Zombies and Covid-19 - Trust the models, or your gut?

MOD510: Mandatory project #3

Deadline: 15 November 2020 (23:59)

Oct 27, 2020

Learning objectives. By completing this project, the student will:

- Implement a solver for systems of first-order ordinary differential equations (ODEs) in Python.
- Learn how to escape a zombie invasion using wisdom from simple *compartment models*.
- Describe the spread of the Corona virus with a compartment model.
- Constrain model input parameters by comparing with data.

Project overview. In this project we are modeling the worst, but hoping (solving) for the best. If zombies invade a little village in Norway, how much time do the inhabitants have to prevent the apocalypse? Is it even possible? To answer these questions, we set up and solve deterministic compartment models. After simulating the zombie epidemic, we apply the same kind of model to real data for outbreaks of the Corona (SARS-CoV-2) virus.

Before we start:.

In this project, more than any other, you have to have control over numerical uncertainty and instabilities - your very future may depend on it!

Figure 1: Prepare for the worst (From the AMC show "The Walking Dead")

1 Exercise 1: General ODE solver

The goal of this exercise is to implement a solver that can handle *any* system of ordinary differential equations (ODEs) of the form

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t)\,, \tag{1}$$

$$\mathbf{y}(0) = \mathbf{y_0}\,, \tag{2}$$

where $t \geq 0$ is a real number, $\mathbf{y(t)}$ is the unknown solution vector with initial condition $\mathbf{y_0}$, and $\mathbf{f}$ is a vector-valued function that can depend upon both y and t.

It is important that the solver is general, because later you are going to apply it to a series of different compartment models.

Part 1.

- Implement a general ODE solver class or function.
- The solver should be able to switch between different numerical schemes, depending on user input:
  - Implement as one option the Forward (explicit) Euler scheme.
  - Implement as another option the (explicit) Runge-Kutta fourth order scheme (see [6], and the lecture notes)
- The solver should work regardless of the size of the system.

To be able to write a reusable solver, your code must be agnostic about the particular choice of right-hand side function. If you are unsure about how to do this, the Appendix contains suggestions with example code to get you started!

Part 2. To test that the solver works, we shall consider the second-order differential equation

$$x''(t) + \omega^2 x(t) = 0\,, \tag{3}$$

where $x_0 = x(0)$ and $v_0 = x'(0)$. The analytical solution is:

$$x(t) = x_0 \cdot \cos(\omega t) + \frac{v_0}{\omega} \cdot \sin(\omega t)\,. \tag{4}$$

- First, make a plot in which you show the analytical solution for several combinations of ω, x_0, and v_0.

Part 3. Equation (3) is a second order ODE, and it is therefore not in the form required by your numerical solver. However, there is a standard trick we can use to convert it into a *system of first order equations*: first, we introduce a new variable $v = v(t) = x'(t)$, and then we let

$$\mathbf{y} = \begin{pmatrix} x \\ v \end{pmatrix}\,.$$

We end up with,

$$\frac{d\mathbf{y}}{dt} = \begin{pmatrix} v \\ -\omega^2 x \end{pmatrix}\,, \tag{5}$$

which is of the form (1):

- Let $v_0 = 0$, $x_0 = 1$, and $\omega = 1$. Solve the system (5) with both the Forward Euler and the Runge-Kutta4 method.
- Simulate for at least $t = 3P$, where $P = 2\pi/\omega$, and start by letting $dt = 0.1$ (you may want to test different step sizes)
- Compare with the analytical solution.

2 Exercise 1.1 (OPTIONAL): Adaptive step size

- As a third solver option, implement an adaptive version of the Runge-Kutta fourth order scheme, using Richardson extrapolation.
- Compare the performance of the adaptive scheme to the non-adaptive schemes.

To be able to do so, you may have to modify your previous code.

3 Modeling the apocalypse: The SZ-model

Compartment models [7] are widely used to study how an epidemic disease might spread in a population. In these models, the population is partitioned into compartments based on a set of possible "disease states", and differential equations are set up to describe how individuals "flow" from one compartment to another. The equations can be either deterministic or stochastic. While the latter type of model is more realistic, we will only study deterministic models in this project.

