SIR Model Implementation and Analysis

Your Name

March 11, 2025

1 Introduction

This document provides a detailed explanation of the implementation and analysis of the SIR (Susceptible-Infected-Recovered) model using Python. The SIR model is a mathematical model used to simulate the spread of infectious diseases. The code uses numerical methods (Euler's method and the 4th-order Runge-Kutta method) to solve the differential equations of the SIR model and analyzes the results.

2 Code Explanation

2.1 Importing Libraries

The code begins by importing necessary libraries for numerical computations and plotting.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
```

- `numpy`: A library for numerical computations in Python.
- `matplotlib.pyplot`: A library for creating plots and visualizations.
- `GridSpec`: A tool from `matplotlib` for creating flexible subplot layouts.

2.2 Creating the SIR Model State

The `create_sir_state` function initializes the parameters of the SIR model.

```
def create_sir_state(beta, gamma, S0, I0, R0):
"""
Initialize the SIR model parameters.
The SIR model divides the population into three groups:
- Susceptible (S): People who can catch the disease.
```

```
        - Infected (I): People who currently have the disease
            and can spread it.
        - Recovered (R): People who have recovered and are
            immune.

        Parameters:
        beta (float): Infection rate, which determines how
            quickly the disease spreads.
        gamma (float): Recovery rate, which determines how
            quickly people recover.
        S0 (float): Initial number of susceptible individuals.
        I0 (float): Initial number of infected individuals.
        R0 (float): Initial number of recovered individuals.

        Returns:
        dict: A dictionary containing all the model parameters
            and the total population.
        """
        return {
            'beta': beta,  # Infection rate
            'gamma': gamma,  # Recovery rate
            'S0': S0,  # Initial susceptible population
            'I0': I0,  # Initial infected population
            'R0': R0,  # Initial recovered population
            'N': S0 + I0 + R0  # Total population (sum of S, I,
                and R)
        }
```

- `beta`: The infection rate, which determines how quickly susceptible individuals become infected.
- `gamma`: The recovery rate, which determines how quickly infected individuals recover.
- `S0`, `I0`, `R0`: The initial number of susceptible, infected, and recovered individuals, respectively.
- `N`: The total population, calculated as the sum of `S0`, `I0`, and `R0`.

2.3 Calculating Derivatives for the SIR Model

The `sir_derivatives` function calculates the rate of change for the susceptible, infected, and recovered populations.

```
        def sir_derivatives(state, t, state_dict):
        """
        Compute the rate of change (derivatives) for the SIR
            model over time.
        This function calculates how the number of susceptible,
             infected, and recovered individuals changes.
```

```
5
6           Parameters:
7           state (array): Current state of the population [S, I, R
                ].
8           t (float): Current time (not used directly but required
                 for compatibility).
9           state_dict (dict): Dictionary containing model
                parameters (beta, gamma, N).
10
11          Returns:
12          array: The rate of change for [S, I, R] as [dS/dt, dI/
                dt, dR/dt].
13          """
14          S, I, R = state  # Unpack the current state into
                susceptible, infected, and recovered
15          N = state_dict['N']  # Total population
16          # Normalize the population to prevent overflow (divide
                by total population)
17          S = S / N
18          I = I / N
19          R = R / N
20          # Calculate the rate of change for each group:
21          dSdt = -state_dict['beta'] * S * I * N  # Susceptible
                decrease due to infection
22          dIdt = state_dict['beta'] * S * I * N - state_dict['
                gamma'] * I * N  # Infected increase from new
                infections and decrease from recoveries
23          dRdt = state_dict['gamma'] * I * N  # Recovered
                increase from recoveries
24          return np.array([dSdt, dIdt, dRdt])  # Return the rates
                 of change
```

- `state`: The current state of the population, represented as an array `[S, I, R]`.
- `t`: The current time (not used in this function but required for compatibility with some numerical solvers).
- `state_dict`: A dictionary containing the model parameters (`beta`, `gamma`, `N`).
- The derivatives are calculated as:

$$\frac{dS}{dt} = -\beta \cdot S \cdot I \cdot N$$
$$\frac{dI}{dt} = \beta \cdot S \cdot I \cdot N - \gamma \cdot I \cdot N$$
$$\frac{dR}{dt} = \gamma \cdot I \cdot N$$

2.4 Solving the SIR Model Using Euler's Method

The `euler_method` function solves the SIR model using Euler's method.

