Assignment 1: Epidemic rate equations

Practical information

Deadline: Saturday 8/2 23.59

Resources:

- ERDA for file storage
- Jupyter for the Terminal to access DAG
- DAG for testing, visualization, and benchmarks

Hand-in:

- A report of up to 3 pages in length (excluding the code)
- Your code both as source file as well as in pdf
- Use the template on Absalon to include your code in the report

Introduction

A simple mathematical description of the spread of a disease in a population is the so-called SIR model, which divides the (fixed) population of *N* individuals into three ”compartments” that may vary as a function of time, *t*:

- *S(t)* are those susceptible but not yet infected with the disease.
- *I(t)* are the number of infectious individuals. These are infected and can infect others.
- *R(t)* are those individuals who have recovered from the disease and now have immunity to it.

The SIR model describes the change in the population of each of these compartments in terms of two parameters, β and γ. β describes how infectious the disease is: on average, in absence of immunity, every infected person infects β others. This rate is reduced by the fraction of the population that is susceptible (i.e. are not immune yet), S/N. γ is the mean recovery rate: That is, 1/γ is the mean period of time during which an infected individual can pass it on.

The differential equations describing this model were first derived by Kermack and McKendrick [Proc. R. Soc. A, 115, 772 (1927)]. The inspiration from this example come from Chris Hill, Learning Scientific Programming with Python. Here are the equations:

$$\frac{dS}{dt} = -\beta \frac{S}{N}$$

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

$$\frac{dR}{dt} = \gamma I$$

Task 1: Implement and validate the SIR model (4 points)

Write a C++ code which integrates these equations based on SIR.cpp. Every place that has to be modified has been marked with “Todo”. Show the correctness by checking that your code does the right thing in at least 4 different limits. This can for example be the evolution after 1 step in different limits of β, γ, β/γ, initial conditions, whether it converges, timescale of convergence or whatever you find appropriate. Argue that the result of your code is correct.

Task 2: Use and plot the SIR model (2 points)

Simulate an epidemic with parameters $\beta = 0.2, \gamma = (10\ days)^{-1}$ spreading in a population of 1000 people. For initial parameters of *S*, *I* and *R*, think about how a population starts. Save the data into a file, and load and plot it in python, showing *S*, *I* and *R* on the same plot as a function of time t. How did you choose the timestep Δt? Plot both a simulation with a high Δt and one with so low Δt that the plot has steps.

Task 3: Timestep convergence (4 points)

We want to explore how different step sizes affect the accuracy of the numerical solution, and what is sufficient. For this, we want to calculate the maximum deviation between a simulation with timestep Δt and another with twice the resolution Δt/2. In python pseudocode, this can be written roughly as

```
diff = np.max(y0[::2] – y1)

err = diff.max(diff)
```

where y0 is the data with double the resolution of y1, both being 2d arrays with rows of the timesteps, and columns of the 3 quantities we are integrating (S, I and R)

With the same parameters as before, calculate this maximal error and plot it as a function of the timestep size Δt for at least 5 timestep sizes in the range 1 day to 1e-6 days. You can do this manually but automating it using a loop in the C++ code is much encouraged. If you make Δt small (which will show something interesting) writing out the answer is prohibitively slow. In this case, just write out t, S, I and R periodically (e.g. every *k* steps or after a fixed amount of days).

Also make a plot of diff (as defined above) as a function of time for both a large and a small Δt. Comment on what you see.

Task 4 (Optional!!!): Integrate C++ with Python. (up to 2 extra points)

This is an exercise for the interested, showing how to integrate C++ with python. This can be very useful to squeeze out performance from Python in some cases e.g. for Monte Carlo sampling of expensive functions, or when the algorithm has to be written in terms of for-loops. Only do this if you find it interesting (the time/point cost benefit is bad :)) and the total number of points is still capped at 10.

Sometimes, C++ (and other low-level languages such as Fortran, C, Rust) is used because for large projects, they are better suited than e.g. python. Other times, we use it to speed up the 1% of our code which takes 99% of the time, meaning that we can continue to use python for convenience. This could for instance be the log-likelihood in a Monte Carlo sampler or the integrand in a SciPy quad integral. This is how NumPy manages to be often orders of magnitudes faster than the equivalent python code. In this case we need to “bind” the two languages together. This exercise demonstrates this.

Rather than running the C++ code, saving a file and then loading and plotting in python, you are now tasked with calling the C++ code as a function inside python. For this, use the cpp file

SIR_python.cpp that uses pybind11++ library, which can be installed with `pip install pybind11`. You might also need to `apt-get python3-dev` if you are on Linux.

After implementing the SIR model, you should be able to compile it by uncommenting all code in the Makefile and then calling make. Afterwards, you should be able to import and run the C++ code from e.g. a jupyter notebook like:

```
import SIR_python

SIR_python.integrate_system(S,I,R,beta, gama, dt, n_times, return_every)
```

Using this, fit the (slightly noisy) data in Unknown_SIR_data.csv. This data has S0=1000, I0=1 and R0=0, but determining beta and gamma is up to you! **Show your fitted values and your fit on the data**.

Note that you must restart the kernel to reload a new version after it has been compiled.