We first consider the SZ-model, which consists of only two compartments:

1. S - Susceptible: humans that risk being turned into zombies; by being bitten or scratched by a zombie.
2. Z - Zombies.

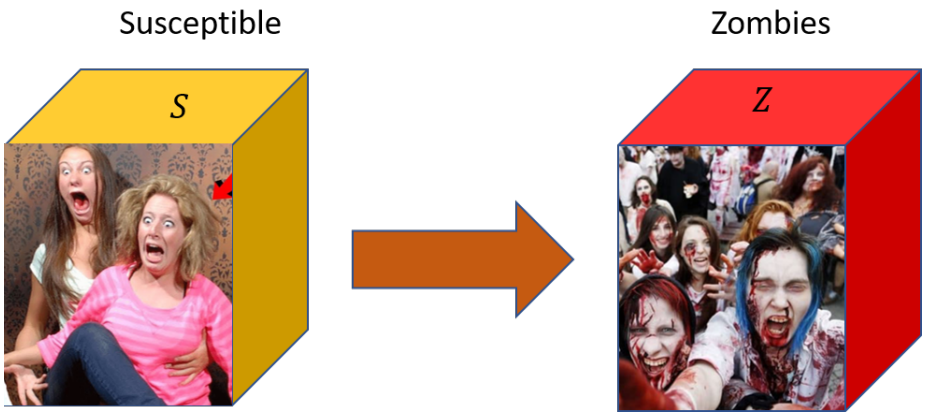

Figure 2: The SZ-model. Note that only transport from the class of exposed humans to zombies are allowed. Zombies are *never* cured; if you are bitten you are dead, dead, dead, ...

Let N be the total population size. For each time t, let $S(t)$ denote the number of humans, and $Z(t)$ the number of zombies. To develop a model, we need to calculate the rate of flow between the two compartments in figure 2. We start by making some observations:

- During each time interval Δt, a certain number of individuals will come into contact with each other.
- We only care about human-zombie encounters. Interactions between two humans, or between two zombies, are not relevant for the process of "zombification".

- Whenever a human meets a zombie, there is a certain probability that the human becomes infected.

We shall take our imagined population of humans and zombies to be *well mixed*, meaning that pairs of individuals interact with equal probability. Let $\mathcal{C}(N)$ denote the rate at which *any* individual in the population contacts *any* another individual, i.e., the average number of contacts made per unit time. Then, we can estimate the change in the human population from time t to $t + \Delta t$ as:

$$S(t + \Delta t) - S(t) = -\mathcal{C}(N) \cdot \Delta t \cdot p \cdot q \cdot S(t)\,. \tag{6}$$

where p denotes the conditional probability that a given contact is between a human and a zombie, and q is the probability that such an encounter leads to infection. Because of the well mixed assumption, a good assumption is that $p = Z(t)/N$; thus, the challenge consists in estimating $\mathcal{C}(N)$ and q. In principle, both of these parameters may vary in time, but for now we shall regard them as constant. By merging them into a single factor, β, we get

$$S(t + \Delta t) - S(t) = -\beta \cdot \Delta t \cdot \frac{S(t)Z(t)}{N}\,, \tag{7}$$

Finally, by dividing by Δt and letting $\Delta t \to 0$, we obtain the following ODE:

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta \cdot \frac{S(t)Z(t)}{N}\,. \tag{8}$$

Similarly, the evolution of the zombie population is given by:

$$\frac{\mathrm{d}Z(t)}{\mathrm{d}t} = +\beta \cdot \frac{S(t)Z(t)}{N}\,. \tag{9}$$

How to interpret β?.

By saying that β is constant, we have made two very strong assumptions:

- People make the same number of contacts regardless of the population size, and independent of time.
- The probability of becoming a zombie, given that you meet one, always stays the same.

In reality, β is time-dependent, as it implicitly accounts for a lot of biomedical, physical, and sociological factors. For example, in the beginning of a zombie outbreak, β is likely to be large, because humanity might not yet understand the severity of the situation, or they may be in denial. As people start to realize the danger and fight back against the zombies, we expect that β will decrease!

4 Exercise 2: No hope?

Clearly, if there are no infected individuals at time zero, the above equations predict that nothing will happen later either. We shall therefore assume that the initial number of zombies is close to one; typically $Z_0 = Z(0) = 1$.

