7. ODE 1

> *The mathematical discipline of differential equations furnishes the explanation of all those elementary manifestations of nature which involve time.*
> –Norwegian Mathematician Sophus Lie

The topic of this chapter is to find *approximate solutions* to *ordinary differential equations*.

Let us briefly recall what an ordinary differential equation (ODE) is. A rather arbitrarily chosen example for an ODE (here, of second order) is

$$y''(x) + 4y'(x) + \sqrt[3]{y(x)} + \cos(x) = 0. \tag{7.1}$$

Equations like this are normally satisfied by many functions $y(x)$: the problem has many solutions. In order to specify a uniquely solvable problem, one needs to fix *initial values*, i.e., the value of y and its first derivative at some point, say, at $x = 0$:

$$y''(x) + 4y'(x) + \sqrt[3]{y(x)} + \cos(x) = 0, \quad y(0) = 1,\ y'(0) = -2. \tag{7.2}$$

This is a so-called *initial-value problem* (IVP). Another variant is to specify the value of $y(x)$, but not of its derivative, at two different points:

$$y''(x) + 4y'(x) + \sqrt[3]{y(x)} + \cos(x) = 0, \quad y(0) = 2,\ y(1) = 1. \tag{7.3}$$

This is called a *boundary value problem* (BVP).

Both IVPs and BVPs have a unique solution (under certain mathematical conditions). However, while one can show on abstract grounds that these solutions exist, it is often not practicable to find an explicit expression for them. The best one can hope for is to approximate the solution numerically.

So what *is* a numerical solution to a differential equation?
When solving a differential equation with analytic techniques the goal is to find an expression for the solution in terms of known functions. In a numerical solution the goal is typically to divide the domain for the solution function into a fine partition by introducing a grid of points, just like we did with numerical differentiation and integration, and then to approximate the solution to the differential equation at each point in that partition. Hence, the end result will be a list of approximate solution values at the grid points.

In this chapter we will examine some of the more common ways to create approximations of solutions to initial value problems. Moreover, we will lean heavily on Taylor Series to give us ways to accurately measure the order of the errors that we make in the process.

In this chapter we will often think of the argument of the function described by the ODE as time, but of course the methods are agnostic to the interpretation of the independent variable.

We will first concentrate on first-order differential equations of the form $x' = f(x)$. The geometric interpretation of this is that we are looking for a function $x(t)$ whose slope at every x is known to us, given by $f(x)$.

7.1. Euler's Method

Exercise 7.1. Consider the differential equation $x' = -0.5x$ with the initial condition $x(0) = 6$.

a. Since we know that $x(0) = 6$ and we know that $x'(0) = -0.5x(0)$ we can approximate the value of x at some future time step. Let us go 1 unit forward in time. That is, approximate $x(1)$ knowing that $x(0) = 6$ and $x'(0) = -3$.
Hint: We know a value, a slope, and the size of the step that we would like to move in the t direction.

$$x(1) \approx \_\_\_\_\_ \tag{7.4}$$

b. Use your answer from part (a) for time $t = 1$ to approximate the x value at time $t = 2$. Then use that value to approximate the value at time $t = 3$. Repeat the process to approximate the value of x at times $t = 2, 3, 4$. Record your answers in the table below. Then find the analytic solution to this differential equation and record the x values at the appropriate times.

t	**0**	**1**	**2**	**3**	**4**
Approximation of $x(t)$	6				
Exact value of $x(t)$	6				

c. The "approximations of x" that you found in part (b) are a **numerical approximation** of the solution to the differential equation. You should notice that your numerical solution is pretty far off from the actual solution for most values of t. Why? What could be the sources of this error and how could we fix it?

d. In Figure 7.1 you see a slope field and the exact solution to the differential equation $x' = -0.5x$ with $x(0) = 6$. Mark your approximate solutions at times $t = 1$, $t = 2$, ..., $t = 4$ on the plot and connect them with straight lines.

