UiO : Department of Informatics
University of Oslo

Modeling the spread of a pandemic

Final project in IN1900, fall 2023

November 9, 2023

About this project

Here are some useful hints on how to solve this project. Read this carefully before you start working:

1. Solve the problems in the order they are presented. One problem builds on the previous one, so they have to be solved in the correct order.

2. Use the version of the ODESolver class found here: `https://sundnes.github.io/solving_odes_in_python/`. There are multiple versions of the ODESolver class found on previous years' IN1900 pages and in the source code for Langtangen's book. Since these versions are almost identical but behave slightly differently, you should avoid confusion by using the version specified here.

3. Since the problems build on each other, the project becomes easier if you know that each component works exactly as it should. It is useful to spend some time making sure that everything works before moving on to the next phase of the project.

4. If you want feedback on your project, the deadline for handing it in is November 17 at 16.59. The program files should be uploaded to devilry as usual, and you should include an example of how you ran each file ("kjøreeksempel") in the usual way. It is not necessary to include plots. If any of your programs do not work properly, and you are not able to solve the problem, you should still include a "kjøreeksempel" that includes the error message you got and/or some comments about what went wrong.

5. As always, collaboration is encouraged. Since this is not a mandatory project, it is perfectly ok to work together on the same files. However, if two people submit identical files, please include a comment with this information, to avoid double work in reviewing the project and providing feedback.

6. Since the project is not mandatory, but very relevant for the exam, you choose yourself how many problems you want to solve and submit. Problem 1 is intended as a simple "warm-up" exercise, and everyone is strongly encouraged to solve at least problems 1, 2 and 3 to be prepared for the exam.

Modeling a pandemic with ODEs and Python

Problem 1. The SEEIIR model
In this exercise we will implement an ODE-based version of the SEEIIR model used by the Norwegian Institute of Public Health to describe the spread of the Covid19 pandemic. The model is described in Chapter 5 of the lecture notes Solving Ordinary Differential Equations in Python[1]. The model has six categories, S, E1, E2, I, Ia, and R, and is referred to as a SEEIIR model in the lecture notes.

a) Copy the entire class `SEEIIR` from page 92-93 of the lecture notes, or directly from the source code provided with the notes[2]. Implement a test function `test_SEEIIR()` to verify that the call function of the class works correctly. Inside the test function, you can create an instance of the class using the following parameter values:
`beta=0.4; r_ia =0.1; r_e2=1.25`
`lmbda_1=0.33; lmbda_2=0.5; p_a=0.4; mu=0.2`.
The call the instance with arguments `t=0` and `u=[1,1,1,1,1,1]`, and verify that the output is a list with these values:
`[-0.156666666666, -0.1733333333333, -0.302, 0.3, -0.068, 0.4]`.
Remember to compare the values with a tolerance (for instance `tol=1e-10`) since the outputs are floats.

b) Make a function `solve_SEEIIR(T,dt,S_0,E2_0)` for solving the system of differential equations. Choose a solver from the `ODESolver` class hierarchy. The equations should be solved from time 0 to `T`, where `T` and the time step `dt` are given as arguments to the function. The other arguments (`S_0,E2_0`) are initial conditions for the S and $E2$ categories. All other initial conditions are set to zero, so the complete initial condition for the ODE system should be `[S_0, 0, E2_0, 0, 0, 0]`. The function should return arrays `t,u` containing the solution and the time.

c) Make a function `plot_SEEIIR(t,u)` for visualizing the components $S(t), I(t)$, $Ia(t)$, and $R(t)$ in the same plot. These are often the most interesting variables in epidemiology. Include a legend with labels for each curve. Finally, call the

[1] https://sundnes.github.io/solving_odes_in_python/
[2] https://github.com/sundnes/solving_odes_in_python/blob/master/docs/src/chapter5/SEEIIR.py

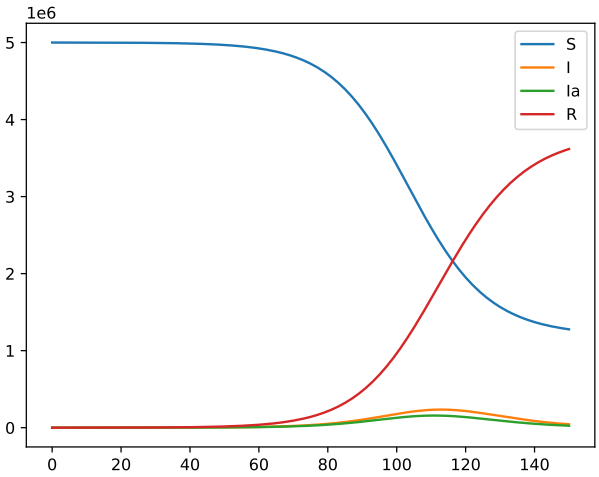

Figure 1: Solution of the SEEIIR model. The plot shows the dynamics of the categories S, I, Ia, R.

functions from a)-b) and the new plot function to solve the SEEIIR model for initial values `S_0=5e6, E2_0=100`, all other initial values zero, `T=150` and `dt=1.0` (the time is given in days). The resulting plot should be similar to the one in Figure 1.

