Numerical solution of ordinary differential equations

Anne Kværnø, modified by André Massing

Apr 14, 2020

The Python codes for this note are given in `ode.py`.

1 Introduction: Whetting your appetite

The topic of this note is the numerical solution of systems of ordinary differential equations (ODEs). This has been discussed in previous courses, see for instance the web page Differensialligninger from Mathematics 1, as well as in Part 1 of this course, where the Laplace transform was introduced as a tool to solve ODEs analytically.

Before we present the first numerical methods to solve ODEs, we quickly look at a number of examples which hopefully will will serve as test examples throughout this topic.

1.1 Scalar first order ODEs

A scalar, first-order ODE is an equation on the form

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0,$$

where $y'(t) = \frac{dy}{dt}$. The *inital condition* $y(t_0) = y_0$ is required for a unique solution.

Notice.

It is common to use the term *initial value problem (IVP)* for an ODE for which the inital value $y(t_0) = y_0$ is given, and we only are interested in the solution for $t > t_0$. In these lecture notes, only initial value problems are considered.

Example 1.1. *Population growth and decay processes.*

One of the simplest possible IVP is given by

$$y'(t) = \lambda y(t), \quad y(t_0) = y_0. \tag{1}$$

For $\lambda > 0$ this equation can present a simple model for the growth of some population, e.g., cells, humans, animals, with unlimited resources (food, space etc.). The constant λ then corresponds to the *growth rate* of the population.

Negative $\lambda < 0$ typically appear in decaying processes, e.g., the decay of a radioactive isotopes, where λ is then simply called the *decay constant*.

The analytical solution to (1) is

$$y(t) = y_0 e^{\lambda(t-t_0)} \tag{2}$$

and will serve us at several occasions as reference solution to assess the accuracy of the numerical methods to be introduced.

Example 1.2. *Time-dependent coefficients.*

Given the ODE

$$y'(t) = -2ty(t), \quad y(0) = y_0.$$

for some given initial value y_0. The general solution of the ODE is

$$y(t) = Ce^{-t^2},$$

where C is a constant. To determine the constant C, we use the initial condition $y(0) = y_0$ yielding the solution

$$y(t) = y_0 e^{-t^2}.$$

1.2 Systems of ODEs

A system of m ODEs are given by

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2, \ldots, y_m), & y_1(t_0) &= y_{1,0} \\ y_2' &= f_2(t, y_1, y_2, \ldots, y_m), & y_2(t_0) &= y_{2,0} \\ &\vdots & &\vdots \\ y_m' &= f_m(t, y_1, y_2, \ldots, y_m), & y_m(t_0) &= y_{m,0} \end{aligned}$$

or more compactly by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

where we use boldface to denote vectors in $\mathbb{R}^m$,

$$\mathbf{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{pmatrix}, \qquad \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} f_1(t, y_1, y_2, \ldots, y_m), \\ f_2(t, y_1, y_2, \ldots, y_m), \\ \vdots \\ f_m(t, y_1, y_2, \ldots, y_m), \end{pmatrix}, \qquad \mathbf{y}_0 = \begin{pmatrix} y_{1,0} \\ y_{2,0} \\ \vdots \\ y_{m,0} \end{pmatrix}.$$

Example 1.3. *Lotka-Volterra equation.*

The Lotka-Volterra equation is a system of two ODEs describing the interaction between preys and predators over time. The system is given by

$$\begin{aligned} y'(t) &= \alpha y(t) - \beta y(t) z(t) \\ z'(t) &= \delta y(t) z(t) - \gamma z(t) \end{aligned}$$

where x denotes time, $y(t)$ describes the population of preys and $z(t)$ the population of predators. The parameters α, β, δ and γ depends on the populations to be modeled.

Example 1.4. *Spreading of diseases.*

Motivated by the ongoing corona virus pandemic, we consider a (simple!) model for the spreading of an infectious disease, which goes under the name SIR model.

The SIR models divides the population into three population classes, namely

S(t): number individuals **susceptible** for infection,

I(t): number **infected** individuals, capable of transmitting the disease,

R(t): number **removed** individuals who cannot be infected due death or to immunity after recovery

The model is of the spreading of a disease is based on moving individual from S to I and then to R. A short derivation can be found in [**?**, Ch. 4.2]. The final ODE system is given by

$$S' = -\beta SI \tag{3}$$
$$I' = \beta SI - \gamma I \tag{4}$$
$$R' = \gamma I, \tag{5}$$

where β denotes the infection rate, and γ the removal rate.

1.3 Higher order ODEs

An initial value ODE of order m is given by

$$u^{(m)} = f(t, u, u', \ldots, u^{(m-1)}), \qquad u(t_0) = u_0, \quad u'(t_0) = u_0', \quad \ldots, \quad u^{(m-1)}(t_0) = u_0^{(m-1)}.$$

Here $u^{(1)} = u'$ and $u^{(m+1)} = \frac{du^{(m)}}{dx}$, for $m > 0$.

Example 1.5. .

Van der Pol's equation is a second order differential equation, given by:

$$u^{(2)} = \mu(1 - u^2)u' - u, \qquad u(0) = u_0, \quad u'(0) = u_0'.$$

where $\mu > 0$ is some constant. As initial values $u_0 = 2$ and $u_0' = 0$ are common choices.

Later in the note we will see how such equations can be rewritten as a system of first order ODEs. Systems of higher order ODEs can be treated similarly.

2 Euler's method

Now we turn to our first numerical method, namely Euler's method, known from Mathematics 1. We quickly review two alternative derivations, namely one based on *numerical differentiation* and one on *numerical integration*.

