Øving 12 - Numeriske metoder for differensiallikninger I - LF

Obligatoriske oppgaver

1 Given the equation

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

with excact solution

$$y(x) = \frac{2}{2 - x^2}, \text{ for } x \in [0, 2)$$

we want to approximate $y(1)$ by a forward Euler method and a Runge-Kutta 4 - method. Denoting

$$f(x, y) = xy^2$$

we want to solve the equation

$$\frac{dy}{dx} = f(x, y), \qquad y(0) = 1.$$

For a small time-step $h = 1/N$, where $N \in \{1, 2, 3, \dots\}$ is a fixed number, we approximate by

$$\frac{y_{n+1} - y_n}{h} = f(x_n, y_n),$$

where

$$x_n = nh, y_0 = y(0).$$

We get the scheme

$$\begin{cases} y_{n+1} = y_n + hf(nh, y_n), \\ y_0 = y(0). \end{cases}$$

Then y_N is the approximate value of $y(1)$. The error is given by

$$|y_N - y(1)|,$$

and below we have implemented this scheme into Python. We also have implemented a Runge-Kutta method (RK4), and computed error estimates for this method. For both methods we have found the size of $h > 0$ such that

$$|y_N - y(1)| \leq 10^{-5}.$$

Results:

1. Step size forward Euler: 10^{-6}.

2. Step size RK4: 0.1

This is evidence that the RK4-method is far better than the forward Euler method.

Python:

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

# eksakt lsning
def y(x):
    return 2.0/(2.0-x**2)

# vi skriver om likningen saa vi faar den paa formen y'=f(x,y)
def f(x,y):
    return x*np.power(y,2)

# regner ut euler-approksimasjonen i x=1 med steglengde h
def euler(x,y,n):
    h = 1.0/n
    for i in range(n):
        y = y + h*f(x,y)
        x = x+h
    return y

def rk4(x,y,n):
    h = 1.0/n
    for i in range(n):
        k1 = f(x,y)
        k2 = f(x+h/2.0,y+k1*h/2.0)
        k3 = f(x + h / 2.0, y + k2 * h / 2.0)
        k4 = f(x+h,y+h*k3)
        y = y + h/6.0*(k1+2*k2+2*k3+k4)
        x = x+h
    return y

# I koden under har vi valgt aa oke antall steg med en faktor paa 10 hver ...
    gang. Grunnen til at dette er aa foretrekke
# framfor aa oke antall steg med en av gangen er at euler konvergerer ...
    veldig tregt, saa aa ke antall steg med en tar
# veldig lang tid. Her faar vi ikke nyaktig svar paa hvor mange steg vi ...
    trenger, men dette gir oss ikke saa mye mer
# informasjon vi faar ved aa se paa narmeste tierpotens uansett.

if __name__ == "__main__":
    x0 = 0
    y0 = 1
    err = 1
    tol = 10.0**-5
    n_e = 0.1
    yn = 0
    while err>tol:
        n_e = int(n_e * 10)
        yn = euler(x0,y0,n_e)
        err = np.abs(y(1)-yn)
    h_e = 1.0/n_e
    n_r = 0.1
    err = 1
    while err>tol:
        n_r = int(n_r * 10)
```