Filename: `SEEIIR0.py`

Problem 2. Introduce classes in the SEIR model

In this exercise we will extend the class from Problem 1 to support more advanced functionality such as time dependent model parameters. We will also introduce a new solver class. To simplify the notation and save a bit of typing we will from hereon refer to it as a SEIR model even though we still consider two distinct E- and two distinct I-categories. We will create a module called `SEIR.py`, which includes two classes:

- The class `ProblemSEIR` which defines the ODE model.
- A solver class `SolverSEIR` to solve the SEIR system of ODEs.

a) Write the class `ProblemSEIR`, which has four methods; `__init__`, `set_initial_condition`, `get_population`, and `__call__`.

The constructor should take as arguments all the model parameters `beta`, `r_ia`, `r_e2`, `....` The parameter `beta` in the SEIR model can be constant or function of time. The implementation of `ProblemSEIR` should be such that beta can be given either as a constant or as a Python function. The constructor can look something like this:

```
def __init__(self, beta, r_ia = 0.1, r_e2=1.25, \
             lmbda_1=0.33, lmbda_2=0.5, p_a=0.4, mu=0.2):

    if isinstance(beta, (float, int)): # is it a number?
        self.beta = lambda t: beta    # wrap as function
    elif callable(beta):
        self.beta = beta
    ...
```

The method `set_initial_condition(self, S_0,E2_0)` shall create and store a list `self.initial_condition` containing the initial values of $S, E1, E2, I, Ia$, and R (in this particular order). The initial values for S and $E2$ are passed as arguments to the method. All other initial conditions can be set to zero.

The method `get_population(self)` should simply return the value of the population of the region, which is equal to the sum of all the initial conditions (i.e., $S_0 + E2_0$ since the others are zero).

Finally, write a special method `__call__(self, t, u)` which returns the right hand side of the ODE system defining the SEIR model, just as the class in Problem 1. Remember that the attribute `self.beta` is now a function of time, and it needs to be treated as such inside the `__call__` method.

b) Write a test function `test_ProblemSEIR()` to verify that the class works correctly. Inside the test function you create an instance of the class and call the `set_initial_condition` method with suitable arguments. Then include three assert statements to verify that the class behaves as expected:

1. Assert that the attribute `initial_condition` has the expected value
2. Assert that the method `get_population` returns the expected result.
3. Use the same test as for the `SEIR` function in Problem 1 to assert that that the call method works.

All these tests can be put inside the same test function.

c) Now we will create a class `SolverSEIR` with three methods; a constructor, a method `solve(self, method)`, and a method `plot(self,states)` for plotting the solution. The constructor should take the arguments `problem` (an instance of class `ProblemSEIR`), `T` (the final time) and `dt`, and store them as attributes.

The `solve(self, method)` shall solve the SEIR system of ODEs by a method of your choice from the `ODESolver`. Use the following sketch for this method:

```
def solve(self, method=RungeKutta4):
    solver = method(self.problem)
    solver.set_initial_condition(...)
    """
    Insert code here to calculate
    the number of time steps from T and dt
    """
    t = np.linspace(...)
    t, u = solver.solve(t)
```

Finally, the `plot(self,states)` shall take a list of strings as input, which specifies the specific variables we want to plot. For instance, the call `solver.plot(['I','Ia','R'])` should plot the variables I, Ia, and R.

