problems week 1

Magnus Chr. Hvidtfeldt

Technical University of Denmark, Lyngby, DK,

s255792@dtu.dk

```
from sympy import *
import numpy as np
import matplotlib.pyplot as plt
init_printing()
x = symbols('x')
```

1 Indledende begreber

1.1 Horners algoritme

```
def horner(a, x=x):
    n = len(a) - 1
    p = a[-1]
    for i in range(n-1, -1, -1):
        p = a[i] + x*p
    display(p)

#  a0, a1, a2, a3
b = [5, 3, -7, 2]
t = [11, 9, 7, 5]
horner(t, exp(x))
```

$$((5e^x + 7)\,e^x + 9)\,e^x + 11$$

1.2 Effektiv beregning

We are given the equation

$$y = 5e^{3x} + 7e^{2x} + 9e^x + 11$$

With the rule $e^{px} = (e^x)^p$ in mind, we can rewrite it to

$$y = 5(e^x)^3 + 7(e^x)^2 + 9(e^x)^1 + 11$$

We see that $x = e^x$, so we can say, for $y = 5x^3 + 7x^2 + 9x + 11$, we use horners algorithm:

$$y = e^x \cdot (e^x \cdot (5e^x + 7) + 9) + 11$$

We check it with horners algorithm and find that is the correct expression.

1.3 Omskrivning af udtryk

1.3.1 First problem

We want to rewrite the function $f(x) = \sqrt{x^4+4} - 2$ to be more robust in terms of cancellation. Since we have a root, we can expand it, and then remove the radical as such

$$\frac{(\sqrt{x^4+4}+2)(\sqrt{x^4+4}-2)}{(\sqrt{x^4+4}+2)} \tag{1.1}$$

$$(\sqrt{x^4+4}+2)(\sqrt{x^4+4}-2) = x^4 \tag{1.2}$$

So our expression becomes

$$f(x) = \frac{x^4}{\sqrt{x^4+4}+2} \tag{1.3}$$

Which should be more robust over cancellations.

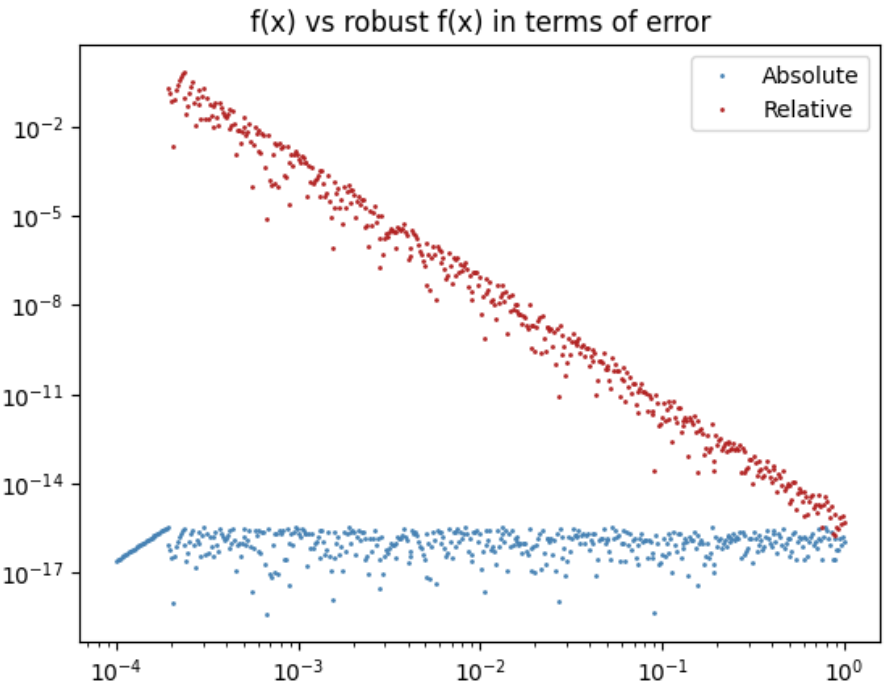
1.3.2 Second problem

```
f_simpl = (x**4+4)**(1/2)-2
f_rob = x**4/((x**4+4)**(1/2)+2)

x = np.logspace(-4, 0, 500)

abserr = abs(f_simpl - f_rob)
relerr = abs(f_simpl - f_rob) / abs(f_simpl)

plt.loglog(x, abserr, '.', markersize=2, color="steelblue")
plt.loglog(x, relerr, '.', markersize=2, color="firebrick")
plt.legend(["Absolute", "Relative"])
plt.title("f(x) vs robust f(x) in terms of error")
plt.show()
```



When plotting the functions, its clear to see that the rewritten form is more robust, as it can handle values that exceed 10^{-16}.

