# HPPC Assignment 1

Søren Gundesen (wjq812), Nicholas Posborg (zkg114)

February 2026

## 1 Introduction

In this project we study the spread of an infectious disease using the SIR model, which divides a fixed population of $N$ individuals into three compartments: $S(t)$ (susceptible), $I(t)$ (infected/infectious), and $R(t)$ (recovered and immune). The dynamics are controlled by two parameters: $\beta$ (infection rate) and $\gamma$ (recovery rate), where $1/\gamma$ is the mean infectious period. The model is described by the coupled differential equations

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -\beta\frac{IS}{N}, \qquad \frac{\mathrm{d}I}{\mathrm{d}t} = \beta\frac{IS}{N} - \gamma I, \qquad \frac{\mathrm{d}R}{\mathrm{d}t} = \gamma I. \tag{1}$$

We solve these equations numerically by discretizing time with a timestep $\Delta t$ and investigating how the choice of $\Delta t$ affects the solution.

## 2 Implementation of SIR Model

We implemented the SIR model in `C++` using an explicit (Forward) Euler time-stepping scheme. First, we defined a `take_step` function that updates the three state variables $S$ (susceptible), $I$ (infected), and $R$ (recovered). In each call, the time derivatives ($\mathrm{d}S/\mathrm{d}t$, $\mathrm{d}I/\mathrm{d}t$, $\mathrm{d}R/\mathrm{d}t$) are evaluated and the solution is advanced by one timestep $\Delta t$. The updated values are returned as a new state vector, which is then used as input for the next iteration.

In the `main` function, we specified the initial conditions, model parameters, and timestep size, and opened an output file for storing the results. The system was integrated forward in time for a fixed simulation duration, writing $(t, S, I, R)$ to the file at a chosen output frequency. Finally, the saved data were loaded in Python and plotted to visualize the evolution of $S(t)$, $I(t)$, and $R(t)$ for different timestep sizes.

## 3 Correctness test

To check our model for correctness we ran it with varying model parameters to see if the results were as we would expect. In figure 1 the results of tests $\beta$ values $I_0$ respectively is shown. These results correspond to expectations as we see the $\beta$ value having an impact on the timescale at which the system reaches a steady state, and how the steady state looks. We also check $\beta = 0$ and see this makes the system quite boring to analyse. When varying the initial amount of infected we see that for values $I_0 \neq 0$ the value of $I_0$ does not impact how the steady state looks but only the rate at which this state is reached. For $I_0 = 0$ we again see that the system is entirely steady.

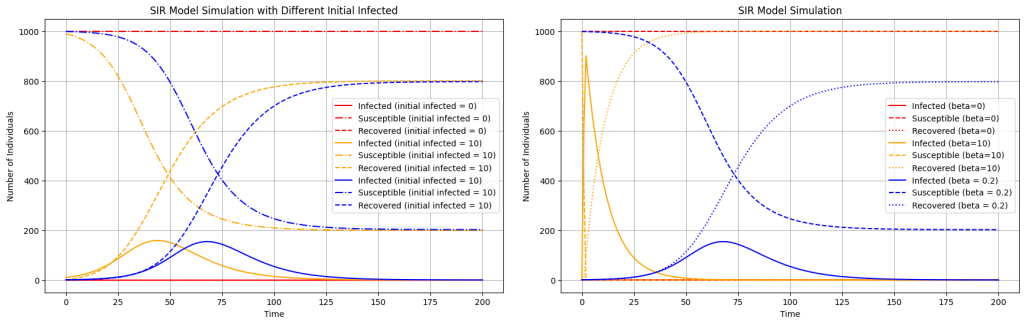


Figure 1: Results of model with different $\beta$ and $I_0$ values

# 4 Results

Results for different dt values can be seen in figure 2, we chose $\Delta t = 0.1$ as the most appropriate as we thought it was the best mix of efficiency and precision.

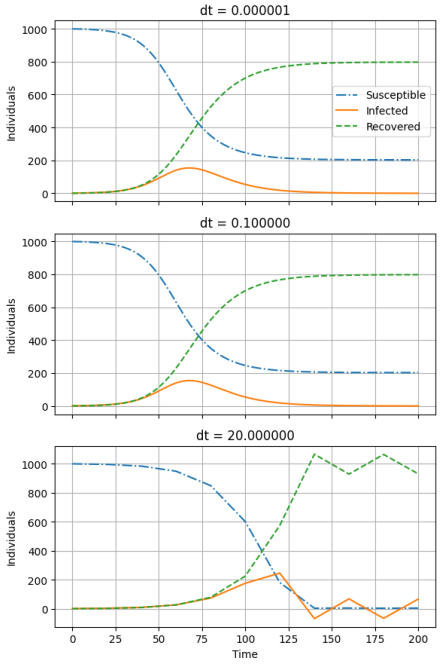


Figure 2: Results from model with different dt values

# 5 Timestep convergence

We see a exponential correlation between timestep size and the resulting error ratio. When plotting the error as a function of t we see that the error is far larger in the interval where

