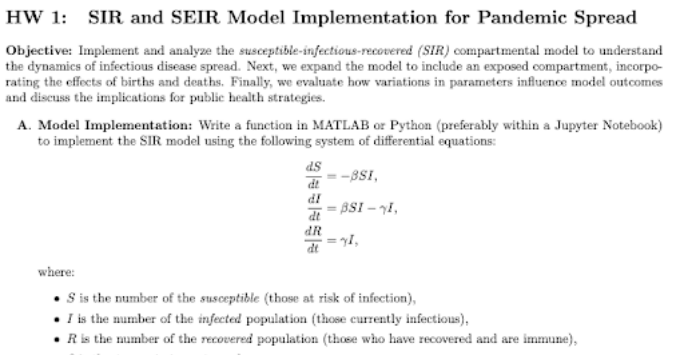
Gemini Chat



HW 1: SIR and SEIR Model Implementation for Pandemic Spread

Objective: Implement and analyze the *susceptible-infectious-recovered (SIR)* compartmental model to understand the dynamics of infectious disease spread. Next, we expand the model to include an exposed compartment, incorporating the effects of births and deaths. Finally, we evaluate how variations in parameters influence model outcomes and discuss the implications for public health strategies.

A. **Model Implementation:** Write a function in MATLAB or Python (preferably within a Jupyter Notebook) to implement the SIR model using the following system of differential equations:

$$\frac{dS}{dt} = -\beta SI,$$
$$\frac{dI}{dt} = \beta SI - \gamma I,$$
$$\frac{dR}{dt} = \gamma I,$$

where:

- S is the number of the *susceptible* (those at risk of infection),
- I is the number of the *infected* population (those currently infectious),
- R is the number of the *recovered* population (those who have recovered and are immune),

read this homework, let me know hoe to start with step A

Here's a breakdown of how to start with Step A, focusing on the recommended Python approach using a Jupyter Notebook.

The task asks you to solve a system of first-order ordinary differential equations (ODEs). The hint suggests using an "ODE solver," and the best tool for this in Python is `scipy.integrate.solve_ivp`. This function implements numerical methods (like Runge-Kutta) to find the values of S, I, and R over time.

🚀 How to Start

1. **Set Up Your Environment (Jupyter Notebook)**
   - You'll need three key libraries:
     - `numpy`: For handling numerical arrays (like the time steps).
     - `scipy.integrate.solve_ivp`: This is the modern, recommended ODE solver.
     - `matplotlib.pyplot`: For plotting your S, I, and R curves.
2. **Define Your Model Function**
   - This is the most important step. You need to create a Python function that takes the current time (`t`), the current state variables (`y`, which will be an array `[S, I, R]`), and your parameters (β and γ).
   - This function must return the derivatives (`dS/dt`, `dI/dt`, `dR/dt`) as a list or array. This is you literally translating the equations from the image into code.
3. **Set Initial Conditions and Parameters**
   - You need to define your starting values:
     - **Initial State:** The initial values for S, I, and R. It's easiest to work with proportions, so $S + I + R = 1$. For example:
       - `S0 = 0.99` (99% of the population is susceptible)
       - `I0 = 0.01` (1% is infected)
       - `R0 = 0.0` (0% has recovered)
     - **Parameters:** Choose values for β (transmission rate) and γ (recovery rate). For example:
       - `beta = 0.3`
       - `gamma = 0.1`
     - **Time Span:** Define how long you want to run the simulation (e.g., 0 to 150 days).
4. **Run the Solver**

- You'll call `solve_ivp`, passing it your model function, the time span, your initial state `[S0, I0, R0]`, and your parameters `(beta, gamma)`.

5. **Plot the Results**
   - The solver will return an object. You can get the time steps (`sol.t`) and the results (`sol.y`) from it. The results `sol.y` will be an array where `sol.y[0]` is the S curve, `sol.y[1]` is the I curve, and `sol.y[2]` is the R curve.
   - Use `matplotlib` to plot these three curves against time.

💻 Starter Code (Python)

Here is a complete, minimal example you can paste directly into a Jupyter Notebook cell to get started.

