1 Introduction to ODE Initial Value Problems

In this chapter, we begin our exploration of numerical methods for solving ordinary differential equations (ODEs). We will focus on initial value problems (IVPs).

Definition 1.1 (ODE Initial Value Problem)**.** An ODE initial value problem is generally expressed in the form:

$$y'(t) = f(t, y(t))$$

subject to an initial condition:

$$y(0) = y_0$$

Here, $y(t)$ is the unknown function we want to find. $y(t)$ can be a scalar function or an n-dimensional vector function $y(t) = (y_1(t), y_2(t), \ldots, y_n(t))^T$. In the vector case, the equation represents a system of n coupled first-order ODEs:

$$\begin{aligned} y_1'(t) &= f_1(t, y_1, y_2, \ldots, y_n) \\ y_2'(t) &= f_2(t, y_1, y_2, \ldots, y_n) \\ &\vdots \\ y_n'(t) &= f_n(t, y_1, y_2, \ldots, y_n) \end{aligned}$$

The condition $y(0) = y_0$ specifies the state of the system at the initial time $t = 0$.

Definition 1.2 (Order of an ODE)**.** The order of an ODE is determined by the highest-order derivative present in the equation. For the standard form $y'(t) = f(t, y)$, the order is 1.

Numerically, we primarily focus on methods for first-order ODEs. This is sufficient because higher-order ODEs can be converted into a system of first-order ODEs by introducing auxiliary variables.

Example 1.3 (Reduction of Order: Newton's Second Law)**.** Consider Newton's second law of motion for a particle of mass m:

$$y''(t) = \frac{F(t, y(t), y'(t))}{m}$$

This is a second-order ODE. We might have initial conditions for position and velocity:

$$y(0) = y_0, \quad y'(0) = v_0$$

To convert this into a first-order system, we introduce a new variable for velocity, $v(t) = y'(t)$. Then the system becomes:

$$\begin{aligned} v'(t) &= \frac{F(t, y(t), v(t))}{m} \\ y'(t) &= v(t) \end{aligned}$$

This is a system of two first-order ODEs for the variables $y(t)$ and $v(t)$, with initial conditions $y(0) = y_0$ and $v(0) = v_0$. A numerical method capable of solving first-order systems can now be applied.

Example 1.4 (Lotka-Volterra Model)**.** The Lotka-Volterra equations are a classic example of a non-linear system of first-order ODEs used to model predator-prey dynamics. Let $y_1(t)$ be the population of the prey (e.g., rabbits) and $y_2(t)$ be the population of the predator (e.g., foxes). The model is:

$$\begin{aligned} y_1'(t) &= \alpha_1 y_1 - \beta_1 y_1 y_2 \\ y_2'(t) &= \beta_2 y_1 y_2 - \alpha_2 y_2 \end{aligned}$$

Here, $\alpha_1, \beta_1, \alpha_2, \beta_2$ are positive parameters:

- $\alpha_1 y_1$: Exponential growth rate of prey in the absence of predators.
- $-\beta_1 y_1 y_2$: Decrease in prey population due to predation, proportional to the rate of encounters between predators and prey.
- $\beta_2 y_1 y_2$: Growth of predator population due to consumption of prey.
- $-\alpha_2 y_2$: Exponential decay rate of predators in the absence of prey.

We can solve this system numerically using Python libraries like SciPy.

Listing 1: Python code (`l_v.py`) for solving the Lotka-Volterra equations using `scipy.integrate.odeint`.

```
#!/usr/bin/python3
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Constants in model
alpha1=1.2
beta1=0.6
alpha2=0.8
beta2=0.3

# Function that evaluates the RHS of the ODE. It has two components,
# representing the changes in prey and predator populations.
def f(y,t):
    return np.array([alpha1*y[0]-beta1*y[0]*y[1], \
           -alpha2*y[1]+beta2*y[0]*y[1]])

# Specify the range of time values where the ODE should be solved at
time=np.linspace(0,70,500)

# Initial conditions, set to the initial populations of prey and predators
yinit=np.array([1.0, 1.0])

# Solve ODE using the "odeint" library in SciPy
y=odeint(f,yinit,time)

# Plot the solutions
plt.figure()
p1,=plt.plot(time,y[:,0])
p2,=plt.plot(time,y[:,1])
plt.legend([p1,p2],["Prey","Predators"])
plt.xlabel('t')
plt.ylabel('Population')
# plt.show() # Removed for automated execution
```

The `odeint` function from `scipy.integrate` provides a convenient way to solve ODE initial value problems. It takes the function defining the right-hand side of the ODE system, the initial conditions, and the time points where the solution is desired. It returns an array containing the solution $y(t)$ at the requested time points.

The plot shows the characteristic oscillations of the Lotka-Volterra model. The predator and prey populations exhibit cyclical behavior, where peaks in the predator population lag behind peaks in the prey population.