Add the following code at the bottom of the file:

```
if __name__ == "__main__":
  test_ProblemSEIR()

  S_0 = 5e6
  E2_0 = 100
  problem = ProblemSEIR(beta=0.4)
  problem.set_initial_condition(S_0,E2_0)
  solver = SolverSEIR(problem,T=150,dt=1.0)
  solver.solve()
  solver.plot(['S','I','Ia','R'])
```

The test function should of course pass with no output, and remaining lines should result in a plot that looks exactly like the one you got in Problem 1

Filename: `SEIR.py`

Problem 3. Simulate a pandemic outbreak
In this exercise we will use the classes `ProblemSEIR` and `SolverSEIR` from Problem 2 to simulate possible scenarios of the Covid19 pandemic in Norway. The code should be written in a separate file `outbreak.py`, which should import from `SEIR.py`.

a) Create a function for simulating the Covid19 outbreak in Norway. The function should look like this

```
def outbreak_Norway(beta, num_days, dt):
    ...
```

and should call the methods of the `ProblemSEIR` and `SolverSEIR` classes to construct and solve the problem, and then plot the two variables I and Ia. Use the same initial conditions as in the previous problems. Call the function using `beta=0.33, num_days =150, dt =1.0`. Find the peak of the I category, either by a visual inspection of the plot or by a suitable call to the builtin `max` function. Estimates from the early phase of the pandemic indicated that about 20% of the infected cases would need hospital care, and 5% would need a mechanical ventilator. There are around 700 ventilators in Norwegian hospitals. How does this number compare to your estimate?

b) Until now we have assumed that β is constant. The β parameter describes the probability that a contagious person (in $E2, I, Ia$) meets and infects a susceptible person. In reality, β depends on numerous factors, including the infectiousness of the disease itself, immunity of the population resulting from vaccination and previous infections, as well as the general behaviour of the population. We will now extend our model to use a piecewise constant β.

Epidemiologists often refer to the reproduction number R of an epidemic, which is the average number of new persons that an infected person infects. The critical number is $R = 1$, since if $R < 1$ the epidemic will decline, while for $R > 1$ it will grow exponentially. In the simplest models, the relationship between R and β is $R = \beta\tau$, where τ is the mean duration of the infectious period. In our model, which has multiple infectious categories, we have

$$R = r_{e2}\beta/\lambda_2 + r_{ia}\beta/\mu + \beta/\mu,$$

since the mean durations of the $E2$ period is $1/\lambda_2$ and the mean duration of both I and Ia is $1/\mu$. The choice of $\beta = 0.33$ gives $R \approx 2.62$, which is the value used by the Institute of Public Health (FHI) to model the early stage of the outbreak in Norway, from mid February to mid March 2020. As we all know, severe restrictions on travel and social interactions were introduced in Norway on March 12, 2020. These restrictions substantially reduced the number of contacts between infected and susceptible persons, which is represented in the model as a reduction in the parameter β.

Write a function `beta(t)` which represents the piecewise constant β:

$$\beta(t) = \begin{cases} 0.33 & \text{for } t < 30 \\ 0.083 & \text{for } t > 30 \end{cases}$$

Call the function `outbreak_Norway` from a) with the new piecewise constant β. What happens? How does the plot compare to the one you got in a)?

Filename: `outbreak.py`

Problem 4. Managing a pandemic

As we are all too aware, the Covid19 pandemic has lasted quite a while and has come in multiple waves. Numerous mutations of the virus affected its infectiousness, and frequent changes in the restrictions on travel and social interactions also impacted the spread of the disease. In the models run by FHI to understand the pandemic and predict incoming waves, these changes were reflected as piecewise constant values of the β parameter, similar to the model implemented in the previous problem, but with many more steps. The file `beta_values.txt` includes a list of suitable β values.[3] The values are based on the parameter used by FHI in their real-world pandemic models, but slightly adapted to fit better in our model.

Implement a piecewise constant β as a class `Beta`. The class should have (at least) three methods:

- A constructor `__init__(self,filename)`, which takes a file on the format of the `beta_values.txt` as argument, do the necessary processing of the file, and store the β values and time intervals in suitable data structures. This can be done in multiple ways, but one option is to store two lists of equal lenghts; one containing the β values and one containing the start date for the interval when each β value applies. (With this approach you can ignore the end dates for each interval). We can assume that $t = 0$ in our model corresponds to the date 15.02.2020, so the list of "dates" can be integers counting from that date. These lists can then be used as lookup tables inside the `__call__` method to evaluate β.

  For converting from the dates in the file to an integer counting from the first date, the `datetime` module is useful. The most relevant parts of the `datetime` module are two classes called `date` and `timedelta`, so a suitable import statement can be

  ```
  from datetime import date, timedelta
  ```

  It may also be useful to implement a method to convert from the date format used in the file to a `date` object. Such a method may, for instance, look like this:

  ```
  str2date(self, date_str):
      #convert a date string on the format dd.mm.yyyy
      #to a datetime object
      day, month, year = (int(n) for n in date_str.split('.'))
      return date(year,month,day)
  ```

  If you have two dates represented as `date` objects, they can be substracted to yield a `timedelta` object, and the number of days can be accessed as an attribute of the `timedelta` class named `days`. Here's a short example:

  ```
  #date0 and date1 are instances of class datetime.date
  delta = date1-date0
  n_days = delta.days
  ```

[3] The file can be downloaded from *https://www.uio.no/studier/emner/matnat/ifi/IN1900/h22/ressurser/live_programmering/beta_values.txt*.

