MNUM Exam 2015

Telmo Baptista

January 12, 2020

1 Exercise 1

```python
from math import cos, sin, sqrt, exp, log

def euler(f, t, T, dt):
    return dt * f(t, T)

# $T_a = 37$
dT = lambda t, T: -0.25 * (T-37)

t0 = 5
T0 = 3
h  = 0.4

t = t0
T = T0
for i in range(2):
    T += euler(dT, t, T, h)
    t +=  h

print(f"T = {T:.5f}")
```

Program Output:

```
T = 9.46000
```

2 Exercise 2

- The iterator $x_{n+1} = x_n + i$ is easy to calculate and never looses precision in calculations (discarding when the integer reaches that limit the machine can represent), however this iterator doesn't allow high precision values that are useful for the normal calculations.

  If the calculations involve big numbers, let's say, in the order of the millions 10^6, then, depending on the situation, the error involved on skipping all the real numbers between two consecutive integers x_n and x_{n+1} isn't significant, and in that case this iterator can be really useful as it's the most efficient, as the operation only involves additions or subtractions of integers.

- The iterator $x_{n+1} = x_n + h$ is a bit heavier to calculate compared to the previous one, and there's also chance of loosing precision between calculations, as the representation of floating numbers on the machine isn't perfect. Despite these disadvantages this iterator grants the method higher precision as it can represent the spectrum of the real numbers between x_n and x_{n+1}.

  When working with situations that require high precision on the decimal places, then this iterator results in less error than the previous iterator, at the cost of being heavier to calculate and prone to loss of precision due to floating point representation.

- The iterator $x_{n+1} = x_0 + h * i$ in theory does the same thing as the iterator above, but the operations involved in this one are heavier to calculate, as it requires a multiplication every iteration, and so it's more prone to errors on floating point representation than the iterator above. The advantage of this iterator resides on the fact of being able to get the n-th value of the succession with a single operation instead of needing to do n iterations.

- The iterator $x_{n+1} = x_n + \frac{1}{2^m}$ depends on how the increment, $\frac{1}{2^m}$ is implemented.

  If the parameter m is fixed throughout the whole process and it's calculated on the beginning, then this iterator functions the same way as the second iterator, with the limitation of only having steps approximate to negative powers of 2. However, if the step is calculated in each step, then it adds computational complexity as it needs to calculate both a power and a division, which can lead to precision errors on the value that's being added.

3 Exercise 3

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

$$b = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$$

```python
from math import cos, sin, sqrt, exp, log

def echelon(A, b):
    rows, cols = len(A), len(A[0])

    for i in range(rows):
        pivot = A[i][i]

        for j in range(cols):
            A[i][j] /= pivot
        b[i] /= pivot

        for i2 in range(i+1, rows):
            coef = A[i2][i]
            for j in range(i, cols):
                A[i2][j] -= A[i][j] * coef
            b[i2] -= b[i]*coef

    return A, b



def gauss(A, b):

    rows, cols = len(A), len(A[0])
    echelon(A, b)

    for i in range(rows):
        for j in range(cols):
            print(f"{A[i][j]:.5f}  ", end='\t')
        print(f"{b[i]:.5f}")

    sols = []
    for i in range(rows-1, -1, -1):
        x = b[i]
        for j in range(cols-1, i, -1):
            x -= A[i][j] * sols[cols-1-j]
        sols.append(x)
    sols.reverse()

    return sols

A = [[1, 1/2, 1/3], [1/2, 1/3, 1/4], [1/3, 1/4, 1/5]]

b = [-1, 1, 1]

sols = gauss(A, b)

for i in range(len(sols)):
    print(f"x({i}) = {sols[i]:.5f}")
```

Program Output:

```
1.00000   0.50000   0.33333    −1.00000
0.00000   1.00000   1.00000    18.00000
0.00000   0.00000   1.00000   −30.00000
```

$x(0) = -15.00000$

$x(1) = 48.00000$

$x(2) = -30.00000$

$(A + \delta A).(x + \delta x) = (b + \delta b)$
$\Rightarrow A.\delta x = \delta b - \delta A.x$

Using Maxima:

```
(%i2)DA:da+zeromatrix(3,3);
```

(%o2) $$\begin{pmatrix} da & da & da \\ da & da & da \\ da & da & da \end{pmatrix}$$

```
(%i3)DB:db+zeromatrix(3,1);
```

(%o3) $$\begin{pmatrix} db \\ db \\ db \end{pmatrix}$$

```
(%i4)X:[x0,x1,x2];
```