Part 1.

- Show that the analytical solution to the SZ-model, equations (8) and (9), is

$$S(t) = \frac{(S_0 + Z_0)\frac{S_0}{Z_0}\exp(-\beta t)}{1 + \frac{S_0}{Z_0}\exp(-\beta t)}, \tag{10}$$

$$Z(t) = \frac{S_0 + Z_0}{1 + \frac{S_0}{Z_0}\exp(-\beta t)}, \tag{11}$$

where $S_0 = S(0)$, and thus $S_0 + Z_0 = N$.

Part 2. The SZ-model predicts that the entire human population will be turned into zombies as $t \to \infty$:

- Prove this mathematically.

Part 3. Sokndal and Dirdal are two small villages in Rogaland. Two scientists from these places went to an international conference on numerical methods in Haiti. During the conference, an excursion was arranged to a rural area, and a strange tomato salad consisting of, among other things, pufferfish venom, was served [4]. After arriving back in Norway, both scientists got a fever, stopped eating, and subsequently started to behave suspiciously.

Figure 3: A view from Sokndal (left) and Dirdal (right).

Dirdal has about 683 inhabitants, and Sokndal 3305 [2]. We want to use the SZ-model to investigate a potential zombie invasion in these two places:

- Assume that $\beta = 0.06$ 1/hour. Use the analytical solution, equation (11), to calculate how the number of zombies changes as a function of time in Dirdal and Sokndal.
- Make a figure where you compare the results.

Part 4. Use your ODESolver to compute numerical solutions to the case you studied in the previous part:

- Make one plot in which you compare the ForwardEuler results to the analytical solution.
- Make another plot where you apply the RungeKutta4 method.
- Roughly which time steps do you need to take in order to get accurate solutions?

Part 5. Later on in a zombie outbreak, people become more aware and not so easily fooled by the hordes of undead; after all, zombies are not known to be particularly bright. To capture this behavior, the zombie-infection rate will now be assumed to decline exponentially:

$$\beta(t) = \beta_0 e^{-\lambda t}. \tag{12}$$

Suppose moreover that, after a specific time T, the probability of infection has been reduced to 60% of its initial value. We can then estimate λ from:

$$e^{-\lambda T} = 0.6, \tag{13}$$

from which it follows that

$$\lambda = -\frac{1}{T} \ln 0.6. \tag{14}$$

Sokndal versus Dirdal?.

In Sokndal, it takes 48 hours for the probability of being infected by a zombie to drop by 40 %. In Dirdal, it takes 72 hours. One explanation for this is that the inhabitants of Dirdal are more tolerant of unorthodox behavior, with the consequence that it takes more time for them to discover the zombies. However, anecdotal evidence suggests that people in Sokndal behave in quite strange ways also, as might be inferred from the lyrics of their local singer-songwriter Tønes.

Based on the information you have been given above:

- Calculate the numerical value of λ for both the Dirdal and the Sokndal zombie outbreak.

- Solve the extended SZ-model numerically.
- Roughly how many people survive the zombie outbreak in the two locations?

Again, you should compare the different numerical schemes with each other.

5 The SEZR-model

In the SZ-model with time-dependent β, some of the humans might survive, but that is only because the probability of infection eventually drops to zero; the zombies are still roaming around. However, we know from data on real outbreaks (e.g., [8]) that the humans will eventually fight back and start killing zombies. The SZ-model furthermore supposes that bitten humans turn into zombies instantaneously. While this conservative assumption are supported by certain movies and TV shows, other sources [3] indicate that a latency period of about $1/\sigma = 24$ hours may be more realistic. Both of the above facts should be acknowledged in our models. We therefore refine our model by adding two new compartments:

- E: *exposed* humans, people that have been bitten or scratched by a zombie, but who still remain non-contagious to others.
- R: *removed* (killed) zombies.

The system of equations (15)-(18) will hereafter be referred to as the $SEZR$-model, see figure 4.