    1. Why are we using straight lines to connect the points?

2. What do you notice about your approximate solutions?

3. Why is it helpful to have the slope field in the background on this plot?

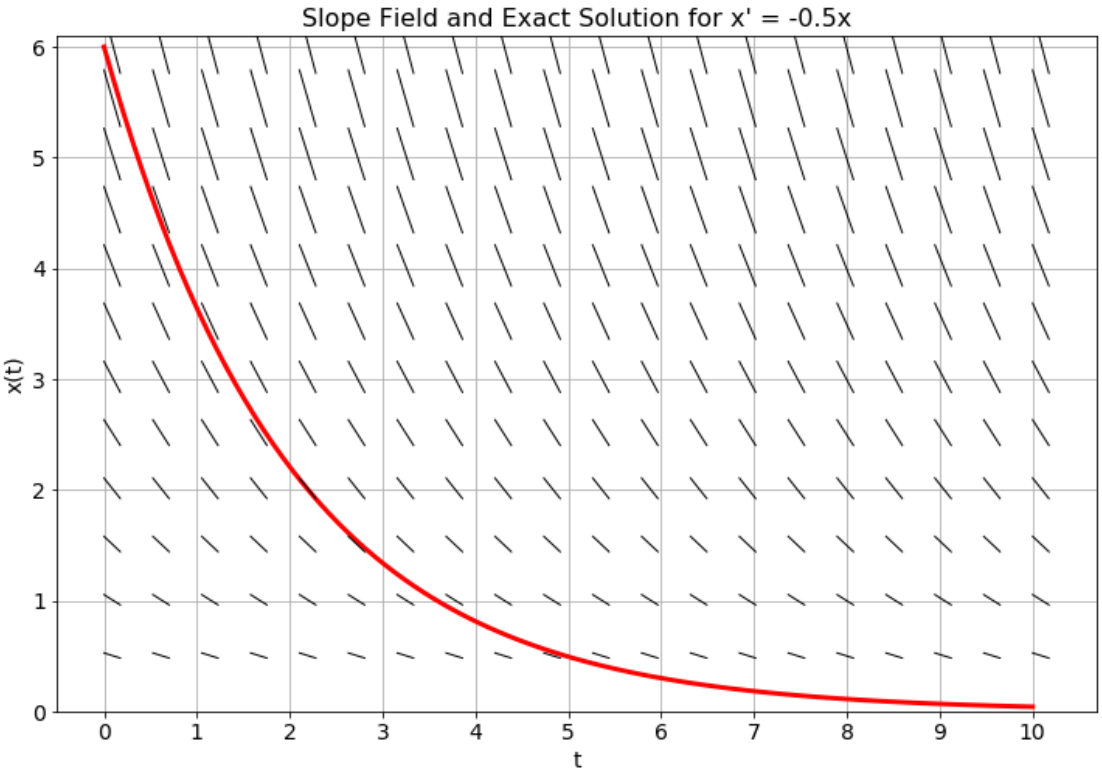

Figure 7.1.: Plot your approximate solution on top of the slope field and the exact solution.

Exercise 7.2. In Figure 7.2 you see the analytic solution with initial condition $x(0) = 5$ and a slope field for an unknown differential equation.

a. Use the slope field and a step size of $\Delta t = 1$ to plot approximate solution values at $t = 1$, $t = 2$, ..., $t = 10$. Connect your points with straight lines. The collection of line segments that you just drew is an approximation to the solution of the unknown differential equation.

b. Use the slope field and a step size of $\Delta t = 0.5$ to plot approximate solution values at $t = 0.5$, $t = 1$, $t = 1.5$, ..., $t = 10$. Again, connect your points with straight lines to get an approximation of the solution to the unknown differential equation.

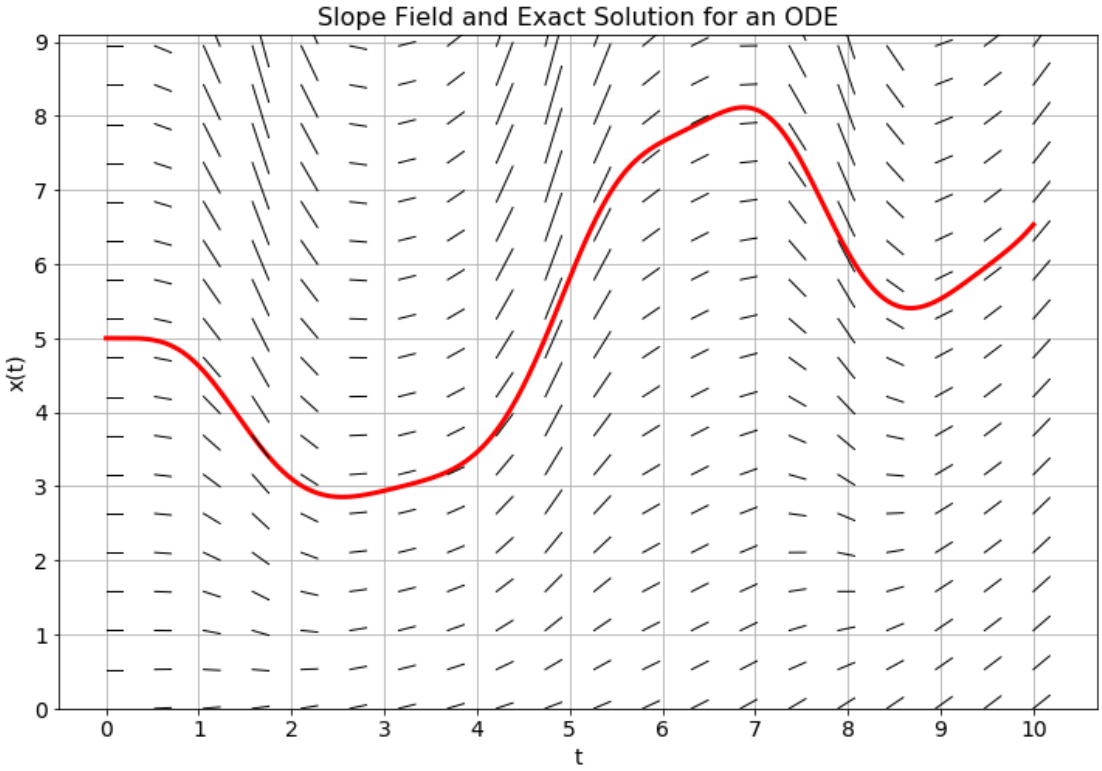

Figure 7.2.: Plot your approximate solution on top of the slope field and the exact solution.

c. If you could take Δt to be very very small, what difference would you see graphically between the exact solution and your collection of line segments? Why?

The notion of approximating solutions to differential equations is simple in principle:

- make a discrete approximation to the derivative and
- step forward through time as a difference equation.

The challenging part is making the approximation to the derivative(s). There are many methods for approximating derivatives, and that is exactly where we will start.

