

BMI 500: Homework 11

Name: Loyani Loyani

Program: CSI PhD

Department: Biomedical Informatics, Emory University

Contact: loyani.loyani@emory.edu

Question: HW 1- SIR and SEIR Model Implementation for Pandemic Spread

Implementation

<https://github.com/loyanie/SIR-and-SEIR-Model-Implementation/blob/main/README.md>

Disclaimer: ChatGPT was used to complete HW 1 to understand the concepts of numerical methods and how they are implemented with Python libraries

Chat History

<https://chatgpt.com/share/69150088-bae8-800e-b38f-26253d97ea85>

Chat history

You said:

what is a python library for ODE to iteratively solve differential equations

ChatGPT said:

A few excellent **Python libraries** can be used to **iteratively solve ordinary differential equations (ODEs)**, depending on your use case — from simple numerical integration to complex system simulation. Here are the main ones:



1. SciPy (scipy.integrate)



Best for: General-purpose ODE solving (e.g., physics, biology, engineering)

- **Functions:**

- `solve_ivp()` — modern interface supporting several solvers
(RK45, Radau, BDF, etc.)
- `odeint()` — classic function (wrapper for LSODA from ODEPACK)

- **Example:**

```
from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt
def dydt(t, y):
    return -2 * y + np.sin(t)
sol = solve_ivp(dydt, [0, 10], [1], method='RK45',
t_eval=np.linspace(0, 10, 100))
plt.plot(sol.t, sol.y[0])
plt.xlabel('t')
plt.ylabel('y(t)')
plt.show()
```



2. SymPy (sympy.solvers.ode)

Best for: Symbolic (analytic) solutions or verifying numerical solvers

- Solves ODEs analytically if possible.
- Can generate numerical functions via `lambdify()` for iterative use.
- **Example:**

```
from sympy import Function, dsolve, Eq, Derivative, symbols
t = symbols('t')
y = Function('y')
ode = Eq(Derivative(y(t), t), -2 * y(t) + 1)
sol = dsolve(ode)
print(sol)
```



3. ODEINT from torchdiffeq

Best for: Neural ODEs and deep learning integration (PyTorch-based)

- Used for differentiable ODE solvers in neural networks.
- Example:**

```
from torchdiffeq import odeint import torch
class ODEFunc(torch.nn.Module):
    def forward(self, t, y):
        return -2 * y + torch.sin(t)
y0 = torch.tensor([1.0])
t = torch.linspace(0, 10, 100)
y = odeint(ODEFunc(), y0, t)
```



4. JAX-based libraries (diffraex, jax.experimental.ode)

Best for: Differentiable solvers with automatic differentiation (for research in ML/physics)

- Example: **Diffraex**

```
import jax.numpy as jnp
import diffraex
def f(t, y, args):
    return -2 * y + jnp.sin(t)
solver = diffraex.Tsit5()
y0 = jnp.array([1.0])
t0, t1 = 0, 10
saveat = diffraex.SaveAt(ts=jnp.linspace(0, 10, 100))
sol = diffraex.diffeqsolve(diffraex.ODETerm(f), solver, t0, t1, dt0=0.1, y0=y0,
                            saveat=saveat)
```



Summary Table

Library	Type	Differentiable	Symbolic	Typical Use
scipy.integrate	Numerical	✗	✗	General ODE solving
sympy.solvers.ode	Symbolic	✗	✓	Analytic math, derivations

Library	Type	Differentiable	Symbolic	Typical Use
torchdiffeq	Numerical	✓	✗	Neural ODEs, deep learning
diffra	Numerical	✓	✗	JAX-based differentiable solvers

Would you like me to show an example where the solver **iteratively steps through the ODE (looping over time)** — for instance, implementing Euler's or Runge–Kutta manually?

You said:

Which numerical method (e.g., Runge–Kutta, ODE solvers or discretizing the differential equations) works better in modelling the dynamics of infectious disease spread and how can it be implemented in Python

ChatGPT said:

Thought for 8s

Short answer: there isn't a single "best" method—it depends on the model and data. For most compartmental epidemic models (SIR/SEIR and extensions), a **variable-step Runge–Kutta (RK45/DOPRI5) or an implicit stiff solver (BDF/Radau)** works best in practice. Use:

- **RK45 (explicit, adaptive)** when the system is not stiff (smooth dynamics, moderate R_0 , no extremely fast/slow processes).
- **BDF or Radau (implicit, stiff)** when you add **fast** processes (e.g., exposed → infectious is hours while recovery is weeks), **strong interventions** that create sharp changes, or **age/heterogeneity** that introduce disparate time scales.
- **Discrete-time difference equations** if your data arrive daily/weekly and you want likelihoods that match reporting cadence.
- **Stochastic (Gillespie / τ-leaping)** when populations are small or you care about extinction/variability (outbreak seeding, early phase, metapopulations).