The $SEZR$-model

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta(t) \cdot \frac{S(t)Z(t)}{N}\,, \tag{15}$$

$$\frac{\mathrm{d}E(t)}{\mathrm{d}t} = \beta(t) \cdot \frac{S(t)Z(t)}{N} - \sigma \cdot E(t)\,, \tag{16}$$

$$\frac{\mathrm{d}Z(t)}{\mathrm{d}t} = \sigma \cdot E(t) - (\alpha + \omega(t))\,\frac{S(t)Z(t)}{N}\,, \tag{17}$$

$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = (\alpha + \omega(t))\,\frac{S(t)Z(t)}{N}\,. \tag{18}$$

In the above equations, the parameter α can be thought of as the average rate of zombie killing, while the function $\omega(t)$ represents a series of violent attacks launched by humans at specific points in time. Following Langtangen et al. [9], we define:

$$\omega(t) = a \sum_{i=1}^{m} \exp\left(\frac{1}{2}(\frac{t - T_i}{T_\sigma})^2\right)\,, \tag{19}$$

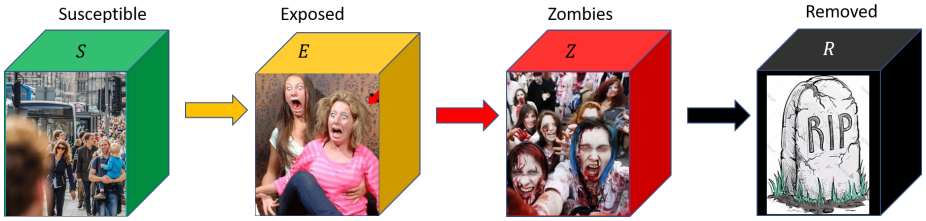

Figure 4: SEZR model.

where a and T_σ are constant parameters, and where $T_1, T_2, \ldots, T_m$ are the times at which the humans launch violent attacks.

The reproduction number.

The *basic reproduction number*, is a number that determines whether a disease is able to spread in population. It represents the average number of new infected cases produced, in an entirely susceptible population, by a typical infected individual [5]. In the SEZR-model, it is a good approximation to define it as

$$\mathcal{R}_0 \simeq \frac{\beta(t)}{\alpha + \omega(t)}. \tag{20}$$

6 Exercise 3: Counter-strike

Part 1.

- Implement the violent attack function, Eq. (19), as a Python function.
- Plot the function for the following choice of values $a = 40 \cdot \beta_0$ and $T_\sigma = 1$
- What is the physical interpretation of the parameters a and T_σ?

Part 2. For simplicity we shall again suppose that $\beta(t) = \beta_0$ is constant. Choose either Sokndal or Dirdal. Run a few simulations:

- Start by letting $\alpha > 0$, but set $\omega(t) = 0$. Is it possible to survive the zombie apocalypse if $\beta/\alpha > 1$?
- Turn on violent attacks. Does it now seem possible to survive if $\beta/\alpha > 1$?

If the number of zombies drops below one, you might want to set $\beta \to 0$. Similarly, if $E(t) < 1$ you could set $\sigma \to 0$. After all, there is no such thing as a fractional zombie or human. (Or is there ...?)

7 Exercise 4: Compartment model for Covid-19

In the final part of the project we wish to study the transmission of Covid-19 (Corona virus disease) by means of a compartment model with five compartments:

The $SEIRD$-model

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta(t) \cdot \frac{S(t)I(t)}{N}\,, \tag{21}$$

$$\frac{\mathrm{d}E(t)}{\mathrm{d}t} = \beta(t) \cdot \frac{S(t)I(t)}{N} - \sigma \cdot E(t)\,, \tag{22}$$

$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = \sigma \cdot E(t) - \gamma \cdot I(t)\,, \tag{23}$$

$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = (1-f) \cdot \gamma \cdot I(t)\,, \tag{24}$$

$$\frac{\mathrm{d}D(t)}{\mathrm{d}t} = f \cdot \gamma \cdot I(t)\,. \tag{25}$$

In this model the *basic reproduction number* is given by $\mathcal{R}_0 = \beta(t)/\gamma$.

Notice that we have changed the notation for the infectious compartment from $Z(t)$ to $I(t)$. The exposed compartment is the same as before, but we have split people who no longer have the disease into two categories:

1. $R(t)$ is the number of people who have become immune ("Recovered").
2. $D(t)$ denotes the number of dead individuals.