Python's `scipy.integrate` module offers powerful tools like `odeint` (based on LSODA from the FORTRAN library ODEPACK) and the more flexible `ode` class. MATLAB also provides excellent ODE solvers, with `ode45` (based on an explicit Runge-Kutta (4,5) formula) being very popular. These solvers often employ sophisticated techniques like adaptive time-stepping, where the step size h is adjusted automatically during integration to meet accuracy requirements efficiently.

2 One-Step Integration Methods

We now turn to developing our own numerical methods for solving $y'(t) = f(t, y)$. The goal is to generate a sequence of approximations y_k to the true solution $y(t_k)$ at discrete time points $t_k = k \cdot h$, where h is the step size, starting from $y_0 = y(t_0)$.

Definition 2.1 (One-Step Methods)**.** One-step methods compute the approximation y_{k+1} at time t_{k+1} using only the information from the previous step, y_k at time t_k. They have the general form:

$$y_{k+1} = y_k + h \cdot \phi(t_k, y_k, t_{k+1}, y_{k+1}, h)$$

where ϕ is an increment function that depends on the method.

2.1 Euler's Method (Forward Euler)

Euler's method is the simplest one-step method. It can be derived in two ways:

2.1.1 Derivation via Finite Difference

Approximate the derivative $y'(t_k)$ using a forward finite difference:

$$y'(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{h}$$

Substituting this into the ODE $y'(t_k) = f(t_k, y(t_k))$ and replacing the true solution $y(t_k)$ with its approximation y_k, we get:

$$\frac{y_{k+1} - y_k}{h} = f(t_k, y_k)$$

Rearranging gives the Forward Euler formula:

$$y_{k+1} = y_k + hf(t_k, y_k) \tag{1}$$

2.1.2 Derivation via Quadrature

Integrate the ODE $y'(s) = f(s, y(s))$ from t_k to t_{k+1}:

$$\int_{t_k}^{t_{k+1}} y'(s)ds = \int_{t_k}^{t_{k+1}} f(s, y(s))ds$$

By the Fundamental Theorem of Calculus, the left side is $y(t_{k+1}) - y(t_k)$. Thus:

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))ds$$

We can approximate the integral using a numerical quadrature rule. The simplest rule is the $n = 0$ Newton-Cotes formula (the rectangle rule) using the left endpoint t_k:

$$\int_{t_k}^{t_{k+1}} f(s, y(s))ds \approx (t_{k+1} - t_k)f(t_k, y(t_k)) = hf(t_k, y(t_k))$$

Substituting this approximation and replacing $y(t_k)$ with y_k gives the Forward Euler formula again:

$$y_{k+1} = y_k + hf(t_k, y_k)$$

Example 2.2 (Python Implementation of Euler's Method)**.** Let's apply Euler's method to the test equation $y' = \lambda y$ with $y(0) = 1$. The exact solution is $y(t) = e^{\lambda t}$. The Euler step is $y_{k+1} = y_k + h(\lambda y_k) = (1 + h\lambda)y_k$.

Listing 2: Python code (`euler.py`) implementing Forward Euler for $y' = \lambda y$.

```
#!/usr/bin/python3
from math import exp

# Initial variables and constants
y=1
t=0
h=0.1
lam=0.5

# Apply Euler step until t=2:
while t<=2:

    # Analytical solution
    yexact=exp(lam*t)
    print(t,y,yexact,y-yexact)

    # Euler step
    y=y+h*(lam*y)

    # Update time
    t=t+h
```

This code iteratively applies the Euler step, calculating the numerical solution 'y' and comparing it to the exact solution 'yexact' at each time step. The output prints the time, numerical solution, exact solution, and the error.

Let's visualize the results for $h = 0.1$ and $h = 0.05$.

The plots show that the numerical solution approximates the exact solution. As the step size h is decreased (from 0.1 to 0.05), the numerical solution becomes visibly closer to the exact solution. We will later analyze this convergence formally.

2.2 Backward Euler Method

If we use the $n = 0$ Newton-Cotes quadrature rule but evaluate the function at the right endpoint t_{k+1} instead of the left, we get:

$$\int_{t_k}^{t_{k+1}} f(s, y(s))ds \approx hf(t_{k+1}, y(t_{k+1}))$$

Substituting this into the integrated ODE form gives the Backward Euler formula:

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}) \tag{2}$$

2.3 Explicit vs. Implicit Methods

The Forward Euler method (1) is an **explicit** method. Given y_k, we can directly calculate y_{k+1} because y_{k+1} appears only on the left-hand side.

The Backward Euler method (2) is an **implicit** method. The unknown y_{k+1} appears on both sides of the equation (inside the function f). To find y_{k+1}, we generally need to solve an equation at each time step.

For example, applying Backward Euler to $y' = \sin(ty)$:

$$y_{k+1} = y_k + h\sin(t_{k+1}y_{k+1})$$

To find y_{k+1}, we need to solve the equation $F(y_{k+1}) = 0$, where $F(y_{k+1}) = y_{k+1} - y_k - h\sin(t_{k+1}y_{k+1})$. This is typically a non-linear equation that can be solved using root-finding methods like Newton's method.