Below are minimal, reliable Python patterns for each.

1) Deterministic ODE SIR

with `solve_ivp` (RK45 & BDF)

```
import numpy as np from scipy.integrate import solve_ivp import matplotlib.pyplot as plt # SIR with
optional time-varying beta(t) to reflect interventions def beta_t(t): # example: intervention at day 30
halves transmission return 0.3 if t < 30 else 0.15 def sir_rhs(t, y, N, gamma): S, I, R = y beta =
beta_t(t) dS = -beta * S * I / N dI = beta * S * I / N - gamma * I dR = gamma * I return [dS, dI, dR] N =
1_000_000 I0 = 10 R0 = 0 S0 = N - I0 - R0 gamma = 1/7 # average infectious period 7 days y0 = [S0,
I0, R0] t_span = (0, 160) t_eval = np.arange(0, 161) # Non-stiff: RK45 sol_rk = solve_ivp(sir_rhs,
t_span, y0, t_eval=t_eval, args=(N, gamma), method="RK45", rtol=1e-6, atol=1e-9) # Stiff: BDF (good
fallback if RK struggles) sol_bdf = solve_ivp(sir_rhs, t_span, y0, t_eval=t_eval, args=(N, gamma),
method="BDF", rtol=1e-6, atol=1e-9) plt.plot(sol_rk.t, sol_rk.y[1], label="I(t) RK45") plt.plot(sol_bdf.t,
sol_bdf.y[1], "--", label="I(t) BDF") plt.xlabel("Days"); plt.ylabel("Infectious"); plt.legend(); plt.show()
```

Tips:

- Start with **RK45**; if you see tiny steps, warnings, or instability, switch to **BDF**.
- Use **events** in `solve_ivp` to detect peaks, thresholds, or extinction.

2) Discrete-time (difference equation) model (data-aligned)

```
import numpy as np import matplotlib.pyplot as plt N = 1_000_000 I0 = 10 R0 = 0 S0 = N - I0 - R0
gamma = 1/7 beta = 0.25 # per-day transmission T = 160 S = np.zeros(T+1); I = np.zeros(T+1); R =
np.zeros(T+1) S[0], I[0], R[0] = S0, I0, R0 for t in range(T): new_inf = beta * S[t] * I[t] / N new_rec =
gamma * I[t] # Euler-style daily update (can add process noise or reporting noise here) S[t+1] = S[t] -
new_inf I[t+1] = I[t] + new_inf - new_rec R[t+1] = R[t] + new_rec plt.plot(I, label="I[t] (daily)")
plt.xlabel("Days"); plt.ylabel("Infectious"); plt.legend(); plt.show()
```

When your observations are daily counts, this discrete model often simplifies inference (state-space models, particle filters, negative-binomial observation models).

3) Stochastic simulation (Gillespie SSA)

```
import numpy as np import matplotlib.pyplot as plt
rng = np.random.default_rng(1)
def gillespie_sir(N, beta, gamma, S0, I0, R0, t_max):
    t, S, I, R = 0.0, S0, I0, R0
    ts = [t]
    ls = [I]
    while t < t_max and I > 0:
        rate_inf = beta * S * I / N
        rate_rec = gamma * I
        rate_tot = rate_inf + rate_rec
        if rate_tot <= 0:
            break
        next_event_time = t + rng.exponential(1.0 / rate_tot)
        if rng.random() < rate_inf / rate_tot:
            S -= 1
            I += 1
        else:
            I -= 1
            R += 1
        ts.append(next_event_time)
        ls.append(I)
    return np.array(ts), np.array(ls), N, beta, gamma
```

S0, I0, R0, t_max = 50_000, 0.25, 1/7
plt.step(ts, ls, where="post")
plt.xlabel("Days")
plt.ylabel("Infectious (stochastic)")
plt.show()

Use this when randomness matters (small populations, early seeding, fade-out probabilities). You can run many trajectories to get confidence bands.

How to choose, in practice

- **Teaching / baseline fits to incidence curves:** RK45 is simple and fast.
- **Large, multi-group/age models or strong intervention switches:** BDF/Radau handles stiffness robustly.
- **Daily surveillance data & likelihood-based inference:** discrete-time state-space model.
- **Early outbreak, importations, elimination probability:** Gillespie/ τ -leaping.
- **Parameter inference:** wrap any of the above
in `scipy.optimize.curve_fit`, `jax`/`torch` autodiff (for differentiable solvers
like `diffraffx`/`torchdiffeq`), or probabilistic programming (PyMC, NumPyro, Stan via
`CmdStanPy`).

If you tell me your specific model (e.g., SEIR with vaccination + waning + seasonality, or age-structured contact matrices), I'll tailor the solver choice and give you ready-to-run code.