Data for the Corona virus are readily available. We will use data found at the Github repository 'Center for Systems Science and Engineering (CSSE) at Johns Hopkins University'. We have already extracted country-level data for you, and stored it in a processed format in the text file `corona_data.dat`. Data for the Hubei province in China, where it is believed that the virus first arose, is also included in the text file.

The reason for doing so is that while the original data were organized by date since January 22 2020, we would like to plot the data versus the time of the first confirmed case. This makes it easier to apply the same model to different locations.

Part 1.

- Plot the total number of a) confirmed cases and b) deaths of Covid-19 for Norway, Sweden, the Hubei province in China, and a couple of other locations of your own choice.

Consider to make more than one Python function to achieve this, for example one that reads the data file into a Pandas Data Frame and returns it, and another function to plot confirmed cases and/or deaths.

You might also want to normalize the data rather than showing absolute numbers, e.g., to plot the number of confirmed cases / deaths per 100 000 inhabitants (find relevant source(s) for population data)

Part 2. Next, we want to fit our model to the data. To this end, you can for example use `scipy.optimize.curve_fit` function.

As before, calculate $\beta(t)$ from equation (12). If we assume the mean infectious period to be approximately 20 days [1], $1/\gamma$=20, and that the mean incubation period, $1/\sigma$, is 5.1 days [10], the only unknowns are β_0, λ, and the death rate, f.

- Fit the $SEIRD$ model (i.e. determine β_0, and λ) to the confirmed number of cases for Hubei, assuming a death rate of 5 %.
- Plot 1) the number of confirmed cases, 2) the number of deaths, and 3) the results of the fitted model *in the same plot*. Include a secondary y-axis where you plot the basic reproduction number versus time.

Part 3.

- Use the same model you fitted to the Hubei data to predict disease transmission in the other places you considered before (Norway, Sweden, etc.)

Part 4.

- Tune the model individually to the different places, so that you get a better match than in the previous part (you might want to change the death rate)

8 In your analysis, discuss (at least) the following points

Zombie How the Euler and Runge Kutta4 methods perform in terms of both accuracy and speed.

Zombie How the SEZR-model could be further improved.

Covid-19 Strengths and weaknesses of the presented SEIRD-model for Covid-19.

Covid-19 In particular, comment on the model we have chosen for $\beta(t)$. Do you have any better suggestions?

9 Appendix A: Implementing ODE solvers using standalone functions

To implement a general ODE solver, you should make use of the fact that in Python, functions are *first-class objects*. That is, functions are objects just like any other (e.g., numbers and strings), which means that functions can passed in as input arguments to other functions, they can be return values of other functions, etc. As an example to whet your appetite, consider the following code:

```
def forward_euler_step(f, yn, tn, dt):
    return yn + dt * f(yn, tn)
```

The function `forward_euler_step` has four input arguments: 1) The right-hand side function of the ODE, 2) the current solution, 3) the current time, and 4) the time step to use when advancing the solution forward in time to the next time step. The code is completely general; all we require is that `f` is a function that takes two input arguments, Y and t, e.g.:

```
def f(y, t):
    return y

def g(y, t):
    return -y

# Set initial condition & time step:
t0 = 0
y0 = 1
dt = 0.1

# Compute solution at t=dt for two different choices of the
# right-hand side function:
y1_f = forward_euler_step(f, y0, t0, 0.1)
y1_g = forward_euler_step(g, y0, t0, 0.1)
```

The above code will even work for functions of several variables, provided that `f` is a vectorized function and that `yn` is an array of the correct size. As an example of this, consider the SZ-model introduced in the main text. While we could reduce this system of two equations to a single equation by using that $S(t) + Z(t) = N$, we ignore this and write it in the general form (1) by setting

$$\mathbf{y}(t) = \begin{pmatrix} S(t) \\ Z(t) \end{pmatrix}. \tag{26}$$

$$\mathbf{f}(\mathbf{y}(t), t) = \begin{pmatrix} -\frac{\beta S(t) Z(t)}{N} \\ +\frac{\beta S(t) Z(t)}{N} \end{pmatrix}. \tag{27}$$

$$. \tag{28}$$

We can implement the vector-valued right-hand side function like this:

```
# We start by defining the model parameters outside of the function
N = 683
beta = 0.06

# Next, we use the current parameters to create a function with
# only Y and t as input arguments:
def rhs_SZ(Y, t):
    S, Z = Y
    return np.array([-beta*S*Z/N, beta*S*Z/N])
```