```
        def euler_method(state_dict, t_span, h):
        """
        Solve the SIR model using Euler's method, a simple
            numerical method for solving differential equations.
        Euler's method approximates the solution by taking
            small steps in time.

        Parameters:
        state_dict (dict): Dictionary containing model
            parameters.
        t_span (array): Time span [t_start, t_end] for the
            simulation.
        h (float): Step size for the Euler method (smaller
            steps = more accurate results).

        Returns:
        tuple: A tuple containing time values and the
            corresponding S, I, R values.
        """
        t_start, t_end = t_span  # Unpack the start and end
            times
        n_steps = int((t_end - t_start) / h) + 1  # Calculate
            the number of steps
        t_values = np.linspace(t_start, t_end, n_steps)  #
            Create an array of time values
        S_values = np.zeros(n_steps)  # Initialize an array to
            store susceptible values
        I_values = np.zeros(n_steps)  # Initialize an array to
            store infected values
        R_values = np.zeros(n_steps)  # Initialize an array to
            store recovered values

        # Set initial conditions
        S_values[0] = state_dict['S0']  # Initial susceptible
            population
        I_values[0] = state_dict['I0']  # Initial infected
            population
        R_values[0] = state_dict['R0']  # Initial recovered
            population

        # Iterate over each time step
        for i in range(1, n_steps):
        state = np.array([S_values[i-1], I_values[i-1],
            R_values[i-1]])  # Current state
        derivatives = sir_derivatives(state, t_values[i-1],
            state_dict)  # Calculate derivatives
```

```
        # Update the population values using Euler's method
        S_values[i] = max(0, min(state_dict['N'], S_values[i-1]
            + h * derivatives[0]))  # Update susceptible
        I_values[i] = max(0, min(state_dict['N'], I_values[i-1]
            + h * derivatives[1]))  # Update infected
        R_values[i] = max(0, min(state_dict['N'], R_values[i-1]
            + h * derivatives[2]))  # Update recovered

        # Ensure the total population remains constant (S + I +
            R = N)
        if (total := S_values[i] + I_values[i] + R_values[i])
            != 0:
        S_values[i] *= state_dict['N'] / total
        I_values[i] *= state_dict['N'] / total
        R_values[i] *= state_dict['N'] / total

        return t_values, S_values, I_values, R_values  # Return
            the results
```

- `t_span`: The time span for the simulation, specified as `[t_start, t_end]`.
- `h`: The step size for the Euler method. Smaller step sizes result in more accurate solutions.
- The function iterates over each time step, updating the values of `S`, `I`, and `R` using the derivatives calculated by `sir_derivatives`.
- The total population is kept constant by normalizing the values of `S`, `I`, and `R` at each step.

2.5 Solving the SIR Model Using the Runge-Kutta 4th Order Method

The `rk4_method` function solves the SIR model using the 4th-order Runge-Kutta method.

```
        def rk4_method(state_dict, t_span, h):
        """
        Solve the SIR model using the 4th order Runge-Kutta
            method (RK4).
        RK4 is a more accurate numerical method compared to
            Euler's method.

        Parameters:
        state_dict (dict): Dictionary containing model
            parameters.
        t_span (array): Time span [t_start, t_end] for the
            simulation.
        h (float): Step size for the RK4 method.
```