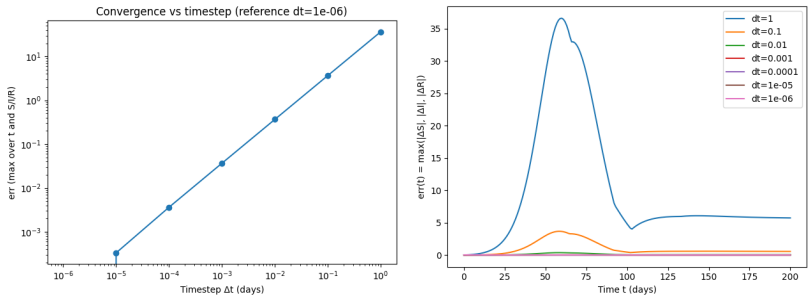


Figure 3: Left: Convergence vs dt. Right: Error vs t for each dt value

the gradient of the parameters is the largest. Perhaps dynamic time steps would be a good idea, so dt varies based on the magnitude of the change in parameter space.

# 6 Python integration

Instead of running the C++ code and reading CSV files in Python, we used `pybind11` to expose the SIR integrator as a Python module. After compiling with `make`, we can call the C++ solver directly in a notebook using `import SIR_python` and `SIR_python.integrate_system(...)`. This keeps the time-stepping in fast C++, while Python is used for fitting and plotting.

We fitted $\beta$ and $\gamma$ to `Unknown_SIR_data.csv` (with $S_0 = 1000$, $I_0 = 1$, $R_0 = 0$) by minimizing a least-squares mismatch between the data and the model. The best fit was $\beta = 0.2495$ and $\gamma = 0.0771$, and the resulting fit is shown in Fig. 4.

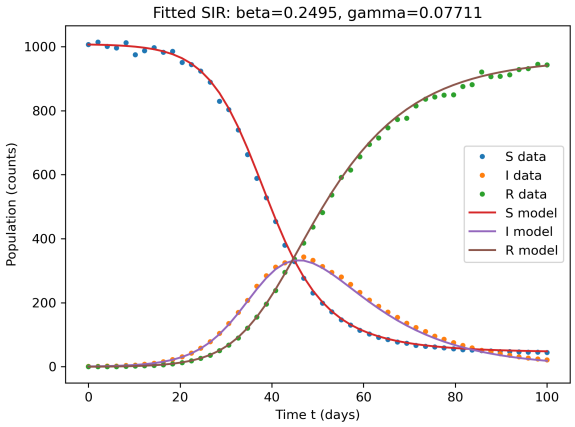


Figure 4: SIR fit to `Unknown_SIR_data.csv`. Points are the noisy data and lines are the fitted model evaluated using the C++ solver called from Python.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <sstream>
#include <iomanip>
#include <string>
#include <filesystem>

// state: vector of S, I, R
std::vector<double> take_step(std::vector<double> state, double beta, double gamma, 
    // Unpack current state
    double S = state[0];
    double I = state[1];
    double R = state[2];

    // Total population
    double N = S + I + R;

    // Time derivatives
    double dSdt = -beta * S * I / N;
    double dIdt =  beta * S * I / N - gamma * I;
    double dRdt =  gamma * I;

    // Forward Euler update
    double S_new = S + dSdt * dt;
    double I_new = I + dIdt * dt;
    double R_new = R + dRdt * dt;

    // Return updated state vector
    return {S_new, I_new, R_new};
}

int main(int argc, char* argv[]) {
    // Model parameters
    double beta = 0.2;
    double gamma = pow(10, -1);    // 0.1
    int N = 1000;

    int population = N;

    // Initial condition
    double infected = 1;
    double susceptible = population - infected;
    double immune = 0.0;

    // state is S, I, R
    std::vector<double> state = {susceptible, infected, immune};
```

```
// Timestep size
double dt = 0.1;
if (argc >= 2) {
    dt = std::stod(argv[1]);
}

// Fixed simulation end time
double t_end = 200.0;

// write output every 1 day
double write_dt = 1.0;

// How many Euler steps correspond to write_dt?
double ratio = write_dt / dt;
long long write_every = llround(ratio);