Definition 7.1 (Euler's Method)**.** Euler's Method is a technique for approximating the solution to the differential equation $x'(t) = f(t, x(t))$. Recall from Exercise 5.9 that the first derivative of a function can be discretized as

$$x'(t) = \frac{x(t+h) - x(t)}{h} + \mathcal{O}(h) \tag{7.5}$$

where $h = \Delta t$ is the step size (or the size of each partition in the domain), so the differential equation $x'(t) = f(t, x(t))$ becomes

$$\frac{x(t+h) - x(t)}{h} \approx f(t, x(t)). \tag{7.6}$$

Rewriting as a difference equation, letting $x_{n+1} = x(t_n + h)$ and $x_n = x(t_n)$, we get

$$x_{n+1} = x_n + hf(t_n, x_n) \tag{7.7}$$

A way to think about Euler's method is that at a given point, the slope is approximated by the value of the right-hand side of the differential equation and then we step forward h units in time following that slope. Figure 7.3 shows a depiction of the idea. Notice in the figure that in regions of high curvature Euler's method will deviate a lot from the exact solution to the differential equation. However, taking the limit as h tends to 0 theoretically gives the exact solution at the trade off of needing infinite computational resources.

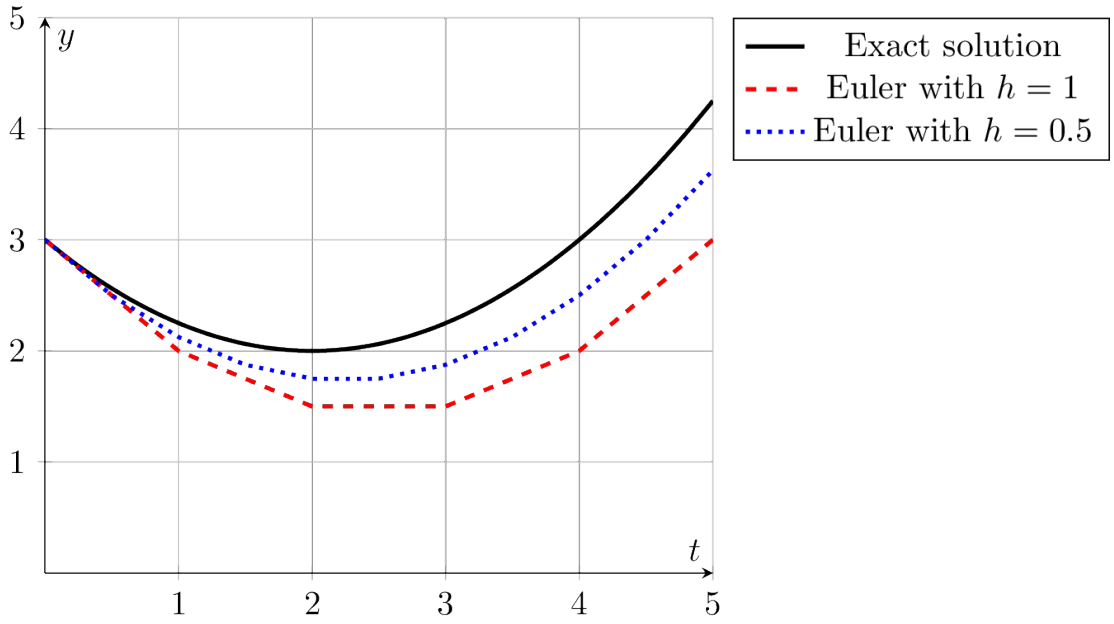

Figure 7.3.: Numerical solutions to a differential equation using Euler's method.

Exercise 7.3. Why would Euler's method do poorly in regions where the solution exhibits high curvature?

Exercise 7.4. Consider the differential equation $x'(t) = -2x(t)/3 + 4t$ with initial condition $x(0) = 6$. By hand perform four steps of the Euler method with stepsize $h = 1/2$ to obtain an approximation for $x(1/2), x(1), x(3/2)$ and $x(2)$.

Exercise 7.5. Write code to implement Euler's method for initial value problems. Your function should accept as input a Python function $f(t, x)$, an initial condition, a start time, an end time, and the value of $h = \Delta t$. The output should be vectors for t and x that you can easily plot to show the numerical solution. The code below will get you started.

```
def euler_1d(f, x0, t0, tmax, dt):
    """
    Solves a first-order ordinary differential equation using the Euler method.

    Parameters:
```

```
        f    : function, the function defining the differential equation. It should
               take two arguments, the independent variable t and the dependent
               variable x, and return the derivative of x with respect to t.
        x0   : float, the initial value of the dependent variable.
        t0   : float, the initial value of the independent variable.
        tmax : float, the maximum value of the independent variable.
        dt   : float, the time step.

    Returns:
       tuple containing two numpy arrays:
            - t : vector of time values.
            - x : vector of solution values at each time value.
    """
    # We need an integer number of steps so we round the number of steps
    # and then adjust dt to match.
    N = round((tmax - t0)/dt)
    dt = (tmax - t0)/N

    # Set up the time grid
    t = np.linspace(t0, tmax, N+1)
    # set up an array for x that is the same size as t
    x = np.zeros(len(t))

    # fill in the initial condition
    x[0] = ???

    for n in range(???): # think about how far we should loop
        # advance the solution forward in time with Euler
        x[n+1] = ???
    return t, x
```

Exercise 7.6. Test your code from the previous exercise on a first order differential equation where you know an analytic solution. For example you could use the differential equation

$$x' = -\frac{1}{3}x + \sin(t) \quad \text{where} \quad x(0) = 1. \tag{7.8}$$

This has the analytic solution

$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right). \tag{7.9}$$

Make a plot of the approximate solution and the exact solution on the same plot for $t \in [0, 10]$ The partial code below should get you started.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function giving x' in terms of t and x
f = lambda t, x: ???
x0 = ???   # initial condition
t0 = ???   # initial time
tmax = ??? # final time
dt = ???   # time step (your choice, but make it small)
t, x = euler_1d(f, x0, t0, tmax, dt)
plt.plot(t, x, 'b-', label='Euler')

# Define a function giving the analytic solution
x_exact = lambda t: ???
# We will plot the exact solution at a higher resolution
t_highres = np.linspace(t0, tmax, 100)
plt.plot(t_highres, x_exact(t_highres), 'r--', label='Exact')
plt.grid()
plt.show()
```

Experiment with different values for `dt` and see how the numerical solution changes.

Exercise 7.7. The goal of this problem will be to compare the maximum error when you solve the differential equation from the previous exercise on the interval $t \in [0, 10]$ with the Euler method for various values of Δt.

Write a function that produces a plot with the value of Δt on the horizontal axis and the value of the associated absolute error on the vertical axis. You should use a log-log plot. Obviously you will need to run your code many times at many different values of Δt to build your data set. The following incomplete code will get you started.

