Next time: One pdf containing the solution that is to be graded. Naming-scheme for directory is Sheet xy_Surname1_Surname2-Surname3. More then one .ipynb per task is also not advisab

Exercise 1

Otherwise we will not grade your submission.

April 27, 2022

1	2	2
3/4	4/6	7/10

Exercise 1 : Numerical stability

a/b

[1]:
```python
import numpy as np

import matplotlib.pyplot as plt

#%matplotlib widget

#declaring the functions f and g
def f(x):
    return (x**3+1/3)-(x**3-1/3)

def g(x):
    return ((3+x**3/3)-(3-x**3/3))/x**3

#declaring range
x = np.logspace(-100, 100, 2000)

#analysicly solved
correct_result = 2/3

#not necessary
f_x = f(x)
g_x = g(x)

#numerical getting range for (f-2/3)/f > 1%
indexes_f_perc = np.where(np.abs((f_x-correct_result)/correct_result) > 0.01)
#2 index necessary because of np.shape(...) = (array([]))
range_f_perc = [x[indexes_f_perc[0][0]], x[indexes_f_perc[0][-1]]]

#numerical getting range for f = 0
indexes_f_0 = np.where(f_x == 0)
range_f_0 = [x[indexes_f_0[0][0]], x[indexes_f_0[0][-1]]]

#numerical getting range for (g-2/3)/g > 1%
indexes_g_perc = np.where(np.abs((g_x-correct_result)/correct_result) > 0.01)
```

consider a smaller range here

very efficient!

```
range_g_perc = [x[indexes_g_perc[0][0]], x[indexes_g_perc[0][-1]]]

#numerical getting range for g = 0
indexes_g_0 = np.where(g_x == 0)
range_g_0 = [x[indexes_g_0[0][0]], x[indexes_g_0[0][-1]]]

#answers getting printed here
#inf found out empirically
print(f'Range for x values of f(x) with more than 1% deviation:
 ↪[{range_f_perc[0]},inf) ')
print(f'Range for x values of f(x) where f(x) is zero:[{range_f_0},inf) ')

print(f'Range for x values of g(x) with more than 1% deviation:␣
 ↪[-inf,{range_g_perc[1]}] ')
print(f'Range for x values of g(x) where f(x) is zero: [-inf,{range_g_0[1]}] ')
```

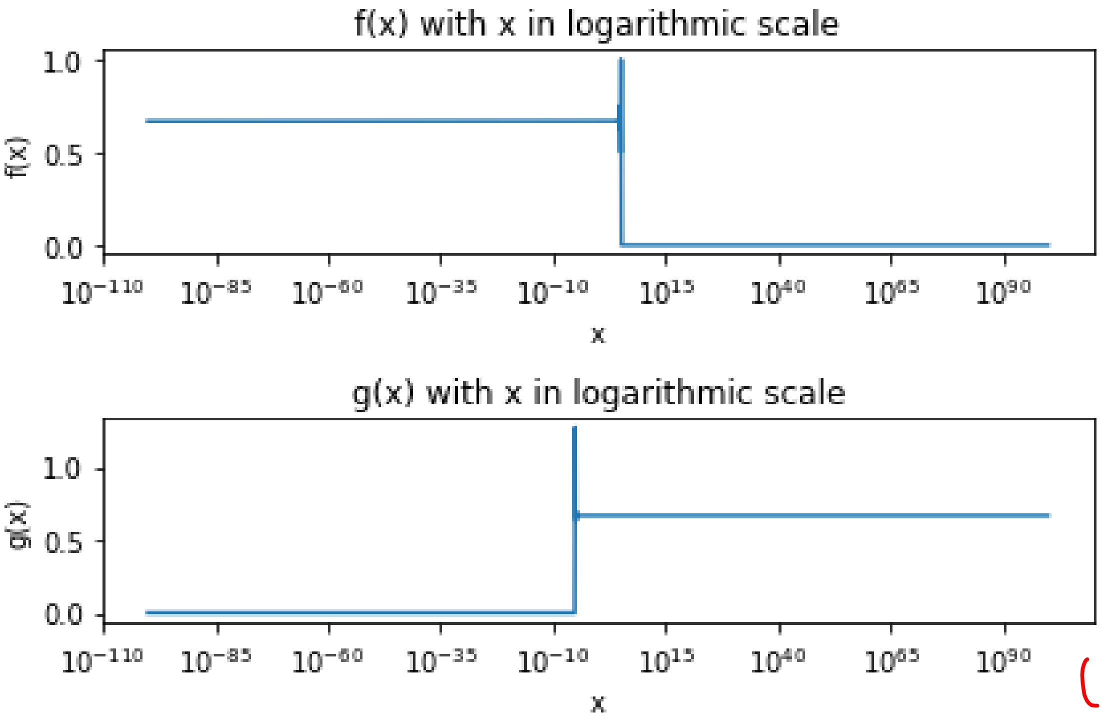
```
Range for x values of f(x) with more than 1% deviation:[44908.2532494585,inf)
Range for x values of f(x) where f(x) is zero:[[178906.57974916475, 1e+100],inf)
Range for x values of g(x) with more than 1% deviation:
[-inf,2.8036504221225594e-05]
Range for x values of g(x) where f(x) is zero: [-inf,7.037585948831987e-06]
```

