Numerical solution of ordinary differential equations

Anne Kværnø

Oct 27, 2020

With revisions by **Markus Grasmair**.

1 Introduction

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 webpage Differensialligninger from Mathematics 1.

Scalar ODEs. A scalar ODE is an equation of the form

$$y'(x) = f(x, y(x)), \qquad y(x_0) = y_0,$$

where $y'(x) = \frac{dy}{dx}$. The *inital condition* $y(x_0) = y_0$ is required for a unique solution.

NB! It is common to use the term *initial value problem (IVP)* for an ODE for which the inital value $y(x_0) = y_0$ is given, and we only are interested in the solution for $x > x_0$. In this note, only initial value problems are considered.

Example 1: The general solution of the ODE

$$y'(x) = -2xy(x)$$

is the function

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

where C is a constant that depends on the initial condition $y(x_0)$. For instance, we obtain for $x_0 = 0$ and $y(0) = 1$ the solution

$$y(x) = e^{-x^2},$$

Systems of ODEs. A system of m ODEs is given by

$$\begin{aligned}
y_1' &= f_1(x, y_1, y_2, \dots, y_m), & y_1(x_0) &= y_{1,0} \\
y_2' &= f_2(x, y_1, y_2, \dots, y_m), & y_2(x_0) &= y_{2,0} \\
&\vdots & &\vdots \\
y_m' &= f_m(x, y_1, y_2, \dots, y_m), & y_m(x_0) &= y_{m,0}
\end{aligned}$$

or, more compactly, by

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)), \qquad \mathbf{y}(x_0) = \mathbf{y}_0$$

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

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

Example 2: The Lotka-Volterra equation is a system of two ODEs describing the interaction between predators and prey over time. The system is given as

$$\begin{aligned} y'(x) &= \alpha y(x) - \beta y(x) z(x), \\ z'(x) &= \delta y(x) z(x) - \gamma z(x). \end{aligned}$$

Here x denotes time, $y(x)$ describes the population of the prey species, and $z(x)$ the population of predators. The parameters α, β, δ, and γ depend on the populations to be modelled.

Higher order ODEs. An initial value ODE of order m is given by

$$u^{(m)} = f(x, u, u', \dots, u^{(m-1)}), \qquad u(x_0) = u_0, \quad u'(x_0) = u_0', \quad \dots, \quad u^{(m-1)}(x_0) = u_0^{(m-1)}.$$

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

Example 3: 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. Common choices for initial values are $u_0 = 2$ and $u_0' = 0$.

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 Numerical methods for solving ODEs

In this note, we will discuss some techniques for the numerial solution of ordinary differential equations. For simplicity or presentation, we will develop and discuss these methods mostly based on scalar ODEs. The same methods, however, are equally applicable for systems of equations.

All the methods that we will discuss are so-called *one-step methods*. Given the ODE and the initial values (x_0, y_0), we choose some step size h and let $x_1 = x_0 + h$. Based on this information, we calculate an approximation y_1 to $y(x_1)$. Then, we repeat this process starting from (x_1, y_1) in order to calculate an approximation y_2 of $y(x_2)$, where $x_2 = x_1 + h$. This process is repeated until some final point, here called x_{end} is reached.

In one-step methods, the approximation y_{k+1} of $y(x_{k+1})$ does not depend on the values of y_{k-1}, y_{k-2}, $\dots$, y_0. The main alternative to this type of methods are *multi-step methods*, where the approximation y_{k+1} of $y(x_{k+1})$ takes into account those values as well.

It should be emphasized that this strategy only will find approximations to the exact solution in some discrete points x_n, $n = 0, 1, \dots$.

3 Euler's method

Let us start with the simplest example, Euler's method, known from Mathematics 1.

We are given an IVP

$$y'(x) = f(x, y(x)), \qquad y(x_0) = y_0.$$

Choose some step size h. The trick is as follows:

Do a Taylor expansion (*Preliminaries*, section 4) of the exact (but unknown) solution $y(x_0+h)$ around x_0:

$$y(x_0+h) = y(x_0) + hy'(x_0) + \frac{1}{2}h^2y''(x_0) + \cdots .$$

Assume the step size h 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:

$$y(x_0+h) \approx y(x_0) + hy'(x_0) = y_0 + hf(x_0, y_0)$$

giving

$$y_1 = y_0 + hf(x_0, y_0).$$

Repeating this, results in

Euler's method.

- Given a function $f(x,y)$ and an initial value (x_0, y_0).
- Choose a step size h.
- For $i = 0, 1, 2, \ldots$

$$\begin{aligned} y_{n+1} &= y_n + hf(x_n, y_n), \\ x_{n+1} &= x_n + h. \end{aligned}$$

4 Implementation

We would like to make this implementation more like a test platform. It should be simple to implement and test methods other than Euler's. That is why the implementaion here is divided in two parts:

- `ode_solver`: This is a generic solver, and can be used by other methods than Euler's.
- `euler`: This function performs one step of Euler's method.

Start by calling the necessary modules:

```
from numpy import *
from matplotlib.pyplot import *
```

```
def euler(f, x, y, h):
    # One step of the Euler method
    y_next = y + h*f(x, y)
    x_next = x + h
    return x_next, y_next
```

```
def ode_solver(f, x0, xend, y0, h, method=euler):
    '''
    Generic solver for ODEs
       y' = f(x,y), y(a)=y0
    Input: f, the integration interval x0 and xend,
           the stepsize h and the method of choice.

    Output: Arrays with the x- and the corresponding y-values.
    '''

    # Initializing:
    y_num = array([y0])    # Array for the solution y
    x_num = array([x0])     # Array for the x-values

    xn = x0                 # Running values for x and y
    yn = y0

    # Main loop
    while xn < xend - 1.e-10:             # Buffer for truncation errors
        xn, yn = method(f, xn, yn, h)     # Do one step by the method of choice

        # Extend the arrays for x and y
        y_num = concatenate((y_num, array([yn])))
        x_num = append(x_num,xn)

    return x_num, y_num
```

The function `method`, which performs one step with a given method, can be changed, but the call of the function has to be of the following form:

`x_next, y_next = method(f, x, y, h)`.