Implicit methods require more computational work per step than explicit methods. However, as we will see later, they often have superior stability properties, which can allow for much larger time steps h, especially for certain types of problems (stiff ODEs).

2.4 Trapezoid Method

If we use the $n = 1$ Newton-Cotes quadrature rule (the trapezoid rule) to approximate the integral:

$$\int_{t_k}^{t_{k+1}} f(s, y(s))ds \approx \frac{h}{2}[f(t_k, y(t_k)) + f(t_{k+1}, y(t_{k+1}))]$$

This leads to the Trapezoid Method:

$$y_{k+1} = y_k + \frac{h}{2}[f(t_k, y_k) + f(t_{k+1}, y_{k+1})] \tag{3}$$

Like Backward Euler, the Trapezoid method is implicit, as y_{k+1} appears on both sides.

These three methods (Forward Euler, Backward Euler, Trapezoid) are fundamental examples of one-step methods. They form the basis for understanding more complex integration schemes.

3 ODE Convergence

A crucial property of any numerical method is convergence: does the numerical solution y_k approach the true solution $y(t_k)$ as the step size h tends to zero? Without convergence, a numerical method is unreliable.

Analyzing convergence for ODEs is subtle because errors made at one step can propagate and affect subsequent steps. We need to distinguish between the error accumulated over many steps (global error) and the error introduced in a single step (local or truncation error).

Definition 3.1 (Global Error)**.** The global error at step k is the difference between the true solution and the numerical approximation at time t_k:

$$e_k = y(t_k) - y_k$$

Definition 3.2 (Truncation Error (for explicit one-step methods))**.** The (local) truncation error T_k at step k measures how well the true solution satisfies the numerical formula, scaled by h. For an explicit one-step method $y_{k+1} = y_k + h\phi(t_k, y_k, h)$, the truncation error is defined by substituting the true solution $y(t)$ into the formula:

$$T_k = \frac{y(t_{k+1}) - y(t_k)}{h} - \phi(t_k, y(t_k), h)$$

This can be rearranged as:

$$y(t_{k+1}) = y(t_k) + h\phi(t_k, y(t_k), h) + hT_k$$

hT_k represents the error introduced in a single step if we start the step exactly on the true solution ($y_k = y(t_k)$). The global error e_k accumulates from these local truncation errors $T_0, T_1, \ldots, T_{k-1}$.

3.1 Convergence Theorem for Euler's Method

To prove convergence, we often need an assumption about the function $f(t, y)$ defining the ODE.

Definition 3.3 (Lipschitz Condition)**.** A function $f(t, y)$ satisfies a Lipschitz condition with respect to y on an interval $[a, b]$ if there exists a constant $L_f \geq 0$ (the Lipschitz constant) such that for all $t \in [a, b]$ and all relevant u, v:

$$\|f(t, u) - f(t, v)\| \leq L_f \|u - v\|$$

This condition essentially bounds how quickly the function f can change as y changes. It limits the "steepness" of f with respect to y.

Theorem 3.4 (Convergence of Euler's Method)**.** Suppose we apply Euler's method $y_{k+1} = y_k + hf(t_k, y_k)$ to solve $y' = f(t, y)$, $y(0) = y_0$ up to time $T = Mh$. If $f(t, y)$ satisfies a Lipschitz condition with constant L_f with respect to y, and the truncation error at step j is T_j, then the global error $e_k = y(t_k) - y_k$ is bounded by:

$$\|e_k\| \leq \frac{e^{L_f t_k} - 1}{L_f} \max_{0 \leq j \leq k-1} \|T_j\|$$

for $k = 0, 1, \ldots, M$. (If $L_f = 0$, the bound is $\|e_k\| \leq t_k \max \|T_j\|$).

Preuve. We start with the definitions of global error and truncation error. The numerical scheme is $y_{k+1} = y_k + hf(t_k, y_k)$. The true solution satisfies $y(t_{k+1}) = y(t_k) + hf(t_k, y(t_k)) + hT_k$.

Subtracting the numerical scheme from the true solution equation:

$$y(t_{k+1}) - y_{k+1} = (y(t_k) - y_k) + h(f(t_k, y(t_k)) - f(t_k, y_k)) + hT_k$$
$$e_{k+1} = e_k + h(f(t_k, y(t_k)) - f(t_k, y_k)) + hT_k$$

