UiO : Department of Informatics
University of Oslo

Modeling the Covid19 pandemic in Norway

Final project in IN1900, fall 2021

November 3, 2021

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. 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. To simplify the programming a bit, example codes and templates are provided for some of the functionality. These code segments are provided here in the project description, and some of them are also available in template files which can be downloaded from the course website.[1] It is recommended to download and use these files.
5. Some of the provided example codes have missing lines which you have to fill in yourself. These places are marked either with `...` or with triple-quoted strings in the code. Regular comments (starting with #) are mostly explanations of the code that is there, and are not supposed to be replaced by your own code.
6. The total number of points available is 24. The number of points for each exercise is provided in the headline.
7. The deadline for handing in the project is November 28 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.
8. As always, collaboration is encouraged, but everyone needs to write and submit their own python files.

[1] See files `template_seir.py` and `template_interaction.py` in https://www.uio.no/studier/emner/matnat/ifi/IN1900/h21/ressurser/live__programmering

Modeling the spread of a pandemic

Problem 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[2]. 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[3]. Rename the function to `SEIR(u,t)`, and keep the rest of the function unchanged, with all model parameters defined as local variables inside the function. Use the following values for the variables:
`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`.
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:
`[-0.156666666666, -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_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`, 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 `u,t` containing the solution and the time.

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

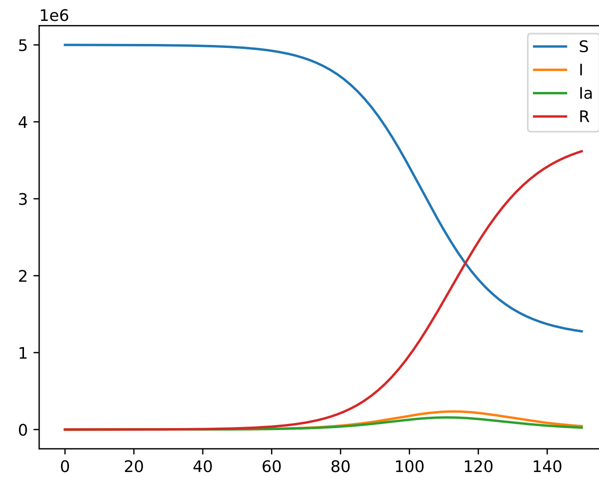

Figure 1: Solution of the SEIR model. The plot shows the dynamics of the categories S, I, Ia, R.

c) Make a function `plot_SEIR(u,t)` 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.

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=150` and `dt=1.0` (the time is given in days). The resulting plot should be similar to the one in Figure 1.
Filename: `seir_func.py`

Problem 2. Introduce classes in the SEIR model (8 points)
In this exercise we will implement the SEIR model from Problem 1 as a class, and extend the model with functionality to describe disease spread in a 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. This class holds the region-specific initial conditions for the categories $S, E1, \ldots, R$. After we have solved the SEIR model it will also include the individual solution components as attributes.
- 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 and method is implemented **exactly** as specified below.

a) Create a class `Region` which has three methods; a constructor (`__init__`), 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):
    self.name = name
    self.S_0 = S_0
    self.E1_0 = 0
    ...
```

The argument `name` is a text string specifying the name of the region, and the other arguments 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)` takes two arrays as arguments; `u` contains the solution of the SEIR system and `t` contains the time. The method should pull out the individual SEIR values from the argument `u` (using array slicing) 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:[4]

[4]Or use the template file found here:
https://www.uio.no/studier/emner/matnat/ifi/IN1900/h21/ressurser/live_programmering/template_seir.py

```python
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)) #a dummy solution array
    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 and prints the attributes of the class to verify that they are saved correctly. Then a dummy solution array `u` is created and passed to the `set_SEIR_values` method to verify that it works. 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`, `split_solution`, and `__call__`.

The constructor should take as arguments all the model parameters `beta`, `r_ia`, `r_e2, ...` and `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:

```python
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)): # is it a number?
        self.beta = lambda t: beta     # wrap as function
    elif callable(beta):
        self.beta = beta
    """
    Put code here for storing the other parameters
    and the region as attributes.
    """
    self.set_initial_condition()        # method call
```

The method `set_initial_condition(self)` 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 should be extracted from the class attribute `region`, which has all these initial conditions as its own attributes.

