8. ODE 2

8.1. Searching for a Better Method

OK. Ready for some experimentation? We are going to build a few experiments that eventually lead us to a more powerful method for finding numerical solutions to first order differential equations than the midpoint method. It is up to you to decide how much time to spend on experimentation in this section. If you are pressed for time, you can skip forward to Section 8.2.

Exercise 8.1. Let us talk about the Midpoint Method for a moment. The geometric idea of the midpoint method is outlined in the bullets below. Draw a picture along with the bullets.

- You are sitting at the point (t_n, x_n).
- The slope of the solution curve to the ODE where you are standing is

$$\text{slope at the point } (t_n, x_n) \text{ is: } m_n = f(t_n, x_n)$$

- You take a half a step forward using the slope where you are standing. The new point, denoted $x_{n+1/2}$, is given by

$$\text{location a half step forward is: } x_{n+1/2} = x_n + \frac{\Delta t}{2} m_n.$$

- Now you are standing at $(t_n + \frac{\Delta t}{2}, x_{n+1/2})$ so there is a new slope here given by

$$\text{slope after a half of an Euler step is: } m_{n+1/2} = f(t_n + \Delta t/2, x_{n+1/2}).$$

- Go back to the point (t_n, x_n) and step a full step forward using slope $m_{n+1/2}$. Hence the new approximation is

$$x_{n+1} = x_n + \Delta t \cdot m_{n+1/2}$$

Exercise 8.2. The midpoint method only uses the slope at the midpoint. Perhaps one can find a better method by taking some combination of slopes calculated at different points. Here we will build four slopes:

- $m_n = f(t_n, x_n)$: the slope at the start of the step.
- A half-step forward using m_n gives $x_{n+1/2} = x_n + \frac{\Delta t}{2} m_n$, and we evaluate the slope there:
$$m_{n+1/2} = f\big(t_n + \tfrac{\Delta t}{2},\, x_{n+1/2}\big)\,.$$
- A half-step forward using $m_{n+1/2}$ gives a second estimate of the midpoint,
$$x^*_{n+1/2} = x_n + \tfrac{\Delta t}{2}\, m_{n+1/2},$$
and we evaluate the slope there:
$$m^*_{n+1/2} = f\big(t_n + \tfrac{\Delta t}{2},\, x^*_{n+1/2}\big)\,.$$
- A full step forward using $m^*_{n+1/2}$ gives an estimate of the endpoint,
$$x^*_{n+1} = x_n + \Delta t\, m^*_{n+1/2},$$
and we evaluate the slope there:
$$m^*_{n+1} = f(t_n + \Delta t,\, x^*_{n+1})\,.$$

1. Draw a picture showing all four points where slopes are calculated.
2. Experiment with different ways to combine these four slopes as a linear combination
$$\text{estimate of slope} = c_n m_n + c_{n+1/2} m_{n+1/2} + c^*_{n+1/2} m^*_{n+1/2} + c^*_{n+1} m^*_{n+1},$$
and then advance with
$$x_{n+1} = x_n + \Delta t \cdot \text{estimate of slope}.$$
The code below implements all four slope calculations and compares your method against Euler and midpoint on the differential equation $x'(t) = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$, whose exact solution is
$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right).$$
The plots show the maximum absolute error vs. step size on a log-log scale; the slope in that plot gives the order of the method.

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

# *********
# You should copy your 1d euler and midpoint functions here.
# We will be comparing to these two existing methods.
# *********

def ode_new_method(f, x0, t0, tmax, dt):
    N = round((tmax - t0)/dt)
    dt = (tmax - t0)/N
    t = np.linspace(t0, tmax, N+1) # set up the times
    x = np.zeros(len(t))            # set up the x
    x[0] = x0                       # initial condition
    for n in range(len(t)-1):
        m_n = f(t[n], x[n])
        x_n_plus_half = x[n] + dt/2 * m_n
        m_n_plus_half = f(t[n] + dt/2, x_n_plus_half)
        x_n_plus_half_star = x[n] + dt/2 * m_n_plus_half
        m_n_plus_half_star = f(t[n] + dt/2, x_n_plus_half_star)
        x_n_plus_1_star = x[n] + dt * m_n_plus_half_star
        m_n_plus_1_star = f(t[n] + dt, x_n_plus_1_star)
        estimate_of_slope = # This is where you get to play
        x[n+1] = x[n] + dt * estimate_of_slope
    return t, x