Notice the clever use of tuple unpacking here; the first line of the function definition is a short-hand equivalent to

```
S = Y[0]
Z = Y[1]
```

With this approach, we can take a Forward-Euler step as follows:

```
t0 = 0
y0 = np.array([682, 1])  # notice that y0 is now an array of size 2
dt = 0.1

# Compute solution at t=dt:
y1 = forward_euler_step(rhs_SZ, y0, t0, 0.1)
```

So far, we have only implemented a single time step. To implement a solver that computes the solution at a sequence of time steps $0, \Delta t, 2\Delta t, 3\Delta t, \ldots$, we can call the function `forward_euler_step` inside yet another function, e.g.:

```
def ode_solver_forward_euler(f, y0, dt, max_time):

    no_time_steps = int(max_time/dt)

    times = [0]
    solutions = [y0]
    for _ in range(no_time_steps):
        # insert missing code here
    return np.array(times), np.array(solutions)
```

Once finished, the function can be executed as follows:

```
t_num, y_num = ode_solver_forward_euler(rhs_SZ, y0, 0.01, 100)
```

Of course, we would have to make additional modifications to accommodate different choices of numerical schemes.

10 Appendix B: Be aware of variable scope!

In the above code examples, the model parameters β and N were defined outside of the function `f`, but the function still had access to the values. In general when running Python code, this is only possible if the following conditions are met:

- The function definition is placed within the same scope as the variables it uses (i.e., β and N), and
- The function definition comes after the variables have been declared.

It should be mentioned that Jupyter notebooks work differently. This is because you can run cells in any order, which means that even if you delete the declaration of β and N, their previous values will be stored in memory until you reset the Python kernel. This also implies that if at any time you execute a cell which changes β or N, all subsequent calls to the ODE solver will use the latest values!

For a very simple example illustrating this behavior, consider the following function, defined in a scope where no variable `x` has previously been assigned:

```
def test_function():
    print(x)
```

Trying to execute the function will lead to a *NameError*:

```
test_function()
```

On the other hand, we can get the code to work by first typing, e.g.

```
x = 1
```

and then re-running the cell executing the function. However, if we restart the Python kernel, the call to `test_function` will not work anymore! This is because we are once more in a situation where the test function is called *before* `x` has been assigned.

To summarize, you should be very careful when using variables defined outside of a function inside the function. Sometimes it may be necessary, but remember that the order in which you execute your code could then be very important!

See this link for a detailed explanation of variable scope in Python.

11 Appendix C: Passing function arguments using `*args` and `*kwargs`

Another way you can implement your ODE solver is to make use of a special Python syntax that lets you pass in an arbitrary number of parameters to a function. For example, you can extend the above solver functions by adding `*args` to the end of their input argument lists:

```
def forward_euler_step(f, yn, tn, dt, *args):
    return yn + dt * f(yn, tn, *args)

def ode_solver_forward_euler(f, y0, dt, max_time, *args):

    no_time_steps = int(max_time/dt)
```

```
    times = [0]
    solutions = [y0]
    for _ in range(no_time_steps):
        # insert missing code here
    return np.array(times), np.array(solutions)
```

Next, we re-define the right-hand side function to explicitly include β and N as input parameters:

```
def f_SZ(Y, t, beta, N):
    S, Z = Y
    return np.array([-beta*S*Z/N, beta*S*Z/N])
```

Finally, you can run the solver by typing:

```
t_num, y_num = ode_solver_forward_euler(f_SZ, y0, 0.01, 100, beta, N)
```

See this tutorial on RealPython for a good introduction to how you can use this technique. The tutorial also explains how you may do the same with keyword arguments (`**kwargs`).

12 Appendix D: Implementing ODE solvers using object-oriented programming

A more advanced method for implementing your ODE solvers is to use classes. You can of course code everything inside a single class, as you probably did in Project 2. However, here it might be an even better idea to code in an object-oriented way by making several classes:

- A base class called, e.g., `ODESolver`.
- Several subclasses, one for each numerical scheme, e.g.. `ForwardEuler`, `RungeKutta4`, etc.