Numerical example 1: Test the implementation of Euler's method on the problem

$$y'(x) = -2xy(x), \qquad y(0) = 1, \qquad 0 \leq x \leq 1,$$

which has the analytic solution

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

Try with different step sizes, for instance $h = 0.1$, $h = 0.05$ and $h = 0.01$. In each case, compare the numerical solution with the exact one.

The following script solves the equation numerically.

```
# Numerical experiment 1

# The right hand side of the ODE
def f(x, y):
    return -2*x*y

# The exact solution, for verification
def y_exact(x):
    return exp(-x**2)

x0, xend = 0, 1               # Integration interval
y0 = 1                        # Initial value for y
h = 0.1                       # Stepsize

# Solve the equation
x_num, y_num = ode_solver(f, x0, xend, y0, h)

# Plot of the exact solution
x = linspace(x0, xend, 101)
plot(x, y_exact(x))

# Plot of the numerical solution
plot(x_num, y_num, '.-')

xlabel('x')
ylabel('y(x)')
legend(['Exact', 'Euler']);
```

We can also make a plot of the error in each step:

```
# Calculate and plot the error in the x-values
error = y_exact(x_num)-y_num
plot(x_num, error, '.-')
xlabel('x')
ylabel('Error in Eulers metode')
print('Max error = ', max(abs(error)))  # Print the maximum error
```

Numerical exercise 1: Repeat the example on a logistic equation, given by

$$y' = y(1-y), \qquad y(0) = y_0,$$

on the interval $[0, 10]$. Use $y_0 = 0.1$ as initial value. For comparision, the exact solution is

$$y(x) = \frac{1}{1-(1-\frac{1}{y_0})e^{-x}}.$$

Solve the equation numerically by using different step sizes h, and try different initial values.

4.1 Systems of ODEs

Euler's method works equally well for systems of m ODEs

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)), \qquad \mathbf{y}(x_0) = \mathbf{y}_0$$

Here, Euler's method is defined as

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(x_n, \mathbf{y}_n), \qquad n = 0, \dots, N-1.$$

The implementation above can be used without any changes. The only difference from the scalar ODE case is that $y_n \in \mathbb{R}^m$ and $\mathbf{f}\colon \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$. That is, the function $\mathbf{f}$ that defines the right hand side of the ODE takes a scalar x and an array $\mathbf{y}_n$ of length m as inputs, and returns an array of length m.

Numerical example 2: Solve the Lotka-Volterra equation

$$\begin{aligned} y_1'(x) &= \alpha y_1(x) - \beta y_1(x) y_2(x), & y_1(0) &= y_{1,0}, \\ y_2'(x) &= \delta y_1(x) y_2(x) - \gamma y_2(x), & y_2(0) &= y_{2,0}. \end{aligned}$$

In this example, use the parameters and initial values

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

Solve the equation over the interval $[0, 20]$, and use $h = 0.02$. Try also other step sizes, e.g. $h = 0.1$ and $h = 0.002$.

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 for the numerical solutions as well? Check for different values of h.

```
# Numerical example 2, system of equations.

# The right hand side of the ODE
# NB! y is an array of dimension 2, and so is dy.
def lotka_volterra(x, y):
    alpha, beta, delta, gamma = 2, 1, 0.5, 1     # Set the parameters
    dy = array([alpha*y[0]-beta*y[0]*y[1],       #
                delta*y[0]*y[1]-gamma*y[1]])
    return dy

x0, xend = 0, 20            # Integration interval
y0 = array([2, 0.5])        # Initial values
```

```
# Solve the equation
x_lv, y_lv = ode_solver(lotka_volterra, x0, xend, y0, h=0.02)

# Plot the solution
plot(x_lv,y_lv);
xlabel('x')
title('Lotka-Volterra equation')
legend(['y1','y2'],loc=1);
```

4.2 Higher order ODEs

What about higher order ODEs? Can they be solved by Euler's method as well?

To that end, we consider the m-th order ODE

$$u^{(m)}(x) = f\big(x, u(x), u'(x), \dots, u^{(m-1)}\big).$$

For a unique solution, we assume that the initial values

$$u(x_0), u'(x_0), u''(x_0), \dots, u^{(m-1)}(x_0)$$

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

$$\begin{aligned} y_1' &= y_2, & y_1(x_0) &= u(x_0), \\ y_2' &= y_3, & y_2(x_0) &= u'(x_0), \\ &\vdots & &\vdots \\ y_{m-1}' &= y_m, & y_{m-1}(x_0) &= u^{(m-2)}(x_0), \\ y_m' &= f(x, y_1, y_2, \cdots, y_{m-1}, y_m), & y_m(x_0) &= u^{(m-1)}(x_0). \end{aligned}$$

This is nothing but a system of first order ODEs, and can be solved by Euler's method exactly as before. Note, though, that we are mainly interested in the first component of the solution, which corresponds to the original function u.

Numerical example 3: 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:

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

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

```
# Numerical example 3

# Define the ODE
def van_der_pol(x, y):
    mu = 2
    dy = array([y[1],
                mu*(1-y[0]**2)*y[1]-y[0] ])
    return dy
```

```
# Solve the equation
x_vdp, y_vdp = ode_solver(van_der_pol, x0=0, xend=20, y0=array([2,0]), h=0.1)

# Plot the solution
plot(x_vdp,y_vdp);
xlabel('x')
title('Van der Pol\'s equation')
legend(['y1','y2'],loc=1);
```

5 Error analysis

When an ODE is solved by Euler's method over some interval $[x_0, x_{end}]$, how will the error at x_{end} (or some arbitrary point) depend on the number of steps. Or more spesific, choose the number of steps N, let the step size be $h = (x_{end} - x_0)/N$, such that $x_{end} = x_N$, what can we say about the error $e_N = y(x_{end}) - y_N$?