f = lambda t, x: -(1/3.0) * x + np.sin(t)
exact = lambda t: (1/10.0)*(19*np.exp(-t/3) + \
                  3*np.sin(t) - \
                  9*np.cos(t))

x0 = 1    # initial condition
t0 = 0    # initial time
tmax = 3  # max time
dt = 2.0**(-np.arange(1, 8, 1))
err_euler = np.zeros(len(dt))
err_midpoint = np.zeros(len(dt))
err_ode_new_method = np.zeros(len(dt))
for n in range(len(dt)):
    t, xeuler = euler_1d(f, x0, t0, tmax, dt[n])
    err_euler[n] = np.max(np.abs(xeuler - exact(t)))
    t, xmidpoint = midpoint_1d(f, x0, t0, tmax, dt[n])
```

```
    err_midpoint[n] = np.max(np.abs(xmidpoint - exact(t)))
    t, xtest = ode_new_method(f, x0, t0, tmax, dt[n])
    err_ode_new_method[n] = np.max(np.abs(xtest - exact(t)))

plt.loglog(dt, err_euler, 'r*-',
           dt, err_midpoint, 'b*-',
           dt, err_ode_new_method, 'k*-')
plt.grid()
plt.legend(['euler', 'midpoint', 'new method'])
plt.show()
```

3. Work with your group to fill in the following summary table. If you tried combinations not listed below, add them.

	c_n	$c_{n+1/2}$	$c^*_{n+1/2}$	c^*_{n+1}	**Order of Error**	**Name**
1	1	0	0	0	$\mathcal{O}(\Delta t)$	Euler method
2	0	1	0	0	$\mathcal{O}(\Delta t^2)$	Midpoint Method
3	1/2	1/2	0	0		
4	1/4	2/4	0	1/4		
5	0	0	1	0		
6	0	1/2	1/2	0		
7	1/4	1/4	1/4	1/4		
8	1/5	2/5	1/5	1/5		
9	1/5	1/5	2/5	1/5		
10	1/6	2/6	2/6	1/6		
11	1/8	3/8	3/8	1/8		
12						
13						

Many of the resulting numerical ODE methods likely had the same order of accuracy, but some may have been much better or much worse. State which linear combination of slopes seems to have been the best, draw a picture of what this method does, and clearly state what the order of the error means about this method.

8.2. Runge-Kutta Method

Definition 8.1 (The Runge-Kutta 4 Method)**.** The Runge-Kutta 4 (RK4) method for approximating the solution to the differential equation $x' = f(t, x)$ approximates the

slope m at the point t_n by using the following calculations:

$$\begin{aligned} k_1 &= f(t_n, x_n) \\ k_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2) \\ k_4 &= f(t_n + h, x_n + hk_3) \\ m &= \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

It then uses that slope to advance by one time step:

$$x_{n+1} = x_n + hm = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

The order of the error in the RK4 method is $\mathcal{O}(\Delta t^4)$.

Exercise 8.3. Of course there is a price to pay for the improvement provided by the RK4 method over our earlier methods. How many evaluations of the function $f(t, x)$ do we need to make at every time step of the RK4 method? Compare this to the Euler method and the midpoint method.

Exercise 8.4. Let us step back for a second and just see what the RK4 method does from a nuts-and-bolts point of view. Consider the differential equation $x' = x$ with initial condition $x(0) = 1$. The solution to this differential equation is clearly $x(t) = e^t$. Take $\Delta t = h$ and perform 1 step of the RK4 method BY HAND to approximate the value $x(h)$. You should find that $x(h)$ is given by a polynomial in h. Does that polynomial look familiar? Is this consistent with the fact that the RK4 method is a 4th order method?

Exercise 8.5. Write a Python function that implements the Runge-Kutta 4 method in one dimension.

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

def rk4_1d(f, x0, t0, tmax, dt):
    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):
        # the interesting bits of the code go here
    return t, x
```

Test the problem on several differential equations where you know the solution.

