Problem 1

We saw in class how Taylor series/roundoff errors fight against each other when deciding how big a step size to use when calculating numerical derivatives. If we allow ourselves to evaluate our function f at four points ¿ and $x \pm 2\delta$ ¿,

a) what should our estimate of the first derivative at x be? Rather than doing a complicated fit, I suggest thinking about how to combine the derivative from $x \pm \delta$ with the derivative from $x \pm 2\delta$ to cancel the next term in the Taylor series.

We have that:

$$f(x \pm \delta) = f(x) \pm f'(x)\delta + f''(x)\delta^2 + \frac{\delta^3}{6} f'''(x) + o(\delta^4)$$

If we do $f(x+\delta) - f(x-\delta)$ we get:

$$f(x+\delta) - f(x-\delta) = 2\delta f'(x) + \frac{\delta^3}{3} f'''(x) + o(\delta^5)$$

Similarly, we have

$$f(x \pm 2\delta) = f(x) \pm 2\delta f'(x) + \frac{4\delta^2}{2!} f''(x) \pm \frac{8\delta^3}{3!} f^{(3)}(x) + o(\delta^4)$$

So that

$$f(x+2\delta) - f(x-2\delta) = 4\delta f'(x) + \frac{8\delta^3}{3} f^{(3)}(x) + o(\delta^4)$$

Next eliminate the terms of third order:

$$8f(x+\delta) - 8f(x-\delta) - f(x+2\delta) + f(x-2\delta) = 12\delta f'(x) + o(\delta^4)$$

And therefore we get the desired formula:

$$f'(x) = \frac{-f(x+2\delta) + 8f(x+\delta) - 8f(x-\delta) + f(x-2\delta)}{12\delta}$$

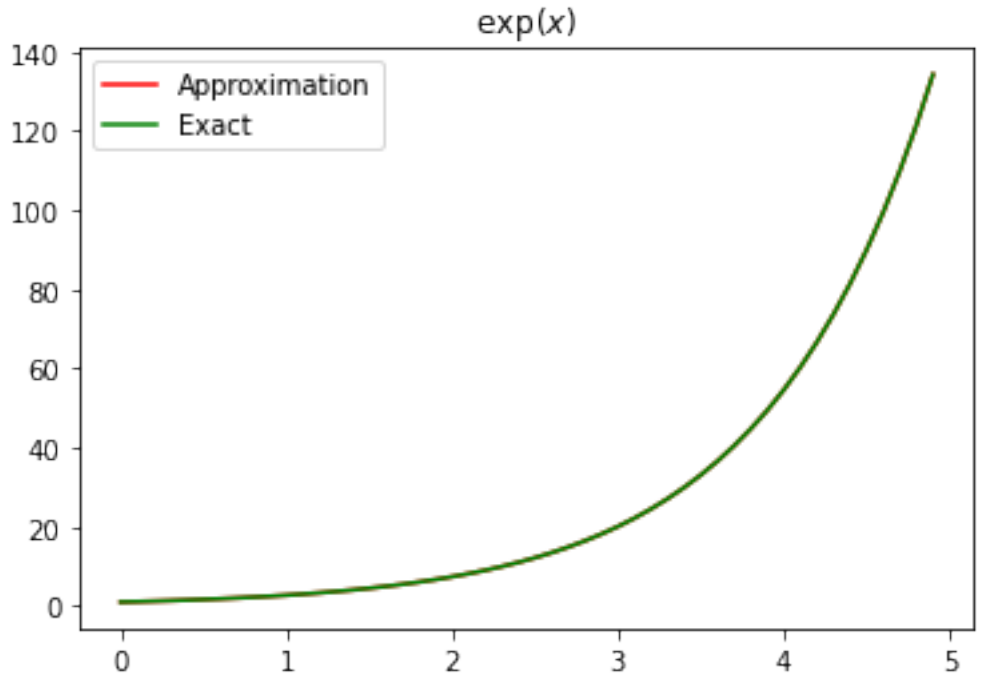
b) Now that you have your operator for the derivative, what should δ be in terms of the machine precision and various properties of the function?

Let ϵ machine precision (10^{-7} in 32 bits and 10^{-16} in 64 bits). Then we have that:

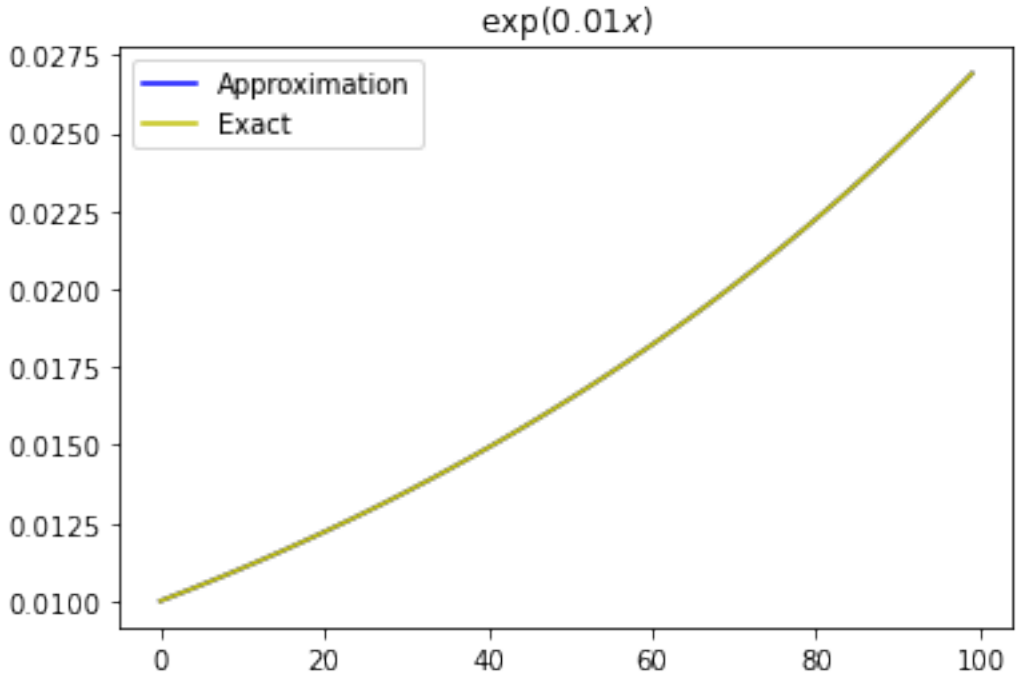
$$\delta \approx \sqrt{\varepsilon \left| \frac{f(x)}{f''(x)} \right|}$$