this is no range

c)

[2]:
```
fig, ax = plt.subplots(2,1)



ax[0].plot(x, f(x))
ax[0].set_xlabel('x')
ax[0].set_ylabel('f(x)')
ax[0].set_xscale('log')
ax[0].set_title('f(x) with x in logarithmic scale')
ax[1].plot(x, g(x))
ax[1].set_xlabel('x')
ax[1].set_ylabel('g(x)')
ax[1].set_xscale('log')
ax[1].set_title('g(x) with x in logarithmic scale')


fig.tight_layout()
None
```

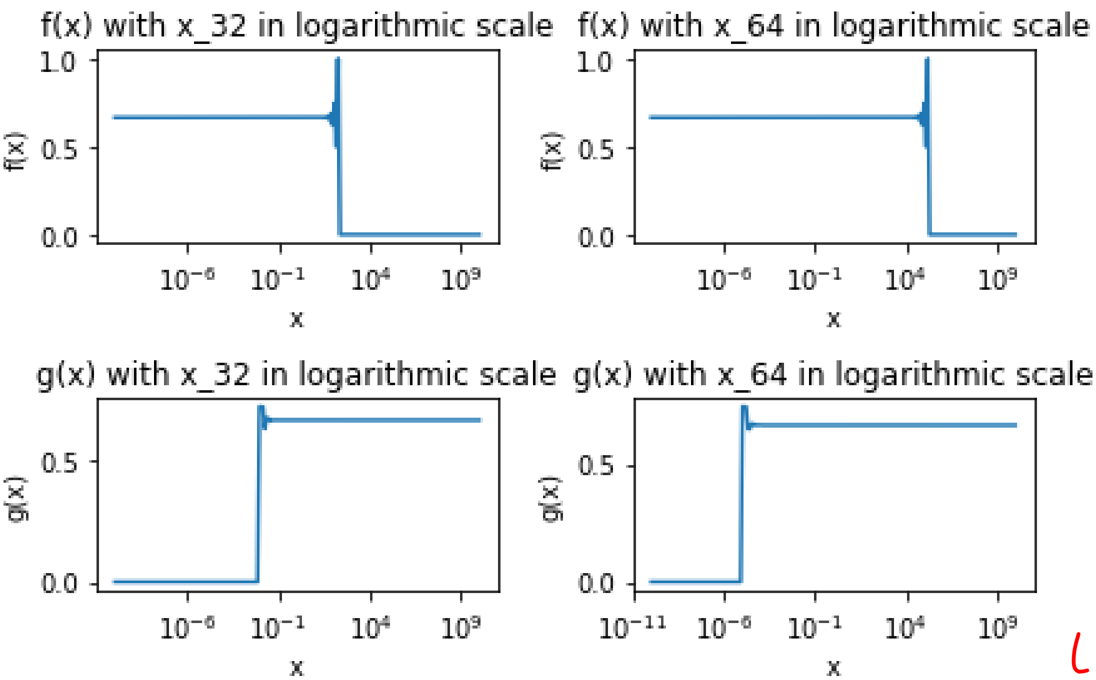
one can barely see the effect here as the x-range is way too big



d)