If you want to try this strategy, the code below illustrates how you could start:

```
class ODESolver:
    """
    Base class for ODE solvers.
    Subclasses of this class will implement a particular numerical
    scheme for solving a system of first-order ODEs of the form

        dY/dt = f(Y, t), Y(0)=Y0,

    where Y=Y(t) can be either a function of a single real-valued
    variable, or a vector of such functions.

    Note that the current implementation presumes a constant step
    size for the integration. This assumption will of course have
    to be relaxed if adaptive step size control is to be used.
    """
```

```
    def __init__(self, f, y0, dt):
        """
        :param f: The right-hand side function of the ODE (system).
        :param y0: The initial condition.
        :param dt: Constant step size.
        """
        # Trick: Ensure that f will return an array even if the
        # user returns a list
        self.f_ = lambda y, t: np.array(f(y, t), dtype='float')
        self.y0_ = y0
        self.dt_ = dt

        # Set simulation history to "None" before initializing
        self.t_ = None
        self.y_ = None

    def reset_model(self):
        """
        Resets the model before starting up a new simulation.
        """
        self.t_ = [0]
        self.y_ = [self.y0_]

    def solve(self, max_t, verbose=0):

        self.reset_model()

        no_steps = int(max_t / self.dt_)

        current_t = 0
        dt = self.dt_
        for _ in range(no_steps):
            self.advance(dt)
            current_t += dt
            if verbose:
                msg = 'Done computing solution at'
                msg += ' t={}.'.format(current_t)
                print(msg)
        return np.array(self.t_), np.array(self.y_)

    def advance(self, dt):
        err = "advance() cannot be called on base class ODESolver."
        err += " Must call on an instance of an ODESolver subclass!"
        raise NotImplementedError(err)

class ForwardEuler(ODESolver):

    def __init__(self, f, y0, dt):
        super().__init__(f, y0, dt)  # call superclass __init__ method

    def advance(self, dt):
        tn, yn = self.t_[-1], self.y_[-1]
        f = self.f_
        self.t_.append(tn+dt)
        self.y_.append(yn + dt * f(yn, tn))
```

References

[1] Coronavirus modelling at the NIPH. https://www.fhi.no/en/id/infectious-diseases/coronavirus/

coronavirus-modelling-at-the-niph-fhi/. Accessed: 2019-10-26.

[2] Store Norske Leksikon. https://snl.no. Accessed: 2020-10-23.

[3] Max Brooks. *The Zombie Survival Guide: Complete Protection From the Living Dead*. Broadway Books, 2003.

[4] Wade Davis. *The Serpent and the Rainbow*. Simon and Schuster, 2010.

[5] Odo Diekmann, Johan Andre Peter Heesterbeek, and Johan AJ Metz. On the definition and the computation of the basic reproduction ratio r 0 in models for infectious diseases in heterogeneous populations. *Journal of mathematical biology*, 28(4):365–382, 1990.

[6] Aksel Hiorth. *Computational Engineering and Modeling*. https://github.com/ahiorth/CompEngineering, 2019.

[7] William Ogilvy Kermack and Anderson G. McKendrick. A contribution to the mathematical theory of epidemics—i. *Proceedings of the Royal Society of London. Series A, Containing papers of a mathematical and physical character*, 115(772):700–721, 1927.

[8] Robert Kirkman. *The Walking Dead Vol. 1: Days Gone Bye*, volume 1. Image Comics, 2004.

[9] Hans Petter Langtangen, Kent-Andre Mardal, and Pål Røtnes. Escaping the zombie threat by mathematics. *Zombies in the Academy-Living Death in Higher Education*, 2013.

[10] Stephen A. Lauer, Kyra H. Grantz, Qifang Bi, Forrest K. Jones, Qulu Zheng, Hannah R. Meredith, Andrew S. Azman, Nicholas G. Reich, and Justin Lessler. The incubation period of coronavirus disease 2019 (covid-19) from publicly reported confirmed cases: Estimation and application. *Annals of internal medicine*, 172(9):577–582, 2020.