Miniproject 3

Linus Falk

October 30, 2022

Introduction

In this miniproject we implement numerical methods in Python to solve ODE's. We take a look on explicit methods as well as implicit and invesitigate the computational order of convergence for the different methods.

Task 1

In this section we solve IVP using Eulers explicit method (or RK1). This method only depend on the current and past steps [1]:

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

An example of implementation of Eulers method to solve initial value problems (IVP) in Python code could look like this:

```
def Euler_exp(t0, y0, tn, n):
    h = (tn - t0) / n
    res = []
    for j in range(n):
        k1 = h * f1(t0, y0)

        yn = y0 + k1
        res.append(yn)
        y0 = yn
        t0 = t0 + h
    return res
```

Here t0 and y0 are the given initial values, tn is the final time and n is the number of steps (which depend on the step size h). We also have the function f1 that is given in the IVP.

We are solving these three IVP:

- a) $y'(t) = te^{-t} - y(t) =$, 0 $\leq$t $\leq$ 10, y(0) = 1, with exact solution $y(t) = (1 + 0.5t^2)e^{-t}$
- b) $y'(t) = \left[\cos(y(t))\right]^2$, 0 $\leq$t $\leq$ 10, y(0) = 0, wtih exact solution $y(t) = \tan^{-1}(t)$
- c) $y'(t) = \frac{t^3}{y(t)}$, 0 $\leq$t $\leq$ 10, y(0) = 1, with exact solution $y(t) = \sqrt{0.5t^4 + 1}$

We examine the order if convergence by calculating the exact solution at the end time tn and calculate the absolute value of the error with the approximation for different step size h. We examine the convergence by the error and step size in logarithmic scale. By calculating the slope of these results we obtain the order of convergence.

```
stp = []
s = 0.2
for x in range(6):
    stp.append(s)
    s = s/2

t0 = 0
y0 = 1
tn = 10
analyticres = (1+0.5*tn**2)*math.exp(-tn)

result1=[]; logscale =[]
for k in range(len(stp)):
    n = int((tn - t0) / stp[k])
    result1.append(math.log(abs(Euler_exp(t0, y0, tn,
        n)[-1] - analyticres)))
    logscale.append(math.log(stp[k]))


print(result1[-1]-result1[0])/(logscale[-1]-logscale
    [0]))
```

By changing f1 and the initial values we can compare the convergence of the solutions with the following result

- a) RK1 order of convergence: 0.9744
- b) RK1 order of convergence: 0.9980
- c) RK1 order of convergence: 0.9977

Task 2

In this task we are asked to solve the IVP from the previous task with the change of using Euler backward and the trapezoidal method. Eulers backward method

with order 1 in time is a implicit method, meaning that we have the looked for approximation of the next step on both sides in the equation, see below [2]:

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

To solve this approximation there is need to solve a non linear system also. An implementation in Python could look like this:

```
def func(x,*data):
    t0, h, y0 = data
    return x - y0 - h * f1(t0+h,y0)

def RK1_imp(t0, y0, tn, n):
    h = (tn - t0) / n
    res = []
    for j in range(n):
        data = (t0, h, y0)
        y = fsolve(func,y0, args=data)[0]
        res.append(y)
        y0 = y
        t0 += h
    return res
```

The trapezoidal method is also an implicit method but with a order of 2 i time. With this method we approximate the solution by integrating the equation:

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

We obtain an equation with the approximation of the next step on both sides of the equation and we therefore need to solve a non linear system again for the next step. Python implementation could look similar to Euler implicit:

```
def func2(x,*data):
    t0, h, y0 = data
    return x - y0 - h/2 * (f1(t0,y0)+f1(t0+h,x))

def trap(t0, y0, tn, n):
    h = (tn - t0) / n
    res = []
    for j in range(n):
        data = (t0, h, y0)
        y = fsolve(func2,y0, args=data)[0]
        res.append(y)
        y0 = y
        t0 += h
    return res
```

We use the method of calculating the slope as in task 1 and obtain the following result for theses two methods:

- **a)** Euler implicit method order of convergence: 0.9933
- **a)** Trapezoidal method order of convergence: 1.9979
- **b)** Euler implicit order of convergence: 0.9980
- **b)** Trapezoidal method order of convergence: 2.0004
- **c)** Euler implicit order of convergence: 1.0027
- **c)** Trapezoidal method order of convergence: 2.0065

The result corresponds to the theoretical order of 1 for Euler implicit and 2 for the trapezoidal method.