Numerical example 4: Solve the equation of Example 1,

$$y'(x) = -2xy(x), \qquad y(0) = 1,$$

with exact solution $y(x) = e^{-x^2}$, over the interval $[0, 1]$. Use different step sizes h, and for each h, measure the error at $x = 1$.

```
# Numerical example 4
def f(x, y):                  # The right hand side of the ODE
    return -2*x*y

def y_exact(x):              # The exact solution
    return exp(-x**2)

h = 0.1                       # The stepsize
x0, xend = 0, 1               # Integration interval
y0 = 1                        # Initial value

print('h           error\n---------------------')

# Main loop
for n in range(10):
    x_num, y_num = ode_solver(f, x0, xend, y0, h)   # Solve the equation
    error = abs(y_exact(xend)-y_num[-1])            # Error at the end point
    print(format('{:.3e}   {:.3e}'.format( h, error)))
    h = 0.5*h                                       # Reduce the stepsize
```

The table generated from this code shows that whenever the step size is reduced with a factor of 0.5, so is the error. Therefore, we expect

$$|y(x_{end}) - y_N| \approx Ch, \qquad h = \frac{x_{end} - x_0}{N}.$$

The method seems to be of order 1, see *Preliminaries*, section 3.1.

In the following we will prove that this is in fact the case. The error analysis will be done on a scalar equation, but it can as well be extended to systems of equations.

5.1 Local and global errors

In this discussion we have to consider two kinds of errors:

- *Local truncation error* d_{n+1}: This is the error done on one step, starting from $(x_n, y(x_n))$.
- *Global error* e_n: This is the difference between the exact and the numerical solution after n steps, that is $e_n = y(x_n) - y_n$.

In the following, we will see how to express the local truncation error, and we will see how the global and the local errors are related. We will use all this to find an upper bound for the global error at the end point $x_N = x_{end}$. The technique described here is quite standard for this type of error analysis.

Let us start with the local truncation error. Euler's method is nothing but the first two terms of the Taylor expansion of the exact solution. As a consequence, the local truncation error is the remainder term $R_2(x)$ (see *Preliminaries*, section 4):

$$d_{n+1} = y(x_n + h) - \big(y(x_n) + hy'(x_n)\big) = \frac{1}{2}h^2 y''(\xi), \qquad \xi \in (x_n, x_n + h).$$

Next, we use the fact that $y'(x_n) = f(x_n, y(x_n))$ and obtain the following two expressions:

$$\begin{aligned} y(x_n + h) &= y(x_n) + hf(x_n, y(x_n)) + d_{n+1}, && \text{the equation above} \\ y_{n+1} &= y_n + hf(x_n, y_n), && \text{Euler's method} \end{aligned}$$

We subtract the second from the first, use that $e_n = y(x_n) - y_n$, and finally use Result 3 in *Preliminaries*, section 5. This yields the expression

$$e_{n+1} = e_n + h\big(f(x_n, y(x_n)) - f(x_n, y_n)\big) + d_{n+1} = e_n + hf_y(x_n, \eta)e_n + d_{n+1},$$

where $f_y = \frac{\partial f}{\partial y}$, and η is some value between y_n and $y(x_n)$. We now take the absolute value on each side, and apply the triangle inequality:

$$|e_{n+1}| = |e_n + hf_y(x_n, \eta)e_n + d_{n+1}| \le |e_n| + h|f_y(x_n, \eta)||e_n| + |d_{n+1}|.$$

Assume now that there exist positive constants D and L satisfying

$$|f_y(x, y)| \le L \qquad \text{and} \qquad |y''(x)| \le 2D$$

for all values of x, y. From the inequality above we get that

$$|e_{n+1}| \le (1 + hL)|e_n| + Dh^2.$$

Since $y_0 = y(x_0)$ we get $e_0 = 0$. The inequality above therefore results in the following estimates for the global errors:

$$\begin{aligned} &|e_1| \le Dh^2 \\ &|e_2| \le (1 + hL)|e_1| + Dh^2 \le \big((1 + hL) + 1\big)Dh^2 \\ &|e_3| \le (1 + hL)|e_2| + Dh^2 \le \big((1 + hL)^2 + (1 + hL) + 1\big)Dh^2 \\ &\quad\vdots \\ &|e_N| \le (1 + hL)|e_{N+1}| + Dh^2 \le \sum_{n=0}^{N-1}(1 + hL)^n Dh^2 \end{aligned}$$

We will now apply two well known results:

- The sum of a truncated geometric series:

  $$\sum_{n=0}^{N-1} r^n = \frac{r^N - 1}{r - 1} \text{ for } r \in \mathbb{R}.$$

- The series of the exponential:

  $$e^x = 1 + x + \frac{1}{2}x^2 + \cdots = 1 + x + \sum_{n=2}^{\infty} \frac{x^n}{n!}$$

which proves that $1 + x < e^x$ whenever $x > 0$.

Using these results, we obtain that

$$\sum_{n=0}^{N-1}(1+hL)^n = \frac{(1+hL)^N - 1}{(1+hL)-1} < \frac{(e^{hL})^N - 1}{hL} = \frac{e^{hLN}-1}{hL} = \frac{e^{L(x_{end}-x_0)}-1}{hL},$$

where the last equality holds because $(x_{end} - x_0) = hN$. Finally, we plug this into the inequality for $|e_N|$ above, and we see that we have proved the the following upper bound for the global:

$$|y(x_{end}) - y_N| = |e_N| \leq \frac{e^{L(x_{end}-x_0)}-1}{L} Dh = Ch,$$

where the constant $C = \frac{e^{L(x_{end}-x_0)}-1}{L} D$ depends on the length of the integration interval $x_{end} - x_0$, of certain properties of the equation (L and D), but *not* on the step size h.

The numerical solution converges to the exact solution since

$$\lim_{N\to\infty} |e_N| = 0.$$

If the step size is reduced by a factor of 0.5, so will the error. This is in agreement with the previous numerical result.

Remark: Following the proof of the error estimate for Euler's method, we see that a local truncation error of size h^2 leads to a final, global error of size h. Intuitively, this is because we need to take roughly $1/h$ steps in order to reach the point x_{end}, although a precise proof is quite a bit more complicated than that, as we have seen. However, what this also should mean is that a method with a truncation order of size h^{p+1} should lead to a global error of h^p. The results of the next section show that, under certain conditions, this is indeed the case.