Exercise 8.6 (RK4 Error Plot)**.** Make a log-log plot of the error in the RK4 method for a differential equation whose solution you know exactly (you could use $x' = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$ on the domain $t \in [0, 10]$, similar to what you did for the Euler method and the midpoint method earlier). Check that the error plot shows that the error is $\mathcal{O}(\Delta t^4)$.

Exercise 8.7 (RK4 in Several Dimensions)**.** Modify your Runge-Kutta 4 code to work for any number of dimensions. You may want to start from your `euler()` and `midpoint()` functions that already do this. You will only need to make minor modifications from there. Then test your new generalized RK4 method on all of the same problems which you used to test your `euler()` and `midpoint()` functions.

8.3. The Backward Euler Method

We have now built up a variety of numerical ODE solvers. All of the solvers that we have built thus far are called **explicit** numerical differential equation solvers since they try to advance the solution explicitly forward in time. Wouldn't it be nice if we could literally just say, *what slope is going to work best in the future time steps … let us use that*? Seems like an unrealistic hope, but let's see.

Definition 8.2 (Backward Euler Method)**.** We want to solve $x' = f(t, x)$ so:

- Approximate the derivative by looking forward in time(!)

$$\frac{x_{n+1} - x_n}{h} \approx f(t_{n+1}, x_{n+1})$$

- Rearrange to get the difference equation

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

- We will always know the value of t_{n+1} and we will always know the value of x_n, but we do not know the value of x_{n+1}. In fact, that is exactly what we want. The major trouble is that x_{n+1} shows up on both sides of the equation. Can you think of a way to solve for it? …you have code that does this step!!!
- This method is called the **Backward Euler** method and is known as an **implicit method** since it does not explicitly give an expression for x_{n+1} but instead it gives an equation that still needs to be solved for x_{n+1}. We will see the main advantage of implicit methods in Section 8.4.

Exercise 8.8. Let us take a few steps through the backward Euler method on a problem that we know well: $x' = -0.5x$ with $x(0) = 6$.
Let us take $h = 1$ for simplicity, so the backward Euler iteration scheme for this particular differential equation is

$$x_{n+1} = x_n - \frac{1}{2}x_{n+1}.$$

Notice that x_{n+1} shows up on both sides of the equation. A little bit of rearranging gives

$$\frac{3}{2}x_{n+1} = x_n \quad \Longrightarrow \quad x_{n+1} = \frac{2}{3}x_n.$$

1. Complete the following table.

t	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Euler Approx. of x	6	3	1.5	0.75			
Back. Euler Approx.of x	6	4	2.667	1.778			
Exact value of x	6	3.64	2.207	1.339			

2. Compare now to what we found for the midpoint method on this problem as well.

Exercise 8.9. By hand, apply the backward Euler method with stepsize $h = 1/2$ to the differential equation $x' = xt$ with initial condition $x(0) = 1$ to obtain an approximation for $x(1/2)$. Do your calculations in terms of fractions.

Example 8.1. The previous problem could potentially lead you to believe that the backward Euler method will always result in some other nice difference equation after some algebraic rearranging. That is not true! Let us consider a slightly more complicated differential equation and see what happens

$$x' = -\frac{1}{2}x^2.$$

1. Recall that the backward Euler approximation is

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

   Let us take $h = 1$ for simplicity (we will make it smaller later). What is the backward Euler formula for this particular differential equation?

2. You should notice that your backward Euler formula is now a quadratic function in x_{n+1}. That is to say, if you are given a value of x_n then you need to solve a quadratic polynomial equation to get x_{n+1}.

3. Let us take $x_0 = 6$ and $h = 1$. What is the value of x_1 using the backward Euler method?

Did you get $x_1 = \sqrt{13} - 1$? (The quadratic has a second root $-\sqrt{13} - 1$, which we discard because $x_0 > 0$ and we expect $x_1 > 0$.)