Task 3

Here we are asked to solve IVP b) from task 1 with the Taylor series method and calculate the order of convergence and compare it to Euler implicit and Trapezoidal method. We begin with a description of the Taylor series method, second and third order [3]:

$$\begin{aligned} y_{k+1} &= y_0 + hf(t_k, y_k) + \frac{h^2}{2} f'(t_k, y_k) \\ y_{k+1} &= y_0 + hf(t_k, y_k) + \frac{h^2}{2} f'(t_k, y_k) \frac{h^3}{6} f''(t_k, y_k) \\ f'' &= f_x + f_y y' \end{aligned} \tag{4}$$

Implementing this in Python code with our function $f = \cos^2(y(t))$ gives us:

```
def taylor1(t0, y0, tn, n):
    h = (tn - t0) / n
    res = []
    for j in range(n):
        yn= y0 + h * f(t0,y0)-(h**2/2) * 2*math.sin(y0
            )*(math.cos(y0))**3
        res.append(yn)
        y0 = yn
        t0 = t0 + h
    return res


def taylor2(t0, y0, tn, n):
    h = (tn - t0) / n
    res = []
    for j in range(n):
        yn=y0+h*f(t0,y0)- (h**2/2) * 2 *math.sin(y0)*(
            math.cos(y0))**3 + (h**3/6) *
```

```
            (-2*(math.cos(y0))**6 + 6*(math.cos(y0))**4*(
                math.sin(y0))**2)
            res.append(yn)
            y0 = yn
            t0 = t0 + h
        return res
```

We use the same method from previous task to calculate the convergence order and get the following result:

- taylor1 order of convergence: 2.0223
- taylor2 order of convergence: 3.0925

We see that we get an increase of 1 and two orders with first and second derivative is included in Taylor 2 and 3 comparing to Eulers method.

Task 4

In this task we are asked to solve the following IVP:

$$y'(t) = -y(t) + t^{0.1}(1.1 + t), \quad y(0) = 0 \tag{5}$$

with the exact solution:

$$y(t) = t^{1.1} \tag{6}$$

The method of choice is a method of order 2 in time, Runge-Kutta 2 and a Python implementation is presented below:

```
def RK2(t0, y0, tn, n):
    h = (tn - t0) / n
    res = []
    for j in range(n):
        k1 = h * f(t0, y0)
        k2 = h * f(t0 + h/2, y0 + k1/2)

        yn = y0 + k2
        res.append(yn)
        y0 = yn
        t0 = t0 + h
    return res
```

In contrary to previous tasks we are also asked to compute the error convergence at different stop time (tn): 1, 2, 3, 4, 5. The process of calculating the order is same as in previous tasks except for this. The result is presented below:

- RK2 (tn=1) order of convergence: 1.5461
- RK2 (tn=2) order of convergence: 1.5677
- RK2 (tn=3) order of convergence: 1.5966
- RK2 (tn=4) order of convergence: 1.6412

- RK2 (tn=5) order of convergence: 1.7078

The order should be 2 but since its not we can start to suspect that the IVP is stiff. Testing the problem with RK4 and with different stop times gives very different orders of convergence in line with what we could see in the case with a stiff problem.

References

[1] Wikipedia. *Eulers method*, 2022. https://en.wikipedia.org/wiki/Euler_method.

[2] Wikipedia. *Trapezoidal*, 2022. https://en.wikipedia.org/wiki/Trapezoidal_rule_(differential_equations).

[3] King Saud University. *Numerical methods*, 2022. https://faculty.ksu.edu.sa/sites/default/files/l10_2.pdf.