5.2 A general convergence result

A one-step method applied to a system of ODEs $\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x))$ can be written in the generic form

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{\Phi}(x_n, \mathbf{y}_n; h),$$

where the *increment function* $\mathbf{\Phi}$ typically depends on the function $\mathbf{f}$ and some parameters defining the method.

Definition: Order of a method.

A method is of order $p > 0$ if there is a constant $C > 0$ such that

$$\|\mathbf{e}_N\| = \|\mathbf{y}(x_{end}) - \mathbf{y}_N\| \leq Ch^p,$$

where N is the number of steps taken to reach x_{end} using the step size $h = (x_{end} - x_0)/N$.

The local truncation error $\mathbf{d}_{n+1}$ of such a method is

$$\mathbf{d}_{n+1} = \mathbf{y}(x_{n+1}) - (\mathbf{y}(x_n) + h\mathbf{\Phi}(x_n, \mathbf{y}(x_n); h))$$

Replace the absolute values in the above proof with norms (*Preliminaries*, section 1), and the argument above can be used to prove the following:

Theorem: Convergence of one-step methods.

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

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

and the local truncation error satisfies

$$\|\mathbf{y}(x+h) - (\mathbf{y}(x) + h\mathbf{\Phi}(x, \mathbf{y}(x), h))\| \le Dh^{p+1}$$

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

$$\|\mathbf{e}_N\| = \|\mathbf{y}(x_{end}) - \mathbf{y}_N\| \le Ch^p, \qquad \text{with } C = \frac{e^{M(x_{end}-x_0)} - 1}{M} D.$$

It can be proved that the first of these conditions is satisfied for all the methods that will be considered here, provided that the function $\mathbf{f}$ is continuously differentiable.

Heun's method. We will now discuss a first, improved alternative to Euler's method:

Assume that we want to solve an

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)).$$

Its exact solution can be written in integral form as

$$\mathbf{y}(x_n + h) = \mathbf{y}(x_n) + \int_{x_n}^{n_n+h} \mathbf{y}'(x)\, dx = \mathbf{y}(x_n) + \int_{x_n}^{x_n+h} \mathbf{f}(x, \mathbf{y}(x)) dx.$$

We now replace the integral on the right hand side by a numerical approximation using the trapezoidal rule:

$$\mathbf{y}(x_n + h) \approx \mathbf{y}(x_n) + \frac{h}{2}\big(\mathbf{f}(x_n, \mathbf{y}(x_n)) + \mathbf{f}(x_{n+1}, \mathbf{y}(x_{n+1}))\big).$$

Then we replace $\mathbf{y}(x_n)$ and $\mathbf{y}(x_{n+1})$ by the approximations $\mathbf{y}_n$ and $\mathbf{y}_{n+1}$. The resulting method is the *trapezoidal rule for ODEs*, given by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}\big(\mathbf{f}(x_n, \mathbf{y}_n) + \mathbf{f}(x_{n+1}, \mathbf{y}_{n+1})\big).$$

This is an example of an *implicit* method. The numerical approximation $\mathbf{y}_{n+1}$ appears on both sides of this equation as is therefore only implicitly given: If $x_n, \mathbf{y}_n$ is known, a nonlinear equation has to be solved in order to find $\mathbf{y}_{n+1}$, and this has to be done for each step. There are important applications where this actually makes sense, which we will discuss in a later lecture in the context of *stiff ODEs*.

However, for the moment we want to avoid this additional complication and thus replace the $\mathbf{y}_{n+1}$ on the right hand side by some approximation. One natural possibility here is to apply one step of Euler's method. This results in Heun's method:

$$\begin{aligned}
\mathbf{u}_{n+1} &= \mathbf{y}_n + h\mathbf{f}(x_n, \mathbf{y}_n), \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{2}\big(\mathbf{f}(x_n, \mathbf{y}_n) + \mathbf{f}(x_{n+1}, \mathbf{u}_{n+1})\big).
\end{aligned}$$

The method is commonly written in the form

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(x_n, \mathbf{y}_n), \\
\mathbf{k}_2 &= \mathbf{f}(x_n + h, \mathbf{y}_n + h\mathbf{k}_1), \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2).
\end{aligned}$$

The increment function for this method is

$$\mathbf{\Phi}(x, \mathbf{y}; h) = \frac{1}{2}\big(\mathbf{f}(x, \mathbf{y}) + \mathbf{f}(x + h, \mathbf{y} + h\mathbf{f}(x, \mathbf{y}))\big).$$

Implementation. One step of Heuns's method is implemented as follows:

```
def heun(f, x, y, h):
    # One step of Heun's method
    k1 = f(x, y)
    k2 = f(x+h, y+h*k1)
    y_next = y + 0.5*h*(k1+k2)
    x_next = x + h
    return x_next, y_next
```

Numerical example 5: Let us compare the numerical solution from Euler's and Heun's methods on the scalar test problem

$$y' = -2xy, \qquad y(0) = 1$$

with the exact solution $y(x) = e^{-x^2}$ on the interval $[0, 1]$. Use $h = 0.1$ for Euler's method and $h = 0.2$ for Heun's metode. Thus both require a total of 10 function evaluations, and the total amount of computational work is comparable.

```
# Numerical experiment 5

def f(x, y):            # The right hand side of the ODE
    return -2*x*y

def y_exact(x):         # The exact solution
    return exp(-x**2)

h = 0.1                 # The stepsize
x0, xend = 0, 1         # Integration interval
y0 = 1                  # Initial value

# Solve the equations
xn_euler, yn_euler = ode_solver(f, x0, xend, y0, h, method=euler)
xn_heun, yn_heun = ode_solver(f, x0, xend, y0, 2*h, method=heun)

# Plot the solution
x = linspace(x0, xend, 101)
plot(xn_euler, yn_euler, 'o')
plot(xn_heun, yn_heun, 'd')
plot(x, y_exact(x))
legend(['Euler','Heun','Exact']);
xlabel('x')
ylabel('y');
```