```
        yn = rk4(x0,y0,n_r)
        err = np.abs(y(1)-yn)
    h_r = 1.0/n_r
    print("Step-size euler:",h_e)
    print("Step-size rk4:",h_r)
```

2 We study the differential equation

$$\frac{dy}{dx} = f(x, y),$$

where

$$f(x, y) = -30y.$$

Implicit Euler.

The implicit Euler method is given by

$$\frac{y_{i+1} - y_i}{h} = f(x_{i+1}, y_{i+1}).$$

Note that we evaluate f in the points (x_{i+1}, y_{i+1}). Inserting our special f yields the scheme

$$y_{i+1} = y_i + h(-30y_{i+1}),$$

or

$$y_{i+1} = \frac{1}{1 + 30h} y_i.$$

Forward Euler.

The forward Euler scheme is given by

$$y_{i+1} = y_i - 30hy_i = (1 - 30h)y_i.$$

The schemes.

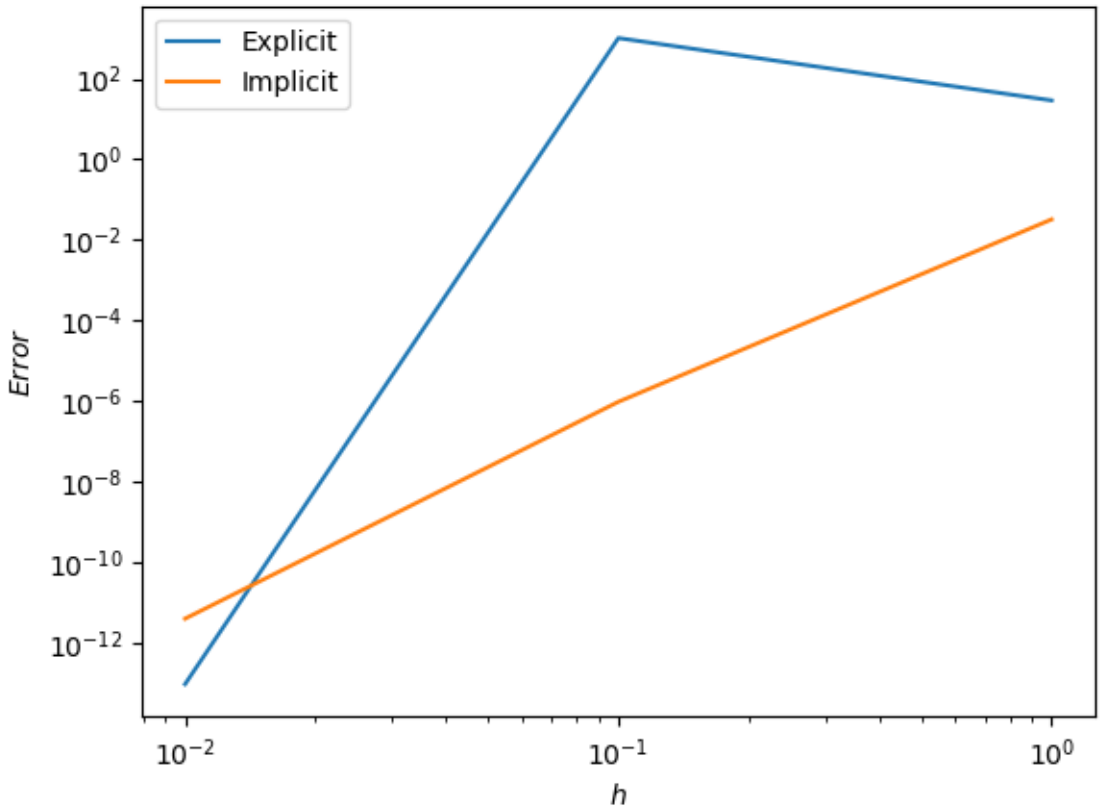
1. Forward Euler: $y_{i+1} = (1 - 30h)y_i$.
2. Implicit Euler: $y_{i+1} = \frac{1}{1+30h} y_i$.

Both schemes have initial value:

$$y_0 = y(0) = 1.$$

Implementation.

We implemented the two schemes in Python. We get the error plot of Figure 1. As seen from the plot, the implicit Euler gives smaller error than the forward Euler for most of the step sizes h.



Figur 1: Error plot.

Python:

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

# eksakt losning
def y_e(x):
    return np.exp(-30*x)

def f(x,y):
    return -30*y

def explicit_euler(x,y,n):
    h = 1.0/n
    for i in range(n):
        y = y + h*f(x,y)
        # x = x+h
    return y

def implicit_euler(x,y,n):
    h = 1.0/n
    for i in range(n):
        y = y/(1+30*h)
        # x = x+h
```

```
    return y

if __name__ == "__main__":
    x0 = 0
    y0 = 1
    err_e = np.zeros(3)
    err_i = np.zeros(3)
    for i in range(3):
        n = 10**i
        y = explicit_euler(x0,y0,n)
        err_e[i] = np.abs(y-y_e(1))
    for i in range(3):
        n = 10**i
        y = implicit_euler(x0,y0,n)
        err_i[i] = np.abs(y-y_e(1))

    # plotter feilene mot h
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot([1,0.1,0.01],err_e,label=r'Explicit')
    ax.plot([1,0.1,0.01],err_i,label=r'Implicit')
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.set_xlabel(r'$h$')
    ax.set_ylabel(r'$Error$')
    ax.legend()
    plt.show()
```

3 We study the differential equation

$$\frac{dy}{dx} = f(x, y),$$

where

$$f(x, y) = y^2 - y^3.$$

Trapezoidal method.

