Final Exam (A)		
EPFL	**Course:**	Numerical Analysis - Sections SIE-GC
	Lecturer:	Pablo Antolín, Espen Sande
	Date:	15/01/2024 **Duration:** 3h 00
Sciper:	**Student:**	**Section:**

Evaluation table				
Ex.1	**Ex.2**	**Ex.3**	**Ex.4**	**Total**

PLEASE READ CAREFULLY ALL THE INSTRUCTIONS OF THE EXAM

WRITE YOUR SURNAME, NAME, SCIPIER on every page, PLEASE!

All Python code and all the results (the plots and images if they are requested, too) MUST BE TRANSCRIBED ON THESE SHEETS, which must be submitted at the end of the exam.

It is not possible to submit the Python/Jupyter code electronically.

Scratch paper is provided for your personal notes, but it will not be considered for the evaluation of the exam. It must be returned at the end of the exam.

Exercise 3 is of multiple-choice type and this is the only case throughout this exam where it is not necessary to justify your answer.

Write everything with a blue or black pen.

AUTHORIZED MATERIAL

The only authorized material is: an handwritten formulary on a double-sized A4 sheet.

No other sheets, notes or books (paper or electronics), or calculator, mobile phone, tablet, laptop or other electronic devices. The access to Internet (e-mail, websites) is obviously prohibited.

Informations for identification (log-in):

Username and password: `Your GASPAR account`

Virtual machine:

```
SB-MATH-EXAM
```

Sciper: **Student:** **Section:**

Exercise 1 (1.4 points)

a) The iteration matrix for the Jacobi method is given by $B_J = I - D^{-1}A$, where D is the diagonal matrix with entries $d_{ii} = a_{ii}$ for $i = 1, ..., n$. In this case, we have $D = 4I$ and consequently $D^{-1} = \frac{1}{4}I$. Therefore, the iteration matrix reads

$$B_J = I - \frac{1}{4}A = \begin{bmatrix} 0 & -\frac{\theta}{4} & & & \\ -\frac{1}{4} & 0 & -\frac{\theta}{4} & & \\ & \ddots & \ddots & \ddots & \\ & & -\frac{1}{4} & 0 & -\frac{\theta}{4} \\ & & & -\frac{1}{4} & 0 \end{bmatrix}.$$

Following the hint we can write

$$\lambda_i(B_J) = 2\sqrt{\frac{\theta}{16}}\cos\left(\frac{\pi i}{n+1}\right) = \frac{1}{2}\sqrt{\theta}\cos\left(\frac{\pi i}{n+1}\right).$$

Then,

$$\rho(B_J) = \max_{1\leq i\leq n} |\lambda_i(B_J)| = \frac{1}{2}\sqrt{\theta}\max_{1\leq i\leq n}\left|\cos\left(\frac{\pi i}{n+1}\right)\right| = \frac{1}{2}\sqrt{\theta}\left|\cos\left(\frac{\pi}{n+1}\right)\right| = \frac{1}{2}\sqrt{\theta}\left|\cos\left(\frac{\pi n}{n+1}\right)\right|$$

(maximum obtained for $i = 1, n$) and we can conclude that the method converges for

$$\rho(B_J) < 1 \Longleftrightarrow \theta < \frac{4}{\cos\left(\frac{\pi}{n+1}\right)^2}. \tag{1}$$

The spectral radius of the iteration matrix B_J for the Jacobi method is reported in the table below, for different values of n and θ:

	$n = 5$	$n = 100$
$\theta = 2$	0.6124	0.7068
$\theta = 5$	0.9682	1.1175

We observe that the Jacobi method converges for every value of n when $\theta = 2$ and only for $n = 5$ when $\theta = 5$. This result is in agreement with the previous computations. Indeed, we have

$$\theta = 5 \not< \frac{4}{\cos\left(\pi/101\right)^2} = 4.0039$$

for which the condition (1) is not verified.

b) We can solve the linear system by using the following commands:

```
import numpy as np
from examen_aux import jacobi

n = 50
theta = 2

A = 4*np.eye(n) + theta*( np.diag(np.ones(n-1),-1)
                    + np.diag(np.ones(n-1),1) )
b = np.ones(n)
x0 = 0. * b
tol = 1e-6
nmax = 500

xJ,iterJ,resJ = jacobi(A,b,x0,nmax,tol)
```

We get: $iterJ = 48$, $resJ(end) = 5.3190 \cdot 10^{-06}$.