Take norms and apply the triangle inequality:

$$\|e_{k+1}\| \leq \|e_k\| + h\|f(t_k, y(t_k)) - f(t_k, y_k)\| + h\|T_k\|$$

Apply the Lipschitz condition $\|f(t_k, y(t_k)) - f(t_k, y_k)\| \leq L_f \|y(t_k) - y_k\| = L_f \|e_k\|$:

$$\|e_{k+1}\| \leq \|e_k\| + hL_f \|e_k\| + h\|T_k\|$$

$$\|e_{k+1}\| \leq (1 + hL_f)\|e_k\| + h\|T_k\|$$

Let $T_{max} = \max_{0 \leq j \leq k-1} \|T_j\|$. Then $\|e_{k+1}\| \leq (1 + hL_f)\|e_k\| + hT_{max}$. This is a difference inequality. We can unroll it, starting from $\|e_0\| = \|y(t_0) - y_0\| = 0$:

$$\|e_1\| \leq (1 + hL_f)\|e_0\| + h\|T_0\| = h\|T_0\| \leq hT_{max}$$
$$\|e_2\| \leq (1 + hL_f)\|e_1\| + h\|T_1\| \leq (1 + hL_f)(hT_{max}) + hT_{max}$$
$$\|e_k\| \leq hT_{max} \sum_{j=0}^{k-1} (1 + hL_f)^j$$

This is a geometric series sum with $r = 1 + hL_f$. The sum is $\frac{r^k - 1}{r - 1} = \frac{(1+hL_f)^k - 1}{(1+hL_f) - 1} = \frac{(1+hL_f)^k - 1}{hL_f}$. So,

$$\|e_k\| \leq hT_{max} \frac{(1 + hL_f)^k - 1}{hL_f} = \frac{T_{max}}{L_f}[(1 + hL_f)^k - 1]$$

We use the inequality $1 + x \leq e^x$ for $x \geq 0$. Let $x = hL_f$. Then $1 + hL_f \leq e^{hL_f}$. Therefore, $(1 + hL_f)^k \leq (e^{hL_f})^k = e^{khL_f} = e^{L_f t_k}$. Substituting this back gives the final bound:

$$\|e_k\| \leq \frac{T_{max}}{L_f}(e^{L_f t_k} - 1)$$

□

3.1.1 Examples of Lipschitz Conditions

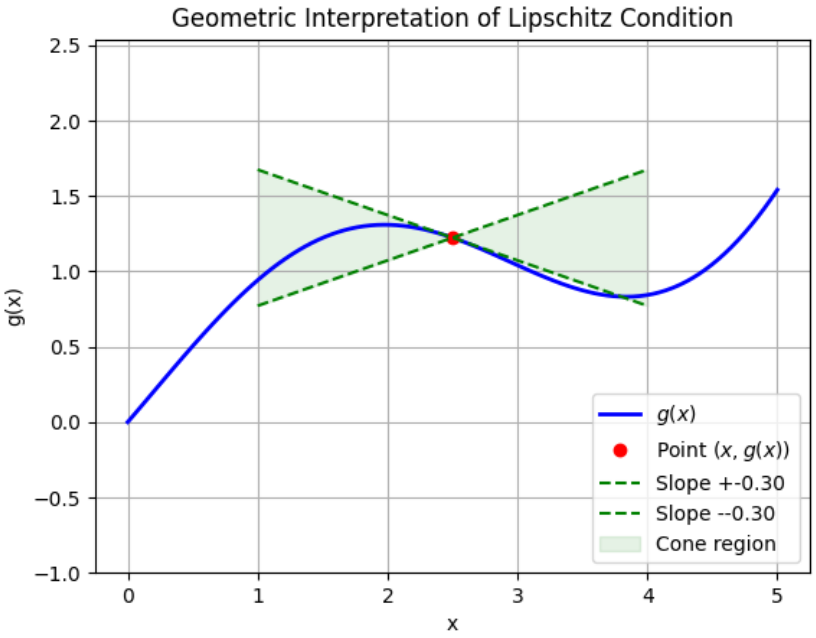

Figure 4: Geometric view of the Lipschitz condition. The graph of the function $g(x)$ must lie entirely within the "cone" formed by lines with slopes $\pm L_f$ emanating from any point $(x, g(x))$ on the curve.

Example 3.5. If $g(x)$ is continuously differentiable on $[a, b]$, then by the Mean Value Theorem, for any $x, y \in [a, b]$, there exists θ between x and y such that $g(x) - g(y) = g'(\theta)(x - y)$. Therefore, $|g(x) - g(y)| = |g'(\theta)||x - y|$. We can choose $L_f = \max_{\theta \in [a,b]} |g'(\theta)|$. If g' is bounded, g is Lipschitz.

Example 3.6. $g(x) = |x|$. This function is not differentiable at $x = 0$. However, by the reverse triangle inequality, $||x| - |y|| \leq |x - y|$. Thus, $g(x) = |x|$ is Lipschitz continuous with $L_f = 1$.

Example 3.7. $g(x) = \sqrt{x}$ on $[0, 1]$. Consider $y = 0$. Then $|g(x) - g(0)| = |\sqrt{x}|$. We need $|\sqrt{x}| \leq L_f|x - 0| = L_f|x|$. This requires $\frac{|\sqrt{x}|}{|x|} = \frac{1}{\sqrt{x}} \leq L_f$. However, as $x \to 0^+$, $1/\sqrt{x} \to \infty$. No finite L_f can satisfy this for all $x \in (0, 1]$. Therefore, $\sqrt{x}$ is not Lipschitz continuous on $[0, 1]$. Graphically, its slope becomes infinite at $x = 0$, violating the finite-slope cone condition.