// Because dt is floating point
double tol = 1e-9 * std::max(1.0, std::fabs(ratio));
if (std::fabs(ratio - static_cast<double>(write_every)) > tol) {
    std::cerr << "ERROR: write_dt/dt is not an integer (within tolerance).\n";
    std::cerr << "write_dt =" << write_dt << ", dt =" << dt << ", ratio =" <<
    return 1;
}

// Create filename based on dt
std::ostringstream base;
base << "filename_dt" << std::fixed << std::setprecision(6) << dt;
std::string base_name = base.str();
for (char& c : base_name) if (c == '.') c = 'p';

// Put outputs in a dedicated "data" folder
std::filesystem::create_directories("data");
std::string filename = "data/" + base_name + ".csv";

// Open output file and write header
std::ofstream outFile(filename);
outFile << "Time,Infected,Susceptibile,Recovered\n";

//Start step
long long step = 0;

// Main time integration loop
while (true) {
     // Current time from step number
    double current_time = step * dt;

    // Stop condition
```

```
        if (current_time > t_end + 1e-12) break;

        // Write output only at selected steps, so output times match across dt
        if (step % write_every == 0) {
            outFile << std::fixed << std::setprecision(12) << current_time << ","
                    << state[1] << ","   // Infected
                    << state[0] << ","   // Susceptible
                    << state[2] << "\n"; // Recovered
        }

        // Take one Euler step forward in time
        state = take_step(state, beta, gamma, dt);
        step += 1;
    }

    outFile.close();
    return 0;
}
```

```cpp
#include <vector>
#include <cmath>
#include <stdexcept>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

// Function to take a step in the SIR model
// state: vector of S, I, R
// beta: infection rate
// gamma: recovery rate
// dt: time step
std::vector<double> take_step(const std::vector<double>& state, double beta, double g
    // Unpack the state
    double S = state[0];
    double I = state[1];
    double R = state[2];

    // Total population
    double N = S + I + R;

    // SIR derivatives
    double dSdt = -beta * S * I / N;
    double dIdt =  beta * S * I / N - gamma * I;
    double dRdt =  gamma * I;

    // Forward Euler update
    double S_new = S + dSdt * dt;
    double I_new = I + dIdt * dt;
    double R_new = R + dRdt * dt;

    // Return updated state vector
    return {S_new, I_new, R_new};
}

// Function simulating num_steps of the SIR model, saving the state every return_ever
// Returns an array with columns: [t, S, I, R]
pybind11::array_t<double> integrate_system(double S0, double I0, double R0,
                                           double beta, double gamma, double dt,
                                           int num_steps, int return_every) {
    // Basic argument validation
    if (num_steps < 0) {
        throw pybind11::value_error("num_steps must be >= 0");
    }
    if (return_every <= 0) {
        throw pybind11::value_error("return_every must be >= 1");
    }
    if (dt <= 0.0) {
        throw pybind11::value_error("dt must be > 0");
```

```
    }

    // Initial state vector
    std::vector<double> state = {S0, I0, R0};

    // Number of records we will store:
    int num_records = (num_steps / return_every) + 1;

    // Allocate output array: rows = num_records, cols = 4 (t, S, I, R)
    pybind11::array_t<double> out({num_records, 4});
    auto out_m = out.mutable_unchecked<2>();

    int rec = 0;

    // Store initial condition at time t=0
    out_m(rec, 0) = 0.0;
    out_m(rec, 1) = state[0];
    out_m(rec, 2) = state[1];
    out_m(rec, 3) = state[2];
    rec += 1;

    // Time integration loop
    for (int step = 1; step <= num_steps; step++) {
        // Advance one Euler step
        state = take_step(state, beta, gamma, dt);

        // Save the state only at the chosen output cadence
        if (step % return_every == 0) {
            double t = step * dt;
            out_m(rec, 0) = t;
            out_m(rec, 1) = state[0];
            out_m(rec, 2) = state[1];
            out_m(rec, 3) = state[2];
            rec += 1;
        }
    }

    return out;
}

PYBIND11_MODULE(SIR_python, m) {
    m.doc() = "Python bindings for a simple SIR model (Forward Euler)";

    // Expose the integrate_system function to Python
    m.def("integrate_system", &integrate_system,
          pybind11::arg("S0"), pybind11::arg("I0"), pybind11::arg("R0"),
          pybind11::arg("beta"), pybind11::arg("gamma"), pybind11::arg("dt"),
          pybind11::arg("num_steps"), pybind11::arg("return_every"));
```

}