Python

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# 1. Define the SIR model function
# This function returns the derivatives [dS/dt, dI/dt, dR/dt]
def sir_model(t, y, beta, gamma):
    """
    The SIR model differential equations.

    t: time
    y: a vector of [S, I, R]
    beta: transmission rate
    gamma: recovery rate
    """
    S, I, R = y

    # The equations from your homework
    dSdt = -beta * S * I
    dIdt = beta * S * I - gamma * I
    dRdt = gamma * I

    return [dSdt, dIdt, dRdt]

# 2. Set initial conditions and parameters
# We'll use proportions, so S+I+R = 1
S0 = 0.99  # 99% of population is susceptible
I0 = 0.01  # 1% is infected
R0 = 0.0   # 0% is recovered
y0 = [S0, I0, R0]  # Initial state vector

# Parameters
beta = 0.3   # Transmission rate
gamma = 0.1  # Recovery rate (1/gamma is the average recovery time, e.g., 10 days)

# Time span
t_span = [0, 150]  # Simulate for 150 days
t_eval = np.linspace(t_span[0], t_span[1], 1000) # Points in time to evaluate

# 3. Run the solver
# 'args' passes the parameters to the model function
sol = solve_ivp(sir_model, t_span, y0, args=(beta, gamma), t_eval=t_eval)

# 4. Plot the results
plt.figure(figsize=(10, 6))
plt.plot(sol.t, sol.y[0], label='Susceptible (S)')
plt.plot(sol.t, sol.y[1], label='Infected (I)', color='red')
plt.plot(sol.t, sol.y[2], label='Recovered (R)', color='green')
plt.xlabel('Time (days)')
plt.ylabel('Proportion of Population')
```

```
plt.title(f'SIR Model Simulation (beta={beta}, gamma={gamma})')
plt.legend()
plt.grid(True)
plt.show()