3.2 Order of Accuracy

The convergence theorem shows that if the truncation error T_j goes to zero as $h \to 0$, then the global error e_k also goes to zero. The rate at which T_j approaches zero determines the rate of convergence.

Definition 3.8 (Order of Accuracy)**.** A one-step method has order of accuracy p if its local truncation error satisfies $T_k = O(h^p)$ as $h \to 0$.

For a method of order $p \geq 1$, the convergence theorem implies that the global error e_k is also $O(h^p)$.

Example 3.9 (Order of Euler Methods)**.** *Forward Euler:* The truncation error is $T_k = \frac{y(t_{k+1})-y(t_k)}{h} - f(t_k, y(t_k))$. Since $f(t_k, y(t_k)) = y'(t_k)$, we have $T_k = \frac{y(t_{k+1})-y(t_k)}{h} - y'(t_k)$. By Taylor expansion, $y(t_{k+1}) = y(t_k) + hy'(t_k) + \frac{h^2}{2}y''(\theta_k)$ for some $\theta_k \in (t_k, t_{k+1})$. Substituting this gives:

$$T_k = \frac{(y(t_k) + hy'(t_k) + \frac{h^2}{2}y''(\theta_k)) - y(t_k)}{h} - y'(t_k) = \frac{h}{2}y''(\theta_k)$$

Assuming y'' is bounded, $T_k = O(h)$. Forward Euler is a first-order accurate method ($p = 1$).

Backward Euler: The truncation error is $T_k = \frac{y(t_{k+1})-y(t_k)}{h} - f(t_{k+1}, y(t_{k+1})) = \frac{y(t_{k+1})-y(t_k)}{h} - y'(t_{k+1})$. Taylor expanding $y(t_k)$ around t_{k+1}: $y(t_k) = y(t_{k+1}) - hy'(t_{k+1}) + \frac{h^2}{2}y''(\psi_k)$ for some $\psi_k \in (t_k, t_{k+1})$. Rearranging gives $\frac{y(t_{k+1})-y(t_k)}{h} = y'(t_{k+1}) - \frac{h}{2}y''(\psi_k)$.

$$T_k = (y'(t_{k+1}) - \frac{h}{2}y''(\psi_k)) - y'(t_{k+1}) = -\frac{h}{2}y''(\psi_k)$$

Assuming y'' is bounded, $T_k = O(h)$. Backward Euler is also a first-order accurate method ($p = 1$).

Example 3.10 (Order of Trapezoid Method)**.** The truncation error is $T_k = \frac{y(t_{k+1})-y(t_k)}{h} - \frac{1}{2}[f(t_k, y(t_k)) + f(t_{k+1}, y(t_{k+1}))]$. Replacing f with y':

$$T_k = \frac{1}{h}\int_{t_k}^{t_{k+1}} y'(s)ds - \frac{1}{2}[y'(t_k) + y'(t_{k+1})]$$

This expression is precisely the error of the trapezoid rule quadrature applied to the function $y'(s)$ over the interval $[t_k, t_{k+1}]$, divided by h. The error of the trapezoid rule for an integral $\int_a^b g(s)ds$ is known to be $-\frac{(b-a)^3}{12}g''(\xi)$ for some $\xi \in (a, b)$. Here $b - a = h$ and $g = y'$. So the integral error is $-\frac{h^3}{12}y'''(\xi)$.

$$T_k = \frac{1}{h}\left(-\frac{h^3}{12}y'''(\xi)\right) = -\frac{h^2}{12}y'''(\xi)$$

Assuming y''' is bounded, $T_k = O(h^2)$. The Trapezoid method is a second-order accurate method ($p = 2$).

Example 3.11 (Verification of Order)**.** Consider $y' = y$, $y(0) = 1$. We compute the global error at $t = 1$ using Forward Euler and Trapezoid methods for decreasing step sizes h.

As seen in Table **??**, when we halve the step size h:

- The error for Forward Euler (E_{Euler}) is roughly halved. This is consistent with $O(h^1)$ accuracy ($p = 1$).
- The error for the Trapezoid method (E_{Trap}) is roughly divided by four. This is consistent with $O(h^2)$ accuracy ($p = 2$).

This numerical experiment supports the theoretical orders of accuracy. Higher-order methods generally provide much better accuracy for a given step size, or allow larger step sizes for a given accuracy target.

4 ODE Stability

While convergence analysis tells us what happens as $h \to 0$, stability analysis concerns the behavior of the numerical solution for a fixed, non-zero step size h. We want methods that produce bounded, non-growing solutions when the true solution of the ODE is itself bounded or decaying. Stability is crucial for practical computation, as we cannot take infinitely small steps. Ideally, the stability properties of the numerical method should mimic those of the underlying ODE.