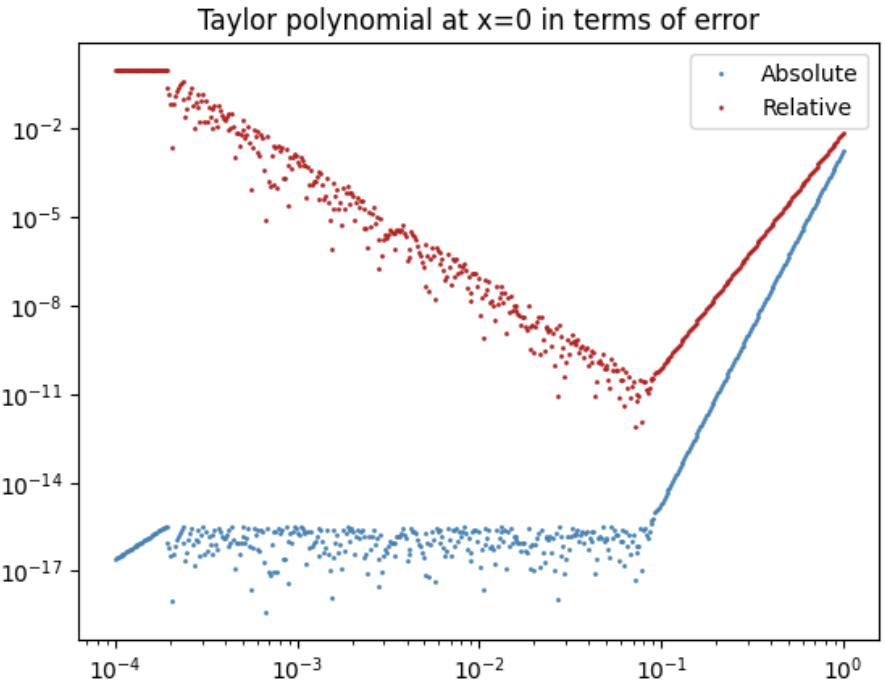
1.3.3 Third problem

Make a plot fo the taylorpolynomial T_B of the difference.

```
t_pol = 1/4*x**4 - 1/64 * x ** 8
x = np.logspace(-4, 0, 500)

abserrt = abs(t_pol - f_simpl)
relerrt = abs((t_pol - f_simpl)) / abs(t_pol)

plt.loglog(x, abserrt, '.', markersize=2, color="steelblue")
plt.loglog(x, relerrt, '.', markersize=2, color="firebrick")
plt.legend(["Absolute", "Relative"])
plt.title("Taylor polynomial at x=0 in terms of error")
plt.show()
```