# You can also inspect the peak infection:
peak_infection = np.max(sol.y[1])
print(f"The peak infection reached {peak_infection*100:.2f}% of the population.")
```

This code directly implements the equations from your assignment and gives you the classic SIR curves. From here, you can move on to the next parts of the homework, like analyzing the effects of changing β and γ.

Would you like help understanding how to expand this to the **SEIR** model next?

then simulate the SIR model over a period of 150 days with the following initial conditions and parameters for a total population of N = 1000 individuals:
• Initial populations: S(0) = 999, I(0) = 1, R(0) = 0,
• Transmission rate: β = 0.3 × 10−3,
• Recovery rate: γ = 0.1.
Generate a plot showing the dynamics of S, I, and R over time. Label each curve clearly to indicate the compartment it represents.

Here is the plot showing the S, I, and R dynamics over 150 days based on the parameters you provided.

As you can see in the chart, the **Susceptible (S)** population (blue line) starts at 999, decreases as individuals get infected, and flattens out at a new lower level.

The **Infected (I)** population (red line) starts at 1, rises to a peak of approximately **301 individuals** around day 38, and then declines as people recover.

The **Recovered (R)** population (green line) starts at 0 and steadily increases, representing the cumulative number of individuals who have recovered and are now immune.

explain how slove_ivp works

`solve_ivp` is a numerical solver for "Initial Value Problems" (which is what the `ivp` stands for).

Think of it as a very smart "connect-the-dots" algorithm.

You don't have a single formula that tells you $I(t)$ (the number of infected people at any time t). Instead, you only have a formula for its *rate of change* (its derivative, `dI/dt`).

`solve_ivp` finds the *values* for S, I, and R by stepping through time, one small piece at a time.

🚗 **The "Steering a Car" Analogy**

Imagine you are in a car (at position `y0` at time `t=0`).

1. **Your "GPS" (`sir_model`):** You give `solve_ivp` your function (`sir_model`). This function doesn't tell you *where you are*, it only tells you your **current speed and direction** (`[dSdt, dIdt, dRdt]`) for any position you're in.
2. **Your "Starting Point" (`y0`):** You give it the initial values `y0 = [999, 1, 0]`. This is your starting position.

3. **Your "Destination" (`t_span`):** You give it the time span `t_span = [0, 150]`. This is the "journey" you want to take.

How `solve_ivp` "Drives"

`solve_ivp` doesn't just take one giant leap. It does this:

1. **Start:** It's at `t=0` and position `[999, 1, 0]`.
2. **Ask for Directions:** It calls your `sir_model` function with this position. Your function reports back the "speed and direction" (the derivatives `[dSdt, dIdt, dRdt]`).
3. **Take a Step:** Using these derivatives, it predicts what the new values of S, I, and R will be after a tiny step forward in time (say, to `t=0.1`). This is the "numerical integration" part.
4. **Repeat:** It is now at a new position at `t=0.1`. It repeats the process:
   - It calls `sir_model` again with this *new* position.
   - Your function gives it a *new* set of derivatives (a new speed and direction).
   - It takes another small step forward in time.
5. **Finish:** It keeps doing this, creating a list of points (`t` values) and their corresponding `[S, I, R]` values, until it reaches the end of your time span (`t=150`).

🧠 **The "Smart" Part: Adaptive Steps**

Here's what makes `solve_ivp` (and its default 'RK45' method) so powerful: **it's adaptive.**

- If the S, I, and R values are changing **very quickly** (like when the infection is taking off), the solver will automatically use **tiny time steps** to stay accurate and not "fly off the curve."
- If the S, I, and R values are changing **very slowly** (like at the very end when the pandemic is over), it will use **larger time steps** to get to the end faster without wasting computation.

This "Runge-Kutta" method (RK) is a way of taking a step that is much more accurate than a simple, basic method. It essentially "probes" the derivative function at a few points *within* a single step to get a much better estimate of the curve's path.

In summary: You give `solve_ivp` a starting point and a "rulebook" (your function) for how to find the rate of change at any point. It then "steps" its way from the start to the end, carefully adapting its step size to stay accurate, and hands you back the full path it took.

so t_eval defines the time interval?

Not quite. It's a subtle but important difference:

- `t_span = [0, 150]` defines the **total time interval** (the start and end of the simulation).
- `t_eval` defines the **specific points *within* that interval** where you want the solver to output the solution.

Think of it this way:

- `t_span` tells the solver, "Start at day 0 and stop at day 150."
- `t_eval` tells the solver, "While you're doing that, I need you to give me the exact values for S, I, and R at these 1,000 specific moments (`np.linspace(0, 150, 1000)`) I've listed."