The complication with the backward Euler method is that you have a nonlinear equation to solve at every time step

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

Notice that this is the same as solving the equation

$$G(x_{n+1}) := x_{n+1} - hf(t_{n+1}, x_{n+1}) - x_n = 0.$$

You know the values of $h = \Delta t$, t_{n+1} and x_n, and you know the function f, so, in a practical sense, you should use some sort of numerical method to solve that equation – at each time step. Since $G'(x_{n+1}) = 1 - h\,\partial f/\partial x$, applying Newton's method would require computing the partial derivative of f with respect to x, which is not always available or convenient. So our next best option is to use the secant method. You can use your code from Exercise 4.20 for the function `secant_1d()`.

Exercise 8.10. Complete the function `backward_euler_1d()` below. How do you define the function `G` inside the `for` loop and what starting values do you use for the secant method?

```
import numpy as np
from scipy import optimize
def backward_euler_1d(f, x0, t0, tmax,dt):
    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):
        G = lambda y: ??? # define this function
        # give appropriate starting values for the secant method
        x[n+1] = secant_1d(G, ???)
    return t, x
```

Test your `backward_euler_1d()` function on several differential equations where you know the solution.

Exercise 8.11. You can test your `backward_euler_1d()` function on the differential equation

$$x' = -\frac{1}{3}x + \sin(t) \qquad \text{with } x(0) = 1$$

like we have for many of our past algorithm since we know that the solution is

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

The following plot lets you compare the backward Euler method to the forward Euler method and the exact solution. You just need to first define the `euler_1d()` function (which you have done in Exercise 7.5) and the `backward_euler_1d()` function (which you have just completed in Exercise 8.10).

```
import matplotlib.pyplot as plt
f = lambda t, x: -x/3+np.sin(t)
x_exact = lambda t: (19*np.exp(-t/3)+3*np.sin(t)-9*np.cos(t)) / 10
x0 = 1
t0 = 0
tmax = 4
dt = 0.1
t, x_euler = euler_1d(f,x0,t0,tmax,dt)
t, x_backward = backward_euler_1d(f,x0,t0,tmax,dt)
plt.plot(t, x_euler, 'g-.', label = "Euler")
plt.plot(t, x_backward, 'b--', label = "Backwards")
plt.plot(t, x_exact(t), 'r-', label = "Exact")
plt.legend()
```

8.4. Numerical Instabilities

Exercise 8.12. It may not be obvious at the outset, but the Backward Euler method will actually behave better than the (forward) Euler method in some sense. Let us take a look. Consider, for example, the really simple linear differential equation $x' = -10x$ with $x(0) = 1$ on the interval $t \in [0, 2]$. The analytic solution is $x(t) = e^{-10t}$. Write Python code that plots the analytic solution, the Euler approximation, and the Backward Euler approximation on top of each other. Use a time step that is larger than you normally would (such as $\Delta t = 0.25$ or $\Delta t = 0.5$ or larger). What do you notice? Why do you think this is happening?

Exercise 8.13. Consider the linear differential equation $x' = -cx$ with $c > 0$ a constant. If you solve this with the forward Euler method with a step size h, then each step can be written in the form

$$x_{n+1} = \text{???} \cdot x_n.$$

Because the exact solution to this differential equation $x(t) = e^{-ct}$ goes towards 0 as t goes to infinity, the Euler method should also go towards 0 as n goes to infinity. What is the condition on the ??? in the equation above that ensures that the Euler method has x_n going towards 0 as n goes to infinity? What condition does this impose on the step size h?

If you now solve this same ODE with the same step size with the backward Euler method, you can solve the equation for x_{n+1} exactly to find that

$$x_{n+1} = \text{???} \cdot x_n.$$

Again you would like this to go towards 0 as n goes to infinity. What condition does this impose on the step size h now?

Because the observations you made in the previous two exercises are so important, I lectured about them. Here are the lecture notes:

The stability of a numerical method is the property that, for a given step size, the numerical solution behaves qualitatively like the exact solution. In particular, if the exact solution decays to zero, the numerical solution should also decay to zero rather than grow without bound. If the numerical solution grows when the exact solution decays, we say the method is **unstable** for that step size.