c) We can solve the linear system for different values of n with the following commands

```
import numpy as np
from matplotlib import pyplot as plt
from examen_aux import jacobi

theta = 1.
ns = [4,8,16,32,64]
niter = []

tol = 1e-6
nmax = 20000
for n in ns:
    A = 4*np.eye(n) + theta*( np.diag(np.ones(n-1),-1) + np.diag(np.ones(n-1),1) )
    b = np.ones(n)
    x0 = 0. * b
    _,iterJ,_ = jacobi(A,b,x0,nmax,tol)
    niter += [iterJ]

plt.figure()
plt.semilogx(ns,niter,'r*-')
plt.grid()
plt.show()

theta = 1
ns = list(range(4, 204, 4))
nmax = 500
tol = 1.0e-6
err = []
res = []

for n in ns:
   A = 4.0 * np.eye(n) + theta * np.diag(np.ones(n-1),1)  + np.diag(np.ones(n-1),-1)
   b = 6.0 * np.ones(n)
   b[0] = 5
   b[-1] = 5

   x0 = np.zeros(n)
   x = np.ones(n)

   xJ, _, _ = jacobi(A, b, x0, nmax, tol)

   err.append(np.linalg.norm(x - xJ) / np.linalg.norm(x))
   res.append(np.linalg.norm(b - A @ xJ) / np.linalg.norm(b))



plt.loglog(ns, err)
plt.loglog(ns, res)
plt.grid()
plt.show()
```

Figure 1 contains the obtained plots for the relative error and the normalized residual as a function of n for the Jacobi method.

We can conclude that the residual is a good estimator of the error. Indeed, we can see that the matrix A is well-conditioned for all values of n (the condition number for $n = 200$ is equal to 7).

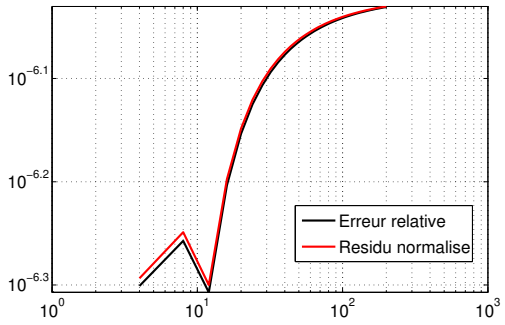

Figure 1: Relative error and normalized residual for different values of n for the Jacobi method.

We recall that the error and the residual at iteration k are linked as follows

$$\frac{\|\mathbf{x} - \mathbf{x^{(k)}}\|}{\|\mathbf{x}\|} \leq \mathcal{K}(A) \frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|},$$

the residual is therefore a good estimator of the error as long as the condition number of the matrix is close to 1.

Exercise 2 (1.4 points)

a) Consider a grid of $M+1$ uniformly spaced points $a = x_0 < x_1 < \ldots < x_M = b$, which defines M intervals of length $h = (b-a)/M$. The composite Simpson quadrature formula $Q_h^{simp}(f)$ to approximate $I(f) = \int_a^b f(x)\,dx$ is

$$Q_h^{simp}(f) = \sum_{i=1}^{M} \frac{h}{6}\left(f(x_{i-1}) + 4f\left(\frac{x_{i-1}+x_i}{2}\right) + f(x_i)\right).$$

The degree of exactness is 3.

b) If $f \in \mathcal{C}^4$, one can show that

$$\left|\int_a^b f(x)dx - Q_h(f)\right| \leq C \max_{x\in[a,b]} |f''''(x)|h^4,$$

with C a constant. Hence, $Q_h^{simp}(f)$ has order 4, as long as f is regular enough.

The fourth derivative of f appears in the constant C.

c) We use the following commands:

```
import numpy as np
from matplotlib import pyplot as plt
from examen_aux import simp
a,b = 0,1

f = lambda x: np.exp(x)*np.sin(x)
exact = .5 * ( 1 + np.exp(1) * (np.sin(1)-np.cos(1)) )

M = 2 ** np.arange(1, 6)
errsimp = []

for i in range(len(M)):
    intsimp = simp( a, b, M[i], f)
    errsimp += [abs(intsimp-exact)]

H = (b-a)/M

plt.figure()
plt.loglog(H, errsimp, 'r*-')
plt.loglog(H,H/100,'k--')
plt.loglog(H,H**2/100,'k^-')
plt.loglog(H,H**4/100,'kd-')
plt.grid()
plt.legend(['error','order 1','order 2','order 4'])
```

We obtain the graph in Figure 1 (left). We see that the convergence order is 4, as predicted by the theory.

d) This time we obtain the graph in Figure 1 (center). The order is no longer 4 but only 1.5, due to the reduced regularity of the function $\sqrt{x}$.