```
10
11          Returns:
12          tuple: A tuple containing time values and the
                corresponding S, I, R values.
13          """
14          t_start, t_end = t_span  # Unpack the start and end
                times
15          n_steps = int((t_end - t_start) / h) + 1  # Calculate
                the number of steps
16          t_values = np.linspace(t_start, t_end, n_steps)  #
                Create an array of time values
17          S_values = np.zeros(n_steps)  # Initialize an array to
                store susceptible values
18          I_values = np.zeros(n_steps)  # Initialize an array to
                store infected values
19          R_values = np.zeros(n_steps)  # Initialize an array to
                store recovered values
20
21          # Set initial conditions
22          S_values[0] = state_dict['S0']  # Initial susceptible
                population
23          I_values[0] = state_dict['I0']  # Initial infected
                population
24          R_values[0] = state_dict['R0']  # Initial recovered
                population
25
26          # Iterate over each time step
27          for i in range(1, n_steps):
28          state = np.array([S_values[i-1], I_values[i-1],
                R_values[i-1]])  # Current state
29          t = t_values[i-1]  # Current time
30          # Calculate the four intermediate steps (k1, k2, k3, k4
                ) for RK4
31          k1 = sir_derivatives(state, t, state_dict)
32          k2 = sir_derivatives(state + 0.5 * h * k1, t + 0.5 * h,
                 state_dict)
33          k3 = sir_derivatives(state + 0.5 * h * k2, t + 0.5 * h,
                 state_dict)
34          k4 = sir_derivatives(state + h * k3, t + h, state_dict)
35          # Update the state using the RK4 formula
36          state_new = state + (h/6) * (k1 + 2*k2 + 2*k3 + k4)
37
38          # Apply bounds checking to ensure population values
                stay within valid range
39          S_values[i] = max(0, min(state_dict['N'], state_new[0])
                )  # Update susceptible
40          I_values[i] = max(0, min(state_dict['N'], state_new[1])
                )  # Update infected
41          R_values[i] = max(0, min(state_dict['N'], state_new[2])
                )  # Update recovered
```

```
42
43          # Ensure the total population remains constant (S + I +
                R = N)
44          if (total := S_values[i] + I_values[i] + R_values[i])
                != 0:
45              S_values[i] *= state_dict['N'] / total
46              I_values[i] *= state_dict['N'] / total
47              R_values[i] *= state_dict['N'] / total
48
49          return t_values, S_values, I_values, R_values  # Return
                the results
```

- The RK4 method calculates four intermediate steps (`k1`, `k2`, `k3`, `k4`) to approximate the solution.
- The state is updated using a weighted average of these intermediate steps.
- Like Euler's method, the total population is kept constant by normalizing the values of `S`, `I`, and `R`.

2.6 Comparing Euler's Method and RK4 Method

The `compare_methods` function compares the results of Euler's method and the RK4 method for different step sizes.

```
1           def compare_methods(state_dict, t_span, step_sizes):
2           """
3           Compare the results of Euler's method and the RK4
                method for different step sizes.
4           This function plots the results to visualize the
                differences in accuracy.
5
6           Parameters:
7           state_dict (dict): Dictionary containing model
                parameters.
8           t_span (array): Time span [t_start, t_end] for the
                simulation.
9           step_sizes (list): List of step sizes to compare.
10          """
11          plt.figure(figsize=(15, 10))  # Create a figure for
                plotting
12          gs = GridSpec(2, 2)  # Create a grid for subplots
13          # Plot comparison for susceptible individuals
14          ax1 = plt.subplot(gs[0, 0])
15          ax1.set_title("Susceptible Population")
16          ax1.set_xlabel("Time")
17          ax1.set_ylabel("Number of individuals")
18          # Plot comparison for infected individuals
19          ax2 = plt.subplot(gs[0, 1])
```

```
        ax2.set_title("Infected Population")
        ax2.set_xlabel("Time")
        ax2.set_ylabel("Number of individuals")
        # Plot comparison for recovered individuals
        ax3 = plt.subplot(gs[1, 0])
        ax3.set_title("Recovered Population")
        ax3.set_xlabel("Time")
        ax3.set_ylabel("Number of individuals")
        # Plot error comparison for infected individuals
        ax4 = plt.subplot(gs[1, 1])
        ax4.set_title("Error Comparison (Infected Population)")
        ax4.set_xlabel("Time")
        ax4.set_ylabel("Absolute Difference")