4.1 Mathematical ODE Stability

First, let's define stability for the ODE itself, $y' = f(t, y)$. Consider two solutions, $y(t)$ and $\hat{y}(t)$, originating from slightly different initial conditions $y(0) = y_0$ and $\hat{y}(0) = \hat{y}_0$.

Definition 4.1 (Stability of an ODE)**.** The ODE solution is stable if small changes in the initial condition lead to only small changes in the solution for all future times. Formally: for every $\epsilon > 0$, there exists a $\delta > 0$ such that if $\|y_0 - \hat{y}_0\| < \delta$, then $\|y(t) - \hat{y}(t)\| < \epsilon$ for all $t \geq 0$.

Definition 4.2 (Asymptotic Stability of an ODE)**.** The ODE solution is asymptotically stable if it is stable, and additionally, the difference between solutions eventually vanishes: $\|y(t) - \hat{y}(t)\| \to 0$ as $t \to \infty$.

Remark 4.3. In the context of ODEs and PDEs, the term "stability" often refers to the well-posedness or sensitivity of the mathematical problem itself, as defined above, as well as the behavior of the numerical method. This differs slightly from other areas of numerical analysis where "stability" usually refers only to the numerical algorithm's sensitivity to perturbations (like rounding errors), distinct from the problem's "conditioning".

Example 4.4 (Stability of $y' = \lambda y$)**.** For the scalar equation $y' = \lambda y$, the solution is $y(t) = y_0 e^{\lambda t}$. The difference between two solutions is $y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$. Let's consider $\lambda \in \mathbb{C}$, $\lambda = a + ib$. Then $e^{\lambda t} = e^{at}e^{ibt} = e^{at}(\cos(bt) + i\sin(bt))$. The magnitude is $|e^{\lambda t}| = |e^{at}||e^{ibt}| = e^{at} \cdot 1 = e^{at}$. The stability depends only on the real part of λ, $a = \mathrm{Re}(\lambda)$.

- If $\mathrm{Re}(\lambda) < 0$ ($a < 0$): Then $e^{at} \to 0$ as $t \to \infty$. The difference decays. The ODE is asymptotically stable.
- If $\mathrm{Re}(\lambda) = 0$ ($a = 0$): Then $e^{at} = 1$. The difference $|y(t) - \hat{y}(t)| = |y_0 - \hat{y}_0|$ remains constant. The ODE is stable (but not asymptotically stable).
- If $\mathrm{Re}(\lambda) > 0$ ($a > 0$): Then $e^{at} \to \infty$ as $t \to \infty$. The difference grows exponentially. The ODE is unstable.

Plots for real λ:

Example 4.5 (Stability of $y' = Ay$)**.** Consider the linear system $y'(t) = Ay(t)$, where $y \in \mathbb{R}^n$ and A is an $n \times n$ matrix. If A is diagonalizable, $A = V\Lambda V^{-1}$, where Λ is a diagonal matrix of eigenvalues λ_i and V is the matrix of corresponding eigenvectors. Let $z(t) = V^{-1}y(t)$. Then $z'(t) = V^{-1}y'(t) = V^{-1}Ay(t) = V^{-1}(V\Lambda V^{-1})y(t) = \Lambda(V^{-1}y(t)) = \Lambda z(t)$. This transforms the system into n decoupled scalar equations: $z_i'(t) = \lambda_i z_i(t)$. The stability of the original system $y' = Ay$ is governed by the stability of these scalar equations. Assuming the eigenvector matrix V is well-conditioned, the system $y' = Ay$ is stable if and only if all eigenvalues λ_i satisfy $\mathrm{Re}(\lambda_i) \leq 0$, and asymptotically stable if $\mathrm{Re}(\lambda_i) < 0$ for all i.

4.2 Numerical Stability

We now define stability for a numerical method applied to an ODE.

Definition 4.6 (Stability of a Numerical Method)**.** Let y_k and $\hat{y}_k$ be two numerical solutions generated by a method, starting from initial conditions y_0 and $\hat{y}_0$. The method is stable if small perturbations in the initial condition lead to small perturbations in the numerical solution for all steps k. Formally: for every $\epsilon > 0$, there exists a $\delta > 0$ such that if $\|y_0 - \hat{y}_0\| < \delta$, then $\|y_k - \hat{y}_k\| < \epsilon$ for all $k \geq 0$.

The goal is for the numerical method to be stable whenever the underlying ODE is stable. Since ODE stability is problem-dependent, we analyze numerical stability using a standard test problem.

Definition 4.7 (Dahlquist Test Equation)**.** The standard test equation for analyzing numerical stability is the scalar linear ODE:

$$y' = \lambda y, \quad \lambda \in \mathbb{C}$$

The behavior of a numerical method on this simple equation often reveals its stability properties for more general problems. We want the numerical method to be stable for all λ where the ODE is stable, i.e., for $\mathrm{Re}(\lambda) \leq 0$.