This is a different concern from convergence (which asks whether the numerical solution approaches the exact solution as $h \to 0$). Stability asks: for a fixed step size h, does the numerical method respect the qualitative behaviour of the exact solution? A method can be unstable for large h even when it is perfectly accurate for small h, so choosing the step size carefully is essential.

The test equation that is always used for studying the stability of a numerical method is the linear ODE

$$x' = \lambda x.$$

The exact solution to this equation is $x(t) = x_0 e^{\lambda t}$. So if the real part of λ is negative the solution decays, and if the real part of λ is positive the solution grows.

As we will see, applying a numerical method to this equation produces a recurrence of the form

$$x_{n+1} = Q(h\lambda)\, x_n$$

for some **amplification factor** $Q(h\lambda)$, so

$$x_n = Q(h\lambda)^n\, x_0.$$

Setting $z = h\lambda$, the numerical solution decays to zero if and only if $|Q(z)| < 1$. The set of all $z \in \mathbb{C}$ for which $|Q(z)| < 1$ is called the **region of absolute stability** of the method.

Since we want the numerical solution to decay whenever the exact solution does, i.e. whenever $\mathrm{Re}(\lambda) < 0$ (equivalently $\mathrm{Re}(z) < 0$), we want the region of absolute stability to cover as much of the left half-plane as possible. Note that λ may be complex, corresponding to oscillatory decaying solutions.

Example 8.2 (Stability of the Euler Method)**.** The Euler method uses the formula $x_{n+1} = x_n + hf(t_n, x_n)$. Applying this to the test equation $x' = \lambda x$ gives

$$x_{n+1} = x_n + h\lambda x_n = (1 + h\lambda)x_n.$$

So

$$x_{n+1} = (1 + h\lambda)x_n = (1 + h\lambda)^2 x_{n-1} = \cdots = (1 + h\lambda)^{n+1} x_0.$$

Because we know the exact solution, we can calculate the absolute error that the Euler method produces:

$$E_n := |x_n - x(t_n)| = |x_0(1 + h\lambda)^n - x_0 e^{\lambda t_n}|.$$

In the case where $\lambda < 0$ we have that the exact solution decays to zero as $n \to \infty$. So the error will be determined by the behaviour of the first term:

$$\begin{aligned} \lim_{n\to\infty} E_n &= \lim_{n\to\infty} |x_0(1 + h\lambda)^n| \\ &= \begin{cases} 0 & \text{if } |1 + h\lambda| < 1 \\ \infty & \text{if } |1 + h\lambda| > 1. \end{cases} \end{aligned}$$

The error grows exponentially unless $|1 + h\lambda| < 1$. This is the stability condition for the Euler method. It requires us to choose a step size

$$h < \frac{2}{|\lambda|}$$

So the amplification factor for the Euler method is $Q(z) = 1 + z$, and its region of absolute stability is the disk $|1 + z| < 1$, centred at $z = -1$ with radius 1. For real negative λ this recovers the step-size constraint $h < 2/|\lambda|$ derived above.

Example 8.3 (Stability of the Midpoint Method)**.** The midpoint method uses the formula

$$x_{n+1} = x_n + hf(t_n + \frac{h}{2}, x_n + \frac{h}{2} f(t_n, x_n)).$$

Applying this to the test equation $x' = \lambda x$ gives

$$x_{n+1} = x_n + h\lambda \left(x_n + \frac{h}{2}\lambda x_n \right) = x_n \left(1 + h\lambda + \frac{(h\lambda)^2}{2} \right).$$

So the amplification factor is $Q(z) = 1 + z + \frac{z^2}{2}$, and the region of absolute stability is the set of $z \in \mathbb{C}$ for which

$$\left| 1 + z + \frac{z^2}{2} \right| < 1.$$

Implicit methods tend to have a larger region of absolute stability, often including the entire left half plane, in which case they are called "unconditionally stable".

