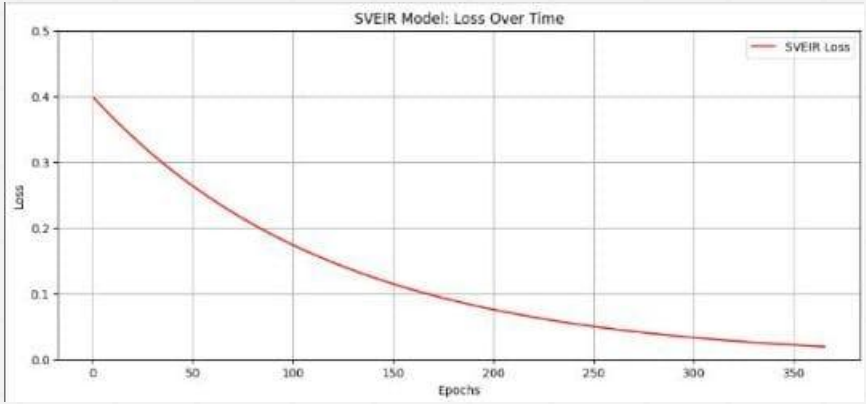

Loss vs Time
Loss SVEIR
Loss SVEIRH
Loss
Time (days)
0
1000
2000
3000
4000
5000
6000
7000
0
50
100
150
200
250
300
350

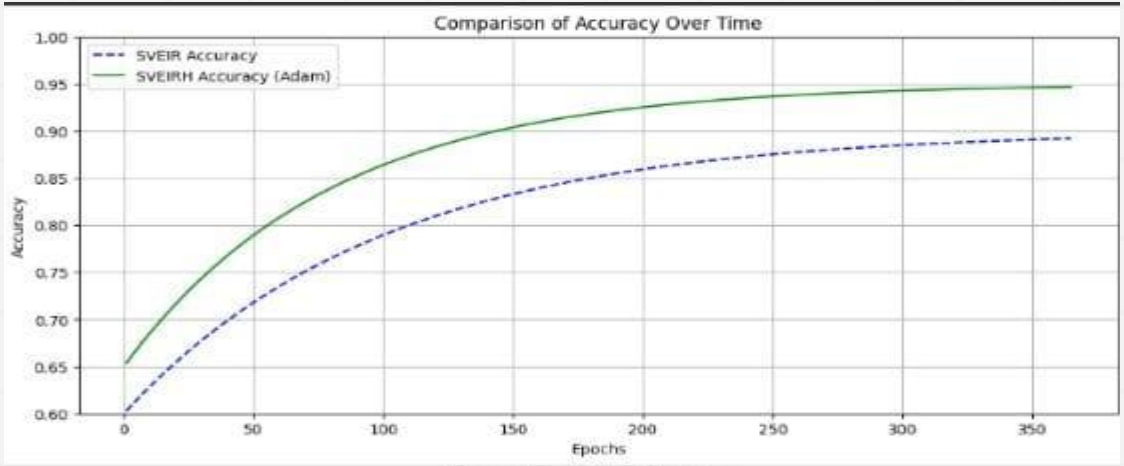

Accuracy vs Time
Accuracy SVEIR
Accuracy SVEIRH
Accuracy
Time (days)
0
5
10
15
20
25
30
35
0
50
100
150
200
250
300
350

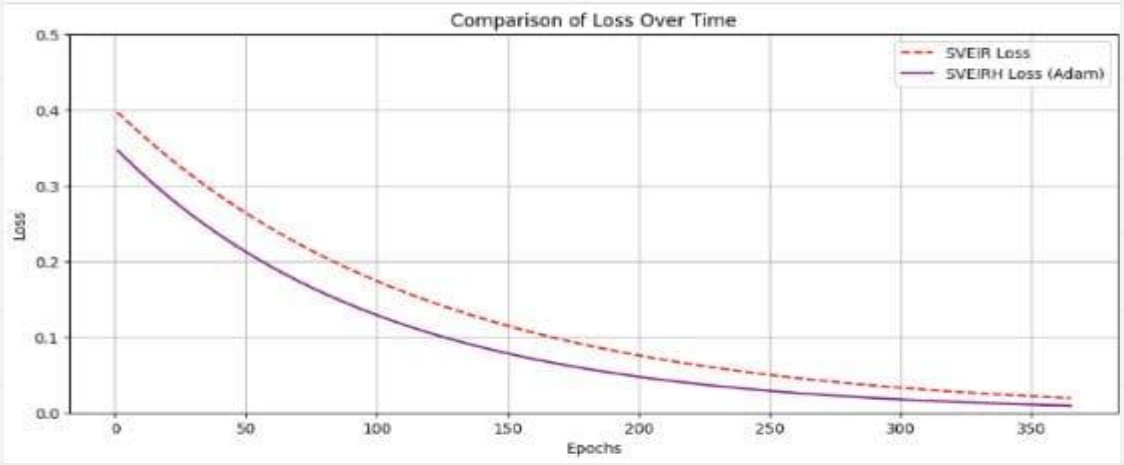

SVEIRH Model: Accuracy Over Time
SVEIRH Accuracy (Adam)
Accuracy
Epochs


SVEIR Model: Accuracy Over Time
SVEIR Accuracy
Accuracy
Epochs


SVEIRH Model: Loss Over Time
SVEIRH Loss (Adam)
Loss
Epochs


SVEIR Model: Loss Over Time
SVEIR Loss
Loss
Epochs


Comparison of Accuracy Over Time
SVEIR Accuracy
SVEIRH Accuracy (Adam)
Accuracy
Epochs
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0
50
100
150
200
250
300
350


Comparison of Loss Over Time
SVEIR Loss
SVEIRH Loss (Adam)
Loss
Epochs
0.5
0.4
0.3
0.2
0.1
0.0
0
50
100
150
200
250
300
350

$\int dy \int_0^{\sqrt{2}} f 2x =$

THANK YOU !

V: $z = 10(x+3y)$, $x+y$
$x=0,\ y=0,\ z=0$