Assignment 1

P1 :

(a) from class or lecture 1 ppt page 23, we know that

by using $\frac{f(x+\delta) - f(x-\delta)}{2\delta}$

$$= f'(x) + \frac{1}{3!} f'''(x) \cdot \delta^2 + \frac{1}{5!} f^{(5)}(x) \cdot \delta^4$$

$$\equiv \Delta\delta$$

similarly $\frac{f(x+2\delta) - f(x-2\delta)}{4\delta}$

$$= f'(x) + \frac{1}{3!} f'''(x) \cdot (2\delta)^2 + \frac{1}{5!} f^{(5)}(x) \cdot (2\delta)^4$$

$$\equiv \Delta 2\delta$$

Now $4 \cdot \Delta\delta - \Delta 2\delta$

$$= 3 \cdot f'(x) + 0 - \frac{1}{10} f^{(5)}(x) \cdot \delta^4$$

so we can use

$$\boxed{\frac{4 \cdot \Delta\delta - \Delta 2\delta}{3}}$$ to estimate $f'(x)$,

$$\boxed{\frac{4\cdot\Delta_\delta - \Delta_{2\delta}}{3}}$$ to estimate $f'(x)$,

which would eliminate $f'''(x)$ error in taylor series, which leaves the leading error due to taylor series to be $\sim \frac{1}{30} f^{(5)}(x)\cdot\delta^4$

(b) Again, by following what we did in class, now the error with leading term is

$$\frac{\varepsilon f}{\delta} + \frac{1}{30} f^{(5)}(x)\cdot\delta^4$$

to minimize this

$$\frac{d}{d\delta}\left(\frac{\varepsilon f}{\delta} + \frac{1}{30} f^{(5)}(x)\cdot\delta^4\right) = 0$$

$$-\frac{\varepsilon f}{\delta^2} + \frac{4}{30} f^{(5)}(x)\cdot\delta^3 = 0$$

$$\Rightarrow \delta^5 = \frac{\varepsilon f}{f^{(5)}}\cdot\frac{30}{4} \qquad \delta \sim \left(\frac{\varepsilon f}{f^{(5)}}\right)^{\frac{1}{5}}$$