2.1 Derivation of Euler's method

Euler's method is the simplest example of a so-called **one step method (OSM)**. Given the IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0,$$

and some final time T, we want to compute to an approximation of $y(t)$ on $[t_0, T]$.

We start from t_0 and choose some (usually small) time step size τ_0 and set the new time $t_1 = t_0 + \tau_0$. The goal is to compute a value y_1 serving as approximation of $y(t_1)$.

To do so, we Taylor expand the exact (but unknown) solution $y(t_0 + \tau)$ around x_0:

$$y(t_0 + \tau) = y(t_0) + \tau y'(t_0) + \frac{1}{2}\tau^2 y''(t_0) + \cdots .$$

Assume the step size τ to be small, such that the solution is dominated by the first two terms. In that case, these can be used as the numerical approximation in the next step:

$$y(t_0 + \tau) \approx y(t_0) + \tau y'(t_0) = y_0 + \tau f(t_0, y_0)$$

giving

$$y_1 = y_0 + \tau_0 f(t_0, y_0).$$

Now we can repeat this procedure and choose the next (possibly different) time step τ_1 and compute a numerical approximation y_2 for $y(t)$ at $t_2 = t_1 + \tau_1$ by setting

$$y_2 = y_1 + \tau_1 f(t_1, y_1).$$

The idea is to repeat this procedure until we reached the final time T resulting in the following

Recipe: Euler's method.

Given a function $f(t, y)$ and an initial value (t_0, y_0).

- Set $t = t_0$.
- `while` $t < T$:

  Choose τ_k

  $y_{k+1} := y_k + \tau f(t_k, y_k)$

  $t_{k+1} := t_k + \tau_k$

  $t := t_{k+1}$

So we can think of the Euler method as a method which approximates the continuous but unknown solution $y(t) : [t_0, T] \to \mathbb{R}$ by a discrete function $y_\Delta : \{t_0, t_1, \dots, t_{N_t}\}$ such that $y_\Delta(t_k) := y_k \approx y(t_k)$.

How to choose τ_i? The simplest possibility is to set a maximum number of steps $N_{\max} = N_t$ and then to chose a *constant time step* $\tau = (T - t_0)/N_{\max}$ resulting in $N_{\max} + 1$ equidistributed points. Later we will also learn, how to choose the *time step adaptively*, depending on the solution's behavior.

Numerical solution between the nodes.

At first we have only an approximation of $y(t)$ at the $N_t + 1$ nodes $y_\Delta : \{t_0, t_1, \dots, t_{N_t}\}$. If we want to evaluate the numerical solution between the nodes, a natural idea is to extend the discrete solution linearly between each pair of time nodes t_k, t_{k+1}. This is compatible with the way the numerical solution can be plotted, namely by connected each pair (t_k, y_k) and (t_{k+1}, y_{k+1}) with straight lines.

Also, in order to compute an approximation at the next point t_{k+1}, Euler's method only needs to know f, τ_k and the solution y_k at the *current* point t_k, but not at earlier points $t_{k-1}, t_{k-2}, \dots$ Thus Euler's method is an prototype of a so-called **One Step Method (OSM)**. We will formalize this concept later.

Interpretation: Euler's method via forward difference operators.

After rearranging terms, we can also interpret the computation of an approximation $y_1 \approx y(t_1)$ as replacing the derivative $y'(t_0) = f(t_0, y_0)$ with a **forward difference operator**

$$f(t_0, y_0) = y'(t_0) \approx \frac{y(t_1) - y(t_0)}{\tau}$$

Thus *Euler's method replace the differential quotient by a difference quotient.*

Alternative derivation via numerical integration. First we recall that for a function $f : [a, b] \to \mathbb{R}$, we can approximate its integral $\int_a^b f(t)\,\mathrm{d}t$ by a *very simple* quadrature rule of the form

$$\int_a^b f(t)\,\mathrm{d}t \approx (b - a) f(a). \tag{6}$$

Question.

What is the degree of exactness of the previous quadrature rule?

Turning to our IVP, we know formally integrate the ODE $y'(t) = f(t, y(t))$ on the time interval $I_k = [t_k, t_{k+1}]$ and then applying the quadrature rule (6) leading to

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(t)\,\mathrm{d}t = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \underbrace{(t_{k+1} - t_k)}_{\tau_k} f(t_k, y(t_k))$$

Sorting terms gives us back Euler's method

$$y(t_{k+1}) \approx y(t_k) + \tau_k f(t_k, y(t_k)).$$

2.2 Implementation of Euler's method

Euler's method can be implemented in only a few lines of code:

```
def explicit_euler(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        ys.append(y + dt*f(t, y))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

Let's test Euler's method with the simple IVP given in Example 1.1.

```
t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 4

# rhs of IVP
f = lambda t,y: lam*y

# Compute numerical solution using Euler
ts, ys_eul = explicit_euler(y0, t0, T, f, Nmax)

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))
ys_ex = y_ex(ts)

# Plot it
plt.plot(ts, ys_ex)
plt.plot(ts, ys_eul, 'ro-')
plt.legend(["$y_{ex}$", "y" ])
```

Plot the solution for various N_t, say $N_t = 4, 8, 16, 32$ against the exact solution given in Example 1.1.

Exercise 1: Error study for the Euler's method

We observed that the more we decrease the constant step size τ (or increase $N_{\max}$), the closer the numerical solution gets to the exact solution.

Now we ask you to quantify this. More precisely, write some code to compute the error

$$\max_{i \in \{0,\dots,N_{\max}\}} |y(t_i) - y_i|$$

for $N_{\max} = 4, 8, 16, 32, 64, 128$. How does the error reduces if you double the number of points?