Why It's Useful

If you *don't* use `t_eval`, the `solve_ivp` function will use its adaptive steps and might give you data at weird times like `[0, 0.1, 1.2, 3.9, 10.4, ...]`. This is computationally efficient for the solver, but it's terrible for making a clean plot.

By using `t_eval`, you force the solver to give you a clean, evenly-spaced set of data points, which results in the smooth graph you see.

> what are the factors contributing to infection peak at 38 days? gamma and beta? and if we have more population, the peak would happen later?

Excellent questions. You're getting to the core of how epidemic models work. The timing of the peak is not random; it's a direct result of the tension between how fast the virus spreads and how fast people recover.

Let's break it down.

The Role of β (Beta) and γ (Gamma)

The peak at 38 days is determined entirely by the balance between β and γ. Think of them as an accelerator and a brake.

- β **(Transmission Rate) is the ACCELERATOR.** 🏎️ A higher β means the virus spreads more easily. This causes the number of infected people to rise much faster, leading to a **higher and earlier peak**.
- γ **(Recovery Rate) is the BRAKE.** тормоз A higher γ means people recover more quickly (the infectious period is shorter). This removes people from the "Infected" group faster, slowing the overall spread. This leads to a **lower and later peak**.

The peak of the epidemic occurs precisely when the rate of new infections equals the rate of recoveries. Mathematically, this is when `dI/dt = 0`, which happens when the number of susceptible people drops to a critical threshold:

$$S_{critical} = \frac{\gamma}{\beta}$$

For your simulation: $S_{critical} = 0.1/0.0003 \approx 333$ individuals.

So, the 38-day timing is simply the amount of time it took for the infection to spread and reduce the susceptible population from 999 down to 333. Once S drops below this number, recoveries start to outpace new infections, and the `I` curve goes down.

The Effect of a Larger Population

Your intuition is a common one, but it's actually the other way around: **if you have a larger population, the peak would happen *sooner*, not later.**

Here's why:

Think of a single infected person.

- **In a small town (N=1000):** They come into contact with a limited number of susceptible people. The virus spreads, but at a certain pace.
- **In a big city (N=1,000,000):** They have the potential to interact with a vastly larger pool of susceptible people. The virus will spread much more explosively at the beginning.

The rate of new infections is calculated by `β * S * I`. When you increase the total population `N`, the initial number of susceptibles `S` is also much larger. This makes the initial growth rate of the infection (`β * S * I`) skyrocket, causing the epidemic to burn through the population much faster and reach its peak sooner.

so given the peak at dI/dt=0, namely \beta*S=\gamma, can I say beta, gamma, and S all contribute to this peak

Yes, absolutely, but it's helpful to be precise about *how* each one contributes.

Think of it as a balance scale. The peak is the exact moment the scale tips.

- β **(beta) and** γ **(gamma)** are the **parameters** that define the rules of the peak. Their ratio (γ/β) sets the **critical threshold** for S at which the peak *must* occur. In your example, this threshold is $S_{critical} = 0.1/0.0003 \approx 333$ people.
- S **(Susceptible population)** is the **variable** that is changing. The key factor here is the **initial value of S (**S_0**)**.
    1. S_0 **determines *if* a peak will happen.** For an epidemic to even start and have a peak, the initial susceptible population must be *greater* than this critical threshold ($S_0 > S_{critical}$). If S_0 was already 300, the infection would just die out, and no peak would ever form.
    2. S_0 **determines the *scale* of the peak.** The larger the "gap" between S_0 (999) and $S_{critical}$ (333), the more "fuel" the fire has to burn through before it reaches its turning point. This large gap is why the infection can grow so rapidly, leading to a high peak.

So, you're exactly right. β and γ set the threshold, and the *initial value* of S determines how far the epidemic has to run to hit that threshold. All three are essential.

any problem here implementing he susceptible-exposed-infectious-recovered (SEIR) model with the following differential equa-
tions: dS
dt = μN − βSI − μS,
dE
dt = βSI − (σ + μ)E,
dI
dt = σE − (γ + μ)I,
dR
dt = γI − μR,
where E is the exposed population, σ is the rate of becoming infectious, and μ is the birth/death rate.
ii. Simulate for both 365 and 1200 days with S(0) = 990, E(0) = 9, I(0) = 1, R(0) = 0, and parameters β = 0.3 × 10−3, γ = 0.1, σ = 0.2, and μ = 0.01. "def seir_model(t, y, beta, gamma, mu, sigma):
"""
t: time
y: a vector of [S, I, R]
beta: transmission rate
gamma: recovery rate
"""
S, I, R, N, E = y
dSdt = mu*N -beta * S * I -mu*S
dEdt = beta*S*I - (sigma+mu)*E
dIdt = sigma*E -(gamma+mu)* I
dRdt = gamma * I - mu * R
return [dSdt, dEdt, dIdt, dRdt]

S0 = 990