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 `split_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 at-

tribute `region`. Having a separate method for this may not seem very useful at this point, but it will be very convenient when we extend the class later.

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. 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 listed above with the following code lines:

```
    problem = ProblemSEIR(nor,beta=0.4)
    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 expected value specified in the `test_SEIR` function in Problem 1.

c) Now we will create a class `SolverSEIR` with two methods; a constructor and a method named `solve`. The constructor should take the arguments `problem` (an instance of class `ProblemSEIR`), `T` (the 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(...)
    """
    Insert code here to calculate
    the number of time steps from T and dt
    """
    t = np.linspace(...)
    u, t = solver.solve(t)
    #Send S, E1, ..., and t back to the region instance:
    self.problem.split_solution(u, t)
```

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

```
  solver = SolverSEIR(problem,T=150,dt=1.0)
  solver.solve()
  nor.plot()
  plt.legend()
  plt.show()
```

The resulting plot should look like the one you got in Problem 1

Filename: `SEIR.py`

Problem 3. The SEIR model across regions (8 points)
The problem class from Problem 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 result will be a complex and interesting ODE model, which is very similar to the models run by Norwegian Institute of Public Health (FHI) to predict the spread of the Covid19 pandemic. 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 $E2_i(t), Ia_i(t), I_i(t), R_i(t)$ 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

$$\frac{dS}{dt} = -\beta\frac{S_i I_i}{N_i} - r_{ia}\beta S_i\sum_{j=1}^{M}\frac{Ia_j}{N_j}e^{-kd_{ij}}$$
$$- r_{e2}\beta S_i\sum_{j=1}^{M}\frac{E2_j}{N_j}e^{-kd_{ij}}$$

where M is the number of regions, N_j is the total population of region j, d_{ij} is the distance between the i-th and the j-th region, and k is a parameter that scales the effect of the interactions between regions. 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.[5]

The derivative for the exposed category $E1_i$ becomes

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

with dS_i/dt given as above. All other parts of the SEIR model remain unchanged.

We will now implement this extended SEIR model as subclasses of the model classes written in Problem 2.

a) Create a subclass of `Region` called `RegionInteraction`. The constructorshould take the same parameters as the `Region` class, and two additional parameters `lat` (latitude) and `long` (longitude). The constructor should convert these values from degrees to radians, by multiplying by $\frac{\pi}{180}$, and store them as attributes. Call the superclass' constructor to store the rest of the parameters as attributes.[6] Create a method `distance(self, other)` which calculates the distance between the `self` region (i) and the `other` 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 $R_{Earth} = 6400$ (km) and $\Delta\sigma_{ij}$ is given by

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

Here, ϕ_i, ϕ_i are the latitudes of the two locations, and λ_i, λ_j are the corresponding longitudes. The arccos function is named `acos` in `math` and `arccos` in `NumPy`. You will use it with a single number as the 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 km. In a main block at the bottom of the file, add code to test that the distance function gives the expected output.[7] 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))
```

The coordinates specified for `innlandet` are for Hamar, so the output should be the approximate distance between Oslo and Hamar.

[5]This property is extremely useful for debugging, since we can run the model with a single region and check that the result is the same as in Problem 2.

[6]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.

[7]Or use the template file found here:
https://www.uio.no/studier/emner/matnat/ifi/IN1900/h21/ressurser/live_programmering/template_interaction.py

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, k=0.01):
    ...
    #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`, `area_name` is a text string containing the name of the total area, and we have introduced an additional parameter `k` which scales the interaction between regions. The `region` argument could, for instance, 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

$$[S1(0), E11(0), E21(0), I1(0), Ia1(0), R1(0), S2(0), E12(0), E22(0), ..., RM(0)]$$

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 += ...
            """
            Insert code to calculate dS, dE1, dE2,
            dI, dIa, dR and put the values at
            the end of the derivative list using +=
            """
    return derivative
```

The method `split_solution(u,t)` should take as first argument a list `u` containing the entire solution. This should be split up into individual SEIR lists, which are then sent to the individual region instances and stored as attributes there (using the method `set_SEIR_values` defined in the `Region` class). 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 split_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 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=0.4)
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=150,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 a simpler problem, where region is a list with only one region (for instance `[oslo]`)).