```
# Insert your code here.
```

```
def error_study(y0, t0, T, f, Nmax_list, solver, y_ex):
    max_errs = []
    for Nmax in Nmax_list:
        ts, ys = solver(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        errors = ys - ys_ex
        max_errs.append(np.abs(errors).max())
        print("For Nmax = {:3}, max ||y(t_i) - y_i||= {:.3e}".format(Nmax,max_errs[-1]))
    print("The computed error reduction rates are")
    max_errs = np.array(max_errs)
    print(max_errs[:-1]/max_errs[1:])

Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, explicit_euler, y_ex)
```

3 Heun's method

Solution. Before we start to look at more exciting examples, we will derive a one step method which is more accurate then Euler's method. Note that Euler's method can be interpreted as being based on a quadrature rule with degree of exactness equal to 0. Let's try to use a better quadrature rule!

Again, we start from the *exact representation*, but this time we use the trapezoidal rule, which has degree of exactness equal to 1, yielding

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \frac{\tau_k}{2}\left(f(t_{k+1}, y(t_{k+1}) + f(t_k, y(t_k))\right)$$

This suggest to consider the scheme

$$y_{k+1} - y_k = \frac{\tau_k}{2}\left(f(t_{k+1}, y_{k+1}) + f(t_k, y(k))\right)$$

But note that starting from y_k, we cannot immediately compute y_{k+1} as it appears also in the expression $f(t_{k+1}, y_{k+1})$! This is an example of an **implicit method**!

To turn this scheme into an explicit scheme, the idea is now to approximate y_{k+1} appearing in f with an explicit Euler step:

$$y_{k+1} = y_k + \frac{\tau_k}{2}\left(f\big(t_{k+1}, y_k + \tau_k f(t_k, y_k)\big) + f(t_k, y_k)\right).$$

Observe that we have now nested evaluations of f. This can be best arranged by computing the nested expression in stages, first the inner one and then the outer one. This leads to the following recipe.

Recipe: Heun's method.

Given a function $f(t, y)$ and an initial value (t_0, y_0).

- Set $t = t_0$.
- `while` $t < T$:

  Choose τ_k

  Compute stage $k_1 := f(t_k, y_k)$

  Compute stage $k_2 := f(t_k + \tau_k, y_k + \tau_k k_1)$

  $y_{k+1} := y_k + \frac{\tau_k}{2}(k_1 + k_2)$

  $t_{k+1} := t_k + \tau_k$

$t := t_{k+1}$

The function `heun` is implemented in `ode.py`:

```
def heun(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        k1 = f(t,y)
        k2 = f(t+dt, y+dt*k1)
        ys.append(y + 0.5*dt*(k1+k2))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

Exercise 2: Comparing Heun with Euler

a) Redo the Example 1.1 with Heun, and plot both the exact solution, y_{eul} and y_{heun} for $N_t = 4, 8, 16, 32$.

```
# Insert code here.
```

```
t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 4

# rhs of IVP
f = lambda t,y: lam*y

# Compute numerical solution using Euler and Heun
ts, ys_eul = explicit_euler(y0, t0, T, f, Nmax)
ts, ys_heun = heun(y0, t0, T, f, Nmax)

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))
ys_ex = y_ex(ts)

# Plot it
plt.plot(ts, ys_ex)
plt.plot(ts, ys_eul, 'ro-')
plt.plot(ts, ys_heun, 'b+-')
plt.legend(["$y_{ex}$", "$y$ Euler", "$y$ Heun" ])
```

b) Redo Exercise 1 with Heun.

```
# Insert code here.
```

```
Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, heun, y_ex)
```

4 Applying Heun's and Euler's method

Solution.

Example 4.1. *The Lotka-Volterra equation revisited.*

Solve the Lotka-Volterra equation

$$y'(t) = \alpha y(t) - \beta y(t)z(t)$$
$$z'(t) = \delta y(t)z(t) - \gamma z(t)$$

In this example, use the parameters and initial values

$$\alpha = 2, \quad \beta = 1, \quad \delta = 0.5, \quad \gamma = 1, \qquad y_{1,0} = 2, \quad y_{2,0} = 0.5.$$

User Euler's method to solve the equation over the interval $[0, 20]$, and use $\tau = 0.02$. Try also other step sizes, e.g. $\tau = 0.1$ and $\tau = 0.002$.

NB! In this case, the exact solution is not known. What is known is that the solutions are periodic and positive. Is this the case here? Check for different values of τ.

```
# Reset plotting parameters
plt.rcParams.update({'figure.figsize': (12,6)})
```

```
# Define rhs
def lotka_volterra(t, y):
    # Set parameters
    alpha, beta, delta, gamma = 2, 1, 0.5, 1
    # Define rhs of ODE
    dy = np.array([alpha*y[0]-beta*y[0]*y[1],
                   delta*y[0]*y[1]-gamma*y[1]])
    return dy

t0, T = 0, 20            # Integration interval
y0 = np.array([2, 0.5])  # Initital values

# Solve the equation
tau = 0.002
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_eul = explicit_euler(y0, t0, T, lotka_volterra, Nmax)

# Plot results
plt.plot(ts, ys_eul)
plt.xlabel('t')
plt.legend(['$y_0(t)$ - Euler', '$y_1(t)$ - Euler'],
        loc="upper right" )
```

Exercise 3: Solving the Lotka-Volterra system using Heun's method

Redo the last example with Heun's method and compare the solutions generated by Euler's and Heun's method.