[3]:
```
#declaration of the arrays with the different data types
x_32 = x = np.logspace(-10, 10, 200, dtype='float32') #type float32
x_64 = x = np.logspace(-10, 10, 200, dtype='float64') #type float64


#starting subplots in a 2x2 grid
plt, ax = plt.subplots(2,2)

ax[0,0].plot(x_32, f(x_32))
ax[0,0].set_xlabel('x')
ax[0,0].set_ylabel('f(x)')
ax[0,0].set_xscale('log')
ax[0,0].set_title('f(x) with x_32 in logarithmic scale')

ax[1,0].plot(x_32, g(x_32))
ax[1,0].set_xlabel('x')
ax[1,0].set_ylabel('g(x)')
ax[1,0].set_xscale('log')
ax[1,0].set_title('g(x) with x_32 in logarithmic scale')

ax[0,1].plot(x_32, f(x_64))
ax[0,1].set_xlabel('x')
```

```
ax[0,1].set_ylabel('f(x)')
ax[0,1].set_xscale('log')
ax[0,1].set_title('f(x) with x_64 in logarithmic scale')

ax[1,1].plot(x_64, g(x_64))
ax[1,1].set_xlabel('x')
ax[1,1].set_ylabel('g(x)')
ax[1,1].set_xscale('log')
ax[1,1].set_title('g(x) with x_64 in logarithmic scale')

plt.tight_layout()



None
```



(v)

same as above

compared to what

For float32 and float64 the calculations remain longer stable, meaning f(x) delivers the correct result for bigger x values, and g(x) delivers numerically correct results for smaller x values

What I think you mean yet unfortunatly did not write is that float64-values remain longes stable than float32-values. This would be correct

3/4

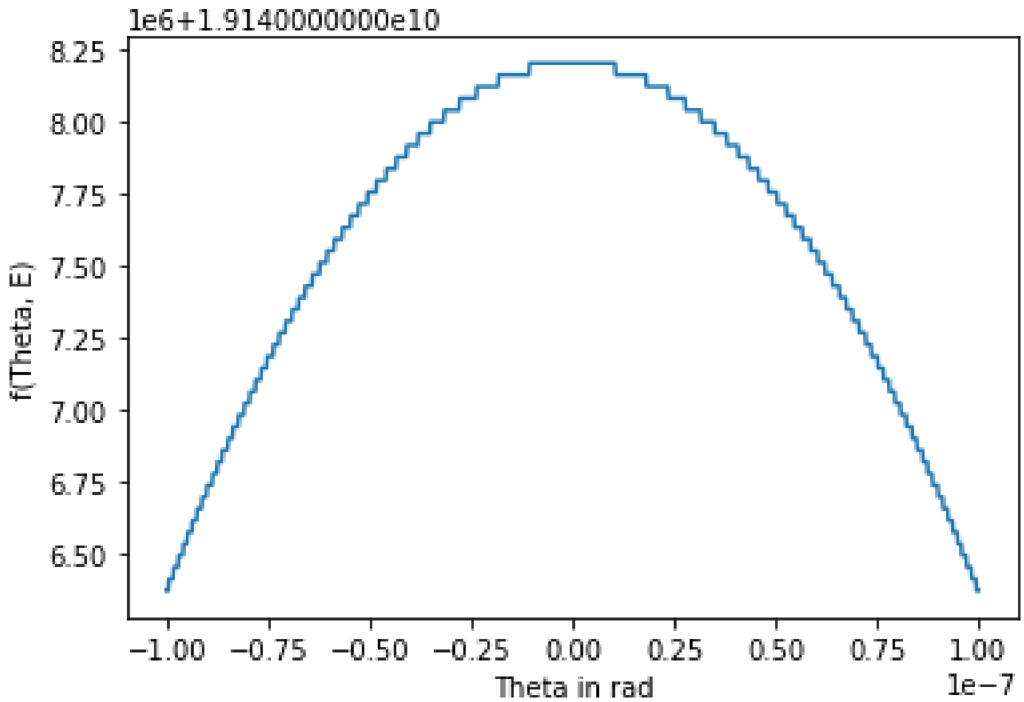
Exercise2

April 27, 2022

Exercise 2 a)