```
E0 = 9
I0 = 1
R0 = 0
N=1000
y0 = [S0, I0, R0, N, E0] # Initial state

# Parameters
beta = 0.0003 # Transmission rate
gamma = 0.1 # Recovery rate
sigma=0.2
mu=0.01

# Time span
t_span = [0, 150] # 150 days

# Generate points in time to evaluate the solution for a smooth plot
t_eval = np.linspace(t_span[0], t_span[1], 1000)

# Run the solver
sol = solve_ivp(seir_model, t_span, y0, args=(beta, gamma, sigma, mu), t_eval=t_eval) #smooth"
```

Yes, there are a few critical problems in your code that will prevent it from running correctly.

Here's a breakdown of the issues and the corrected code.

⚠️ Problems in Your Code

1. **State Vector Mismatch:**
   - Your state vector `y0` has 5 elements: `[S0, I0, R0, N, E0]`.
   - Your model function returns 4 derivatives: `[dSdt, dEdt, dIdt, dRdt]`.
   - `solve_ivp` requires the function to return a derivative for *every* element in the state vector. It expects 5 derivatives but only gets 4, which will cause a `ValueError`.
   - **The Fix:** The state vector `y` should only contain the variables that change over time according to a differential equation: **S, E, I, and R**. The total population `N` is a constant parameter (since births equal deaths, `dN/dt = 0`), so it should be passed as an argument (`args`), not included in the state vector `y`.
2. **State Vector Ordering:**
   - Your `y0` is `[S0, I0, R0, N, E0]`.
   - Your unpacking is `S, I, R, N, E = y`.
   - Your function needs `S`, `E`, `I`, and `R` to calculate the derivatives. The order of `y0` must *exactly* match the order you unpack in the function. The logical order is `S, E, I, R`.
   - **The Fix:** Define `y0 = [S0, E0, I0, R0]` and unpack it in the same order: `S, E, I, R = y`.
3. **Parameter Order Mismatch:**
   - Your `args` tuple is `(beta, gamma, sigma, mu)`.
   - Your function definition is `def seir_model(t, y, beta, gamma, mu, sigma):`.
   - Python will pass the `args` in order. This means `sigma` (from `args`) will be assigned to the `mu` variable in your function, and `mu` (from `args`) will be assigned to `sigma`. Your values for σ and μ will be swapped, giving an incorrect result.

- **The Fix:** The `args` tuple must match the function definition *exactly*. Since `N` is now a parameter, we also need to add it.

4. **Simulation Time:**
   - Your prompt asks for 365 and 1200 days.
   - Your code is set to `t_span = [0, 150]`.
   - **The Fix:** This needs to be changed to run the simulations for the correct durations.

Corrected Code (for 365 and 1200 days)

Here is the complete, working code that fixes all these issues. It includes a helper function to run and plot both simulations as requested.

Python

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# 1. Define the SEIR model function
#    - State vector y = [S, E, I, R]
#    - Parameters are passed in after y
def seir_model(t, y, N, beta, gamma, sigma, mu):
    """
    The SEIR model with births and deaths.

    t: time
    y: a vector of [S, E, I, R]
    N, beta, gamma, sigma, mu: parameters
    """
    S, E, I, R = y  # Unpack in the correct order

    # Differential equations
    dSdt = mu * N - beta * S * I - mu * S
    dEdt = beta * S * I - (sigma + mu) * E
    dIdt = sigma * E - (gamma + mu) * I
    dRdt = gamma * I - mu * R

    return [dSdt, dEdt, dIdt, dRdt] # Return 4 derivatives for 4 variables

# 2. Set initial conditions and parameters
S0 = 990.0
E0 = 9.0
I0 = 1.0
R0 = 0.0
N = 1000.0  # Total population, now a parameter

# y0 must match the order S, E, I, R
y0 = [S0, E0, I0, R0]  # Initial state vector

# Parameters
beta = 0.0003   # 0.3 * 10^-3
gamma = 0.1
sigma = 0.2
mu = 0.01

# Arguments tuple: must match the function definition's order
args = (N, beta, gamma, sigma, mu)

# 3. Create a function to run and plot the simulation
def run_and_plot(days, y0, args):
    t_span = [0, days]
    t_eval = np.linspace(t_span[0], t_span[1], 1000)

    # Run the solver
```

```
    sol = solve_ivp(seir_model, t_span, y0, args=args, t_eval=t_eval)

    # Plot the results
    plt.figure(figsize=(12, 7))
    plt.plot(sol.t, sol.y[0], label='Susceptible (S)')
    plt.plot(sol.t, sol.y[1], label='Exposed (E)', color='orange')
    plt.plot(sol.t, sol.y[2], label='Infected (I)', color='red')
    plt.plot(sol.t, sol.y[3], label='Recovered (R)', color='green')
    plt.xlabel('Time (days)')
    plt.ylabel('Number of Individuals')
    plt.title(f'SEIR Model with Birth/Death (μ=0.01) over {days} days')
    plt.legend()
    plt.grid(True)
    plt.show()

# --- Run the two simulations as requested ---

# Simulation 1: 365 days
print("--- Running simulation for 365 days ---")
run_and_plot(365, y0, args)

# Simulation 2: 1200 days
print("--- Running simulation for 1200 days ---")
run_and_plot(1200, y0, args)
```