(%o4) $$[x_0, x_1, x_2]$$

```
(%i5)DB-DA . X;
```

(%o5) $$\begin{pmatrix} -da\,x_2 - da\,x_1 - da\,x_0 + db \\ -da\,x_2 - da\,x_1 - da\,x_0 + db \\ -da\,x_2 - da\,x_1 - da\,x_0 + db \end{pmatrix}$$

```python
from math import cos, sin, sqrt, exp, log
from copy import deepcopy

def echelon(A, b):
    rows, cols = len(A), len(A[0])

    for i in range(rows):
        pivot = A[i][i]

        for j in range(cols):
            A[i][j] /= pivot
        b[i] /= pivot

        for i2 in range(i+1, rows):
            coef = A[i2][i]
            for j in range(i, cols):
                A[i2][j] -= A[i][j] * coef
            b[i2] -= b[i]*coef

    return A, b
```

```python
def gauss(A, b):

    rows, cols = len(A), len(A[0])
    echelon(A, b)

    for i in range(rows):
        for j in range(cols):
            print(f"{A[i][j]:.5f}  ", end='\t')
        print(f"{b[i]:.5f}")

    sols = []
    for i in range(rows-1, -1, -1):
        x = b[i]
        for j in range(cols-1, i, -1):
            x -= A[i][j] * sols[cols-1-j]
        sols.append(x)
    sols.reverse()

    return sols

A = [[1, 1/2, 1/3], [1/2, 1/3, 1/4], [1/3, 1/4, 1/5]]

b = [-1, 1, 1]

sols = gauss(deepcopy(A), b)

print("Internal Stability")
x0, x1, x2 = sols
da = db = 0.05
new_b = [-da*x2-da*x1-da*x0+db,
                 -da*x2-da*x1-da*x0+db,
                 -da*x2-da*x1-da*x0+db]

internal_stab = gauss(deepcopy(A), new_b)

for i in range(len(internal_stab)):
    print(f"x({i}) = {internal_stab[i]:.5f}")
```

Program Output:

```
InternalStability
1.00000   0.50000   0.33333   -0.10000
0.00000   1.00000   1.00000   -0.60000
0.00000   0.00000   1.00000   -3.00000
x(0) = -0.30000
x(1) = 2.40000
x(2) = -3.00000
```

4 Exercise 4

We pretend to find the solution to

$$f(x) = 0$$

Given that

$$f(x) = e^x - 4x^2$$

The root were already isolated and the intervals are:
$X_1 = [-1, 0]$ $X_2 = [0, 1]$ $X_3 = [4, 5]$
Three Picard-Peano expressions are also given:

$$(1) \quad x_{n+1} = \frac{1}{2}\sqrt{e^{x_n}}$$
$$(2) \quad x_{n+1} = \frac{e^{x_n}}{4x_n}$$
$$(3) \quad x_{n+1} = -\frac{1}{2}\sqrt{e^{x_n}}$$

Whose derivatives are:

$$(1) \quad g_1'(x) = \frac{\sqrt{e^x}}{4}$$
$$(2) \quad g_2'(x) = \frac{e^x}{4x} - \frac{e^x}{4x^2}$$
$$(3) \quad g_3'(x) = -\frac{\sqrt{e^x}}{4}$$