[1]:
```
import numpy as np

import matplotlib.pyplot as plt

#%matplotlib widget

def f(E, theta):
    m= 511
    gamma = E/m
    beta = np.sqrt(1-gamma**(-2))
    print('Beta =',beta)
    return (2+np.sin(theta)**2)/(1-beta**2*np.cos(theta)**2)

E_a = 50e6

theta_a = np.linspace(-10**-7, 10**-7, 1000)

f_a = f(E_a, theta_a)

#indexes_f_a = np.where(np.abs(f_a > 10e6))
#range_f_a = [theta_a[indexes_f_a[0][0]], theta_a[indexes_f_a[0][-1]]]

#print(range)


plt.figure()
plt.plot(theta_a, f_a)
plt.xlabel("Theta in rad")
plt.ylabel('f(Theta, E)')

None
```

```
Beta = 0.9999999999477758
```



f is numerically unstable for values of Theta which are multiples of Pi, because Beta = 0.9999999999477758. When Pi is reached, the denumerator approaches zero, making the calculation unstable. However since the sin and cos function are numerical approximations and Pi is not exact, infinity is not reached. However the values around the "True" pi still approach it, thus making the resulting function grow significantly.

b)

$$\frac{2+\sin(\theta)^2}{1-\beta^2 cos(\theta)} = \frac{2+\sin(\theta)^2}{\sin(\theta)^2+\cos(\theta)^2-\beta^2 cos(\theta)} = \frac{2+\sin(\theta)^2}{\sin(\theta)^2+\cos(\theta)^2\cdot(1-\beta^2)} = \frac{2+\sin(\theta)^2}{\sin(\theta)^2+\frac{\cos(\theta)^2}{\gamma^2}}$$

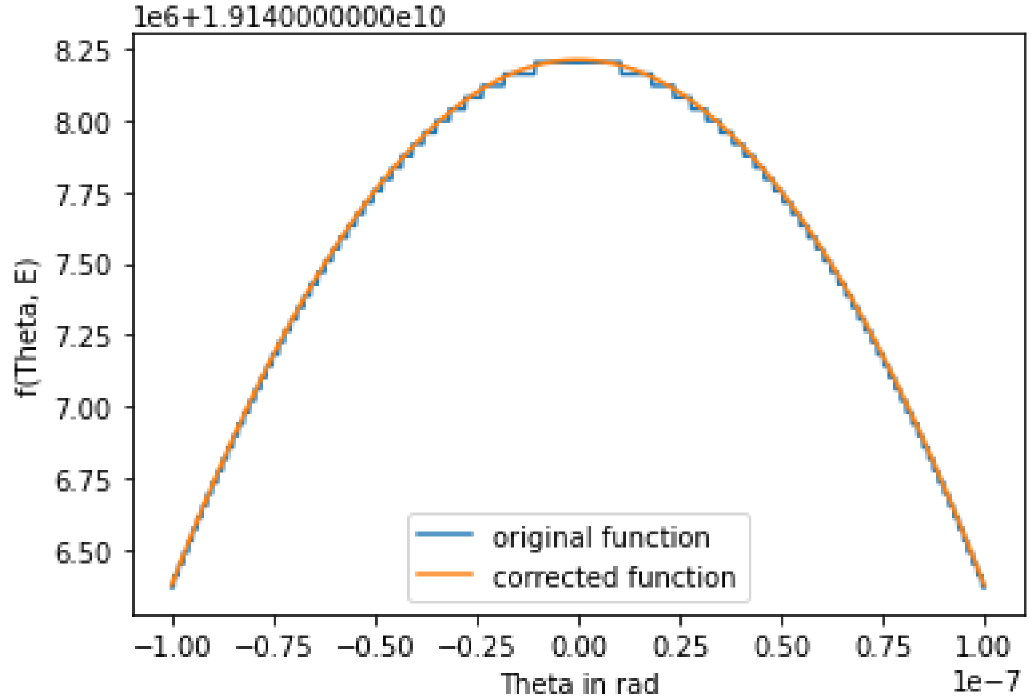
c)

Please be aware of these problems when exporting .pdfs from .ipynb, so you are carefull when using $$ ... $$ environments.

[2]:
```
def f_2(E, theta):
    m = 511
    gamma = E/m
    return (3*np.tan(theta)**2+2)/(np.tan(theta)**2+1/gamma**2) #with tan but␣
 ↪same function as in b
E_a = 50e6

theta_c = np.linspace(-10**-7, 10**-7, 1000)

f_c = f(E_a, theta_c)
f_2_c= f_2(E_a, theta_c)
```

```
plt.figure()
plt.plot(theta_c, f_c, label ='original function')
plt.plot(theta_c, f_2_c, label ='corrected function')
plt.xlabel("Theta in rad")
plt.ylabel('f(Theta, E)')
plt.legend()

None
```

Beta = 0.9999999999477758



d)

same as above, line-breaks can be difficult when exporting .pdfs.