The general trapezoidal rule is given by

$$y_{i+1} = y_i + \frac{h}{2}\Big(f(x_i, y_i) + f(x_{i+1}, y_{i+1})\Big).$$

Inserting our special f, we get

$$y_{i+1} = y_i + \frac{h}{2}\Big(y_{i+1}^2 - y_{i+1}^3 + y_i^2 - y_i^3\Big)$$

Rearranging we get

$$0 = -y_{i+1} + \frac{h}{2}\Big(y_{i+1}^2 - y_{i+1}^3 + y_i^2 - y_i^3\Big) := G(y_{i+1}). \tag{1}$$

Think now of y_i as a constant, a given value, while y_{i+1} is the unknown. We want to solve the equation (denoting $y = y_{i+1}$)

$$G(y) = -y + \frac{h}{2}\Big(y^2 - y^3 + y_i^2 - y_i^3\Big) = 0$$

and to do so, we use the Newton method. We have

$$\frac{dG}{dy} = -1 + \frac{h}{2}\Big(2y - 3y^2\Big),$$

and thus the Newton iterations are given by

$$\begin{aligned} y^{n+1} &= y^n - \frac{G(y)}{G'(y)} \\ &= y^n - \frac{-y + \frac{h}{2}\Big(y^2 - y^3 + y_i^2 - y_i^3\Big)}{-1 + \frac{h}{2}\Big(2y - 3y^2\Big)}. \end{aligned}$$

In the code, we pick $y^0 = y_i$ as a reasonable start value. Note that we need to solve (1) for each iteration of the numerical scheme!

Heuns method.