```
# Create vector with different step sizes
dt = 10**(-np.linspace(0, 4, 50))
# Create vector with the same size as dt for holding the errors
errors = np.zeros_like(dt)
# Loop over the different step sizes to calculate the errors
for i in range(len(dt)):
    # Approximate the solution with Euler's method
    t, x = euler_1d(f, x0, t0, tmax, dt[i])
    errors[i] = ??? # Calculate maximum absolute error
```

```
# Plot the errors with logarithmic axes
plt.loglog(dt, errors)
plt.xlabel('Step size')
plt.ylabel('Maximum error')
plt.grid()
```

3. What does the plot tell you? In general, if you were to divide your value of Δt by 10, how much approximately does that reduce the error?

Exercise 7.8. Shelby solved a first order ODE $x' = f(t, x)$ using Euler's method with a step size of $dt = 0.1$ on a domain $t \in [0, 3]$. To test her code she used a differential equation where she new the exact analytic solution and she found the maximum absolute error on the interval to be 0.15. Jackson then solves the exact same differential equation, on the same interval, with the same initial condition using Euler's method and a step size of $dt = 0.01$. What would be your best estimate of Jackson's maximum absolute error?

Theorem 7.1. *Euler's method is a first order method for approximating the solution to the differential equation $x' = f(t, x)$. Hence, if the step size h of the partition of the domain were to be divided by some positive constant M then the maximum absolute error between the numerical solution and the exact solution would ???*
(Complete the last sentence.)

7.2. Higher-order equations and systems of equations

Exercise 7.9 (Harmonic Oscillator)**.** If a mass is hanging from a spring then Newton's second law, $\sum F = ma$, gives us the differential equation $mx'' = F_{restoring} + F_{damping}$ where x is the displacement of the mass from equilibrium, m is the mass of the object hanging from the spring, $F_{restoring}$ is the force pulling the mass back to equilibrium, and $F_{damping}$ is the force due to friction or air resistance that slows the mass down.

1. Which of the following is a good candidate for a restoring force in a spring? Defend your answer.

    a. $F_{restoring} = -kx$: The restoring force is proportional to the displacement away from equilibrium.

b. $F_{restoring} = -kx'$: The restoring force is proportional to the velocity of the mass.

c. $F_{restoring} = -kx''$: The restoring force is proportional to the acceleration of the mass.

2. Which of the following is a good candidate for a damping force in a spring? Defend your answer.

a. $F_{damping} = -bx$: The damping force is proportional to the displacement away from equilibrium.

b. $F_{damping} = -bx'$: The damping force is proportional to the velocity of the mass.

c. $F_{damping} = -bx''$: The damping force is proportional to the acceleration of the mass.

3. Put your answers to parts (1) and (2) together and simplify to form a second-order differential equation for position:

$$\_\_\_x'' = \_\_\_x' + \_\_\_x \tag{7.10}$$

4. If we want to solve a second order differential equation numerically we need to convert it to first order differential equations (Euler's method is only designed to deal with first order differential equations, not second order). To do so we can introduce a new variable, x_1, such that $x_1 = x'$. For the sake of notational consistency we define $x_0 = x$. The result is a system of first-order differential equations.

$$\begin{aligned} x_0' &= x_1 \\ x_1' &= \_\_\_\_\_ \end{aligned} \tag{7.11}$$

5. The code and Euler's method algorithm that we have created thus far in this chapter are only designed to work with a single differential equation instead of a system, so we need to make some modifications. We can discretize the system of differential equations using Euler's method so that

$$x' = F(t, x) \tag{7.12}$$

where F is a function that accepts a vector of inputs, plus time, and returns a vector of outputs. In the context of this particular problem,

$$F(t, x) = \begin{pmatrix} x_0' \\ x_1' \end{pmatrix} = \begin{pmatrix} x_1 \\ \_\_\_\_\_ \end{pmatrix} \tag{7.13}$$

6. We now need to discretize the derivatives in the system. As with 1D Euler's method, we will use a first-order approximation of the first derivative so that

$$\frac{x_{n+1} - x_n}{h} = F(t_n, x_n) + \mathcal{O}(h). \tag{7.14}$$

Rearranging and solving for x_{n+1} gives

$$x_{n+1} = \_\_\_\_\_ + hF(\_\_\_, \_\_\_). \tag{7.15}$$

7. The following Python function will implement Euler's method. Complete the code.

```
def euler(F, x0, t0, tmax, dt):
    """
    Solves a system of first-order ordinary differential equations
    using the Euler method.

    Parameters:
        F : function, the function defining the system of differential equations.
            It should take two arguments, the independent variable t and the
            dependent variable x (as a 1D numpy array), and return the
            derivative of x with respect to t (as a 1D numpy array).
        x0 : numpy vector, the initial values of the dependent variables.
        t0 : float, the initial value of the independent variable.
        tmax : float, the maximum value of the independent variable.
        dt : float, the time step.