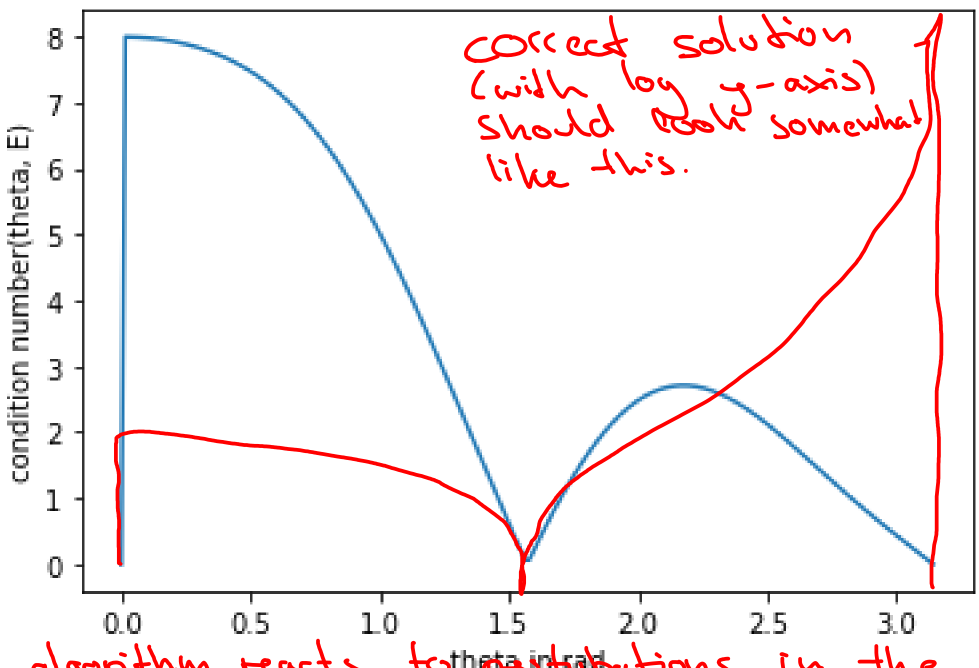
$$K = |\theta \frac{f(\theta)'}{f(\theta)}| f(x)' = \frac{2\sin(\theta)\cos(\theta)}{1-\beta^2\cos(\theta)^2} - \frac{2+\sin(\theta)^2}{(1-\beta^2\cos(\theta))^2} \cdot (2\beta^2\cos(\theta)\sin(\theta)) K = |\theta \cdot \left(\frac{2\sin(\theta)\cos(\theta)}{2+\sin(\theta)^2} - \frac{(2+\sin(\theta)^2)2\beta^2 c}{1-\beta^2 c}\right.$$

f

e)

```
[3]: def condition_number(theta, E):
    m= 511
    gamma = E/m
    beta = np.sqrt(1-gamma**(-2))
    first_summand = 2*np.sin(theta)*np.cos(theta)/(2+np.sin(theta)**2)
```

```
    second_summand = ((2+np.sin(theta)**2)*2*beta**2*np.cos(theta)*np.
 ↪sin(theta))/(1-beta**2*np.cos(theta))
    return np.abs(theta*(first_summand-second_summand))

theta_c = np.linspace(0, np.pi, 200)

condition = condition_number(theta_c, E_a)

plt.figure()
plt.plot(theta_c, condition)
plt.xlabel("theta in rad")
plt.ylabel(' condition number(theta, E) ')

None
```



correct solution (with log y-axis) should look somewhat like this.

#

stability: How the algorithm reacts to perturbations in the data
condition: How the solution ——— " ———

The problem is ill conditioned for values around zero and becomes well coniditioned around Pi/2 #

f)

Stability describes the numeric accuracy of a ~~function~~ algoritm for ~~numeric edge~~ pertubations in the input data cases.

Condition describes the error propagation/amplification of an input variable that already has an error

stability : The ~~function~~ algorithm “creates” the error Condition : An existing error is amplified or dampened

Be careful to not confuse function and algorithm here. We typically use an algorithm to represent a (mathmatical) function.

4/6