Exercise #+∞

November 2022

In this sheet, you can find several implementation-related problems that are of the same form as the ones that could be given in the exam. **This exercise should not be handed in!**

Problem 1.

Consider the following implementation of a quadrature rule:

```
import numpy as np

[a, b] = 0.5, 1

def f(x):
    return  np.exp(-x**2)

S = (f(a)+4*f(.5*(a+b))+f(b))/6
```

a) The formula implemented in the last line has a mistake that will lead to the wrong output `S`. Rewrite that line with the correct formula.

b) Once the mistake is corrected, what numerical method will be implemented?

c) After correcting the mistake, what value will the output `S` have?

Solution.

a) The quadrature formula is missing the length $(b-a)$ of the interval: we should actually have
`S = (b-a)*(f(a)+4*f(.5*(a+b))+f(b))/6`.

b) After correcting the mistake, we will have an implementation for Simpson's rule.

c) We will get

$$S = (b-a)\frac{f(a)+4f((a+b)/2)+f(b)}{6} = (1-0.5)\frac{e^{-0.5^2}+4e^{-0.75^2}+e^{-1^2}}{6} \approx 0.2854843$$

Problem 2.

Consider the following implementation of an iterative method:

```
from numpy import cos, sin, log

x = 0.5
err = abs(sin(x)+log(x))

while err > 1e-6:
    dx = -(sin(x)+log(x))/(cos(x)+1/x)
    x = x + dx
    err = abs(dx)

print(x)
```

a) What method is implemented above?

b) Write down the specific equation being (iteratively) solved by the algorithm.

Solution.

a) Newton's method (Newton–Raphson).

b) The equation being solved is $\sin x + \log x = 0$, or $\sin x = -\log x$.

Problem 3.

We consider the solution of the equation

$$\cos(\cos(x)) = \frac{3x}{2}.$$

a) Show that this equation has a unique solution $\hat{x} \in \mathbb{R}$.

b) If you run the code below, you will obtain a value x that is a numerical approximation of $\hat{x}$. Provide an upper bound for the error $e := |\hat{x} - x|$.

```
import numpy as np

def g(x):
    return 2*np.cos(np.cos(x))/3

x = 0
x_old = 1

while np.abs(x_old-x) > 1e-6:
    x_old = x
    x = g(x)

print(x)
```

Solution.

a) We rewrite this equation as the fixed point equation

$$x = g(x) \qquad \text{with } g(x) = \frac{2}{3}\cos(\cos(x)).$$

Then

$$g'(x) = \frac{2}{3}\sin(\cos(x))\sin(x).$$

Since $|\sin(y)| \leq 1$ for all y, it follows that

$$|g'(x)| \leq \frac{2}{3}$$

for all x. That is, g is a contraction with contraction factor $L = 2/3$. Now the fixed point theorem implies that the equation $g(x) = x$, which is equivalent to $\cos(\cos(x)) = 3x/2$, has a unique solution $\hat{x}$.

b) This code is an implementation of the fixed point iteration

$$x_{k+1} = g(x_k)$$

with the function

$$g(x) = \frac{2}{3}\cos(\cos(x)).$$

Moreover, the while-loop stops as soon as the difference between the current and the previous iterate is at most 10^{-6}, that is, when

$$|x - x_{\text{old}}| \leq 10^{-6}.$$

As shown in part a), the function g is a contraction with contraction factor $L = \frac{2}{3}$. The a-posteriori estimate thus implies that

$$|x - \hat{x}| \leq \frac{L}{1-L}|x - x_{\text{old}}|.$$

With $L = 2/3$ and $|x - x_{\text{old}}| \leq 10^{-6}$, we obtain

$$|x - \hat{x}| \leq \frac{2/3}{1-2/3}10^{-6} = 2 \cdot 10^{-6}.$$

Problem 4.

We consider the numerical solution of the of ODEs

$$y'(t) = -at\,y(t) + b, \qquad y(0) = y_0.$$

Here $a > 0$, $b \in \mathbb{R}$ and $y_0 \in \mathbb{R}$ are given numbers.