    Returns:
        tuple containing two numpy arrays:
            - t : vector of time values.
            - x : array of solutions at each time value. Each column of x
                  corresponds to a different dependent variable.
    """
    # We need an integer number of steps so we round the number of steps
    # and then adjust dt to match.
    N = round((tmax - t0)/dt)
    dt = (tmax - t0)/N
    # Set up the time grid
    t = ???
    # set up an array for x with one row for each dependent variable and one column
    # for each grid point
    x = np.zeros((len(t), len(x0)))
    # store the initial condition in the first row of x
    x[0, :] = x0
    # loop over the different time steps
```

```
    for n in range(???):
        x[n+1, :] = x[???, ???] + dt * F(t[???], x[???, ???])
    return t, x
```

8. To use the `euler()` function to solve the system of equations from part (4), complete the following code. Use a mass of $m = 2$kg, a damping force of $b = 40$kg/s, and a spring constant of $k = 128$N/m. Consider an initial position of $x = 0$ (equilibrium) and an initial velocity of $x_1 = 0.6$m/s.

```
m = 2
b = 40
k = 128
F = lambda t, x: np.array([x[1], ???])
x0 = [???, ???] # initial conditions
t0 = 0
tmax = 5  # pick something reasonable here
dt = 0.01 # your choice.  pick something small
t, x = euler(F, x0, t0, tmax, dt)
```

9. Complete the following code to make a plot that shows both position and velocity versus time.

```
plt.plot(t, x[???, ???], 'b-', t, x[???, ???], 'r--')
plt.grid()
plt.title('Time Evolution of Position and Velocity')
plt.legend(['which legend entry here','which legend entry here'])
plt.xlabel('time')
plt.ylabel('position and velocity')
plt.show()
```

10. Complete the following code to make a second plot, called a phase plot, that shows position versus velocity. In a phase plot, time is implicit (not one of the axes).

```
plt.plot(x[???, ???], x[???, ???])
plt.grid()
plt.title('Phase Plot')
plt.xlabel('???')
plt.ylabel('???')
plt.show()
```

Exercise 7.10 (A Lotka-Volterra Predator-Prey Model)**.** Let $x_0(t)$ denote the number of rabbits (prey) and $x_1(t)$ denote the number of foxes (predator) at time t. The relationship between the species can be modelled by the classic 1920's Lotka-Volterra Model:

$$\begin{cases} x_0' &= \alpha x_0 - \beta x_0 x_1 \\ x_1' &= \delta x_0 x_1 - \gamma x_1 \end{cases} \tag{7.16}$$

where α, β, γ, and δ are positive constants. For this problems take $\alpha = 1$, $\beta \approx 0.1$, $\gamma = 1$, and $\delta = 0.1$.

1. First rewrite the system of ODEs in the form $x' = F(t, x)$ so you can use your `euler()` code.
2. Modify your code from the previous problem so that it works for this problem. Use `tmax = 20` and `dt = 0.05`. Start with initial conditions $x_0(0) = 10$ rabbits and $x_1(0) = 5$ foxes.
3. Create the time evolution plot. What does this plot tell you in context?
4. Create a phase plot. What does this plot tell you in context?
5. If you decrease the step size by a factor of 10, what do you see in the two plots? Why?

7.3. The Midpoint Method

Now we get to improve upon Euler's method. There is a long history of wonderful improvements to the classic Euler's method – some that work in special cases, some that resolve areas where the error is going to be high, and some that are great for general purpose numerical solutions to ODEs with relatively high accuracy. In this section we will make a simple modification to Euler's method that has a surprisingly great pay-off in the error rate.

Exercise 7.11. In Euler's method, if we are at the point t_n then we approximate the slope $x'(t_n) = f(t_n, x_n)$ and use the slope to propagate forward one time step. As you have seen, this method can lead to an overshooting of the exact solution in regions of high curvature. It would be nice to be able to look into the future and get a better approximation of the slope so that we did not miss upcoming curvature. If you could build such a method that looks in to the future, finds a slope in the future, and then uses that slope (instead of the slope from Euler's method) to advance forward in time, how far into the future would you look? Why?

Exercise 7.12. Let us return to the simple differential equation $x' = -0.5x$ with $x(0) = 6$ that we saw in Exercise 7.1. Now we will propose a slightly different method for approximating the solution.

a. At $t = 0$ we know that $x(0) = 6$. If we use the slope at time $t = 0$ to step forward in time then we will get the Euler approximation of the solution. Consider this alternative approach:

- Use the slope at time $t = 0$ and move *half* a step forward.
- Find the slope at the half-way point
- Then use the slope from the half way point to go a full step forward from time $t = 0$.

Perhaps a bit confusing …let us build this idea together:

- What is the slope at time $t = 0$? $x'(0) =$ _____
- Use this slope to step a half step forward and find the x value: $x(0.5) \approx$ _____
- Now use the differential equation to find the slope at time $t = 0.5$. $x'(0.5) =$ _____
- Now take your answer from the previous step, and go one full step forward from time $t = 0$. What x value do you end up with?
- Your answers to the previous bullets should be: $x'(0) = -3$, $x(0.5) \approx 4.5$, $x'(0.5) = -2.25$, so if we take a full step forward with slope $m = -2.25$ starting from $t = 0$ we get $x(1) \approx 3.75$.

b. Repeat the process outlined in part (a) to approximate the solution to the differential equation at times $t = 2, 3, 4$. Also record the exact answer at each of these times by noting that the exact solution is $x(t) = 6e^{-0.5t}$.

t	**0**	**1**	**2**	**3**	**4**
Euler approx of $x(t)$	6				
New approx of $x(t)$	6				
Exact value of $x(t)$	6				

c. Draw a clear picture of what this method is doing in order to approximate the slope at each individual step.

d. How does your approximation compare to the Euler approximation that you found in Exercise 7.1?

Definition 7.2 (The Midpoint Method)**.** The midpoint method is defined by first taking a half step with Euler's method to approximate a solution at time $t_{n+1/2}$. There is no grid point at $t_{n+1/2}$ so we define this as

$$t_{n+1/2} = (t_n + t_{n+1})/2.$$

We then take a full step using the value of f at $t_{n+1/2}$ and the approximate $x_{n+1/2}$.

$$\begin{aligned} x_{n+1/2} &= x_n + \frac{h}{2} f(t_n, x_n) \\ x_{n+1} &= x_n + h f(t_{n+1/2}, x_{n+1/2}) \end{aligned}$$

Exercise 7.13. As in in Exercise 7.4, consider the differential equation $x'(t) = -2x(t)/3 + 4t$ with initial condition $x(0) = 6$. By hand perform one step of the midpoint method with stepsize $h = 1$ to obtain an approximation for $x(1)$.

Exercise 7.14. Complete the code below to implement the midpoint method in one dimension.