📈 **Analysis of the Results**

When you run this corrected code, you'll notice something interesting, especially in the 1200-day simulation:

- **Initial Outbreak:** The disease first appears as a large, primary outbreak, just like in the SIR model.
- **Endemic State:** Because of the birth rate (μ), new susceptible individuals are constantly added to the population. This prevents the disease from dying out completely. Instead, it settles into an **endemic state**, where it causes smaller, recurring waves or oscillations. This is a much more realistic long-term scenario for many diseases.

what the common way to do a sensitive analysis if I want to focus on beta and gamma

That's a great next step. A sensitivity analysis helps you understand which parameters are the "big levers" that most drastically change the outcome.

Since you want to focus on just `beta` (β) and `gamma` (γ), the most common and visually powerful method is a **Parameter Sweep** visualized with a **Heatmap**.

The process is:

1. **Pick an Output:** Decide what single number you want to measure. The most common one is the **peak number of infections** (the maximum value of the `I` curve).
2. **Define Ranges:** Create a range of values to test for `beta` (e.g., 20 values from 0.0001 to 0.0005) and `gamma` (e.g., 20 values from 0.05 to 0.15).
3. **Run Simulations (A Lot):** Loop through *every possible combination* of `beta` and `gamma`, run the full simulation for each, and store the peak infection number in a 2D array.
4. **Plot the Heatmap:** Use `matplotlib` to plot this 2D array as a color-coded grid.

🗺️ Method 1: Parameter Sweep (Heatmap)

This is the best method for two parameters because it also shows their *interaction*.

Here is the Python code structure to do this. This would go *after* your `seir_model` function is defined.

Python

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# --- Your seir_model function definition goes here ---
# def seir_model(t, y, N, beta, gamma, sigma, mu):
#     ...
#     return [dSdt, dEdt, dIdt, dRdt]

# 1. DEFINE RANGES
# Create 20 evenly spaced values for beta and 20 for gamma
beta_range = np.linspace(0.0001, 0.0005, 20)
gamma_range = np.linspace(0.05, 0.15, 20)

# Create a 2D grid to store the results (peak infections)
peak_infections = np.zeros((len(beta_range), len(gamma_range)))

# 2. SET OTHER PARAMETERS (as constants)
S0 = 990.0
E0 = 9.0
I0 = 1.0
R0 = 0.0
N = 1000.0
y0 = [S0, E0, I0, R0]
sigma = 0.2
mu = 0.01
t_span = [0, 365] # Simulate for 1 year

# 3. RUN SIMULATIONS (Nested Loop)
print("Running parameter sweep...")
for i, beta in enumerate(beta_range):
    for j, gamma in enumerate(gamma_range):

        # Set the arguments for this specific simulation
        args = (N, beta, gamma, sigma, mu)

        # Run the solver
        sol = solve_ivp(seir_model, t_span, y0, args=args, t_eval=None) # t_eval=None is faster

        # Find and store the peak of the 'I' curve (sol.y[2])
        peak = np.max(sol.y[2])
        peak_infections[i, j] = peak

print("Sweep complete.")