        # Use the smallest step size RK4 as the reference solution (most accurate)
        smallest_h = min(step_sizes)
        ref_t, ref_S, ref_I, ref_R = rk4_method(state_dict, t_span, smallest_h)
        colors = ["b", "g", "r", "c", "m", "y"]  # Colors for plotting
        line_styles_euler = ["--" for _ in range(len(step_sizes))]  # Line styles for Euler method
        line_styles_rk4 = ["-" for _ in range(len(step_sizes))]  # Line styles for RK4 method

        # Iterate over each step size and plot the results
        for i, h in enumerate(step_sizes):
        if h == smallest_h:
        continue  # Skip the smallest step size (used as reference)
        color = colors[i % len(colors)]  # Choose a color for the current step size
        # Solve using Euler's method
        t_euler, S_euler, I_euler, R_euler = euler_method(state_dict, t_span, h)
        # Solve using RK4 method
        t_rk4, S_rk4, I_rk4, R_rk4 = rk4_method(state_dict, t_span, h)

        # Plot susceptible population
        ax1.plot(t_euler, S_euler, line_styles_euler[i], color=color, label=f"Euler (h={h})")
        ax1.plot(t_rk4, S_rk4, line_styles_rk4[i], color=color, label=f"RK4 (h={h})")
        # Plot infected population
        ax2.plot(t_euler, I_euler, line_styles_euler[i], color=color, label=f"Euler (h={h})")
        ax2.plot(t_rk4, I_rk4, line_styles_rk4[i], color=color, label=f"RK4 (h={h})")
```

```
        # Plot recovered population
        ax3.plot(t_euler, R_euler, line_styles_euler[i], color=color, label=f"Euler (h={h})")
        ax3.plot(t_rk4, R_rk4, line_styles_rk4[i], color=color, label=f"RK4 (h={h})")
        # Calculate and plot errors for infected population
        I_euler_error = np.abs(np.interp(t_euler, ref_t, ref_I) - I_euler)
        I_rk4_error = np.abs(np.interp(t_rk4, ref_t, ref_I) - I_rk4)
        ax4.plot(t_euler, I_euler_error, line_styles_euler[i], color=color, label=f"Euler Error (h={h})")
        ax4.plot(t_rk4, I_rk4_error, line_styles_rk4[i], color=color, label=f"RK4 Error (h={h})")

        # Add legends to the plots
        ax1.legend()
        ax2.legend()
        ax3.legend()
        ax4.legend()
        plt.tight_layout()  # Adjust layout to prevent overlap
        plt.savefig("sir_model_comparison.png", dpi=300)  # Save the plot as an image
        plt.show()  # Display the plot
```

- This function creates a 2x2 grid of subplots to visualize the susceptible, infected, and recovered populations, as well as the error in the infected population.
- The smallest step size RK4 solution is used as a reference for calculating errors.
- The results are plotted for each step size, and the errors are displayed to show the accuracy of each method.

2.7 Analyzing the Effect of Parameters

The `analyze_parameters` function analyzes the effect of different infection rates (`beta`) and recovery rates (`gamma`) on the infected population.

```
        def analyze_parameters(state_dict, t_span, h, beta_values, gamma_values):
        """
        Analyze the effect of different infection rates (beta) and recovery rates (gamma) on the SIR model.
        This function plots how changing these parameters affects the infected population over time.

        Parameters:
```

```
        state_dict (dict): Dictionary containing model parameters.
        t_span (array): Time span [t_start, t_end] for the simulation.
        h (float): Step size for the simulation.
        beta_values (list): List of beta values to analyze.
        gamma_values (list): List of gamma values to analyze.
        """
        plt.figure(figsize=(15, 10))  # Create a figure for plotting
        gs = GridSpec(2, 1)  # Create a grid for subplots
        ax1 = plt.subplot(gs[0])  # First subplot for beta analysis
        ax1.set_title("Effect of Infection Rate (  ) on Infected Population")
        ax1.set_xlabel("Time")
        ax1.set_ylabel("Number of Infected Individuals")
        original_beta = state_dict['beta']  # Save the original beta value
        original_gamma = state_dict['gamma']  # Save the original gamma value
        colors = ["b", "g", "r", "c", "m", "y"]  # Colors for plotting