Example 8.4 (Stability of the Backward Euler Method)**.** We demonstrate this in the case of the backward Euler method, which uses the formula

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

Applying this to the test equation $x' = \lambda x$ gives

$$x_{n+1} = x_n + h\lambda x_{n+1}.$$

So

$$x_{n+1} = \frac{x_n}{1 - h\lambda}.$$

The region of absolute stability is the set of $z \in \mathbb{C}$ for which

$$\left|\frac{1}{1-z}\right| < 1,$$

or, equivalently,

$$|z - 1| > 1.$$

This is almost the entire complex plane, except for the disk of radius 1 centred at $z = 1$. In particular it contains the entire left half-plane, so no upper bound on the step size is needed for stability, even when $|\lambda|$ is very large (as arises in stiff equations, discussed in Section 8.5). We therefore say that the backward Euler method is **unconditionally stable**.

8.5. Stiff Equations

A differential equation is called **stiff** if the exact solution contains a rapidly decaying transient term. This is a problem for explicit numerical methods because the stability condition forces the step size to be very small — small enough to track the transient — even after the transient has disappeared and the remaining solution varies slowly. The step-size restriction therefore persists for the entire computation, making it very slow.

An example of such a rapidly decaying term would be a term of the form $e^{-\gamma t}$ with $\gamma \gg 1$. As an example consider the ODE

$$x' = -100x + \sin t.$$

This is similar to equations we solved in Exercise 7.6, just with a larger negative coefficient in front of x. The solution to this equation is

$$x(t) = Ce^{-100t} + \frac{100}{10001}\sin t - \frac{1}{10001}\cos t.$$

The term e^{-100t} decays very rapidly. We refer to such a term in the solution as a "transient" because it becomes irrelevant after a short time. In spite of the fact that this transient term is irrelevant, it forces us to take a very small step size in order to avoid an instability if we are using an explicit method. Thus for a stiff equation it is advisable to use an implicit method instead.

Exercise 8.14. Verify by substitution that

$$x(t) = Ce^{-100t} + \frac{100}{10001}\sin t - \frac{1}{10001}\cos t$$

is indeed the general solution to $x' = -100x + \sin t$. (Recall how we found the particular solution to $x' = -\frac{1}{3}x + \sin t$ in Exercise 7.6.) Then:

a. What value of C gives the solution satisfying the initial condition $x(0) = 0$?

b. The transient Ce^{-100t} has decayed to 5% of its initial value by time t^*. Find the approximate value of t^*. You can use that $e^{-3} \approx 0.05$.

c. What is the maximum step size that forward Euler can use on this equation and still be stable?

Exercise 8.15. Investigate the stiffness of $x' = -100x + \sin t$ with $x(0) = 0$ numerically.

a. Solve the equation using forward Euler with step sizes $h = 0.1$, $h = 0.01$, and $h = 0.001$. Plot all three numerical solutions together with the exact solution on the interval $t \in [0, 1]$. What do you observe?

b. Repeat with the backward Euler method using the same step sizes. What do you observe?

c. Repeat with the RK4 method. What is the largest step size for which RK4 remains stable on this problem? (Hint: the stability region of RK4 reaches approximately $z \approx -2.8$ along the negative real axis.) Compare with what you found for forward Euler.

d. Explain in your own words why an implicit method is preferable for stiff equations.

Stiffness also arises naturally in systems of differential equations. In a system, different components of the solution can decay at very different rates. If one component decays on a timescale of $1/\gamma_{\text{fast}}$ while another decays on a timescale of $1/\gamma_{\text{slow}}$, an explicit method must use a step size dictated by γ_{fast}, even if we only care about the slow component.