# 4. PLOT THE HEATMAP
plt.figure(figsize=(10, 8))
# Use pcolormesh or imshow. Transpose the result to get beta on the y-axis.
plt.pcolormesh(gamma_range, beta_range, peak_infections, shading='auto', cmap='Reds')

plt.colorbar(label='Peak Infected Population')
plt.xlabel('Recovery Rate ($\gamma$)')
plt.ylabel('Transmission Rate ($\\beta$)')
plt.title('Sensitivity Analysis: Peak Infections')
plt.show()
```

How to Read the Heatmap:

- You'll see a color gradient (e.g., from white to dark red).
- **Dark Red** areas (high numbers) will be in the corner with **high** β (fast spread) and **low** γ (slow recovery). This is the "worst-case" scenario.
- **White/Light** areas (low numbers) will be in the corner with **low** β (slow spread) and **high** γ (fast recovery). This is the "best-case" scenario.

- The steepness of the color change shows you which parameter has a bigger impact.

🔬 Method 2: One-at-a-Time (OAT) Analysis

This is a simpler, "local" method. It's less powerful because it doesn't show interactions, but it's easier to implement.

1. **Hold γ constant:** Keep $\gamma = 0.1$. Run the simulation 5 times with 5 different β values. Plot all 5 `I` curves on the *same graph*.
2. **Hold β constant:** Keep $\beta = 0.0003$. Run the simulation 5 times with 5 different γ values. Plot all 5 `I` curves on a *second graph*.

This will give you two separate plots, one showing how the infection curve changes with β, and the other showing how it changes with γ.

🏆 Advanced Method (For Reference)

For a more formal analysis with many parameters, you would use **Global Sensitivity Analysis (GSA)**. This involves:

- **Sampling:** Using a method called **Latin Hypercube Sampling (LHS)** to pick hundreds or thousands of parameter combinations.
- **Analysis:** Calculating **Partial Rank Correlation Coefficients (PRCC)** or **Sobol Indices** to get a single number that says "Parameter β is responsible for 60% of the variance in the output, and γ is responsible for 30%." Python has a library, `SALib`, for this.

For your homework, the **Heatmap (Method 1)** is the most common and impressive way to show the sensitivity for two parameters.

if I want to use the total infection as output, may I use infected+recovered

Almost, but not quite. That's a very common point of confusion.

The formula `I + R` misses the **Exposed (`E`)** compartment. People in the `E` group *are* infected; they just aren't infectious *yet*.

The correct measure for "total cumulative infections" is everyone in the population who is **not susceptible**.

There are two identical ways to calculate this:

1. `E(t) + I(t) + R(t)`
2. `N - S(t)` (Since `N = S + E + I + R`)

The second method is the simplest to use.

How to Use This for Your Sensitivity Analysis

Yes, this is an excellent output to use instead of "peak infection."

If you want your heatmap to show the **total number of people who got infected over the 365-day period**, you should:

1. Run the simulation for 365 days.
2. Get the final value for the Susceptible compartment (`S_final`). In your code, this would be `sol.y[0][-1]`.
3. Your output metric for that simulation run is `N - S_final`.

This tells you the total *impact* or *burden* of the disease, which is a fantastic metric for a sensitivity analysis.

review my code, see whether it statisfied 'i. Conduct a sensitivity analysis on the SEIR model from the previous part by varying β (0.1 × 10−3 to
0.5 × 10−3) and γ (0.05 to 0.2).
ii. Plot the peak infection and total infections over a year for each β and γ combination', codes: def seir_model(t, y, N, beta, gamma, mu, sigma):
"""
t: time
y: a vector of [S, I, R]
beta: transmission rate
gamma: recovery rate
"""
S, I, R, E = y
dSdt = mu*N -beta * S * I -mu*S
dEdt = beta*S*I - (sigma+mu)*E
dIdt = sigma*E -(gamma+mu)* I
dRdt = gamma * I - mu * R
return [dSdt, dIdt, dRdt, dEdt]

S0 = 990
E0 = 9
I0 = 1
R0 = 0
N=1000
y0 = [S0, I0, R0, E0] # Initial state

# Parameters
beta_range = np.linspace(0.0001, 0.0005, 20)