        # Analyze the effect of different beta values
        for i, beta in enumerate(beta_values):
        state_dict['beta'] = beta  # Update the beta value
        _, _, I_values, _ = rk4_method(state_dict, t_span, h) # Solve the model
        color = colors[i % len(colors)]  # Choose a color for the current beta value
        ax1.plot(np.linspace(t_span[0], t_span[1], len(I_values)), I_values,
        "-", color=color, label=f"  ={beta}")  # Plot the results
        state_dict['beta'] = original_beta  # Restore the original beta value

        ax2 = plt.subplot(gs[1])  # Second subplot for gamma analysis
        ax2.set_title("Effect of Recovery Rate (  ) on Infected Population")
        ax2.set_xlabel("Time")
        ax2.set_ylabel("Number of Infected Individuals")
        # Analyze the effect of different gamma values
        for i, gamma in enumerate(gamma_values):
        state_dict['gamma'] = gamma  # Update the gamma value
        _, _, I_values, _ = rk4_method(state_dict, t_span, h) # Solve the model
        color = colors[i % len(colors)]  # Choose a color for
```

```
            the current gamma value
        ax2.plot(np.linspace(t_span[0], t_span[1], len(I_values
            )), I_values,
        "-", color=color, label=f"  ={gamma}")  # Plot the
            results
        state_dict['gamma'] = original_gamma  # Restore the
            original gamma value

        # Add legends to the plots
        ax1.legend()
        ax2.legend()
        plt.tight_layout()  # Adjust layout to prevent overlap
        plt.savefig("sir_parameter_analysis.png", dpi=300)  #
            Save the plot as an image
        plt.show()  # Display the plot
```

- This function creates two subplots: one for the effect of `beta` and one for the effect of `gamma`.
- The infected population is plotted for each value of `beta` and `gamma`, showing how these parameters influence the spread of the disease.

2.8 Calculating the Basic Reproduction Number (R0)

The `calculate_r0` function calculates the basic reproduction number `R0`.

```
        def calculate_r0(state_dict):
        """Calculate the basic reproduction number R0, which
            indicates how contagious the disease is."""
        return state_dict['beta'] / state_dict['gamma']  # R0 =
             beta / gamma
```

- `R0` is defined as the ratio of the infection rate (`beta`) to the recovery rate (`gamma`).

2.9 Calculating Summary Statistics

The `summary_statistics` function calculates summary statistics for the epidemic.

```
        def summary_statistics(state_dict, t_span, h):
        """
        Calculate and return summary statistics for the
            epidemic, such as the peak number of infected
            individuals,
        the time to reach the peak, and the final size of the
            epidemic.

```

```
        Parameters:
        state_dict (dict): Dictionary containing model
            parameters.
        t_span (array): Time span [t_start, t_end] for the
            simulation.
        h (float): Step size for the simulation.

        Returns:
        dict: A dictionary containing summary statistics.
        """
        _, _, I_values, R_values = rk4_method(state_dict,
            t_span, h)  # Solve the model
        max_infected = np.max(I_values)  # Maximum number of
            infected individuals
        time_to_peak = np.argmax(I_values) * h  # Time to reach
             the peak infection
        final_size = R_values[-1]  # Final number of recovered
            individuals
        r0 = calculate_r0(state_dict)  # Calculate R0
        return {
            'r0': r0,  # Basic reproduction number
            'max_infected': max_infected,  # Peak number of
                infected individuals
            'time_to_peak': time_to_peak,  # Time to reach the
                peak infection
            'final_size': final_size,  # Final size of the
                epidemic
            'attack_rate': final_size / state_dict['N'] * 100
                # Attack rate (percentage of population infected
                )
        }
```

- This function calculates summary statistics for the epidemic, including:
  - `R0`: The basic reproduction number.
  - `max_infected`: The peak number of infected individuals.
  - `time_to_peak`: The time at which the peak infection occurs.
  - `final_size`: The final number of recovered individuals.
  - `attack_rate`: The percentage of the population that was infected during the epidemic.

2.10 Main Function

The `main` function is the entry point of the program.