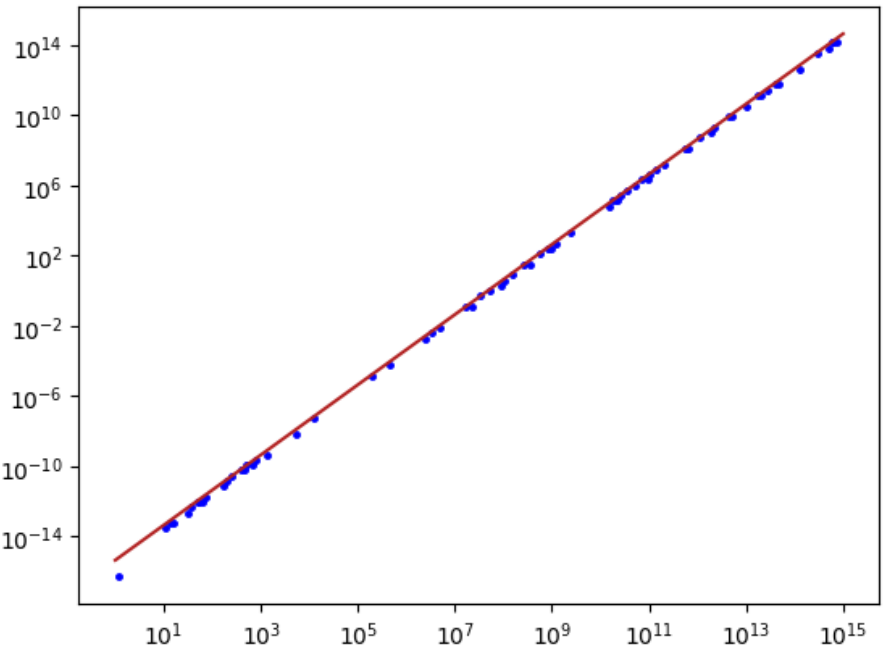
Notice the taylorpolynomial T_8 plot difference looks similar to that of the robust rewrite of $f(x)$.

1.4 Influence of loss of precision

```
x = symbols('x')
def f(x):
    return abs(1 - x**2 - (x-1)**2 + 2*x*(x-1))

x = np.logspace(0, 15, 250)
fx = f(x)
plt.loglog(x, fx, '.b', markersize=5)
plt.loglog(x, 4*x**2*((2.22044604925031*10**(-16)/2)),
↪  color="firebrick")
np.finfo(float).eps
```

$2.22044604925031 \cdot 10^{-16}$



2 Løsning af lineære ligningssystemer

2.1 Skruer og møtrikker

2.1.1 First problem

Let $x_1 =$ skruer, $x_2 =$ møtrikker, $x_3 =$ søm

$$\begin{bmatrix} 3 & 12 & 10 \\ 12 & 0 & 20 \\ 0 & 2 & 30 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 72.3 \\ 99.5 \\ 56.6 \end{bmatrix} \tag{2.1}$$

2.1.2 Second problem

```
A = np.array([[3, 12, 10],
              [12, 0, 20],
              [0, 2, 30]])
b = np.array([72.3, 99.5, 56.5])
display(np.linalg.solve(A,b))
```

```
array([5.5147619 , 3.25785714, 1.66614286])
```

2.1.3 Third problem

```
A = np.array([[3, 12, 10],
              [12, 0, 20],
              [0, 2, 30]])
b = np.array([73.3, 98.4, 57.1])
display(np.linalg.solve(A,b))
```

```
array([5.40095238, 3.35857143, 1.67942857])
```

2.1.4 Fourth problem

For the absolute errors, we have $[0.11, 0.1, 0.02]^T$

The greatest absolute error is 0.11, and it occurs in the first component a_1.

2.2 Calculation times

Here i create a python script to compare the computation times between np.linalg.solve(), np.linalg.inv(), and cramers rule.

```
import time

def m1(A,b):
    t_start = time.time()
    sol = np.linalg.solve(A,b)
    t_end = time.time()
    display(t_end-t_start)

def m2(A,b):
    t_start = time.time()
    sol = np.dot(np.linalg.inv(A), b)
    t_end = time.time()
    display(t_end-t_start)

def m3(A,b):
    t_start = time.time()
    for i in range(1, n+1):
        sol = np.linalg.det(A) / 1
    t_end = time.time()
    display(t_end-t_start)

for i in [100, 200, 400, 800]:
    n = i
    A = np.random.rand(n,n)/5
```

```
    b = np.random.rand(n,1)
    display("i: ", i)
    m1(A,b)
    m2(A,b)
    m3(A,b)
```

```
'i: '
```

100

0.000171184539794922

0.000306129455566406

0.00390505790710449

```
'i: '
```

200

0.000269889831542969

0.000462055206298828

0.0238537788391113

```
'i: '
```

400

0.000844955444335938

0.00160694122314453

0.232136964797974

```
'i: '
```

800

0.00365710258483887

0.00863885879516602

2.07763886451721

We can see that its significantly faster to compute np.linalg.solve(A,b) than the other methods. Note: I was not able to do the fully correct calculation for the third method at $n = 800$.