4.1 Higher order ODEs

How can we numerically solve higher order ODEs using, e.g., Euler's or Heun's method?

Given the m-th order ODE

$$u^{(m)}(t) = f\big(t, u(t), u'(x), \ldots, u^{(m-1)}\big).$$

For a unique solution, we assume that the initial values

$$u(t_0), u'(t_0), u''(t_0), \ldots, u^{(m-1)}(t_0)$$

are known.

Such equations can be written as a system of first order ODEs by the following trick: Let

$$y_1(x) = u(x), \quad y_2(x) = u'(x), \quad y_3(x) = u^{(2)}(x), \quad \dots \quad , y_m(x) = u^{(m-1)}(x)$$

such that

$$\begin{aligned} y_1' &= y_2, & y_1(a) &= u(a) \\ y_2' &= y_3, & y_2(a) &= u'(a) \\ &\vdots & &\vdots \\ y_{m-1}' &= y_m, & y_{m-1}(a) &= u^{(m-2)}(a) \\ y_m' &= f(t, y_1, y_2, \dots, y_{m-1}, y_m), & y_m(a) &= u^{(m-1)}(a) \end{aligned}$$

which is nothing but a system of first order ODEs, and can be solved numerically exactly as before.

Exercise 4: Numerical solution of Van der Pol's equation

Recalling Example 1.5, the Van der Pol oscillator is described by the second order differential equation

$$u'' = \mu(1 - u^2)u' - u, \qquad u(0) = u_0, \quad u'(0) = u_0'.$$

It can be rewritten as a system of first order ODEs:

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= u_0, \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1, & y_2(0) &= u_0'. \end{aligned}$$

a) Let $\mu = 2$, $u(0) = 2$ and $u'(0) = 0$ and solve the equation over the interval $[0, 20]$, using the explicit Euler and $\tau = 0.1$. Play with different step sizes, and maybe also with different values of μ.

b) Repeat the previous numerical experiment with Heun's method. Try to compare the number of steps you need to perform with Euler vs Heun to obtain visually the "same" solution. (That is, you measure the difference of the two numerical solutions in the "eyeball norm".)

```
# Insert code here.
```

```
# Define the ODE
def f(t, y):
    mu = 2
    dy = np.array([y[1],
                   mu*(1-y[0]**2)*y[1]-y[0] ])
    return dy

# Set initial time, stop time and initial value
t0, T - 0, 20
y0 = np.array([2,0])

# Solve the equation using Euler and plot
tau = 0.1
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_eul  = explicit_euler(y0, t0, T, f, Nmax)
plt.plot(ts,ys_eul);

# Solve the equation using Heun
tau = 0.1
Nmax = int(20/tau)
```

```
print("Nmax = {:4}".format(Nmax))
ts, ys_heun = heun(y0, t0, T, f, Nmax)
plt.plot(ts,ys_heun);

plt.xlabel('x')
plt.title('Van der Pols ligning')
plt.legend(['y1 - Euler','y2 - Euler', 'y1 - Heun','y2 - Heun'],loc='upper right');
```

5 One Step Methods

Solution. In the last lecture, we introduced the explicit Euler method and Heun's method, motivating the following definition.

Definition 5.1. *One step methods.*

A one step method defines an approximation to the IVP in the form of a discrete function $\boldsymbol{y}_\Delta : \{t_0, \ldots, t_N\} \to \mathbb{R}^n$ given by

$$\boldsymbol{y}_{k+1} := \boldsymbol{y}_k + \tau_k \Phi(t_k, \boldsymbol{y}_k, \boldsymbol{y}_{k+1}, \tau_k) \tag{7}$$

for some **increment function**

$$\Phi : [t_0, T] \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^+ \to \mathbb{R}^n.$$

The OSM is called **explicit** if the increment function Φ does not depend on $\boldsymbol{y}_{k+1}$, otherwise it is called **implicit**.

Example 5.1. *Increment functions for Euler and Heun.*

The increment functions for Euler's and Heun's method are defined by respectively

$$\Phi(t_k, y_k, y_{k+1}, \tau_k) = f(t_k, y_k), \qquad \Phi(t_k, y_k, y_{k+1}, \tau_k) = \tfrac{1}{2}\left(f(t_k, y_k) + f\big(t_{k+1}, y_k + \tau_k f(t_k, y_k)\big)\right).$$

5.1 Local and global truncation error of OSM

Definition 5.2. *Local truncation error.*

The **local truncation error** $\eta(t, \tau)$ is defined by

$$\eta(t, \tau) = y(t) + \tau\Phi(t, y(t), y(t+\tau), \tau) - y(t+\tau). \tag{8}$$

$\eta(t, \tau)$ is often also called the **local discretization** or **consistency error**.

A one step method is called **consistent of order** $p \in \mathbb{N}$ if there is a constant $C > 0$ such that

$$|\eta(t, \tau)| \leqslant C\tau^{p+1} \quad \text{for } \tau \to 0. \tag{9}$$

A short-hand notation for this is to write $\eta(t, \tau) = \mathcal{O}(\tau^{p+1})$ for $\tau \to 0$.

Example 5.2. .

Euler's method has consistency order $p = 1$.

Definition 5.3. *Global truncation error.*

For a numerical solution $y_\Delta : \{t_0, \ldots, t_N\} \to \mathbb{R}$ the **global truncation error** is defined by

$$e_k(t_k, \tau_k) = y(t_k) - y_k \quad \text{for } k = 0, \ldots, N. \tag{10}$$

A one step method is called **convergent with order** $p \in \mathbb{N}$ if

$$\max_{k \in \{0,1,\ldots,N_t\}} |e_k(t_k, \tau_k)| = \mathcal{O}(\tau^p) \tag{11}$$

with $\tau = \max_k \tau_k$.

