UiO : Department of Informatics
University of Oslo

Modeling Covid19 scenarios in Norway

Final project IN1900, fall 2020

November 17, 2020

Chapter 1

Modeling the spread of a pandemic

About this project

Here are some useful hints on how to solve this project:

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. Through the course of the project you will implement a number of separate model components, and then in the end combine these into a fairly complex model. The final part becomes much easier if each individual component is working correctly. It is therefore important to follow the instructions for each class and method very carefully, and to test that each individual component behaves as expected before moving on to the next step.

4. The total number of points available is 24. The number of points for each exercise is provided in the headline.

5. The deadline for handing in the project is November 22 at 23.59. The program files should be uploaded to devilry as usual. Include an example of how you ran each file ("kjøreeksempel") in the usual way, but 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.

Problem 1.1. The SEIR model as a function (2 points)
In this exercise we will implement an ODE-based version of the SEIR model used by the Norwegian Institute of Public Health to describe the spread of the Covid19 pandemic. The model is described in Chapter 4 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. However, to simplify the notation and save a bit of typing we will here refer to it as a SEIR model even though there are two distinct E- and two distinct I-categories.

a) Copy the entire function `SEEIIR_model(u,t)` from page 47 of the lecture notes, or directly from the source code provided with the notes[2] Rename the function to `SEIR(u,t)`, and keep the rest of the function unchanged, with all model paraemters defined as local variables inside the function. Use the following values for the variables:
`beta =0.5; r_ia =0.1; r_e2=1.25; lmbda_1=0.33; lmbda_2=0.5; p_a=0.4; mu=0.2`.
Implement a test function `test_SEIR()` to verify that the function works correctly. Inside the test function, you should call the `SEIR(u,t)` function with arguments $t = 0$ and $u = [1, 1, 1, 1, 1, 1]$, and verify that the output is a list with these values:
`[-0.19583333333333333, -0.13416666666666668, -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_SEIR(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`, with time step `dt` and initial conditions `[S_0, 0, E2_0, 0, 0, 0]`. The function should return arrays `u,t` containining the time and the solution.

c) Make a function `plot_SEIR(u,t)` for visualising the components $S(t)$, $I(t)$, $Ia(t)$, $R(t)$ in the same plot. These are usually the most interesting variables in epidemiology. Include a legend with labels for each curve.

d) Use the functions from a)-c) to solve the SEIR model for initial values `S_0=5e6, E2_0=100`, all other initial values zero, `T=100` and `dt=1.0` (the time is given in days). The resulting plot should be similar to the one in Figure 1.1.

Filename: `seir_func.py`

[1] https://sundnes.github.io/solving_odes_in_python/
[2] https://github.com/sundnes/solving_odes_in_python/blob/master/docs/src/chapter4/SEEIIR_fun.py

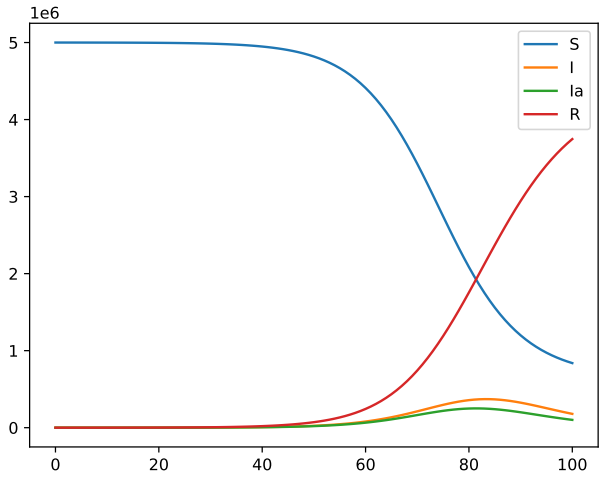

Figure 1.1: Solution of the SEIR model. The plot shows the dynamics of the categories S, I, Ia, and R.

Problem 1.2. Introduce classes in the SEIR model (8 points)
In this exercise we will implement the SEIR model from exercise 1.1 as a class, and extend the model with functionality to describe disease spread in a particular geographical region. The classes will be implemented in a module called `SEIR.py`. We will create three classes:

- `Region`, which can represent a geographical region with specific initial conditions for the categories $S, E1, \ldots, R$.
- The class `ProblemSEIR` which defines the ODE model for a given region.
- A solver class `SolverSEIR` to solve the SEIR system of ODEs for a given region.

Since the classes will later be put together in a more complex model with multiple regions, it is important that each class is implemented **exactly** as specified below.

a) Create a class `Region` which has three methods; a constructor, a method `set_SEIR_values(self, u, t)`, and a method `plot(self)` for plotting the SEIR values.