The following code tries to implement a function that returns the numerical result after N steps of the implicit Euler method for this problem with step length h. However, it contains two errors. Find them!

```
import numpy as np

def iEuler(a,b,y0,h,N):
    y = y0
    t = 0
    for n in range(N):
        y = (y+b)/(1+t*h*a)
        t = t+h
    return(y)
```

Solution.

The implicit Euler method for this equation reads

$$y_{n+1} = y_n + h(-at_{n+1}y_{n+1} + b).$$

Solving for y_{n+1} yields the explicit expression

$$y_{n+1} = \frac{y_n + hb}{1 + aht_{n+1}}. \tag{1}$$

Moreover, we have the update for the time

$$t_{n+1} = t_n + h$$

and the initialisation $t_0 = 0$.

We now look at the actual implementation: The function takes as input the values a, b, $y0$, h, and N, which is fine. Then it initialises y with $y0$, and t with 0, which is fine as well. After that, the function performs N iterations of the for-loop, and finally returns the last value of y as output. All of this appears to be reasonable, so we have to take a look at what happens within the for-loop.

First, we have the update

$$y \leftarrow \frac{y + b}{1 + tha}, \tag{2}$$

and afterwards we update

$$t \leftarrow t + h.$$

What we see immediately when we compare (1) and (2) is that the enumerator should be $y + hb$ instead of $y + b$.

Next we see that the value of t is only updated after the update of y. This means that the update of y is calculated with the old value t_n instead of with t_{n+1}. For instance, in the first step (for $n = 0$), we should compute (since $t_1 = h$)

$$y_1 = \frac{y_0 + hb}{1 + aht_1} = \frac{y_0 + hb}{1 + ah^2}.$$

However, in the code above, the value of t is still equal to 0, and thus we obtain instead the wrong value

$$\tilde{y}_1 = \frac{y_0 + hb}{1 + ah \cdot 0} = y_0 + hb.$$

In order to correct this error, we should update the value of t before the value of y, that is, switch the order of the two lines within the for-loop.

This gives us the following corrected code (you do not have to provide the corrected code in order to get full marks in the exam, unless the question explicitly asks for it):

```
import numpy as np

def iEuler(a,b,y0,h,N):
    y = y0
    t = 0
    for n in range(N):
        t = t+h
        y = (y+h*b)/(1+t*h*a)
    return(y)
```

Problem 5.

Consider the following implementation of a certain Runge–Kutta method:

```
import numpy as np
from numpy import sin, exp

y0 = np.array([pi/3, 0])
T = 10
h = 0.1
ys = [y0]
ts = [0]

def f(t,y):
    f = np.array([y[1], (exp(-t)-1)*sin(y[0])])
    return f

while(ts[-1] < T):
    t, y = ts[-1], ys[-1]
    k1 = f(t,y)
    k2 = f(t+h/2, y+h*k1/2)
    ys.append(y + h*k2)
    ts.append(t + h)
```

a) Write down the Butcher tableau for the method implemented above.

b) Write down the initial value problem being solved in the code.

c) Rewrite this initial value problem as a scalar, second-order ODE (with corresponding initial conditions).

Solution.

a) From the code, we can write

$$\begin{aligned} k_1 &= f(t_n, y_n) = f(t_n + 0 \cdot h,\, y_n + 0 \cdot hk_1 + 0 \cdot hk_2) \quad \Rightarrow \quad c_1 = a_{11} = a_{12} = 0\,. \\ k_2 &= f(t_n + h/2,\, y_n + hk_1/2) \\ &= f(t_n + 0.5h,\, y_n + h[0.5k_1 + 0 \cdot k_2]) \quad \Rightarrow \quad c_1 = a_{21} = 0.5,\ a_{22} = 0\,. \\ y_{n+1} &= y_n + hk_2 = y_n + 0 \cdot hk_1 + 1 \cdot hk_2 \quad \Rightarrow \quad b_1 = 0,\ b_2 = 1\,. \end{aligned}$$