Discussion. If a one step method has convergence order equal to p, the maximum error $e(\tau) = \max_k |e(t_k, \tau)|$ can be thought as a function of the step size τ is of the form

$$e(\tau) = O(\tau^p) \leqslant C\tau^p.$$

This implies that if we change the time step size from τ to $\frac{\tau}{2}$, we can expect that the error decreases from $C\tau^p$ to $C(\frac{\tau}{2})^p$, that is, the error will be reduced by a factor 2^{-p}.

How can we determine the convergence rate by means of numerical experiments? Starting from $e(\tau) = O(\tau^p) \leqslant C\tau^p$ and taking the logarithm gives

$$\log(e(\tau)) \leqslant p\log(\tau) + \log(C).$$

Thus $\log(e(\tau))$ is a linear function of $\log(\tau)$ and the slope of this linear function corresponds to the order of convergence p.

So if you have an *exact solution* at your disposal, you can for an increasing sequence `Nmax_list` defining a descreasing sequence of *maximum* time-steps $\{\tau_0, \ldots, \tau_M\}$ and solve your problem numerically and then compute the resulting exact error $e(\tau_i)$ and plot it against τ_i in a $\log - \log$ plot to determine the convergence order.

In addition you can also compute the experimentally observed convergence rate EOC for $i = 1, \ldots M$ defined by

$$\mathrm{EOC}(i) = \frac{\log(e(\tau_i)) - \log(e(\tau_{i-1}))}{\log(\tau_i) - \log(\tau_{i-1})} = \frac{\log(e(\tau_i)/e(\tau_{i-1}))}{\log(\tau_i/\tau_{i-1})}$$

Ideally, $\mathrm{EOC}(i)$ is close to p.

This is implemented in the following `python` function.

```
def compute_eoc(y0, t0, T, f, Nmax_list, solver, y_ex):
    errs = [ ]
    for Nmax in Nmax_list:
        ts, ys = solver(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        errs.append(np.abs(ys - ys_ex).max())
        print("For Nmax = {:3}, max ||y(t_i) - y_i||= {:.3e}".format(Nmax,errs[-1]))

    errs = np.array(errs)
    Nmax_list = np.array(Nmax_list)
    dts = (T-t0)/Nmax_list

    eocs = np.log(errs[1:]/errs[:-1])/np.log(dts[1:]/dts[:-1])
    return errs, eocs
```

Exercise 5: Convergence order for Euler and Heun

Use the `compute_eoc` function and any of the examples with a known analytical solution from the previous lecture to determine convergence order for Euler's and Heun's method.

Solution. The solution to this exercise is implemented in the `exercise_eoc_study` function as part of the `ode.py` file. You can also find it in the jupyter notebook version of the this lecture note.

5.2 A general convergence result for one step methods

Theorem 5.1. *Convergence of one-step methods.*

Assume that there exist positive constants M and D such that the increment function satisfies

$$\|\mathbf{\Phi}(t, \mathbf{y}; \tau) - \mathbf{\Phi}(t, \mathbf{z}; \tau)\| \leq M\|\mathbf{y} - \mathbf{z}\|$$

and the local trunctation error satisfies

$$\|\boldsymbol{\eta}(t, \tau)\| = \|\mathbf{y}(t + \tau) - (\mathbf{y}(t) + \tau\mathbf{\Phi}(t, \mathbf{y}(t), \tau))\| \leqslant D\tau^{p+1}$$

for all t, $\mathbf{y}$ and $\mathbf{z}$ in the neighbourhood of the solution. In that case, the global error satisfies

$$\max_{k\in\{0,1,\ldots,N_t\}} \|e_k(t_k,\tau_k)\| \leqslant C\tau^p, \qquad C = \frac{e^{M(T-t_0)}-1}{M}D,$$

where $\tau = \max_{k\in\{0,1,\ldots,N_t\}} \tau_k$.

Proof. We omit the proof here.

It can be proved that the first of these conditions are satisfied for all the methods that will be considered here.

Summary.

The convergence theorem for one step methods can be summarized as

"local truncation error behaves like $\mathcal{O}(\tau^{p+1})$" + "Increment function satisfies a Lipschitz condition" $\Rightarrow$ "global truncation error behaves like $\mathcal{O}(\tau^p)$"

or equivalently,

"consistency order p" + "Lipschitz condition for the Increment function" $\Rightarrow$ "convergence order p.

5.3 Convergence properties of Heun's method

Thanks to Theorem 5.1, we need to show two things to prove convergence and find the corresponding convergence of a given one step methods:

- determine the local truncation error, expressed as a power series in the step size τ
- the condition $\|\mathbf{\Phi}(t,\boldsymbol{y},\tau) - \mathbf{\Phi}(t,\boldsymbol{z},\tau)\| \leqslant M\|\boldsymbol{y}-\boldsymbol{z}\|$

Determining the consistency order. The local truncation error is found by making Taylor expansions of the exact and the numerical solutions starting from the same point, and compare. In practice, this is not trivial. For simplicity, we will here do this for a scalar equation $y'(t) = f(t,y(t))$. The result is valid for systems as well

In the following, we will use the notation

$$f_t = \frac{\partial f}{\partial t}, \qquad f_y = \frac{\partial f}{\partial y}, \qquad f_{tt} = \frac{\partial^2 f}{\partial t^2} \qquad f_{ty} = \frac{\partial^2 f}{\partial t \partial y} \qquad \text{etc.}$$

Further, we will surpress the arguments of the function f and its derivatives. So f is to be understood as $f(t,y(t))$ although it is not explicitly written.