The errors of the two approximations are:

```
# Plot the error of the two methods
semilogy(xn_euler, abs(y_exact(xn_euler)- yn_euler), 'o');
semilogy(xn_heun, abs(y_exact(xn_heun)- yn_heun), 'd');
xlabel('x')
ylabel('Error')
legend(['Euler', 'Heun'],loc=3)
```

Let us finally compare the error at x_{end} when the two methods are applied to our test problem, for different values of h:

```
# Print the error as a function of h.
print('Error in Euler and Heun\n')
print('h           Euler       Heun')
print('----------------------------------')
for n in range(10):
    x_euler, y_euler = ode_solver(f, x0, xend, y0, h, method=euler)
    x_heun, y_heun = ode_solver(f, x0, xend, y0, 2*h, method=heun)
    error_euler = abs(y_exact(xend)-y_euler[-1])
    error_heun = abs(y_exact(xend)-y_heun[-1])
    print(format('{:.3e}   {:.3e}   {:.3e}'.format( h, error_euler, error_heun)))
    h = 0.5*h
```

First of all, Heun's method is significantly more accurate than Euler's method, even when the number of function evaluations are the same. Further, we notice that the error in Heun's method is reduced by a factor of approximately 1/4 whenever the step size is reduced by a factor 1/2, indicating that the error $|y(x_{end} - y_N| \approx Ch^2$, and the method is of order 2.

Numerical example 6: Solve the Lotka-Volterra equation from Numerical example 2 by Euler's and Heun's methods, again using twice as many steps for Euler's method than for Heun's method.

- Use $h = 0.01$ for Euler's method and $h = 0.02$ for Heun's method.
- Use $h = 0.1$ for Euler's method and $h = 0.2$ for Heun's method.

```
# Numerical example 6

def lotka_volterra(x, y):       # The Lotka-Volterra equation
    alpha, beta, delta, gamma = 2, 1, 0.5, 1        # Parameters
    dy = array([alpha*y[0]-beta*y[0]*y[1],
                delta*y[0]*y[1]-gamma*y[1]])
    return dy

x0, xend = 0, 20
y0 = array([2, 0.5])
h = 0.01

x_euler, y_euler = ode_solver(lotka_volterra, x0, xend, y0, h, method=euler)
x_heun, y_heun = ode_solver(lotka_volterra, x0, xend, y0, 2*h, method=heun)

plot(x_euler,y_euler)
plot(x_heun, y_heun, '--')
xlabel('x')
title('Lotka-Volterra ligningen')
legend(['y1 (Euler)','y2', 'y1 (Heun)', 'y2'],loc=2)
```

Numerical exercises:

1. Solve Van der Pol's equation by use of Heun's method. Experiment with different choices of the step size h and compare with the results from Numerical experiment 3.
2. Implement the classical Runge–Kutta method and verify numerically that the order of the method is 4. The method is given by

$$
\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(x_n, \mathbf{y}_n) \\
\mathbf{k}_2 &= \mathbf{f}\left(x_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\right) \\
\mathbf{k}_3 &= \mathbf{f}\left(x_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\right) \\
\mathbf{k}_4 &= \mathbf{f}(x_n + h, \mathbf{y}_n + h\mathbf{k}_3) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4).
\end{aligned}
$$

Convergence properties of Heun's method. To prove convergence and to find the order of a method two things are needed:

- the local truncation error, expressed as a power series in the step size h

- the condition $\|\mathbf{\Phi}(x,\mathbf{y};h)-\mathbf{\Phi}(x,\mathbf{z};h)\| \leq M\|\mathbf{y}-\mathbf{z}\|$

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

In the following, we will use the notation

$$f_x=\frac{\partial f}{\partial x},\qquad f_y=\frac{\partial f}{\partial y},\qquad f_{xx}=\frac{\partial^2 f}{\partial x^2}\qquad f_{xy}=\frac{\partial^2 f}{\partial x\partial y}\qquad \text{etc.}$$

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

The Taylor expansion of the exact solution $y(x+h)$ is given by

$$y(x+h)=y(x)+hy'(x)+\frac{h^2}{2}y''(x)+\frac{h^3}{6}y'''(x)+\dots.$$

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

$$\begin{aligned}
y'(x)&=f,\\
y''(x)&=f_x+f_yy'=f_x+f_yf,\\
y'''(x)&=f_{xx}+f_{xy}y'+f_{yx}f+f_{yy}y'f+f_yf_x+f_yf_yy'=f_{xx}+2f_{xy}f+f_{yy}f^2+f_yf_x+(f_y)^2f.
\end{aligned}$$

We then 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

$$y(x_0+h)=y_0+hf+\frac{h^2}{2}(f_x+f_yf)+\frac{h^3}{6}(f_{xx}+2f_{xy}f+f_{yy}f^2+f_yf_x+(f_y)^2f)+\dots,$$

where f and all its derivatives are evaluated in (x_0,y_0). For the numerical solution we get

$$\begin{aligned}
k_1&=f(x_0,y_0)=f,\\
k_2&=f(x_0+h,y_0+hk_1)\\
&=f+hf_x+f_yhk_1+\frac{1}{2}f_{xx}h^2+f_{xy}hhk_1+\frac{1}{2}f_{yy}h^2k_1^2+\dots\\
&=f+h(f_x+f_yf)+\frac{h^2}{2}(f_{xx}+2f_{xy}f+f_{yy}f^2)+\dots,\\
y_1&=y_0+\frac{h}{2}(k_1+k_2)=y_0+\frac{h}{2}\Big(f+f+h(f_x+f_yf)+\frac{h^2}{2}(f_{xx}+2f_{xy}f+f_{yy}f^2)+\dots\Big)\\
&=y_0+hf+\frac{h^2}{2}(f_x+f_yf)+\frac{h^3}{4}(f_{xx}+2f_{xy}f+f_{yy}f^2)+\dots,
\end{aligned}$$

and the local truncation error will be

$$d_1=y(x_0+h)-y_1=\frac{h^3}{12}(-f_{xx}-2f_{xy}f-f_{yy}f^2+2f_yf_x+2(f_y)^2f)+\dots.$$