```
        def main():
        """
```

```
    Main function to run the SIR model simulation and analysis.
    This function sets up the model, runs simulations, and generates plots.
    """
    # Create the initial state of the SIR model
    beta = 0.3  # Infection rate
    gamma = 0.1  # Recovery rate
    S0 = 990  # Initial susceptible population
    I0 = 10  # Initial infected population
    R0 = 0  # Initial recovered population
    state_dict = create_sir_state(beta, gamma, S0, I0, R0)  # Initialize the model

    # Run simulations using Euler's method and RK4 method
    t_span = [0, 100]  # Time span for the simulation (0 to 100 days)
    step_sizes = [0.1, 0.5, 1.0]  # Step sizes to compare
    t_euler, S_euler, I_euler, R_euler = euler_method(state_dict, t_span, 0.1)  # Euler method
    t_rk4, S_rk4, I_rk4, R_rk4 = rk4_method(state_dict, t_span, 0.1)  # RK4 method

    # Plot the basic simulation results
    plt.figure(figsize=(12, 8))  # Create a figure for plotting
    plt.subplot(2, 1, 1)  # First subplot for Euler method
    plt.title("SIR Model - Euler Method")
    plt.plot(t_euler, S_euler, "b-", label="Susceptible")  # Plot susceptible population
    plt.plot(t_euler, I_euler, "r-", label="Infected")  # Plot infected population
    plt.plot(t_euler, R_euler, "g-", label="Recovered")  # Plot recovered population
    plt.xlabel("Time (days)")
    plt.ylabel("Population")
    plt.legend()  # Add a legend
    plt.grid(True)  # Add a grid

    plt.subplot(2, 1, 2)  # Second subplot for RK4 method
    plt.title("SIR Model - RK4 Method")
    plt.plot(t_rk4, S_rk4, "b-", label="Susceptible")  # Plot susceptible population
    plt.plot(t_rk4, I_rk4, "r-", label="Infected")  # Plot infected population
    plt.plot(t_rk4, R_rk4, "g-", label="Recovered")  # Plot recovered population
    plt.xlabel("Time (days)")
    plt.ylabel("Population")
    plt.legend()  # Add a legend
```

```
        plt.grid(True)  # Add a grid

        plt.tight_layout()  # Adjust layout to prevent overlap
        plt.savefig("sir_basic_simulation.png", dpi=300)  # Save the plot as an image
        plt.show()  # Display the plot

        # Compare the accuracy of Euler's method and RK4 method
        compare_methods(state_dict, t_span, step_sizes)

        # Analyze the effect of different beta and gamma values
        beta_values = [0.1, 0.2, 0.3, 0.4, 0.5]  # List of beta values to analyze
        gamma_values = [0.05, 0.1, 0.15, 0.2, 0.25]  # List of gamma values to analyze
        analyze_parameters(state_dict, t_span, 0.1, beta_values, gamma_values)

        # Print summary statistics for the epidemic
        stats = summary_statistics(state_dict, t_span, 0.1)
        print("\nSummary Statistics:")
        print(f"Basic Reproduction Number (R0): {stats['r0']:.2f}")  # Print R0
        print(f"Maximum Number of Infected: {stats['max_infected']:.2f}")  # Print peak infections
        print(f"Time to Peak Infection: {stats['time_to_peak']:.2f} days")  # Print time to peak
        print(f"Final Epidemic Size: {stats['final_size']:.2f}")  # Print final size
        print(f"Attack Rate: {stats['attack_rate']:.2f}%")  # Print attack rate
```

- The `main` function initializes the SIR model, runs simulations using Euler's method and RK4 method, and generates plots.
- It also compares the accuracy of the two methods, analyzes the effect of different parameters, and prints summary statistics.

2.11 Running the Script

The script is executed by calling the `main` function.

```
        if __name__ == "__main__":
        main()  # Run the main function
```

- This block ensures that the `main` function is only executed when the script is run directly, not when it is imported as a module.

3 Conclusion

This document provides a detailed explanation of the implementation and analysis of the SIR model using Python. The code uses numerical methods (Euler's method and the 4th-order Runge-Kutta method) to solve the differential equations of the SIR model and analyzes the results. The results are visualized using plots, and summary statistics are calculated to provide insights into the epidemic.