The Taylor expansion of the exact solution $y(t+\tau)$ is given by

$$y(t+\tau) = y(t) + \tau y'(t) + \frac{\tau^2}{2}y''(t) + \frac{\tau^3}{6}y'''(t) + \cdots.$$

Higher derivatives of $y(t)$ can be expressed in terms of the function f by using the chain rule and the product rule for differentiation.

$$\begin{aligned}
y'(t) &= f,\\
y''(t) &= f_t + f_y y' = f_t + f_y f,\\
y'''(t) &= f_{tt} + f_{ty}y' + f_{yt}f + f_{yy}y'f + f_y f_t + f_y f_y y' = f_{tt} + 2f_{ty}f + f_{yy}f^2 + f_y f_t + (f_y)^2 f.
\end{aligned}$$

Find the series of the exact and the numerical solution around x_0, y_0 (any other point will do equally well). From the discussion above, the series for the exact solution becomes

$$y(t_0+\tau) = y_0 + \tau f + \frac{\tau^2}{2}(f_t + f_y f) + \frac{\tau^3}{6}(f_{tt} + 2f_{ty}f + f_{yy}ff + f_y f_x f + f_y f_t + (f_y)^2 f) + \cdots,$$

where f and all its derivatives are evaluated in (t_0, y_0). For the numerical solution we get

$$\begin{aligned}
k_1 &= f(t_0, y_0) = f, \\
k_2 &= f(t_0 + \tau, y_0 + \tau k_1) \\
&= f + \tau f_t + f_y \tau k_1 + \frac{1}{2} f_{tt} \tau^2 + f_{ty} \tau \tau k_1 + \frac{1}{2} f_{yy} \tau^2 k_1^2 + \cdots \\
&= f + \tau(f_t + f_y f) + \frac{\tau^2}{2}(f_{tt} + 2 f_{ty} f + f_{yy} f^2) + \cdots, \\
y_1 &= y_0 + \frac{\tau}{2}(k_1 + k_2) = y_0 + \frac{\tau}{2}(f + f + \tau(f_t + f_y f) + \frac{\tau^2}{2}(f_{tt} + 2 f_{ty} k_1 + f_{yy} f^2)) + \cdots \\
&= y_0 + \tau f + \frac{\tau^2}{2}(f_t + f_y f) + \frac{\tau^3}{4}(f_{tt} + 2 f_{ty} f + f_{yy} f^2) + \cdots
\end{aligned}$$

and the local truncation error will be

$$\eta(t_0, \tau) = y(t_0 + \tau) - y_1 = \frac{\tau^3}{12}(-f_{tt} - 2 f_{ty} f - f_{yy} f^2 + 2 f_y f_t + 2 (f_y)^2 f) + \cdots$$

The first nonzero term in the local truncation error series is called *the principal error term*. For τ sufficiently small this is the term dominating the error, and this fact will be used later.

Although the series has been developed around the initial point, series around $x_n, y(t_n)$ will give similar results, and it is possible to conclude that, given sufficient differentiability of f there is a constant D such that

$$\max_i |\eta(t_i, \tau)| \leq D \tau^3.$$

Consequently, Heun's method is of consistency order 2.

Lipschitz condition for Φ. Further, we have to prove the condition on the increment function $\Phi(t, y)$. For f differentiable, there is for all y, z some ξ between x and y such that $f(t, y) - f(t, z) = f_y(t, \xi)(y - z)$. Let L be a constant such that $|f_y| < L$, and for all x, y, z of interest we get

$$|f(t, y) - f(t, z)| \leq L |y - z|.$$

The increment function for Heun's method is given by

$$\Phi(t, y) = \frac{1}{2}(f(t, y) + f(t + \tau, y + \tau f(t, y))).$$

By repeated use of the condition above and the triangle inequalitiy for absolute values we get

$$\begin{aligned}
|\Phi(t, y) - \Phi(t, z)| &= \frac{1}{2} |f(t, y) + f(t + \tau, y + f(t, y)) - f(t, z) - \tau f(t + \tau, z + f(t, z)| \\
&\leq \frac{1}{2} \big(|f(t, y) - f(t, z)| + |f(t + \tau, y + \tau f(t, y)) - f(t + \tau, z + \tau f(t, z)| \big) \\
&\leq \frac{1}{2} \big(L|y - z| + L|y + \tau f(t, y) - z - \tau f(t, z)| \big) \\
&\leq \frac{1}{2} \big(2L|y - z| + \tau L^2 |y - z| \big) \\
&= (L + \frac{\tau}{2} L^2) |y - z|.
\end{aligned}$$

Assuming that the step size τ is bounded upward by some τ_0, we can conclude that

$$|\Phi(t, y) - \Phi(t, z)| \leq M |y - z|, \qquad M = L + \frac{\tau_0}{2} L^2.$$

Thanks to Theorem 5.1, we can conclude that Heun's method is convergent of order 2.

6 Runge-Kutta Methods

In the previous lectures we introduced *Euler's method* and *Heun's method* as particular instances of the *One Step Methods*, and we presented the general error theory for one step method.

In this Lecture, we introduce a large family of the one step methods which go under the name **Runge-Kutta methods (RKM)**. We will see that Euler's method and Heun's method are instance of RKMs.