The first nonzero term in the local truncation error series is called *the principal error term*. For h 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 $x_n, y(x_n)$ will give similar results, and it is possible to conclude that, given sufficient differentiability of f, there is a constant D such that

$$|d_n|\leq Dh^3.$$

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

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

The increment function for Heun's method is given by

$$\Phi(x,y) = \frac{1}{2}\big(f(x,y) + f(x+h, y+hf(x,y))\big).$$

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

$$\begin{aligned}
|\Phi(x,y) - \Phi(x,z)| &= \frac{1}{2}|f(x,y) + f(x+h,y+hf(x,y)) - f(x,z) - hf(x+h,z+hf(x,z))| \\
&\le \frac{1}{2}\big(|f(x,y) - f(x,z)| + |f(x+h,y+hf(x,y)) - f(x+h,z+hf(x,z))|\big) \\
&\le \frac{1}{2}\big(L|y-z| + L|y+hf(x,y) - z - hf(x,z)|\big) \\
&\le \frac{1}{2}\big(2L|y-z| + hL^2|y-z|\big) \\
&= \Big(L + \frac{h}{2}L^2\Big)|y-z|.
\end{aligned}$$

Assuming that the step size h is bounded above by some H, we can conclude that

$$|\Phi(x,y) - \Phi(x,z)| \le M|y-z|, \qquad M = L + \frac{H}{2}L^2.$$

In conclusion: Heun's method is convergent of order 2.

6 Error estimation and step size control

To control the global error $y(x_n) - y_n$ is notoriously difficult, and far beyond what will be discussed in this course. To control the local error in each step and adjust the step size accordingly is rather straightforward, as we will see.

6.1 Error estimation

Given two methods, one of order p and the other of order $p+1$ or higher. Assume we have reached a point $(x_n, \mathbf{y}_n)$. One step forward with each of these methods can be written as

$$\begin{aligned}
\mathbf{y}_{n+1} &= \mathbf{y}_n + h\mathbf{\Phi}(x_n, \mathbf{y}_n; h), && \text{order } p, \\
\widehat{\mathbf{y}}_{n+1} &= \mathbf{y}_n + h\widehat{\mathbf{\Phi}}(x_n, \mathbf{y}_n; h), && \text{order } p+1 \text{ or more.}
\end{aligned}$$

Let $\mathbf{y}(x_{n+1}; x_n, \mathbf{y}_n)$ be the exact solution of the ODE through $(x_n, \mathbf{y}_n)$. We would like to find an estimate for *the local error* $\mathbf{l}_{n+1}$, that is, the error in one step starting from $(x_n, \mathbf{y}_n)$,

$$\mathbf{l}_{n+1} = \mathbf{y}(x_{n+1}; x_n, \mathbf{y}_n) - \mathbf{y}_{n+1}.$$

As we already have seen, the local error is found by finding the power series in h of the exact and the numerical solution. The local error is of order p if the lowest order terms in the series where the exact and the numerical solution differ is of order $p+1$. So the local errors of the two methods are

$$\begin{aligned}
\mathbf{y}(x_{n+1}; x_n, \mathbf{y}_n) - \mathbf{y}_{n+1} &= \mathbf{\Psi}(x_n, y_n)h^{p+1} + \dots, \\
\mathbf{y}(x_{n+1}; x_n, \mathbf{y}_n) - \widehat{\mathbf{y}}_{n+1} &= \qquad\qquad\qquad\quad + \dots,
\end{aligned}$$

where $\Psi(x_n, y_n)$ is a term consisting of method parameters and differentials of $\mathbf{f}$ and $\dots$ contains all the terms of the series of order $p+2$ or higher. Taking the difference gives

$$\widehat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} = \mathbf{\Psi}(x_n, \mathbf{y}_n)h^{p+1} + \dots.$$

Assume now that h is small, such that the *principal error term* $\mathbf{\Psi}(\mathbf{x_n}, \mathbf{y_n})h^{p+1}$ dominates the error series. Then a reasonable approximation to the unknown local error $\mathbf{l}_{n+1}$ is the *local error estimate* $\mathbf{le}_{n+1}$:

$$\mathbf{le}_{n+1} = \widehat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} \approx \mathbf{y}(x_{n+1}; x_n, \mathbf{y}_n) - \mathbf{y}_{n+1}.$$

Example 4: Apply Euler's method of order 1 and Heun's method of order 2 with $h = 0.1$ to the equation

$$y' = -2xy, \qquad y(0) = 1.$$

Use this to find an approximation to the error after one step.

Euler's method:

$$y_1 = 1.0 - 0.1 \cdot 2 \cdot 0 \cdot 1.0 = 1.0.$$

Heun's method

$$\begin{aligned} k_1 &= -2 \cdot 0.0 \cdot 1.0 = 0.0, \\ k_2 &= -2 \cdot 0.1 \cdot (1 + 0.0) = -0.2, \\ \widehat{y}_1 &= 1.0 + \frac{0.1}{2} \cdot (0.0 - 0.2) = 0.99. \end{aligned}$$

The error estimate and the local error are respectively

$$le_1 = \widehat{y}_1 - y_1 = -10^{-2}, \qquad l_1 = y(0.1) - y_1 = e^{-0.1^2} - 1.0 = -0.995 \cdot 10^{-2}.$$

So in this case the error estimate is a quite decent approximation to the actual local error.

6.2 Stepsize control

The next step is to control the local error, that is, choose the step size so that $\|\mathbf{le}_{n+1}\| \leq \text{Tol}$ for some given tolerance Tol, and for some chosen norm $\|\cdot\|$.

Essentially:

Given $x_n, \mathbf{y}_n$ and a step size h_n.

- Do one step with the method of choice, and find an error estimate $\mathbf{le}_{n+1}$.
- if $\|\mathbf{le}\|_{n+1} < \text{Tol}$

  Accept the solution $x_{n+1}, \mathbf{y}_{n+1}$.

  If possible, increase the step size for the next step.

- else

  Repeat the step from $(x_n, \mathbf{y}_n)$ with a reduced step size h_n.

In both cases, the step size will change. But how?