The signature of the constructor should look like

```
def __init__(self,name,S_0,E2_0):
...
```

The argument `name` is a text string specifying the name of the region, and the others are the initial conditions for the categories S and $E2$. All other initial conditions should be set to zero. The constructor shall store the region name and all six initial conditions as attributes. You should also add an attribute

`self.population` which is the total population of the region at time t_0 (i.e., the sum of all the initial conditions).

The method `set_SEIR_values(self, u, t)` should take out the SEIR values from the argument `u` and store $S, E1, E2, I, Ia, R$ and t as attributes of the class.

The method `plot(self)` should plot S, I, Ia, and R in the same plot. Label the axes with for instance `plt.xlabel('Time(days')` and `plt.ylabel('Population')` and set the title of the plot to the name of the region. Specify a label for all the different categories (an example could be `plt.plot(self.t, self.S, label='Susceptible')`). Do not include calls to `plt.legend()` or `plt.show()` inside the function. We will later use the method to plot several subplots, and these methods must therefore be called at the end. Put the following code as a main block in the bottom of the file: [3]

```
if __name__ == '__main__':
    nor = Region('Norway',S_0=5e6,E2_0=100)

    print(nor.name, nor.population)
    S_0, E1_0, E2_0 = nor.S_0, nor.E1_0, nor.E2_0
    I_0, Ia_0, R_0 = nor.I_0, nor.Ia_0, nor.R_0
    print(f'S_0 = {S_0}, E1_0 = {E1_0}, E2_0 = {E2_0}')
    print(f'I_0 = {I_0}, Ia_0 = {Ia_0}, R_0 = {R_0}')
    u = np.zeros((2,6))
    u[0,:] = [S_0, E1_0, E2_0, I_0, Ia_0, R_0]
    nor.set_SEIR_values(u,0)
    print(nor.S, nor.E1, nor.E2, nor.I, nor.Ia, nor.R)
```

This code creates a `Region` instance with the same initial conditions as in Problem 1.1 and prints the attributes of the class. Make sure that this code works for your class and gives the expected output.

b) Write the class `ProblemSEIR`, which has five methods; `__init__`, `set_initial_condition`, `get_population`, `solution`, and `__call__`.

The constructor should take in all the model parameters `beta, r_ia, r_e2, ...` and a region, which must be an instance of the class `Region`. 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 should look like this:

```
def __init__(self, region, 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)): # number?
        self.beta = lambda t: beta    # wrap as function
    elif callable(beta):
        self.beta = beta
    """
    Put code here for storing the region and the other
    parameters as attributes.
```

[3] Or use the template file found here:
https://www.uio.no/studier/emner/matnat/ifi/IN1900/h20/ressurser/live_programmering/template_seir.py

```
    """
    self.set_initial_condition()        # method call
```

The method `set_initial_condition(self)` shall store a list `self.initial_condition` containing the initial values of $S(0)$, $E1(0)$, $E2(0)$, $I(0)$, $Ia(0)$, and $R(0)$ (in this particular order). The initial values should be extracted from the class attribute `region`.

The method `get_population(self)` should simply return the value of the population of the region, which is stored in the class attribute `region`.

The method `solution(self, u, t)` calls the method `set_SEIR_values(u, t)` of the class attribute `region`. (The purpose of this method is to take a solution array `u` and store the individual solution components as attributes in the attribute `region`.)

Finally, write a special method `__call__(self, u, t)` which returns the right hand side of the ODE system defining the SEIR model, just as the function in Problem 1.1. Remember that the attribute `self.beta` is now a function of time, and it needs to be treated as such inside `__call__` method.

Extend the main block from above with the following code lines:

```
    problem = ProblemSEIR(nor,beta=0.5)
    problem.set_initial_condition()
    print(problem.initial_condition)
    print(problem.get_population())
    print(problem([1,1,1,1,1,1],0))
```

Make sure that all of these lines work and give the expected output. The output from the last line should be the same as the output in the `test_SEIR` function in Problem 1.1.

c) Now we will create a class `SolverSEIR` with two methods; a constructor and a method named `solve`. The constructor should take the parameters `problem` (which must be an instance of class `ProblemSEIR`), `T` (final time) and `dt`, and store them as attributes. The constructor should also store an attribute called `total_population`, which is obtained by calling the `get_population` method of `problem`.