6.1 Derivation of Runge-Kutta Methods

For a given time interval $I_i = [t_i, t_{i+1}]$ we want to compute y_{i+1} assuming that y_i is given. Starting from the exact expression

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y(t))\,\mathrm{d}t,$$

the idea is now to approximate the integral by some quadrature rule $\mathrm{Q}[\cdot](\{\xi_j\}_{j=1}^s, \{b_j\}_{j=1}^s)$ defined on I_i. Then we get

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y(t))\,\mathrm{d}t \tag{12}$$

$$\approx \tau \sum_{j=0}^{s} b_j f(\xi_j, y(\xi_j)) \tag{13}$$

Now we can define $\{c_j\}_{j=1}^s$ such that $\xi_j = t_i + c_j\tau$ for $j = 1, \ldots, s$

Exercise 6: A first condition on b_j

Question: What value do you expect for $\sum_{j=1}^s b_j$?

A. $\sum_{j=1}^s b_j = \tau$

B. $\sum_{j=1}^s b_j = 0$

C. $\sum_{j=1}^s b_j = 1$

Answer: C.

Solution: **A**: Wrong. **B**: Wrong. **C**: Right.

In contrast to pure numerical integration, we don't know the values of $y(\xi_j)$. Again, we could use the same idea to approximate

$$y(\xi_j) - y(t_i) = \int_{t_i}^{t_i + c_j\tau} f(t, y(t))\,\mathrm{d}t$$

but then again we get a closure problem if we choose new quadrature points. The idea is now to *not introduce even more new quadrature points* but to use same $y(\xi_j)$ to avoid the closure problem. Note that this leads to an approximation of the integrals $\int_{t_i}^{t_i + c_j\tau}$ with possible nodes *outside* of $[t_i, t_i + c_j\tau]$.

This leads us to

$$y(\xi_j) - y(t_i) = \int_{t_i}^{t_i + c_j\tau} f(t, y(t))\,\mathrm{d}t \tag{14}$$

$$\approx c_j\tau \sum_{l=1}^{s} \tilde{a}_{jl} f(\xi_l, y(\xi_l)) \tag{15}$$

$$= \tau \sum_{l=1}^{s} a_{jl} f(\xi_l, y(\xi_l)) \tag{16}$$

where we set $c_j\tilde{a}_{jl} = a_{jl}$.

Exercise 7: A first condition on a_{jl}

Question: What value do you expect for $\sum_{l=1}^{s} a_{jl}$?

A. $\sum_{l=1}^{s} a_{jl} = \frac{1}{c_j}$

B. $\sum_{l=1}^{s} a_{jl} = c_j$

C. $\sum_{l=1}^{s} a_{jl} = 1$

D. $\sum_{l=1}^{s} a_{jl} = \tau$

Answer: B.

Solution: **A**: Wrong. **B**: Right. **C**: Wrong. **D**: Wrong.

Definition 6.1. *Runge-Kutta methods.*

Given b_j, c_j, and a_{jl} for $j, l = 1, \dots s$, the Runge-Kutta method is defined by the recipe

$$Y_j = y_i + \tau \sum_{l=1}^{s} a_{jl} f(t_i + c_l \tau, Y_l) \quad \text{for } j = 1, \dots s, \tag{17}$$

$$y_{i+1} = y_i + \tau \sum_{j=1}^{s} b_j f(t_i + c_j \tau, Y_j) \tag{18}$$

Runge-Kutta schemes are often specified in the form of a **Butcher table**:

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} \tag{19}$$

If $a_{ij} = 0$ for $j \geqslant i$ the Runge-Kutta method is called **explicit**. (Why?)

Note that in the final step, all the function evaluation we need to perform have already been performed when computing Y_j.

Therefore one often rewrite the scheme by introducing **stage derivatives**

$$k_l = f(t_i + c_l \tau, Y_l) \tag{20}$$

$$= f(t_i + c_l \tau, y_i + \tau \sum_{j=1}^{s} a_{lj} k_j) \quad j = 1, \dots s, \tag{21}$$

so the resulting scheme will be (swapping index l and j)

$$k_j = f(t_i + c_j \tau, y_i + \tau \sum_{l=1}^{s} a_{jl} k_l) \quad j = 1, \dots s, \tag{22}$$

$$y_{i+1} = y_i + \tau \sum_{j=1}^{s} b_j k_j \tag{23}$$

Exercise 8: Butcher table for the explicit Euler

Write down the Butcher table for the explicit Euler.

Solution. Define $k_1 = f(t_i, y_i) = f(t_i + 0 \cdot \tau, y_i + \tau \cdot 0 \cdot k_1)$. Then the explicit Euler step $y_{i+1} = y_i + \tau k_1 = y_i + \tau \cdot 1 \cdot k_1$, and thus the Butcher table is given by

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}.$$

Exercise 9: The improved explicit Euler method

We formally derive the **explicit midpoint rule** or **improved explicit Euler method**. Applying the midpoint rule to our integral representatio yields

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y(t))\,\mathrm{d}t \tag{24}$$

$$\approx \tau f(t_i + \tfrac{1}{2}\tau, y(t_i + \tfrac{1}{2}\tau)) \tag{25}$$

Since we cannot determine the value $y(t_i + \frac{1}{2}\tau)$ from this system, we approximate it using a half Euler step

$$y(t_i + \tfrac{1}{2}\tau) \approx y_{t_i} + \tfrac{1}{2}\tau f(t_i, y(t_i))$$

leading to the scheme

$$y_{i+1/2} := y_i + \tfrac{1}{2}\tau f(t_i, y_i) \tag{26}$$

$$y_{i+1} := y_i + \tau f(t_i + \tfrac{1}{2}\tau, y_{i+1/2}) \tag{27}$$

a) Is this a one-step function? Can you define the increment function Φ?