From the discussion above, we have that

$$\|\mathbf{le}_{n+1}\| \approx D h_n^{p+1}.$$

where $\mathbf{le}_{n+1}$ is the error estimate we can compute, D is some unknown quantity, which we assume almost constant from one step to the next. What we want is a step size h_{new} such that

$$\text{Tol} \approx D h_{new}^{p+1}.$$

From these two approximations we get:

$$\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|} \approx \left(\frac{h_{new}}{h_n}\right)^{p+1} \qquad \Rightarrow \qquad h_{new} \approx \left(\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|}\right)^{\frac{1}{p+1}} h_n.$$

That is, if the current step h_n was rejected, we try a new step h_{new} with this approximation. However, it is still possible that this new step will be rejected as well. To avoid too many rejected steps, it is therefore common to be a bit restrictive when choosing the new step size, so the following is used in practice:

$$h_{new} = P \cdot \left(\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|}\right)^{\frac{1}{p+1}} h_n.$$

where the *pessimist factor* $P < 1$ is some constant, normally chosen between 0.5 and 0.95.

6.3 Implementation

We have all the bits and pieces for constructing an adaptive ODE solver based on Euler's and Heuns's methods. There are still some practical aspects to consider:

- The combination of the two methods, implemented in `heun_euler` can be written as

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(x_n, \mathbf{y}_n), \\
\mathbf{k}_2 &= \mathbf{f}(x_n + h, \mathbf{y}_n + h\mathbf{k}_1), \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + h\mathbf{k}_1, && \text{Euler} \\
\widehat{\mathbf{y}}_{n+1} &= \mathbf{y}_n + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2), && \text{Heun} \\
\mathbf{le}_{n+1} &= \|\widehat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1}\| = \frac{h}{2}\|\mathbf{k}_2 - \mathbf{k}_1\|.
\end{aligned}$$

- Even if the error estimate is derived for the lower order method, in this case Euler's method, it is common to advance the solution with the higher order method, since the additional accuracy is for free.
- Adjust the last step to be able to terminate the solutions exactly in x_{end}.
- To avoid infinite loops, add some stopping criteria. In the code below, there is a maximum number of allowed steps (rejected or accepted).
- The main driver `ode_adaptive` is written to make it simple to test other pairs of methods. This is also the reason why the function `heun_euler` returns the order of the lowest order method.

```python
def heun_euler(f, x, y, h):
    '''
    One step with the pair Heun/Euler
    Input: the function f, the present state xn and yn  and the stepsize h
    Output: the solution x and y in the next step, error estimate, and the
            order p of Eulers method (the lowest order)
    '''

    k1 = f(x, y)
    k2 = f(x+h, y+h*k1)
    y_next = y + 0.5*h*(k1+k2)      # Heuns metode (local extrapolation)
    x_next = x + h
    error_estimate = 0.5*h*norm(k2-k1)   # The 2-norm or the error estimate
    p = 1
    return x_next, y_next, error_estimate, p
```

```python
def ode_adaptive(f, x0, xend, y0, h0, tol = 1.e-6, method=heun_euler):
    '''
    Adaptive solver for ODEs
       y' = f(x,y), y(x0)=y0

    Input: the function f, x0, xend, and the initial value y0
           intial stepsize h, the tolerance tol,
           and a function (method) implementing one step of a pair.
    Output: Array with x- and y- values.
    '''

    y_num = array([y0])    # Array for the solutions y
    x_num = array([x0])    # Array for the x-values

    xn = x0                # Running values for  x, y and the stepsize h
    yn = y0
    h = h0
    Maxcall = 100000        # Maximum allowed calls of method
    ncall = 0
    # Main loop
    while xn < xend - 1.e-10:               # Buffer for truncation error
        # Adjust the stepsize for the last step
        if xn + h > xend:
            h = xend - xn

        # Perform one step with the chosen mehtod
        x_try, y_try, error_estimate, p = method(f, xn, yn, h)
        ncall = ncall + 1

        if error_estimate <= tol:
            # Solution accepted, update x and y
            xn = x_try
            yn = y_try
            # Store the solutions
            y_num = concatenate((y_num, array([yn])))
            x_num = append(x_num, xn)

        # else: The step is rejected and nothing is updated.

        # Adjust the stepsize
        h = 0.8*(tol/error_estimate)**(1/(p+1))*h

        # Stop with a warning in the case of max calls to method
        if ncall > Maxcall:
            print('Maximum number of method calls')
            return x_num, y_num

    # Some diagnostic output
    print('Number of accepted steps = ', len(x_num)-1)
    print('Number of rejected steps = ', ncall - len(x_num)+1)
    return x_num, y_num
```

Numerical example 7: Apply the code on the test equation:

$$y' = -2xy, \qquad y(0) = 1.$$

```
# Numerical example 7
def f(x, y):
    return -2*x*y

def y_exact(x):
    return exp(-x**2)

h0 = 100
x0, xend = 0, 1
y0 = 1

x_num, y_num = ode_adaptive(f, x0, xend, y0, h0, tol=1.e-3)

plot(x_num, y_num, '.-', x_num, y_exact(x_num))
title('Adaptive Heun-Euler')
xlabel('x')
ylabel('y')
legend(['Numerical', 'Exact']);
```

The error $|y(x_n) - y_n|$ is:

```
# Plot the error from the adaptive method
error = abs(y_exact(x_num) - y_num)
```

```
semilogy(x_num, error, '.-')
title('Error in Heun-Euler for dy/dt=-2xy')
xlabel('x');
```

And the step size will change like

```
# Plot the step size sequence
h_n = diff(x_num)            # array with the stepsizes h_n = x_{n+1}
x_n = x_num[0:-1]            # array with x_num[n], n=0..N-1
semilogy(x_n, h_n, '.-')
xlabel('x')
ylabel('h')
title('Stepsize variations');
```

Numerical exercises:

1. Solve the Lotka-Volterra equation, use for instance $h_0 = 0.1$ and $\mathrm{Tol} = 10^{-3}$. Notice also how the step size varies over the integration interval.
2. Repeat the experiment using Van der Pol's equation.