```
def midpoint_1d(f, x0, t0, tmax, dt):
    """
    Solves a first-order ordinary differential equation using
    the midpoint method.

    Parameters:
        f    : function, the function defining the differential equation. It should
               take two arguments, the independent variable t and the dependent
               variable x, and return the derivative of x with respect to t.
        x0   : float, the initial value of the dependent variable.
        t0   : float, the initial value of the independent variable.
        tmax : float, the maximum value of the independent variable.
        dt   : float, the time step.

    Returns:
       tuple containing two numpy arrays:
            - t : vector of time values.
            - x : vector of solution values at each time value.
```

```
    """
    N = round((tmax - t0)/dt)
    dt = (tmax - t0)/N
    t = ??? # build the times
    x = ??? # build an array for the x values
    x[0] = # save the initial condition
    # On the next line: be careful about how far you're looping
    for n in range(???):
        slope = ??? # get the slope at the current point
        x_halfstep = ??? # take a half step forward
        x[n+1] = ??? # take a full step forward
    return t, x
```

Test your code on several differential equations where you know the solution (just to be sure that it is working).

```
f = lambda t, x: # your ODE right hand side goes here
x0 = # initial condition
t0 = 0
tmax = # ending time (up to you)
dt = # pick something small
t, x = midpoint_1d(???, ???, ???, ???, ???)
plt.plot(???, ???, ???)
x_exact = lambda t: # your exact solution goes here
plt.plot(???, ???, ???)
plt.legend(['Midpoint', 'Exact'])
plt.grid()
plt.show()
```

Also apply your method to the question in the feedback quiz.

Exercise 7.15. The goal in building the midpoint method was to hopefully capture some of the upcoming curvature in the solution before we overshot it. Consider the differential equation $x' = -\frac{1}{3}x + \sin(t)$ with initial condition $x(0) = 1$ on the domain $t \in [0, 10]$ as in Exercise 7.6. First get a numerical solution with Euler's method using $\Delta t = 1$. Then get a numerical solution with the midpoint method using the same value for $\Delta t = 1$. Plot the two solutions on top of each other along with the exact solution

$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right). \tag{7.17}$$

What do you observe?

Exercise 7.16. Repeat Exercise 7.7 with the midpoint method. Compare your results to what you found with Euler's method. Then use what you have discovered to answer the following, similar to Exercise 7.8:

Shelby solved a first order ODE $x' = f(t, x)$ using the midpoint method with a step size of $dt = 0.1$ on a domain $t \in [0, 3]$. To test her code she used a differential equation where she new the exact analytic solution and she found the maximum absolute error on the interval to be 0.15. Jackson then solves the exact same differential equation, on the same interval, with the same initial condition using the midpoint method and a step size of $dt = 0.01$. What would be your best estimate of Jackson's maximum absolute error?

Exercise 7.17. We have studied two methods thus far: Euler's method and the Midpoint method. In Figure 7.4 we see a graphical depiction of how each method works on the differential equation $y' = y$ with $\Delta t = 1$ and $y(0) = 1$. The exact solution at $t = 1$ is $y(1) = e^1 \approx 2.718$ and is shown in red in each figure. The methods can be summarized in the table below.

Discuss what you observe as the pros and cons of each method based on the table and on the Figure.

Euler's Method	**Midpoint Method**
1. Get the slope at time t_n	1. Get the slope at time t_n
2. Follow the slope for time Δt	2. Follow the slope for time $\Delta t/2$
	3. Get the slope at the point $t_n + \Delta t/2$
	4. Follow the new slope from time t_n for time Δt

When might you want to use Euler's method instead of the midpoint method? When might you want to use the midpoint method instead of Euler's method?

Exercise 7.18 (Midpoint Method in Several Dimensions)**.** Write a function `midpoint()` that can solve a system of first-order ordinary differential equations using the midpoint method. Base your code on the `euler()` code from Exercise 7.9 . You should only have to add one line of code and then be careful about the size of the arrays that are in play. Test your code on several problems. Compare and contrast what you see with your Euler solutions and with your Midpoint solutions.