4.3 Stability Analysis of Forward Euler

Applying Forward Euler $y_{k+1} = y_k + hf(t_k, y_k)$ to the test equation $y' = \lambda y$ gives:

$$y_{k+1} = y_k + h(\lambda y_k) = (1 + h\lambda)y_k$$

Iterating this gives $y_k = (1+h\lambda)^k y_0$. The term $G(h\lambda) = 1 + h\lambda$ is called the **amplification factor**. For the numerical solution y_k to remain bounded or decay (mimicking stable ODEs where $\mathrm{Re}(\lambda) \leq 0$), we require the magnitude of the amplification factor to be less than or equal to one:

$$|G(h\lambda)| = |1 + h\lambda| \leq 1$$

Let $\bar{h} = h\lambda$. We need $|1 + \bar{h}| \leq 1$. Let $\bar{h} = a + ib$. Then $|1 + a + ib| \leq 1$, which means $\sqrt{(1+a)^2 + b^2} \leq 1$, or $(1 + a)^2 + b^2 \leq 1$. This inequality describes the interior and boundary of a disk in the complex $\bar{h}$-plane with radius 1 centered at $(-1, 0)$.

The stability region for Forward Euler (Figure 9) only covers a part of the left half-plane where the ODE is stable. This means that even if $\mathrm{Re}(\lambda) < 0$ (stable ODE), the Forward Euler method might be unstable if h is too large, causing $\bar{h} = h\lambda$ to fall outside the green disk. Therefore, Forward Euler is called **conditionally stable**.

For λ real and negative ($\lambda < 0$), the stability condition $|1 + h\lambda| \leq 1$ becomes $-1 \leq 1 + h\lambda \leq 1$, which simplifies to $-2 \leq h\lambda \leq 0$. Since $h > 0$ and $\lambda < 0$, the right inequality $h\lambda \leq 0$ is always satisfied. The left inequality $-2 \leq h\lambda$ implies $h \leq -2/\lambda$. There is an upper bound on the step size h for stability. The more negative λ is (faster decay in the true solution), the smaller h must be.

- If $\lambda = -10$, need $h \leq -2/(-10) = 0.2$.
- If $\lambda = -200$, need $h \leq -2/(-200) = 0.01$.

Example 4.8 (Instability of Forward Euler)**.** Let's use $h = 0.1$ and examine the behavior for different negative λ. The stability limit is $h \leq -2/\lambda$, or $\lambda \geq -2/h = -2/0.1 = -20$.

Listing 3: Python code (`e_stab.py`) to demonstrate stability/instability of Forward Euler.

```
#!/usr/bin/python3
from math import exp
import numpy as np # Added for linspace if used for plotting later
import matplotlib.pyplot as plt # Added for plotting
```

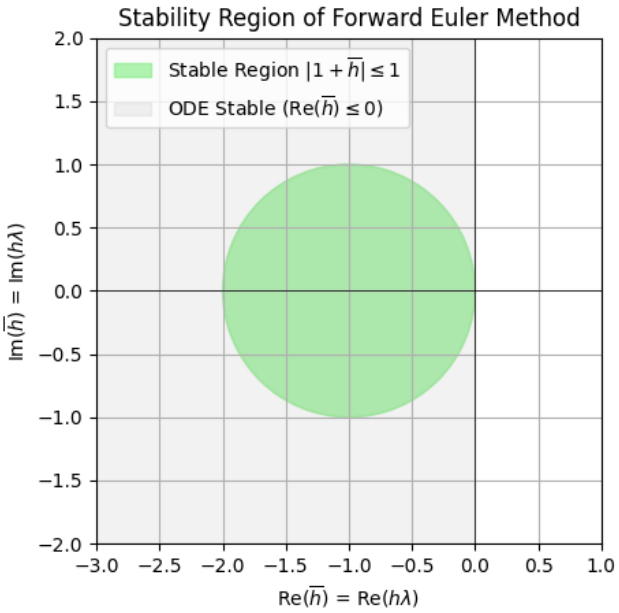

Figure 9: Stability region (green disk) for the Forward Euler method in the complex $\bar{h} = h\lambda$ plane. The method is stable only if $\bar{h}$ lies within this disk. The grey shaded area is the left half-plane where the test ODE $y' = \lambda y$ is mathematically stable.

```
# Initial variables and constants
y=1.0
t=0.0
h=0.1
# Choose the constant in the ODE, dy/dt=lam*y. We need -2<=h*lam<=0 for
    ↪ stability.
# lam=-5   # h*lam = -0.5 (Stable)
# lam=-12.5 # h*lam = -1.25 (Stable, oscillatory)
lam=-21  # h*lam = -2.1 (Unstable)

t_end = 1.0 # Integrate up to t=1
times = [t]
y_numerical = [y]
y_exact_vals = [exp(lam*t)]

print(f"Lambda = {lam}, h = {h}, h*lambda = {h*lam}")

# Apply forward Euler step until t=t_end:
while t < t_end: # Use < to avoid potential floating point issue

    # Analytical solution
    yexact=exp(lam*(t+h)) # Exact solution at next step for comparison
    # print(t,y,yexact,y-yexact) # Original print

    # Euler step
    y=y+h*(lam*y)
```

```
    # Update time
    t=t+h

    times.append(t)
    y_numerical.append(y)
    y_exact_vals.append(exp(lam*t))

# Plotting code can be added here or run separately
# print("Final time:", t, " Final y:", y)
```