Show for $f(x)=\exp(x)$ and $f(x)=\exp(0.01x)$ that your estimate of the optimal δ is at least roughly correct.

```
import numpy as np
import matplotlib.pyplot as plt

x=np.arange(0, 5, 0.1)

eps=1e-7 #also works with epsilon=1e-16
delta=np.sqrt(eps)


def deriv(x):
    d=(-np.exp(x+2*delta)+8*np.exp(x+delta)-8*np.exp(x-delta)
+np.exp(x-2*delta))/(12*delta)
    return d



plt.plot(x, deriv(x), color='r', label='Approximation')
plt.plot(x, np.exp(x), color='g', label='Exact')
plt.title("$\exp(x)$")

plt.legend()

plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt

x=np.arange(0, 100, 1)

eps=1e-7 #also works with epsilon=1e-16
delta=0.1



def deriv(x):
    d=(-np.exp(0.01*(x+2*delta))+8*np.exp(0.01*(x+delta))-
8*np.exp(0.01*(x-delta))+np.exp(0.01*(x-2*delta)))/(12*delta)
    return d



plt.plot(x, deriv(x), color='b', label='Approximation')
plt.plot(x, 0.01*np.exp(0.01*x), color='y', label='Exact')
plt.title("$\exp(0.01x)$")

plt.legend()

plt.show()
```



Problem 2

Write a numerical differentiator with prototype

```
def ndiff(fun, x, full= False):
    epsilon=1e-16

    dx=np.cbrt(epsilon*x)

    deriv= (fun(x+dx)-fun(x-dx))/(2*dx)

    error= np.cbrt (epsilon**2)

    if full==False:
        return deriv
    else:
        return deriv, dx, error
```

We take a value of $dx = \sqrt[3]{\varepsilon x_c}$ with the curvature scale $x_c = x$, since we do not have any more detailed knowledge about the function. According to the numerical recipe chapter on the topic, such a rule of thumb will provide a reasonable default for non-zero x. The fractionnal error is then $\sqrt[3]{\epsilon^2}$.

where fun is a function and x is a value. If full is set to False, ndiff should return the numerical derivative at x. If full is True, it should return the derivative, dx, and an estimate of the error on the derivative. I suggest you use the centered derivative

$$f' \simeq \frac{f(x+dx) - f(x-dx)}{2\,dx}$$

Your routine should estimate the optimal dx then use that in calculating the derivative. If you're feeling ambitious, write your code so that x can be an array, not just a single number. If you do that, you may actually wish to save your code as you might use it in the future.

I wanted to write the function so that it takes in arrays, but I didn't know how to deal with the function being an argument. Indeed, in the above implementtion, if x is an array, then so is dx, so it all depends on the function fun.

Problem 3

Lakeshore 670 diodes (successors to the venerable Lakeshore 470) are temperature-sensitive diodes used for a range of cryogenic temperature measurements. They are fed with a constant $10\,\mu A$ current, and the voltage is read out. Lakeshore provides a chart that converts voltage to temperature, available at https://www.lakeshore.com/products/categories/specification/temperatureproducts/cryogenic-temperature-sensors/dt-670-silicon-diodes, or you can look at the text file I've helpfully copied and pasted (lakeshore.txt). Write a routine that will take an arbitrary

voltage and interpolate to return a temperature. You should also make some sort of quantitative (but possibly rough) estimate of the error in your interpolation as well (this is a common situation where you have been presented with data and have to figure out some idea of how to get error estimates).

The columns of lakeshore.txt are 1) the temperature, 2) the voltage across the diode at that temperature (which is what your experiment will actually measure), and 3) dV/dT. You do not need to use the third column, but you may if you wish.

Your code should support V being either a number or array, and it should return the interpolated temperature, and your estimated uncertainty on the temperature.

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt

data=np.loadtxt ('./lakeshore.txt')

def lakeshore(V, data):
    x=data[:,1]
    y=data[:,0]

    lin=np.array(V)

    interpolation=interpolate.interp1d(x,y, kind='cubic')
    y_lin=interpolation(lin)

    #plt.plot(x,y, '+', label='Points')
    #plt.plot(lin, y_lin, label='Interpolation' )
    #plt.show()

    return y_lin
```

To compute the error, we need a reference function which we don't have in this case.