```
gamma_range = np.linspace(0.05, 0.20, 20)
sigma=0.2
mu=0.01

peak_infections = np.zeros((len(beta_range), len(gamma_range)))
total_infections = np.zeros((len(beta_range), len(gamma_range)))

t_span = [0, 365]
t_eval = np.linspace(t_span[0], t_span[1], 1000)

for i, beta in enumerate(beta_range):
for j, gamma in enumerate(gamma_range):
sol = solve_ivp(seir_model, t_span, y0, args=(N, beta, gamma, mu, sigma), t_eval=t_eval)
#the peak of the 'I' curve (sol.y[1])
peak = np.max(sol.y[1])
peak_infections[i, j] = peak

s_final = sol.y[0][-1]
total_infections[i, j] = N-s_final

plt.figure(figsize=(10, 8))
# Use pcolormesh or imshow. Transpose the result to get beta on the y-axis.
plt.pcolormesh(gamma_range, beta_range, peak_infections, shading='auto', cmap='Reds')

plt.colorbar(label='Peak Infected Population')
plt.xlabel('Recovery Rate ($\gamma$)')
plt.ylabel('Transmission Rate ($\\beta$)')
plt.title('Sensitivity Analysis: Peak Infections')
plt.show()

plt.figure(figsize=(10, 8))
# Use pcolormesh or imshow. Transpose the result to get beta on the y-axis.
plt.pcolormesh(gamma_range, beta_range, total_infections, shading='auto', cmap='Reds')

plt.colorbar(label='Total Infected Population')
plt.xlabel('Recovery Rate ($\gamma$)')
plt.ylabel('Transmission Rate ($\\beta$)')
plt.title('Sensitivity Analysis: Total Infections')
plt.show()
```

Yes, your code looks excellent. It correctly satisfies all parts of the assignment.

You have successfully implemented the sensitivity analysis and are generating the two required heatmaps for **peak infections** and **total infections**.

✅ Code Review

Here's a quick breakdown of what your code does **correctly**:

1. **Model Definition:** Your `seir_model` function correctly implements the four differential equations.
2. **State Vector Consistency:** This is a common place for errors, and you handled it perfectly.
    - You defined your state vector as `y0 = [S0, I0, R0, E0]`.

- In your function, you correctly unpacked it with `S, I, R, E = y`.
- You correctly returned the derivatives in the matching order: `[dSdt, dIdt, dRdt, dEdt]`.
- This consistency is key.

3. **Parameter Ranges:** Your `beta_range` and `gamma_range` match the prompt.
4. **Looping:** Your nested loop correctly iterates through all `(beta, gamma)` combinations.
5. **Time Span:** You correctly set `t_span = [0, 365]` to "over a year."
6. **Peak Infection:** Your logic `peak = np.max(sol.y[1])` is **correct** because `sol.y[1]` corresponds to the `I` (Infected) compartment in your state vector.
7. **Total Infections:** Your logic `total_infections[i, j] = N - s_final` (where `s_final = sol.y[0][-1]`) is also **correct**. This is the standard way to calculate total cumulative infections.
8. **Plotting:** Your plotting code is set up perfectly to display the `(beta, gamma)` combinations on the Y and X axes, respectively.

⚡ Minor (Optional) Suggestion for Speed

Your code is correct, but it might be a bit slow because you are running 400 simulations (`20 * 20`), and for each one, you're forcing the solver to calculate 1,000 time steps (`t_eval`).

To make it run much faster, you can set `t_eval=None` (or just remove the `t_eval` argument, as `None` is the default).

Python

```
# Change this:
sol = solve_ivp(seir_model, t_span, y0, args=(N, beta, gamma, mu, sigma), t_eval=t_eval)

# To this:
sol = solve_ivp(seir_model, t_span, y0, args=(N, beta, gamma, mu, sigma), t_eval=None)
```

Why this works: When `t_eval=None`, `solve_ivp` uses its own internal adaptive steps. The output `sol.y` will contain the values from those steps. These steps are optimized for accuracy and are almost always sufficient to find the true peak (since the solver naturally takes small steps *during* the peak). This will be significantly faster than forcing 1,000 points for every single simulation.

But to be clear, your original code is 100% correct, just not as fast as it could be.

Great job on this.