Since the processing of the input file will require quite a few lines, the constructor can end up a bit long and messy. It may be useful to structure the code a bit by defining one or more methods to perform the file reading and processing, and then calling the method(s) from the constructor.

- A `__call__(self,t)` method, which takes the time as input, and return the β value for the give day. If you created two lists in the constructor, as suggested above, a suitable approach will be to loop through these lists and add an appropriate if test to return the correct value of β. There are many other ways this can be done, and some are probably more elegant, but this is a simple approach that works.

- A method named `plot(self,T)` which takes a time point `T` as argument, and plots the β function for t between 0 and `T`.
  Hint 1: In order for the plot to capture the jumps and actually look like a discontinuous function, you need to use quite a few t values. For instance, using `t =np.linspace(0,T,1000)` should make the plot look fairly good.
  Hint 2: If your `__call__` method contains one or more if tests, the usual approach of sending the entire `t` array to the function will not work. A for loop or a call to `np.vectorize` may then be needed to compute the array of β values.

If the class defined is implemented correctly, the following test block should work. Add it to the bottom of your file.

```
if __name__ == "__main__":
    from outbreak import outbreak_Norway
    beta = Beta('beta_values.txt')
    beta.plot(1000)
    outbreak_Norway(beta,1000,1)
```

The β plot, which shows up first, should look similar to the plot in Figure 2. It is not important that the plot looks exactly the same, but it should be similar. What does the final plot of the solution look like? (The plot may look less interesting than you expect, and does not properly capture the multiple waves of infections observed in Norway.)

Filename: `lockdown.py`

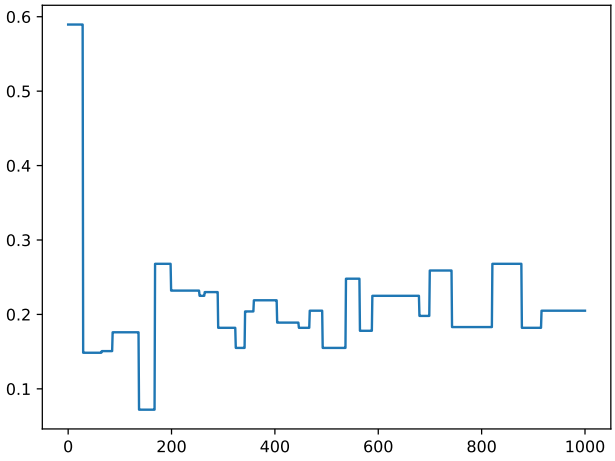

Figure 2: The piecewise constant β parameter read from file.

Problem 5. Extending the model

The plots you got in Problem 4 probably did not look too interesting, and did not really capture the dynamics of the Covid19 pandemic in Norway. There may be several reasons for why the model fails, but one possible explanation is that there is no import of infected people. After the first wave has been crushed by the lockdown, the number of infected persons is too low to start off new waves, even if the reproduction number $R > 1$ in some time intervals. The goal for this last part of the project is to modify the class from Problem 2 to account for import of infected people. There are several ways to include this in the model, and we shall adopt the simple approach of adding a source term to the equation for the $E2$ category. The original equation reads

$$\frac{dE_2}{dt} = \lambda_1(1 - p_a)E_1 - \lambda_2 E_2,$$

and we may simply modify this to

$$\frac{dE_2}{dt} = \lambda_1(1 - p_a)E_1 - \lambda_2 E_2 + \Sigma,$$

to model an influx of Σ people in the $E2$ category per day. While this simple model is not the most realistic, it is suitable for illustrating the impact of infection imports.

Implement a sub-class `SEIRimport(ProblemSEIR)` of the class you created in Problem 2. Reuse as much code as possible from the base class. The new class needs one additional parameter (`sigma`), so the constructor needs to be modified. The `__call__` method also needs to be modified. You can either copy the `__call__` method code directly into the new class and modify it there, or call the method from the base class and then modify the resulting list.

Copy the function `outbreak_Norway` from Problem 3, call it for instance `outbreak_Norway`, and modify it to use the class `SEIRimport` instead of

`ProblemSEIR`. Repeat the last steps from Problem 4 using the modified model with $\Sigma = 10$. Does the plot change from Problem 4?

Filename: `covid19.py`