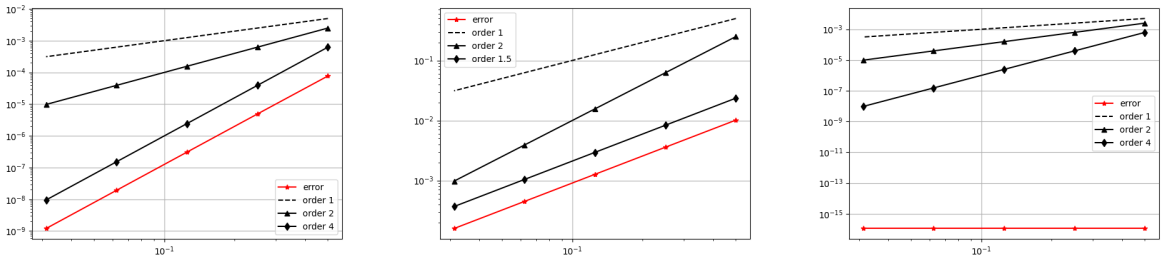

FIGURE 1: Left: error convergence for $f(x) = e^x \sin(x)$. Center: error convergence for $f(x) = \sqrt{x}$. Right: error convergence for $f(x) = \frac{x^3-7x^2+14x-8}{x-4}$.

e) For this function f, we obtain the graph in Figure 1 (right). The error is always very small, around machine precision. This is a consequence of the fact that, in fact, f is a quadratic polynomial:

$$f(x) = \frac{x^3 - 7x^2 + 14x - 8}{x - 4} = \frac{(x-1)(x-4)(x-2)}{x-4} = (x-1)(x-2)$$

Accordingly, the Simpson formula yields the exact result, no matter the choice of M.

Exercise 3 - Multiple choice questions (1.2 points)

Version A

	MC1	MC2	MC3	MC4	MC5	MC6
Answer	b)	a)	a)	b)	a)	c)

Version B

	MC1	MC2	MC3	MC4	MC5	MC6
Answer	e)	b)	d)	c)	c)	e)

Exercise 4 - Implementation (0.5 points)

CODE1 Complete the following Python function which implements the Heun method.

```
import numpy as np
def heun(f,I,u0,N):
    """
    Solve the differential equation:
        u'=f(t,u), t in (t0,T],
        u(t0)=u0
    using the Heun's method on an equispaced grid of stepsize dt=(T-t0)/N.

    Input:
        f: right-hand-side function. It can be evaluated as f(x),
            where x is a scalar value.
        I: integration interval [t0,T].
        u0: initial condition.
        N: number of sub-intervals.

    Output:
        t: vector of time snapshots [t0,t1,...,tN] (with tN=T).
        u: approximation of the solution [u0,u1,...,uN].
        dt: time step.
    """

    dt = (I[1]-I[0])/N
    u = np.zeros(N+1)


    t =  np.linspace(I[0], I[1], N + 1) # TO COMPLETE


    u[0] = u0                           # TO COMPLETE


    for n in range(N):

        # TO COMPLETE below


        z = u[n] + dt * f(t[n], u[n])


        u[n+1] = u[n] + dt / 2 * ( f(t[n], u[n])      # TO COMPLETE
                                 + f(t[n] + dt, z) )




    return t, u, dt
```

CODE2 Complete the following Python function which implements the bisection method.

```
import numpy as np

def bisection( f, a, b, tol, nmax ):
    """
    Tries to find a zero of the continuous function f in the interval [a, b]
    using the bisection method. This function assumes that a zero is found when the
    length of the bisection interval semilength at a certain iteration is smaller
    than the given tolerance.

    Input:
        f: function whose zero is sought. It must be evaluable as f(x),
            where x is a scalar value.
        a, b: Start and end of the interval. They must be such that the values
            of the function f at them have opposite sign.
        tol: Tolerance to be used.
        nmax: Maximum number of iterations to perform.

    Output:
        zero: Computed zero.
        fzero: Value of f evaluated at the computed zero.
        niter: Number of iterations performed.
    """

    fa = f(a)
    fb = f(b)
    niter = 0

    if fa * fb > 0:
        raise Exception("The sign of f at a and b must be different")

    elif np.abs(fa) < tol:
        zero  = a                   # TO COMPLETE
        res   = fa                  # TO COMPLETE
        return zero, res, niter
    elif np.abs(fb) < tol:
        zero  = b                   # TO COMPLETE
        res   = fb                  # TO COMPLETE
        return zero, res, niter

    I = 0.5 * (b - a)       # This is the interval semilength, i.e. the error estimate.

    while I > tol and niter < nmax:
        # TO COMPLETE below
        niter += 1

        zero = 0.5 * (a + b)
        fzero = f(zero)

        if fzero * fa < 0:
            b = zero
            fb = fzero
        elif fzero * fb < 0:
            a = zero
            fa = fzero
        else:
            I = 0

        I = 0.5 * I

    zero = 0.5 * (a + b)
    return zero, f(zero), niter
```