Example 8.5. A simple example comes from the overdamped spring. Consider the second-order ODE

$$x'' + (\gamma + 1)x' + \gamma x = 0,$$

which describes a spring-mass system with damping. This is the system from Exercise 7.9 with $m = 1, b = \gamma + 1$ and $k = \gamma$. Writing $u = x$ and $v = x'$ converts this to the first-order system

$$\begin{pmatrix} u \\ v \end{pmatrix}' = \begin{pmatrix} 0 & 1 \\ -\gamma & -(\gamma+1) \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}.$$

The characteristic equation of the matrix is $\lambda^2 + (\gamma+1)\lambda + \gamma = (\lambda+1)(\lambda+\gamma) = 0$, so the eigenvalues are $\lambda_1 = -1$ and $\lambda_2 = -\gamma$. The general solution is

$$x(t) = C_1 e^{-t} + C_2 e^{-\gamma t}.$$

For large γ the term $e^{-\gamma t}$ is a rapidly decaying transient and the stiffness ratio is $\gamma/1 = \gamma$. An explicit method must take $h < 2/\gamma$ to remain stable, even though the long-term behaviour of x is governed by the much slower rate e^{-t}.

Exercise 8.16. Consider the system from Example 8.5 with initial conditions $u(0) = 1$ and $v(0) = x'(0) = 0$:

$$\begin{pmatrix} u \\ v \end{pmatrix}' = \begin{pmatrix} 0 & 1 \\ -\gamma & -(\gamma+1) \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}, \quad \begin{pmatrix} u(0) \\ v(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

a. Use the initial conditions $u(0) = 1$ and $v(0) = x'(0) = 0$ to find C_1 and C_2.

b. Write down the exact solution $x(t)$ for these initial conditions and plot it.

c. For $\gamma = 100$, how small must the step size h be for the forward Euler method to be stable? How does this compare to the timescale on which the solution $u(t) = x(t)$ actually changes significantly?

Exercise 8.17. Solve the stiff system from Example 8.5 numerically with $\gamma = 100$ on the interval $t \in [0, 5]$.

a. Solve the system with the forward Euler method and step sizes $h = 0.1$, $h = 0.01$, and $h = 0.001$. You can use your code from Exercise 7.9. At which step size does forward Euler first produce a stable solution?

b. Solve the system with the backward Euler method using the same step sizes. Does backward Euler remain stable even for $h = 0.1$? You can use the following function for the higher-dimensional backward Euler method:

```
import numpy as np
from scipy import optimize

def backward_euler(F,x0,t0,tmax,dt):
    N = round((tmax - t0)/dt)
    dt = (tmax - t0)/N
    t = np.linspace(t0, tmax, N+1)
    x = np.zeros((len(t), len(x0)))
    x[0,:] = x0
    for n in range(len(t)-1):
        # G takes a vector y and returns a vector
        G = lambda y: y - dt*F(t[n+1],y) - x[n,:]
        # fsolve with initial guess as the previous time step's vector
        x[n+1,:] = optimize.fsolve(G, x[n,:])
    return t, x
```

c. By what factor can backward Euler take larger steps than forward Euler and still be stable? How does this factor relate to γ?

> ⚠ Warning
>
> Note that while Backward Euler is stable for large h, it is still only a first-order method. Large steps might be stable but could be very inaccurate!

8.6. Algorithm Summaries

Exercise 8.18. Consider the first-order differential equation $x' = f(t, x)$. What is the Runge Kutta 4 method for approximating the solution to this differential equation? What is the order of accuracy of the Runge Kutta 4 method?

Exercise 8.19. Explain in clear language what the Runge Kutta 4 method does geometrically.

Exercise 8.20. Consider the first-order differential equation $x' = f(t, x)$. What is the Backward Euler method for approximating the solution to this differential equation? What is the order of accuracy of the backward Euler method? What is the region of absolute stability for the backward Euler method?

Exercise 8.21. Explain in clear language what the Backward Euler method does geometrically.

8.7. Exam-Style Question

(a) Explain geometrically the difference between the explicit (forward) Euler method and the implicit backward Euler method. What is the fundamental property of backward Euler that makes it more suited for stiff equations? [3 marks]

(b) Show that the backward Euler method is unconditionally stable for the linear test equation $x' = \lambda x$ where $\mathrm{Re}(\lambda) < 0$.

(c) Show that the backward Euler method has a local truncation error of order $O(h)$.

(d) State, but don't derive, the order of accuracy for the Runge-Kutta 4 (RK4) method. How many evaluations of the function $f(t, x)$ are required at each time step? [2 marks]