Solution. Yes it is, and it's increment function is given by

$$\Phi(t_i, y_i, y_{i+1}, \tau) = f(t_i + \tfrac{1}{2}\tau, y_i + \tfrac{1}{2}\tau f(t_i, y_i))$$

b) Can you rewrite this as a Runge-Kutta method? If so, determine the Butcher table of it.

Solution. Define k_1 and k_2 as follows,

$$y_{i+1/2} := y_i + \tfrac{1}{2}\tau \underbrace{f(t_i, y_i)}_{=:k_1} \tag{28}$$

$$y_{i+1} := y_i + \tau f(t_i + \tfrac{1}{2}\tau, y_{i+1/2}) = y_i + \tau \underbrace{f(t_i + \tfrac{1}{2}\tau, y_i + \tau\tfrac{1}{2}k_1)}_{:=k_2}. \tag{29}$$

Then

$$y_{i+1} = y_i + \tau k_2 \tag{30}$$

and thus the Butcher table is given by

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

6.2 Implementation of explicit Runge-Kutta methods

Below you will find the implementation a general solver class `Explicit_Runge_Kutta` which at its initialization takes in a Butcher table and has `__call__` function

```
def __call__(self, y0, f, t0, T, n):
```

and can be used like this

```
# Define Butcher table
a = np.array([[0, 0, 0],
              [1.0/3.0, 0, 0],
              [0, 2.0/3.0, 0]])

b = np.array([1.0/4.0, 0, 3.0/4.0])

c = np.array([0,
              1.0/3.0,
              2.0/3.0])

# Define number of time steps
n = 10

# Create solver
rk3 = Explicit_Runge_Kutta(a, b, c)

# Solve problem (applies __call__ function)
ts, ys = rk3(y0, t0, T, f, Nmax)
```

The complete implementation is given here:

```
class Explicit_Runge_Kutta:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __call__(self, y0, t0, T, f, Nmax):
        # Extract Butcher table
        a, b, c = self.a, self.b, self.c

        # Stages
        s = len(b)
        ks = [np.zeros_like(y0, dtype=np.double) for s in range(s)]

        # Start time-stepping
        ys = [y0]
        ts = [t0]
        dt = (T - t0)/Nmax

        while(ts[-1] < T):
            t, y = ts[-1], ys[-1]

            # Compute stages derivatives k_j
            for j in range(s):
                t_j = t + c[j]*dt
                dY_j = np.zeros_like(y, dtype=np.double)
                for l in range(j):
                    dY_j += dt*a[j,l]*ks[l]

                ks[j] = f(t_j, y + dY_j)

            # Compute next time-step
            dy = np.zeros_like(y, dtype=np.double)
            for j in range(s):
                dy += dt*b[j]*ks[j]

            ys.append(y + dy)
            ts.append(t + dt)

        return (np.array(ts), np.array(ys))
```

Example 6.1. *Implementation and testing of the improved Euler method.*

We implement the **improved explicit Euler** from above and plot the analytical and the numerical solution. Finally, we determine the convergence order.

```
# Define Butcher table for improved Euler
a = np.array([[0, 0],
              [0.5, 0]])
b = np.array([0, 1])
c = np.array([0, 0.5])

# Define rk2
rk2 = Explicit_Runge_Kutta(a, b, c)

t0, T = 0, 1
y0 = 1
lam = 1

# rhs of IVP
f = lambda t,y: lam*y

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))

# EOC test
Nmax_list = [4, 8, 16, 32, 64, 128]

errs, eocs = compute_eoc(y0, t0, T, f, Nmax_list, rk2, y_ex)
print(errs)
print(eocs)
```

6.3 Order conditions for Runge-Kutta Methods

The convergence theorem for one-step methods gave us some necessary conditions to guarantee that a method is convergent order of p:

> "consistency order p" + "Increment function satisfies a Lipschitz condition" $\Rightarrow$ "convergence order p.
>
> "local truncation error behaves like $\mathcal{O}(\tau^{p+1})$" + "Increment function satisfies a Lipschitz condition" $\Rightarrow$ "global truncation error behaves like $\mathcal{O}(\tau^p)$"

It turns out that for f is at least C^1 with respect to all its arguments then the increment function Φ associated with any Runge-Kutta methods satisfies a Lipschitz condition. Thus the next theorem

Theorem 6.1. *Order conditions for Runge-Kutta methods.*

A Runge–Kutta method has consistency order p if and only if all the conditions up to and including p in the table below are satisfied.

p	conditions
1	$\sum b_i = 1$
2	$\sum b_i c_i = 1/2$
3	$\sum b_i c_i^2 = 1/3$ $\sum b_i a_{ij} c_j = 1/6$
4	$\sum b_i c_i^3 = 1/4$ $\sum b_i c_i a_{ij} c_j = 1/8$ $\sum b_i a_{ij} c_j^2 = 1/12$ $\sum b_i a_{ij} a_{jk} c_k = 1/24$

where sums are taken over all the indices from 1 to s.

Proof. Without proof.

Theorem 6.2. *Convergence theorem for Runge-Kutta methods.*

Given the IVP $\boldsymbol{y}' = \boldsymbol{f}(t, \boldsymbol{y}), \boldsymbol{y}(0) = \boldsymbol{y}_0$. Assume $f \in C^1$ and that a given Runge-Kutta method satisfies the order conditions from Theorem 6.1 up to order p. Then the Runge-Kutta method is convergent of order p.