We therefore get

0	0	0
1/2	1/2	0
	0	1

b) Runge–Kutta methods are designed for first-order problems $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y})$, where $\mathbf{y}$ and $\mathbf{f}$ can be either scalars or vectors (arrays, in Python). From the function $\mathbf{f}$ implemented under **def** `f(t,y):`, we see that

$$\mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} y_2 \\ (\mathrm{e}^{-t} - 1)\sin y_1 \end{bmatrix} \quad \text{(remember that Python starts numbering from 0).}$$

We also see from the code that $y_1(0) = \pi/3$ and $y_2(0) = 0$ (the two entries of the vector $\mathbf{y}_0$). The complete initial value problem is therefore

$$\mathbf{y}'(t) := \begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} y_2 \\ (\mathrm{e}^{-t} - 1)\sin y_1 \end{bmatrix}, \text{ with } \mathbf{y}(0) = \begin{bmatrix} \pi/3 \\ 0 \end{bmatrix}.$$

c) The second equation in the 2×2 system reads $y_2' = (\mathrm{e}^{-t} - 1)\sin y_1$, while the first one is simply $y_1' = y_2$, so that $y_2' = y_1''$. Then, eliminating y_2' gives us

$$y_1'' = (\mathrm{e}^{-t} - 1)\sin y_1\,,$$

or, by denoting y_1 as simply y, the second-order scalar ODE

$$y''(t) + (1 - \mathrm{e}^{-t})\sin y(t) = 0\,, \text{ with } y(0) = \pi/3 \text{ and } y'(0) = 0\,.$$

Problem 6.

We consider the heat equation

$$\partial_t u = \partial_{xx} u \qquad \text{for } 0 < x < 1 \text{ and } t > 0$$

with initial conditions

$$u(x, 0) = x^3 \qquad \text{for } 0 < x < 1$$

and boundary conditions

$$\left.\begin{aligned} u(0, t) &= 0 \\ u(1, t) &= 1 \end{aligned}\right\} \qquad \text{for } t > 0.$$

The following code computes a numerical solution of this problem using the explicit Euler method.

```
import numpy as np

M = 10
h = 1/M
x = np.linspace(0,1,M+1)
k = 1/200
N = 400
r = k/h**2

U = np.zeros([M+1,N+1])

U[:,0] = x**3

for n in range(N):
    U[0,n+1] = 0
    U[1:-1,n+1] = U[1:-1,n] + r*(U[0:-2,n] - 2*U[1:-1,n] + U[2:,n])
    U[-1,n+1] = 1
```

How must the last line be changed, if we want to solve the same problem with the boundary conditions

$$\left.\begin{aligned} u(0, t) &= 0 \\ \partial_x u(1, t) &= 1 \end{aligned}\right\} \qquad \text{for } t > 0$$

instead?

Solution.

Our task is to replace the (Dirichlet) boundary condition $u(1, t) = 1$ with the (Neumann)

boundary condition $\partial_x u(1,t) = 1$. For that, we need to change the computation of U_M^{n+1}, which happens in the last line of the code.

For the computation of U_M^{n+1}, we have to combine the information we get from the PDE with the boundary condition. From the discretisation of the PDE, we get the expression

$$U_M^{n+1} = U_M^n + \frac{k}{h^2}\left(U_{M-1}^n - 2U_M^n + U_{M+1}^n\right). \tag{3}$$

By discretising the (Neumann) boundary condition

$$\partial_x u(1,t) = 1$$

with a central difference we obtain the condition

$$\frac{U_{M+1}^n - U_{M-1}^n}{2h} = 1.$$

Solving this for U_{M+1}^n, we get

$$U_{M+1}^n = U_{M-1}^n + 2h.$$

We now insert this into (3) and obtain

$$U_M^{n+1} = U_M^n + \frac{k}{h^2}\left(2U_{M-1}^n - 2U_M^n + 2h\right) = U_M^n + 2\frac{k}{h^2}\left(U_{M-1}^n - U_M^n + h\right).$$

With the abbreviation $r = k/h^2$ (which is also used in the given code), one possible implementation would thus replace the last line in the code above with the following line:

```
    U[-1,n+1] = U[-1,n] + 2*r*(U[-2,n] - U[-1,n] + h)
```