6.4 Runge–Kutta methods

Euler's and Heun's method are both examples of *explicit Runge-Kutta methods* (ERK). Such schemes are given by

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(x_n, \mathbf{y}_n), \\
\mathbf{k}_2 &= \mathbf{f}(x_n + c_2 h, \mathbf{y}_n + h a_{21}\mathbf{k}_1), \\
\mathbf{k}_3 &= \mathbf{f}\big(x_n + c_3 h, \mathbf{y}_n + h(a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2)\big), \\
&\ \ \vdots \\
\mathbf{k}_s &= \mathbf{f}\Big(x_n + c_s h, \mathbf{y}_n + h\sum_{j=1}^{s-1} a_{sj}\mathbf{k}_j\Big), \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + h\sum_{i=1}^{s} b_i \mathbf{k}_i,
\end{aligned}$$

where c_i, a_{ij} and b_i are coefficients defining the method. We always require $c_i = \sum_{j=1}^{s} a_{ij}$. Here, s is the number of *stages*, or the number of function evaluations needed for each step. The vectors $\mathbf{k}_i$ are called stage derivatives. Also implicit methods, like the trapezoidal rule,

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}\big(\mathbf{f}(x_n, \mathbf{y}_n) + \mathbf{f}(x_n + h, \mathbf{y}_{n+1})\big)$$

can be written in a similar form,

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(x_n, \mathbf{y}_n), \\
\mathbf{k}_2 &= \mathbf{f}\Big(x_n + h, \mathbf{y}_n + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2)\Big), \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2).
\end{aligned}$$

But, contrary to what is the case for explicit methods, a nonlinear system of equations has to be solved to find $\mathbf{k}_2$.

Definition: Runge–Kutta methods.

An s-stage Runge-Kutta method is given by

$$\begin{aligned}\mathbf{k}_i &= \mathbf{f}\big(x_n + c_i h, \mathbf{y}_n + h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j\big), \qquad i = 1, 2, \cdots, s, \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + h\sum_{i=1}^{s} b_i \mathbf{k}_i.\end{aligned}$$

The method is defined by its coefficients, which are given in a *Butcher tableau*

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

with

$$c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \cdots, s.$$

The method is *explicit* if $a_{ij} = 0$ whenever $j \geq i$; otherwise it is *implicit.*

A Runge–Kutta methods with an error estimate are usually called *embedded Runge–Kutta methods* or *Runge–Kutta pairs*, and the coefficients can be written in a Butcher tableau as follows

$$\begin{array}{c|ccccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} & \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} & \\ \vdots & \vdots & & & \vdots & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} & \\ \hline & b_1 & b_2 & \cdots & b_s & \text{Order } p \\ \hline & \widehat{b}_1 & \widehat{b}_2 & \cdots & \widehat{b}_s & \text{Order } p+1 \end{array}.$$

The error estimate is then given by

$$\mathbf{le}_{n+1} = h\sum_{i=1}^{s}(\widehat{b}_i - b_i)\mathbf{k}_i.$$

Example 5: The Butcher-tableaux for the methods presented so far are

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

Euler Heun trapezoidal rule

and the Heun-Euler pair can be written as

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

A particular mention deserves also the classical Runge-Kutta method from a previous numerical exercise, which can be written as

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

See this list of Runge–Kutta methods for more.

Order conditions for Runge–Kutta methods. It can be proved that a Runge–Kutta method is of order p 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.

Example 6: Apply the conditions to Heun's method, for which $s = 2$ and the Butcher tableau is

$$\begin{array}{c|cc} c_1 & a_{11} & a_{12} \\ c_2 & a_{21} & a_{22} \\ \hline & b_1 & b_2 \end{array} = \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} .$$

The order conditions are:

$p = 1$ $$b_1 + b_2 = \frac{1}{2} + \frac{1}{2} = 1$$ OK

$p = 2$ $$b_1 c_1 + b_2 c_2 = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$$ OK

$p = 3$ $$b_1 c_1^2 + b_2 c_2^2 = \frac{1}{2} \cdot 0^2 + \frac{1}{2} \cdot 1^2 = \frac{1}{2} \neq \frac{1}{3}$$ Not satisfied

$$\begin{aligned} b_1(a_{11}c_1 + a_{12}c_2) + b_2(a_{21}c_1 + a_{22}c_2) &= \frac{1}{2}(0 \cdot 0 + 0 \cdot 1) + \frac{1}{2}(1 \cdot 0 + 0 \cdot 1) \\ &= 0 \neq \frac{1}{6} \end{aligned}$$ Not satisfied

The method is of order 2.

6.5 A summary of some terms and definitions

There have been quite a few definitions and different error terms in this note. So let us list some of them (not exclusive):

Definitions:

$\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$	the ODE
$\mathbf{y}(x\,;\,x^*, \mathbf{y}^*)$,	the exact solution of the ODE through $(x^*, \mathbf{y}^*)$
$\mathbf{y}(x) = \mathbf{y}(x\,;\,x_0, \mathbf{y}_0)$,	the exact solution of $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$, $\mathbf{y}(x_0) = \mathbf{y}_0$
$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{\Phi}(x_n, y_n; h)$,	one step of the method

Let $\mathbf{\Phi}$ represent a method of order p and $\hat{\mathbf{\Phi}}$ a method of order $p+1$.

Error concepts:

$\mathbf{d}_{n+1} = \mathbf{y}(x_n + h\,;\,x_n, \mathbf{y}(x_n)) - \Big(\mathbf{y}(x_n) + h\mathbf{\Phi}(x_n, \mathbf{y}(x_n); h)\Big)$,	the local truncation error
$\mathbf{l}_{n+1} = \mathbf{y}(x_n + h\,;\,x_n, \mathbf{y}_n) - \Big(\mathbf{y}_n + h\mathbf{\Phi}(x_n, \mathbf{y}_n; h)\Big)$,	the local error
$\mathbf{le}_{n+1} = h\Big(\hat{\mathbf{\Phi}}(x_n, \mathbf{y}_n; h) - \mathbf{\Phi}(x_n, \mathbf{y}_n; h)\Big)$,	the local error estimate, $\quad \mathbf{le}_{n+1} \approx \mathbf{l}_{n+1}$
$\mathbf{e}_n = \mathbf{y}(x_n) - \mathbf{y}_n$	the global error