Running this code (or a plotting version) for different λ values with $h = 0.1$:
Case 1: $\lambda = -5$ *(*$\bar{h} = -0.5$*, inside stability region)*
Case 2: $\lambda = -12.5$ *(*$\bar{h} = -1.25$*, inside stability region)*
Case 3: $\lambda = -21$ *(*$\bar{h} = -2.1$*, outside stability region)*
These examples clearly show that Forward Euler can produce unstable, growing numerical solutions even when the true solution decays, if the step size h is too large relative to $|\lambda|$.

4.4 Stability Analysis of Backward Euler

Applying Backward Euler $y_{k+1} = y_k + hf(t_{k+1}, y_{k+1})$ to $y' = \lambda y$ gives:

$$y_{k+1} = y_k + h\lambda y_{k+1}$$

Solving for y_{k+1}:

$$y_{k+1}(1 - h\lambda) = y_k \implies y_{k+1} = \frac{1}{1 - h\lambda} y_k$$

The amplification factor is $G(h\lambda) = \frac{1}{1-h\lambda}$. For stability, we need $|G(h\lambda)| \leq 1$, which means:

$$\left| \frac{1}{1 - h\lambda} \right| \leq 1 \implies |1 - h\lambda| \geq 1$$

Let $\bar{h} = h\lambda = a + ib$. We need $|1 - (a + ib)| \geq 1$, or $|(1 - a) - ib| \geq 1$. This means $\sqrt{(1-a)^2 + (-b)^2} \geq 1$, or $(1 - a)^2 + b^2 \geq 1$. This inequality describes the exterior and boundary of a disk in the complex $\bar{h}$-plane with radius 1 centered at $(1, 0)$.

The stability region for Backward Euler (Figure 13) contains the entire left half-plane ($\text{Re}(\bar{h}) \leq 0$). This means that if the ODE $y' = \lambda y$ is stable ($\text{Re}(\lambda) \leq 0$), the Backward Euler method will produce a stable numerical solution for *any* step size $h > 0$. Therefore, Backward Euler is called **unconditionally stable** (more precisely, A-stable, meaning its stability region contains the entire left half-plane).

4.5 Comparison and Trade-offs

- **Explicit Methods (e.g., Forward Euler):**
  - Computationally cheap per step (no equation solving).
  - Often conditionally stable: require restrictions on h for stability, especially when $|\lambda|$ is large (stiff problems).
- **Implicit Methods (e.g., Backward Euler, Trapezoid):**
  - Computationally expensive per step (require solving linear or non-linear equations).
  - Often have much larger stability regions (sometimes unconditionally stable).
  - Can allow significantly larger step sizes h compared to explicit methods for stiff problems, potentially leading to overall higher efficiency despite the cost per step.

The choice between explicit and implicit methods depends on the specific problem, particularly its stiffness (presence of widely varying time scales or large negative $\text{Re}(\lambda)$ values) and the desired accuracy.

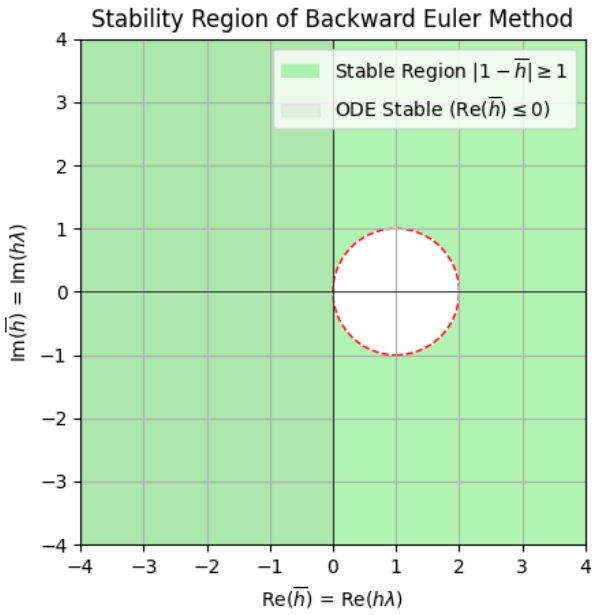

Figure 13: Stability region (green exterior) for the Backward Euler method in the complex $\bar{h} = h\lambda$ plane. The method is stable if $\bar{h}$ lies outside the open disk centered at $(1, 0)$. This region includes the entire left half-plane (grey) where the test ODE is mathematically stable.