Graphing the derivatives with Maxima:

```
(%i2)e1:%e^(x/2)/2;
```

(%o2) $$\frac{e^{\frac{x}{2}}}{2}$$

```
(%i3)e2:%e^x/(4*x);
```

(%o3) $$\frac{e^x}{4\,x}$$

```
(%i4)e3:-%e^(x/2)/2;
```

(%o4) $$-\frac{e^{\frac{x}{2}}}{2}$$

```
(%i5)diff(e1,x);
```

(%o5) $$\frac{e^{\frac{x}{2}}}{4}$$

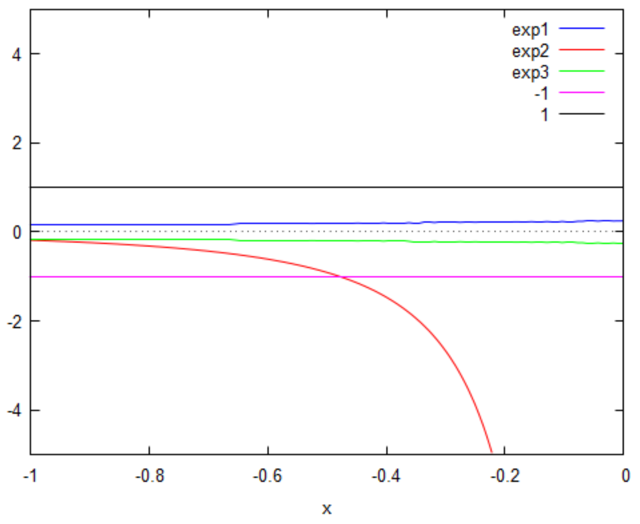
```
(%i6)diff(e2,x);
```

(%o6) $$\frac{e^x}{4\,x} - \frac{e^x}{4\,x^2}$$

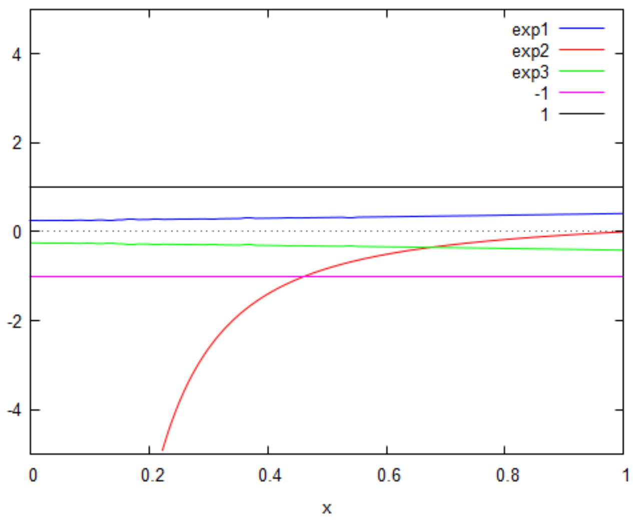
```
(%i7)diff(e3,x);
```

(%o7) $$-\frac{e^{\frac{x}{2}}}{4}$$

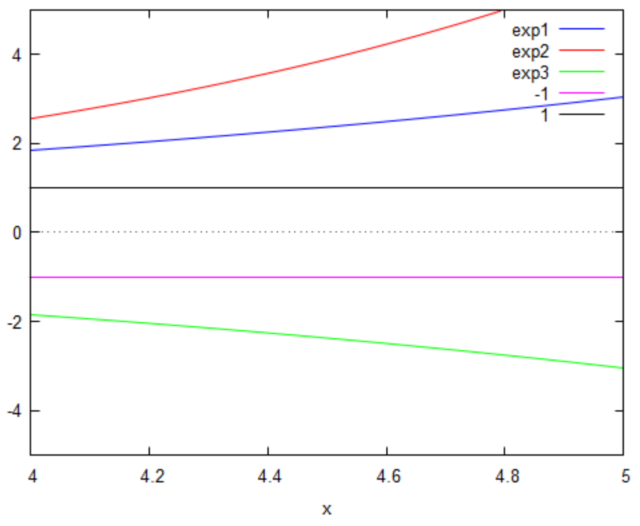
```
(%i24)wxplot2d([%o5,%o6,%o7,-1,1],[x,-1,0],[y,-5,5],
         [legend,"exp1","exp2","exp3","-1","1"]);
```



```
(%i25)wxplot2d([%o5,%o6,%o7,-1,1],[x,0,1],[y,-5,5],
        [legend,"exp1","exp2","exp3","-1","1"]);
```



```
(%i26)wxplot2d([%o5,%o6,%o7,-1,1],[x,4,5],[y,-5,5],
        [legend,"exp1","exp2","exp3","-1","1"]);
```



Analysing the graphs, that correspond to the intervals X_1, X_2, X_3 respectively, we can conclude that:

- First expression converges in the interval X_1 and X_2 as the derivative function is contained in $-1 \leqslant y \leqslant 1$.
- Second expression doesn't have an interval where it's certain to converge, due to the derivative function not being contained in $-1 \leqslant y \leqslant 1$.
- Third expression converges in the interval X_1 and X_2 as the derivative function is contained in $-1 \leqslant y \leqslant 1$.

```
from math import cos, sin, sqrt, exp, log

f = lambda x: exp(x) - 4*x**2
g = lambda x: 2*log(2*x)

x = 1.1

print(f"Iteration 0: X = {x:.5f}")

for i in range(1):
    x = g(x)
    print(f"Iteration {i+1}: X = {x:.5f}")
    print(f"Residue = {f(x):.5f}")
```

Program Output:

```
Iteration 0:  X = 1.10000
Iteration 1:  X = 1.57691
Residue = -5.10664
```