Here we write

$$y_{i+1} = y_i + \frac{h}{2}\Big(f(x_i, y_i) + f(x_{i+1}, y_{i+1}^*)\Big),$$

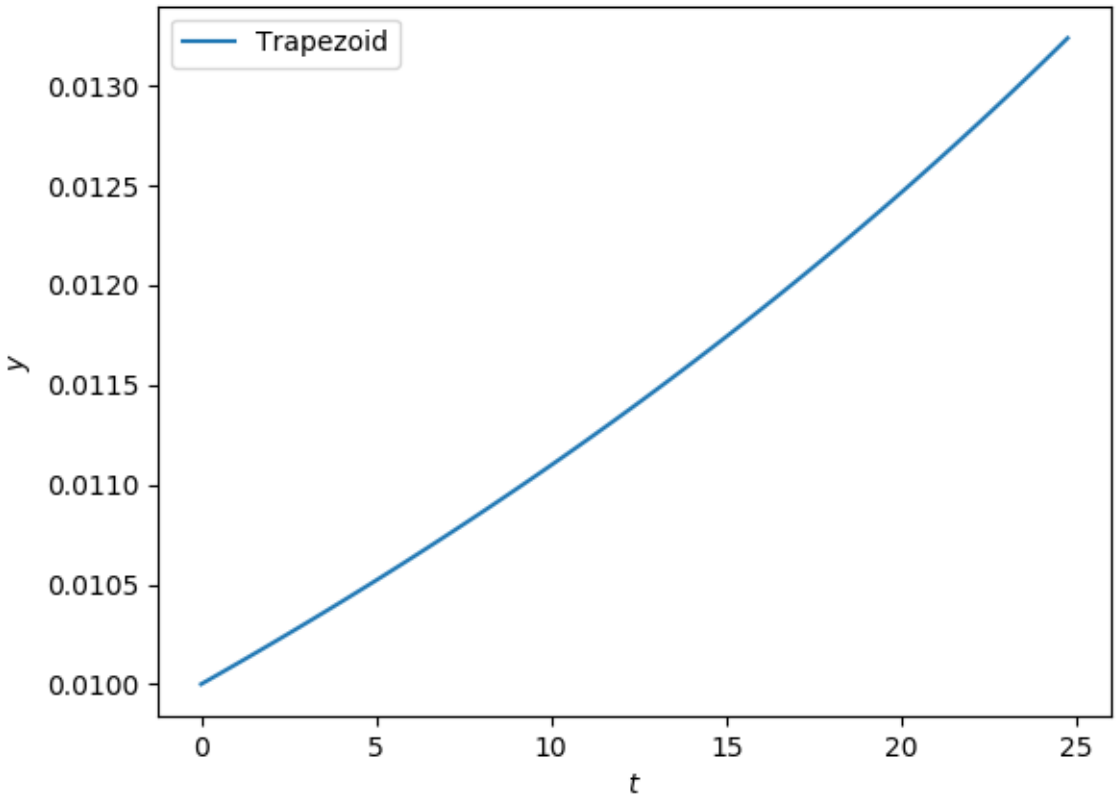
as before, but we let

$$y_{i+1}^* = y_i + hf(x_i, y_i).$$

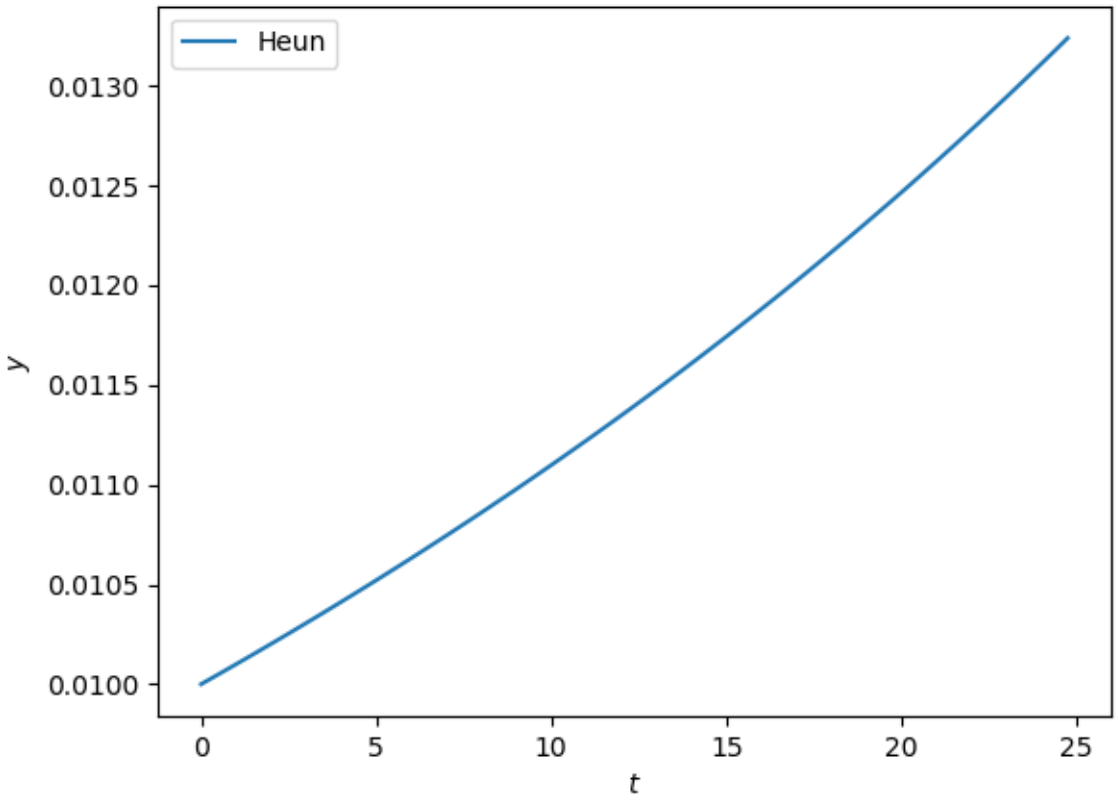
This yields

$$y_{i+1} = y_i + \frac{h}{2}\Big(f(y_i + hf(y_i)) + f(y_i)\Big).$$

Writing out the whole scheme for our f takes a lot of space, so we will not do it. However, implementing the scheme in Python is less complicated, as shown in the code below. Running the code, we get the following plots



Figur 2: Trapezoid rule.



Figur 3: Heun.