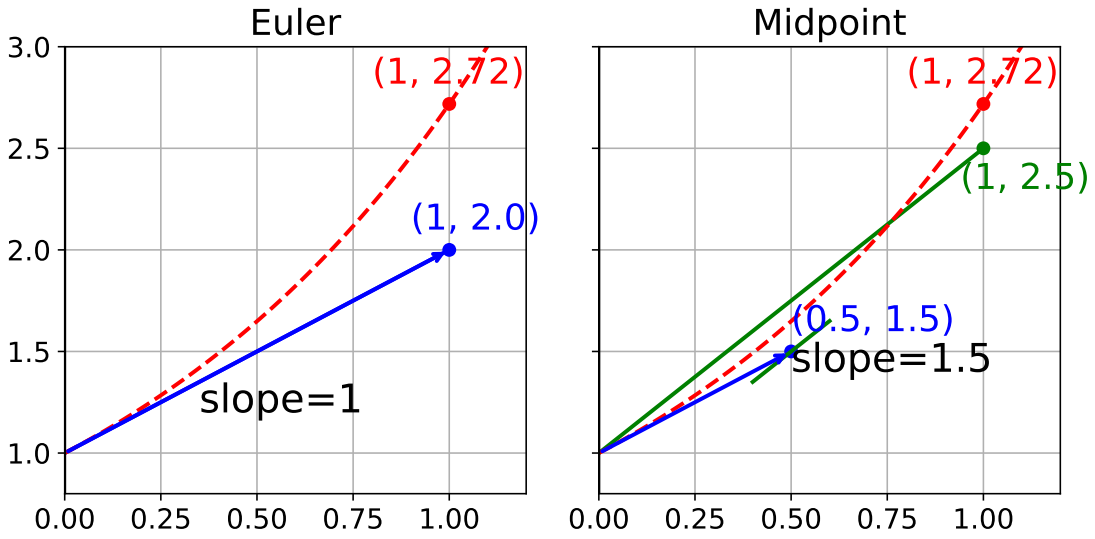

Figure 7.4.: Graphical depictions of two numerical methods. Euler (left) and Midpoint (right). The exact solution is shown in red.

7.4. Numerical Instabilities

This section is for discovery only. The theory behind what you are about to discover will be presented in a later chapter.

Exercise 7.19. Consider the differential equation $x' = -3x$ with initial condition $x(0) = 1$. The exact solution is the exponentially decaying function $x(t) = e^{-3t}$, which quickly approaches 0 as t gets larger.

a. Use Euler's method by hand to approximate $x(0.1)$, $x(0.2)$, and $x(0.3)$ using a stepsize of $\Delta t = 0.1$. Does the numerical solution appear to be decaying towards 0?

b. Now use Euler's method by hand to approximate $x(1)$, $x(2)$, and $x(3)$ using a stepsize of $\Delta t = 1$. What happens to the values of x? Does it look like the exact solution?

Exercise 7.20. The phenomenon you just discovered is called *numerical instability*. It occurs when the stepsize is too large, causing the numerical solution to oscillate or grow exponentially even when the exact solution decays.

a. Use your `euler_1d` function to solve the same differential equation $x' = -3x$ with $x(0) = 1$ on the interval $t \in [0, 10]$. Run it for different values of the stepsize Δt, for example $\Delta t \in \{0.1, 0.5, 0.7, 1.0\}$.

b. Plot the numerical solutions along with the exact solution $x(t) = e^{-3t}$ on the same graph. (You might want to limit the y-axis using `plt.ylim(-5, 5)` to see the details before it blows up completely.)

c. What is the limit on the stepsize Δt before the numerical solution completely loses its mind and grows exponentially? (Experiment with `dt` between 0.5 and 0.7 to find the exact threshold).

d. Try the same experiment using your `midpoint_1d` function. Does the midpoint method also suffer from numerical instability? If so, what is the critical stepsize for the midpoint method on this problem?

Exercise 7.21. Based on your observations in the previous exercises, what is the danger of blindly trusting a numerical solution without considering the stepsize? How might you check if your numerical solution is stable in practice if you don't know the exact solution?

7.5. Algorithm Summaries

Exercise 7.22. Consider the first-order differential equation $x' = f(t, x)$. What is Euler's method for approximating the solution to this differential equation? What is the order of accuracy of Euler's method? Explain the meaning of the order of the method in the context of solving a differential equation.

Exercise 7.23. Explain in clear language what Euler's method does geometrically.

Exercise 7.24. Consider the first-order differential equation $x' = f(t, x)$. What is the Midpoint method for approximating the solution to this differential equation? What is the order of accuracy of the Midpoint method?

Exercise 7.25. Explain in clear language what the Midpoint method does geometrically.

7.6. Truncation Errors

Many of the approximation methods in Numerical Analysis are based on Taylor series expansions and then dropping higher-order terms in the series. The error that is introduced by truncating a Taylor series in this way is called the truncation error. We have seem that in Section 5.5.

The material in this section will be covered by lectures. You will not need to have read this section before the assessment quiz for this week.

We recall from Section 5.5 Taylor's formula: for a function x that is p times differentiable,

$$x(t+h) = x(t) + hx'(t) + \frac{h^2}{2!}x''(t) + \cdots + \frac{h^{p-1}}{(p-1)!}x^{(p-1)}(t) + \frac{h^p}{p!}x^{(p)}(\xi)$$

for some ξ between t and $t+h$.

Her we consider the initial-value problem $x'(t) = f(t, x(t))$ with exact solution $x(t)$. The **local truncation error** at step n is defined as the error made in a single step starting from the exact solution, divided by the step size h:

$$\tau_n = \frac{1}{h}\left|x(t_{n+1}) - x_{n+1}\right|,$$

where x_{n+1} is the value produced by the numerical method when started from the exact value $x(t_n)$.

7.6.1. Truncation Error in Euler's Method

Euler's method produces the update

$$x_{n+1} = x_n + hf(t_n, x_n).$$

Starting from the exact solution value $x(t_n)$, the method gives

$$x_{n+1} = x(t_n) + hf(t_n, x(t_n)) = x(t_n) + hx'(t_n).$$

We compare this to the exact value at the next grid point by expanding $x(t_{n+1}) = x(t_n+h)$ in a Taylor series around t_n:

$$x(t_{n+1}) = x(t_n) + hx'(t_n) + \frac{h^2}{2}x''(\xi_n)$$

for some ξ_n between t_n and t_{n+1}. The local error in one step is therefore

$$x(t_{n+1}) - x_{n+1} = \frac{h^2}{2}x''(\xi_n),$$

so the local truncation error is

$$\tau_n = \frac{1}{h}\left|\frac{h^2}{2}x''(\xi_n)\right| = \frac{h}{2}|x''(\xi_n)|.$$

We see that the local truncation error is of order h and that Euler's method is a **first-order** approximation. This means that halving the step size halves the error.

The **global truncation error** is obtained by summing the local errors over all $N = (b-a)/h$ steps. For some ξ between a and b,

$$\tau = \left|\sum_{n=0}^{N-1}\frac{h^2}{2}x''(\xi_n)\right| \le N\frac{h^2}{2}\max_{t\in[a,b]}|x''(t)| = \frac{b-a}{2}\,h\max_{t\in[a,b]}|x''(t)|.$$

The global truncation error is also of order h, confirming that Euler's method is a first-order method.

7.6.2. Truncation Error in the Midpoint Method

The midpoint method produces the update in two stages:

$$\begin{aligned} x_{n+1/2} &= x_n + \frac{h}{2}f(t_n, x_n),\\ x_{n+1} &= x_n + h\,f\left(t_n + \tfrac{h}{2},\, x_{n+1/2}\right). \end{aligned}$$

Starting from the exact value $x(t_n)$, the half-step gives

$$x_{n+1/2} = x(t_n) + \frac{h}{2}x'(t_n).$$

We now expand f at the half-step point using a Taylor series in two variables. Since $f_t + f_x x' = x''$ (differentiating $x' = f(t, x(t))$ with respect to t), we have

$$\begin{aligned} f\left(t_n + \tfrac{h}{2},\, x_{n+1/2}\right) &= f\left(t_n + \tfrac{h}{2},\, x(t_n) + \tfrac{h}{2}x'(t_n)\right)\\ &= f(t_n, x(t_n)) + \frac{h}{2}f_t(t_n, x(t_n)) + \frac{h}{2}x'(t_n)\,f_x(t_n, x(t_n)) + \mathcal{O}(h^2)\\ &= x'(t_n) + \frac{h}{2}x''(t_n) + \mathcal{O}(h^2). \end{aligned}$$

Substituting into the full-step update gives

$$x_{n+1} = x(t_n) + h\left(x'(t_n) + \frac{h}{2}x''(t_n)\right) + \mathcal{O}(h^3) = x(t_n) + hx'(t_n) + \frac{h^2}{2}x''(t_n) + \mathcal{O}(h^3).$$

We compare this to the Taylor expansion of the exact solution:

$$x(t_{n+1}) = x(t_n) + hx'(t_n) + \frac{h^2}{2}x''(t_n) + \frac{h^3}{6}x'''(\xi_n)$$

for some ξ_n between t_n and t_{n+1}. The local error in one step is

$$x(t_{n+1}) - x_{n+1} = \frac{h^3}{6} x'''(\xi_n),$$

so the local truncation error is

$$\tau_n = \frac{1}{h}\left|\frac{h^3}{6} x'''(\xi_n)\right| = \frac{h^2}{6}|x'''(\xi_n)|.$$

We see that the local truncation error is of order h^2 and that the midpoint method is a **second-order** approximation. This means that halving the step size reduces the error by a factor of four.

The **global truncation error** is obtained by summing over all $N = (b-a)/h$ steps:

$$\tau \leq N \frac{h^3}{6} \max_{t\in[a,b]} |x'''(t)| = \frac{b-a}{6} h^2 \max_{t\in[a,b]} |x'''(t)|,$$

which is of order h^2, confirming that the midpoint method is a second-order method.

7.7. Exam-Style Question

(a) Explain what is meant by *numerical instability* when solving an ordinary differential equation using an explicit finite-difference scheme. What happens to the numerical solution, and how can it be avoided? [3 marks]

(b) Consider the ordinary differential equation $x' = -2x + t$ with initial condition $x(0) = 1$. Perform one step of Euler's method with a step size of $h = 0.5$ to find an approximation for $x(0.5)$. [2 marks]

(c) Now perform one step of the Midpoint method with a step size of $h = 0.5$ to find an approximation for $x(0.5)$ for the same initial value problem. [3 marks]

(d) Consider forming a system of first-order differential equations from the second-order autonomous differential equation for a nonlinear pendulum:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0,$$

where g and L are constants. By introducing a new variable $\omega = \frac{d\theta}{dt}$, write down the corresponding system of two first-order differential equations. [2 marks]

(e) The following incomplete Python code computes the numerical solution to the differential equation $x' = f(t, x)$ using the Midpoint method. Provide the missing code indicated by .... [3 marks]

```python
import numpy as np

def midpoint_1d(f, x0, t0, tmax, dt):
    """
    Solves a first-order ordinary differential equation using the midpoint method.
    """
    N = round((tmax - t0)/dt)
    dt = (tmax - t0)/N
    t = np.linspace(t0, tmax, N+1)
    x = np.zeros(len(t))
    x[0] = x0

    for n in range(len(t) - 1):
        slope = ...
        x_halfstep = ...
        x[n+1] = ...

    return t, x
```