5 Exercise 5

```python
from math import cos, sin, sqrt, exp, log

def trapeze(f, a, b, h):

    n = int(abs(a-b)/h)

    S = h/2 * (f(a) + f(b) + 2 * sum(f(a + i*h) for i in range(1, n)))

    h1 = h/2

    S1 = h1/2 * (f(a) + f(b) + 2 * sum(f(a + i*h1) for i in range(1, 2*n)))

    h2 = h1/2

    S2 = h2/2 * (f(a) + f(b) + 2 * sum(f(a + i*h2) for i in range(1, 4*n)))

    QC = (S1-S)/(S2-S1)

    error = (S2-S1)/3

    return h, S, h1, S1, h2, S2, QC, error

def simpson(f, a, b, h):

    n = int(abs(a-b)/h)

    S = h/3 * (f(a) + f(b) + sum(4*f(a + i*h) if i%2 else 2*f(a + i*h) for i in range(1, n)))

    h1 = h/2

    S1 = h1/3 * (f(a) + f(b) + sum(4*f(a + i*h1) if i%2 else 2*f(a + i*h1) for i in range(1, 2*n)))

    h2 = h1/2

    S2 = h2/3 * (f(a) + f(b) + sum(4*f(a + i*h2) if i%2 else 2*f(a + i*h2) for i in range(1, 4*n)))

    QC = (S1-S)/(S2-S1)

    error = (S2-S1)/15

    return h, S, h1, S1, h2, S2, QC, error



f = lambda x: sqrt(1+(2.5*exp(2.5*x))**2)

a = 0
b = 1
h = 0.125

print("TRAPEZE")
h, S, h1, S1, h2, S2, QC, error = trapeze(f, a, b, h)
print(f"""
h   = {h:.5f}
h'  = {h1:.5f}
h'' = {h2:.5f}
L1  = {S:.5f}
L2  = {S1:.5f}
L3  = {S2:.5f}
QC   = {QC:.5f}
Error = {error:.5f}
""")

print("SIMPSON")
h, S, h1, S1, h2, S2, QC, error = simpson(f, a, b, h)
print(f"""
h    = {h:.5f}
h'   = {h1:.5f}
h''  = {h2:.5f}
L1   = {S:.5f}
L2   = {S1:.5f}
L3   = {S2:.5f}
QC   = {QC:.5f}
Error = {error:.5f}
""")
```

Program Output:

$TRAPEZE$

$h = 0.12500$

$h' = 0.06250$

$h'' = 0.03125$

$L1 = 11.34629$

$L2 = 11.27778$

$L3 = 11.26063$

$QC = 3.99394$

$Error = -0.00572$

$SIMPSON$

$h = 0.12500$

$h' = 0.06250$

$h'' = 0.03125$

$L1 = 11.25550$

$L2 = 11.25495$

$L3 = 11.25491$

$QC = 15.85798$

$Error = -0.00000$

6 Exercise 6

There's multiple methods that can be applied to find the minimum of the function. Methods such as the Gradient method and the Quadric method, or Levenberg-Marquardt Method.
Gradient method consists on following the direction of the gradient, in the opposite way, as the gradient of a level surface points towards the direction the function increases. However this method isn't efficient for surfaces with large areas where the function value doesn't change significantly, and so this method could be used in the beginning of the minimizing process, but near the minimum of this function the gradient method wouldn't bring any advantages.
Quadric Method consists on approximating the surface by a quadric and calculating the minimum of the quadric in order to obtain a better approximation of the minimum of the function that's being studied. This method purpose isn't to be used independently but as an adjustment to other methods, such as the gradient, as near the minimum the quadric approximation is precise enough to be considered a good approximation for the minimum value.
The last method, Levenberg-Marquardt is a method that consists on mixing both methods cited above into a single step, giving a weight, λ, to the gradient. This weight is reduced as we get closer to the minimum, as the gradient method looses its efficiency near the minimum area, and in case we get further from the minimum, the value of the weight is increased in order to give more importance to the graident.
Levenberg-Marquardt is the preferable method to minimize this function, as the surface contains depressions where the gradient method works best, but also has large areas where the function value doesn't change significantly, in which the gradient method isn't as efficient, but the quadric method is. By using Levenberg-Marquardt we can combine both advantages of each method into a single step, at the cost of more calculation processes in each iteration. This disadvantage of requiring more calculation can be optimized by saving the values of the gradient step and the value of the quadric step. This is useful in case the step fails (the function value of the new point is greater than the previous one) where the new point is discarded and there's no need to recalculate the gradient steps and the quadric steps since we stay at the same place.

7 Exercise 7

```python
from math import sin, cos, exp, log, sqrt

f = lambda x: x**3 - 10*sin(x) + 2.8

def bissec(f, a, b):

    for i in range(2):
        m = (a+b)/2

        if f(m) * f(b) < 0:
            a = m
        else:
            b = m

    return a, b

a, b = bissec(f, 1.5, 4.2)

print(f"[ {a:.5f}   ,   {b:.5f}   ]")
```

Program Output:

```
[ 1.50000 , 2.17500 ]
```