Filename: `SEIR_interaction.py`

Problem 4. Simulate Covid19 in Norway (6 points) In this exercise we will use the classes `ProblemInteraction, SolverSEIR` and `RegionInteraction` from Problem 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[8], contains information about all counties in Norway. Write a function `read_regions(filename)`, which 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):
    """
    Insert code here to do the following:
    1. call the read_regins function from a), to read file
    and file and create list of RegionInteraction instances
    2. create problem, an instance of ProblemInteraction
    3. create the solver, an instance of SolverSEIR
    4. call the method solve to solve the seir problem
    """
    plt.figure(figsize=(9, 12)) # set figsize
    index = 1
    """
    insert code to loop over all the regions
    and do the following:
    """
        plt.subplot(4,3,index)
        """
        Call plot method from the current region here
        """
        index += 1

    plt.subplot(4,3, index)
    plt.subplots_adjust(hspace = 0.75, wspace=0.5)
    """
    Insert a call to the plot method from
    problem (the ProblemInteraction instance)
    to plot the solution for all the regions combined
    """
    plt.legend()
```

[8] https://www.uio.no/studier/emner/matnat/ifi/IN1900/h21/ressurser/live_programmering/fylker.txt

```
    plt.show()
```

You may have to adjust the arguments to `plt.figure(figsize=(9, 12))` and `subplots_adjust(...)` to make the figure look nice, since the size and distance between subplots depend on the screen resolution. The point of this function is to create a figure containing four by three subplots, with eleven plots showing the evolution in each county, and the last one showing all t the counties combined.

Call the function using `beta=0.4, num_days =150, 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?s

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 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 $E2$ period is $1/\lambda_2$ and the mean duration of both I and Ia is $1/\mu$. The choice of $\beta = 04$ gives $R = 3.2$, 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 2020. 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.

Implement a Python function representing a piecewise constant β corresponding to the values and time intervals in Table 1, with time zero on February 15 2020. You can either precompute each beta value and insert them directly into the function, or first compute a piecewise constant R which is then converted to β. The number of days between the dates in Table 1 can be calculated by hand or you can use the `datetime` module. The piecewise constant function can be implemented using a number of if-tests in the usual way. Since there are 22 distinct values this will be a bit tedious to write. If you want you can implement the piecewise constant function for the first nine periods, from R0 to R8, and keep it constant after that.

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.[9]

[9] Recall that we cannot send an entire array as argument to a piecewise constant function implemented with if-tests. For plotting the function we therefore need to define an array of zeros and use a for-loop to fill it with the correct values. In the actual model this will not be an issue, since we will always pass a single number (t) as argument to the piecewise constant β function, but plotting the function requires a couple of extra lines.

Time interval		R value
15.02.2020 to 14.03.2020	(R0)	3.2
15.03.2020 to 19.04.2020	(R1)	0.5
20.04.2020 to 10.05.2020	(R2)	0.7
11.05.2020 to 30.06.2020	(R3)	0.7
01.07.2020 to 31.07.2020	(R4)	1.2
01.08.2020 to 31.08.2020	(R5)	1.0
01.09.2020 to 30.09.2020	(R6)	1.0
01.10.2020 to 25.10.2020	(R7)	1.3
26.10.2020 to 04.11.2020	(R8)	1.3
05.11.2020 to 30.11.2020	(R9)	0.81
01.12.2020 to 03.01.2021	(R10)	1.03
04.01.2021 to 21.01.2021	(R11)	0.6
22.01.2021 to 07.02.2021	(R12)	0.8
08.02.2021 to 01.03.2021	(R13)	1.5
02.02.2021 to 24.03.2021	(R14)	1.04
25.03.2021 to 12.04.2021	(R15)	0.76
13.04.2021 to 05.05.2021	(R16)	0.85
06.05.2021 to 26.05.2021	(R17)	1.0
27.05.2021 to 20.06.2021	(R18)	0.7
21.06.2021 to 11.07.2021	(R19)	1.1
12.07.2021 to 03.08.2021	(R20)	1.0
04.08.2021 to 31.08.2021	(R21)	1.2
01.09.2021 -	(R22)	0.78

Table 1: Reproduction numbers used in models of the Covid19 pandemic in Norway. Source: National institute of public health (https://www.fhi.no/publ/2020/koronavirus-ukerapporter).

Solve the model with the piecewise constant β, from February 15 2020 until today. How do the numbers compare with the reported number of cases? Try to experiment with different β values, for instance setting $R = 3.2$ after some period of time. What happens? Since $R = 3.2$ 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