Write a method `solve(self, method)` that solves 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(...)
    #calculate the number of time steps from T and dt
    t = np.linspace(...)
    u, t = solver.solve(t)
    # Send the values of S, E1, E2, I, Ia, R, and t
    # from the Problem class to the Region class:
    self.problem.solution(u, t)
```

Add the following code to the main block at the bottom of the file:

```
solver = SolverSEIR(problem,T=100,dt=1.0)
solver.solve()
nor.plot()
plt.legend()
plt.show()
```

The resulting plot should look like the one you got in Problem 1.1.

Filename: `SEIR.py`

Problem 1.3. The SEIR model across regions (8 points)
The problem class from exercise 1.2 can only model the spread of a disease within one region. In this exercise we will extend our program with subclasses of `ProblemSEIR` and `Region` that permits people in one region to get infected by people from another region. The likelihood of transmission of disease between regions will depend on the distance between the regions.

We introduce subscripts on the categories to specify which region they belong to, such that $S_i(t), E1_i(t), E2_i(t), Ia_i(t), I_i(t)$, and $R_i(t)$ are the number of people in each category in the i-th region at time t. If we examine the model equations of the SEIR model it is natural that the expressions for $E2_i(t), I_a(t), I_i(t), R_i(t)$ will be unchanged from the SEIR model used above, since the transitions in and out of these categories are independent of interactions with the other categories and therefore do not involve interactions with other regions. The transition from S_i to $E1_i$ is different, since it involves interactions of people in the S_i category with people in the infected categories $E2_i, Ia_i$, and I_i. We want to extend the model from above to also take into account interactions with infected people in other regions. We make two assumptions:

1. People in category S_i will interact with and potentially get infected by people in $E2_j, Ia_j$, for $j \neq i$, but not by $I_j, j \neq i$. This assumption is based on the fact that people in the I category are sick (i.e., they have symptoms) and are likely to be isolated at home and not interact with people from other regions. (People in the S_i group can still be infected by people in I_i, i.e., by sick people in the same region.)

2. The level of interaction between people in two regions is a function of the geographical distance between the regions, with longer distance meaning less interaction. This assumption was probably quite accurate before the 20th century, when travel was generally slow, but is not very accurate today. However, it is a reasonable simplification that can easily be replaced by a more realistic model later.

Based on these assumptions, we can derive the following model for disease spread between regions. We have

$$\begin{aligned}\frac{dS_i}{dt} =& -\beta\frac{S_i I_i}{N_i} - r_{ia}\beta S_i \sum_{j=1}^{M} \frac{Ia_j}{N_j} e^{-d_{ij}} \\ & - r_{e2}\beta S_i \sum_{j=1}^{M} \frac{E2_j}{N_j} e^{-d_{ij}},\end{aligned}$$

where M is the number of regions, N_j is the total population of region j, and d_{ij} is the distance between the i-th and the j-th region. Note that the distance from a region to itself, d_{ii}, is always zero, which leaves this part of the expression unchanged from the previous SEIR model. This also means that if we have a single region ($M = 1$), the model is identical to the standard SEIR model presented above. The derivative for the exposed category $E1$ becomes

$$\frac{dE1_i}{dt} = -\frac{dS_i}{dt} - \lambda_1 E1_i.$$

We will now implement this extended SEIR model as subclasses of the model classes written in Problem 1.2.

a) Create a subclass of `Region` called `RegionInteraction`. The constructor should take the same arguments as the `Region` class, and two additional parameters `lat` (latitude) and `long` (latitude). The constructor should convert these values from degrees to radians, by multiplying by $\frac{\pi}{180^\circ}$, and store them as attributes. Call the superclass' constructor to store the rest of the parameters as attributes.[4]

Create a method `distance(self, other)` which calculates the distance between the `self` region (i) and another region (j). The distance is calculated as the arc length between the coordinate points of the two regions:

$$d_{ij} = R_{Earth}\Delta\sigma_{ij},$$

where the radius of the Earth is $R_{Earth} = 64$ given in units of 10^5 m and $\Delta\sigma$ is given by

$$\Delta\sigma_{ij} = \arccos\left(\sin\phi_i \sin\phi_j + \cos\phi_i \cos\phi_j \cos\left(|\lambda_i - \lambda_j|\right)\right).$$

Here, ϕ_i, λ_i are the latitudes and longitudes of the two locations, respectively. The arccos function is named `acos` in math and `arccos` in NumPy. You will use it with a single number as an argument, so both versions will work. **Warning:** Roundoff error may cause problems in the arccos function, since the arguments may become slightly > 1 when a location is compared with itself, and this makes the function return a NaN (Not a Number) value. To avoid this problem, add an if-test inside the distance function to ensure that the argument to arccos is between 0 and 1. The method should return the distance in units of 10^5 m.

In a main block at the bottom of the file, add code to test that the distance function gives the expected output.[5] For instance, the code can look as follows:

```
if __name__ == '__main__':
  innlandet = RegionInteraction('Innlandet',S_0=371385, E2_0=0, \
                                lat=60.7945,long=11.0680)
  oslo = RegionInteraction('Oslo',S_0=693494,E2_0=100, \
                           lat=59.9,long=10.8)
  print(oslo.distance(innlandet))
```

b) Create a subclass `ProblemInteraction` of class `ProblemSEIR`. The signature of the class' constructor should look as follows:

```
def __init__(self, region, area_name, beta, r_ia = 0.1, r_e2=1.25,
             lmbda_1=0.33, lmbda_2=0.5, p_a=0.4, mu=0.2):
             ...
             #store arguments as attributes
```

This is almost identical to the constructor of the superclass `ProblemSEIR`, but the argument `region` should in this case be a list of regions, which are all instances of class `RegionInteraction`, and `area_name` is a text string containing the

[4] For simplicity we represent the location of a region by a single pair of coordinates, which can be, for instance, the center of the region or the location of its capital or other administrative center.

[5] Or use the template file found here:
https://www.uio.no/studier/emner/matnat/ifi/IN1900/h20/ressurser/live_programmering/template_interaction.py

name of the total region. For instance, the list `region` argument could be a list of regions corresponding to counties in Norway, and the `area_name` would then be `'Norway'`. Save the area name as an attribute, and call the superclass' constructor to save all the other arguments as attributes.

The method `get_population(self)` should calculate and return the total population of all the regions in the list `region` combined.

The method `set_initial_condition(self)` must create a (not nested) list `self.initial_condition` with the initial values from all the regions. Loop over all the regions in the list `self.region` to create the list on the form

$$[S_1(0), E1_1(0), E2_1(0), I_1(0), Ia_1(0), R_1(0), S_2(0), E1_2(0), E2_2(0), ..., R_M(0)]$$

If we have M regions the result should be a one-dimensional (non-nested) list of length $6M$. (You should use the `+` or `+=` operators to add the lists together, since using `append` will create a nested list.)

The special method `__call__(self, u, t)` should return a list with the derivatives at time t, in the same order as the list `self.initial_condition`. This method specifies the right hand side of the total ODE system, with $6M$ ODEs, and is the core of our model for pandemic spread. The input argument `u` is a list of length $6M$ that contains the state variables for all the M regions. Inside `__call__` it is convenient to convert this to a nested list of states for the individual regions, and then loop over this list to compute the corresponding derivatives (right hand sides). Finally, we put these derivatives back together as a non-nested list of length $6M$. Below is a sketch of what the implementation can look like:

```
def __call__(self, u, t):
    n = len(self.region)
    # create a nested list:
    # SEIR_list[i] = [S_i, E1_i, E2_i, I_i, Ia_i, R_i]:
    SEIR_list = [u[i:i+6] for i in range(0, len(u), 6)]
    # Create separate lists containing E2 and Ia values:
    E2_list = [u[i] for i in range(2, len(u), 6)]
    Ia_list = ...
    derivative = []
    for i in range(n):
        S, E1, E2, I, Ia, R = SEIR_list[i]
        dS = 0
        for j in range(n):
            E2_other = E2_list[j]
            Ia_other = Ia_list[j]
            dS += ...
        # calculate dE1, dE2, dI, dIa, dR
        # put the values in the end of derivative
    return derivative
```

The method `solution` should take a list `u` containing the entire solution, and split it up into individual SEIR lists that should be sent to all the regions. One way to do this is to first convert `u` from a non-nested list to a nested list containing the individual SEIR-lists, and then to loop over this list and send each list to the correct region. The example below shows how it can be done. You do not have to use this code, but the result should be the same.

```
def solution(self, u, t):
        n = len(t)
        n_reg = len(self.region)
        self.t = t
        self.S = np.zeros(n)
        self.E1 = ...
        SEIR_list = [u[:, i:i+6] for i in range(0, n_reg*6, 6)]
        for part, SEIR in zip(self.region, SEIR_list):
            part.set_SEIR_values(SEIR, t)
            self.S += ...
```

The attributes `self.S`, `self.E1`, `self.E2`, `self.I`, `self.Ia`, and `self.R` should be the total values for all the regions combined, i.e., each SEIR-category summed over all regions.

Create a new method `plot(self)`. the method should create the same kind of plot as class `Region`'s method `plot(self)`, as explained in Problem 1.2. The method in `ProblemInteraction` should plot the S, I, Ia, and R values for all the regions combined (i.e., the sum over all regions), and the title of the plot should be `self.area_name`.

Extend the main block at the bottom of the file with code to demonstrate that the `ProblemInteraction` class is working correctly. For instance, you can add the following code. It may be useful to add the code line by line, and and make sure you get the expected output before adding the next line:

```
    problem = ProblemInteraction([oslo,innlandet],'Norway_east', beta=0.5)
    print(problem.get_population())
    problem.set_initial_condition()
    print(problem.initial_condition) #non-nested list of length 12
    u = problem.initial_condition
    print(problem(u,0)) #list of length 12. Check that values make sense

    #when lines above work, add this code to solve a test problem:
    solver = SolverSEIR(problem,T=100,dt=1.0)
    solver.solve()
    problem.plot()
    plt.legend()
    plt.show()
```

The final plot produced by the code above should show the total number of people in each category, for the regions `oslo` and `innlandet` combined. For instance, the S category should start from a value of 1064879. If some of the lines do not work as expected, it may be useful to debug the code by trying an even simpler problem, where `region` is a list with only one region (for instance `[oslo]`)).
Filename: `SEIR_interaction.py`

Problem 1.4. Simulate Covid19 in Norway (6 points)
In this exercise we will use the classes `ProblemInteraction`, `SolverSEIR` and `RegionInteraction` from Problem 1.3 to simulate scenarios of the Covid19 pandemic in Norway. The code should be written in a separate file `covid19.py`, which should import from `SEIR_interaction.py`.

a) The file `fylker.txt`, which is found on the course web site[6], contains information about all counties in Norway. Write a function that takes a file name as input, reads such a file, and returns a list of `RegionInteraction` instances. Check that your function works properly by creating additional files containing only one or two lines from `fylker.txt`, and verify that you get the expected result.

b) Create a function for simulating the Covid19 outbreak in Norway. The function should create a list of regions, create and solve the problem, and then plot a subplot of the disease dynamics in each region, and one subplot for the total progress for all regions combined. The function could look something like this:

```
def covid19_Norway(beta, filename, num_days, dt):
    # read file and create list of RegionInteraction instances
    # create problem, an instance of ProblemInteraction
    # create the solver, an instance of SolverSEIR
    # call the method solve
    plt.figure(figsize=(9, 12)) # set figsize
    index = 1
    # for each part in problem's attribute region:
        plt.subplot(4,3,index)
        # Call plot method from current part
        index += 1
    plt.subplot(4,3, index)
    plt.subplots_adjust(hspace = 0.75, wspace=0.5)
    # Call plot method from problem
    plt.legend()
    plt.show()
```

You can adjust the arguments to `plt.figure(figsize=(9, 12))` and `subplots_adjust(...)` to make the figure look nice. Call the function using `beta =0.5, num_days =100, dt =1.0`, and the `fylker.txt` input file. Examine the plot of the total cases to find the approximate peak for the I category. 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?

c) 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

[6] https://www.uio.no/studier/emner/matnat/ifi/IN1900/h20/ressurser/live_programmering/fylker.txt

Time interval	R value
February 15 to March 14	4.0
March 15 to April 20	0.5
April 21 to May 10	0.4
May 11 to June 30	0.8
July 1 to July 31	0.9
August 1 to August 30	1.0
September 1 -	1.1

Table 1.1: Reproduction numbers used by the Norwegian Institute of Public Health (FHI) to model the spread of the Covid19 pandemic.

infectiousness of the disease itself and the general behaviour of the population. We will now extend our model to use 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 E_2 period is $1/\lambda_2$ and the mean duration of for I, I_a is $1/\mu$. The choice of $\beta = 0.5$ used above gives $R = 4.0$, which is the value used by the Institute of Public Health (FHI) to model the early stage of the outbreak in Norway, from around February 15 to March 14. Since then, changes in people's behavior have led to variations in the reproduction number, and the models run by FHI have used the numbers given in Table 1.1.

Implement a Python function representing a piecewise constant β corresponding to the values and time intervals above, with time zero on February 15. You can either precompute each beta value and insert them directly into the function, or compute a piecewise constant R which is then converted to β. The number of days between the dates in Table 1.1 can be calculated by hand or you can use the `datetime` module. It may be useful to plot β as a function of t, to verify that you have implemented it correctly, before you try to use it in the model. Solve the model with the piecewise constant β, from February 15 until today. How do the numbers compare with the reported number of cases? Try to experiment with different β values, for instance assuming $R = 4.0$ after September 1st. What happens? Since $R = 4.0$ was the estimated reproduction number in the early stage of the pandemic, it may be seen as representing the "natural" pandemic spread, when there are no restrictions on travel and social interactions.

Filename: `covid19.py`