Python:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt

def f(y):
    return y**2-y**3

# med implisitt trap

def newton(z,h): # z er y_i og blir startgjetningen vaar for forste ...
    iterasjon hver gang denne kalles.
    tol = 10.0**-6
    err = 1.0
    y = x =  z
    while err>tol:
        y = y - ( z-y+0.5*h* ( z**2-z**3+y**2-y**3 ) ...
            )/(-1.0+0.5*h*(2*y-3*y**2) )
        err = np.abs(y-x)
        x = y  # x er forrige verdi av y_i i iterasjonen, brukes for aa ...
            sjekke feilen
    return y
```

```
# denne er ogsaa kjent som den eksplisitte trapesmetoden
def heun(t0,y0,n):
    h = 25/n
    y = np.zeros(n)
    t = np.zeros(n)
    y[0] = y0
    t[0] = t0
    for i in range(n-1):
        y[i+1] = y[i]+0.5*h*(f(y[i])+f( y[i] + h * f(y[i]) ) )
        t[i+1] = t[i]+h
    return t,y


# dette er den vanlige implisitte trapesmetoden
def trapezoid(t0,y0,n):
    h = 25/n
    y = np.zeros(n)
    t = np.zeros(n)
    y[0] = y0
    t[0] = t0
    for i in range(n-1):
        y[i+1] = newton(y[i],h)
        t[i+1] = t[i]+h
    return t,y

def plotfigure(t,y,lab):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot(t,y,label=lab)
    ax.set_xlabel(r'$t$')
    ax.set_ylabel(r'$y$')
    ax.legend()
    return ax

if __name__ == "__main__":
    t0 = 0
    y0 = 0.01
    n = 100
    t,y = trapezoid(t0,y0,n)
    plotfigure(t,y,r'Trapezoid')
    t,y = heun(t0,y0,n)
    plotfigure(t,y,r'Heun')
    plt.show()
```

Anbefalte oppgaver

1 We study systems of ODE's on the form

$$\begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \end{bmatrix} = A \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, \tag{2}$$

where A is a 2×2 matrix.

Forward Euler. We approximate (2) by

$$\begin{bmatrix} \frac{x_{n+1}-x_n}{h} \\ \frac{y_{n+1}-y_n}{h} \end{bmatrix} = A \begin{bmatrix} x_n \\ y_n \end{bmatrix},$$

which yields

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + h \cdot A \begin{bmatrix} x_n \\ y_n \end{bmatrix}.$$

Setting $x_0 = y_0 = 1$, and $h > 0$ this approximates the solution of (2).

Backward Euler.

We do as before.

$$\begin{bmatrix} \frac{x_{n+1}-x_n}{h} \\ \frac{y_{n+1}-y_n}{h} \end{bmatrix} = A \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix}.$$

This yields

$$\Big(I - hA\Big) \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix}. = \begin{bmatrix} x_n \\ y_n \end{bmatrix},$$

where I is the identity matrix. Setting $x_0 = y_0 = 1$, and $h > 0$ this approximates the solution of (2).

Other methods.

The principle for writing the other methods is more or less the same as shown here. You should try it!!

The actual solution.

To finish the exercise we need to write **a)**, **b)** and **c)** on the form (2). We have

$$\textbf{a)} \quad A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \qquad \textbf{b)} \quad A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \qquad \textbf{c)} \quad A = \begin{bmatrix} 1 & 0 \\ -1 & 2 \end{bmatrix}.$$

Code.

Try to implement this scheme in your favorite programming language!

2 We define $y_1(t) := y(t)$ and $y_2(t) := y'(t)$. Using vector notation and the equation, we get

$$\begin{bmatrix} y_1'(t) \\ y_2'(t) \end{bmatrix} = \begin{bmatrix} y'(t) \\ y''(t) \end{bmatrix} = \begin{bmatrix} y_2(t) \\ -\sin(y_1(t)) \end{bmatrix}. \tag{3}$$

This is actually a quite complicated expression, since $\sin(x)$ is a non-linear function.

Forward Euler.

The forward Euler method is straight forward:

$$\begin{bmatrix} y_1^{n+1} \\ y_2^{n+1} \end{bmatrix} = \begin{bmatrix} y_1^n \\ y_2^n \end{bmatrix} + h \cdot \begin{bmatrix} y_2^n \\ -\sin(y_1^n) \end{bmatrix}.$$

Implicit Euler.

If we assume that y_1 is very small, then we can approximate $\sin(y_1) \approx y_1$ (Taylor-series). This leads to a system of ODE's of the form (2), with

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

This can be solved by methods described earlier.

Comments on initial values.

Recall from physics that $y(t)$ describes the angle of the pendulum at time t, and $y'(t)$ describes the angular velocity at time t (or something like this). It is natural to assume that initial velocity is $y'(0) = 0$, and that the start position is small (remember the linearization!!!), for example $y(0) = 0.1$. However, you are free to choose your favorite initial conditions.