for $f = e^x$, $f = f^{(5)}$,

for double $\varepsilon \sim 10^{-16}$,

so $\delta \sim 10^{-3}$

for $f = e^{0.01x}$, $\frac{f}{f^{(5)}} = 10^{10}$

$\varepsilon \sim 10^{-16}$

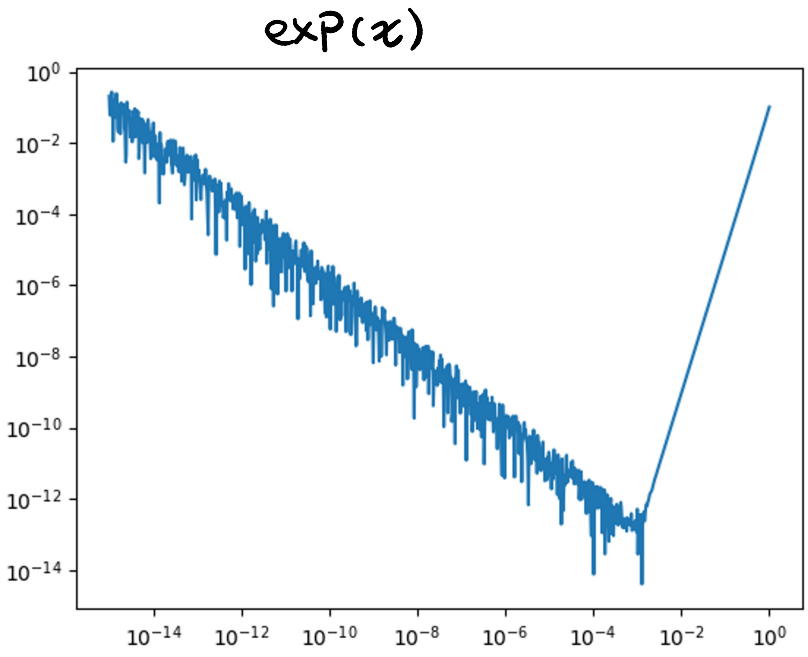
so $\delta \sim 10^{-1}$

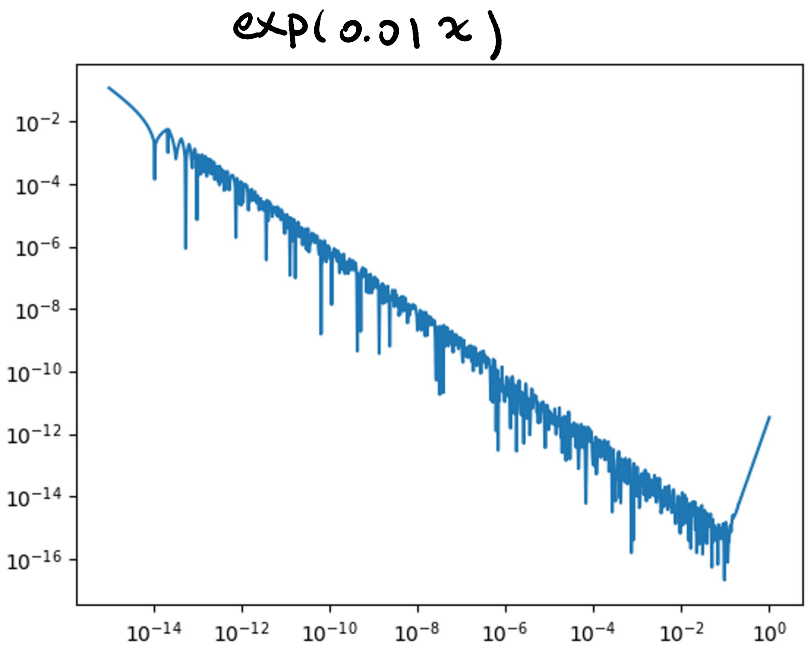
Now test with python.

```
import numpy as np
from matplotlib import pyplot as plt
logdx=np.linspace(-15,0,1001)
dx=10**logdx
def fun1(x):
    return np.exp(x) #define exp^(x)
def fun2(x):
    return np.exp(0.01*x) #define exp^(0.01x)
x0=1
def twosidederiv(fun,x0,dx): #define 2-sided derivative
    y1=fun(x0+dx)
    ym=fun(x0-dx)
    d2=(y1-ym)/(2*dx) #calculate the 2-sided derivative.
    return d2
def foursidederiv(fun,x0,dx): #define 4-sided derivative
    dd = twosidederiv(fun,x0,dx)
    d2d = twosidederiv(fun,x0,2*dx)
    return (4*dd - d2d)/3 #from what we calculated earlier
d4_fun1 = foursidederiv(fun1, x0, dx)
d4_fun2 = foursidederiv(fun2, x0, dx)
# plt.ion()
plt.clf()
#make a log plot of our errors in the derivatives
plt.loglog(dx,np.abs(d4_fun1-fun1(x0)))
plt.savefig('deriv_errors1.png')
plt.show()

plt.clf()
```

```
#make a log plot of our errors in the derivatives
plt.loglog(dx,np.abs(d4_fun2-0.01*fun2(x0))) # derive of exp^0.01*x = 0.01*exp^
0.01*x
plt.savefig('deriv_errors2.png')
plt.show()
```





I copied the codes here and have the explanations written as comments inside the code. Please tell me if this is ok or not.

We can see from the plots that the error for $f(x) = \exp(x)$, the minimum occurs around $\delta = 10^{-3}$, as what we expected.

The error for $f(x) = \exp(0.01x)$, the minimum occurs around $\delta = 10^{-1}$, also as what we expected.

Q2:

We already know that to find the estimate error and the optimal dx, we need to know $f'''(x)$.

we need to know $f'''(x)$.

However, we do not know $f'''(x)$ for an arbitrary function. But we can estimate $f'''(x)$ numerically.

From

https://math.stackexchange.com/questions/1301769/approximation-formula-for-third-derivative-is-my-approach-right

(or really just simple calculation)

we find that

$$f'''(x) \cong \frac{1}{2dx^3}\{f(x+2dx) - f(x-2dx) - 2\cdot[f(x+dx) - f(x-dx)]\} + 3f^{(5)}(x)\,dx^2$$

with round off error, leading err

$$err \cong f^{(5)}(x)dx^2 + \frac{fg\varepsilon}{dx^3}$$

minimize this

$$2f^{(5)}(x)\,dx - \frac{4fg\varepsilon}{3dx^4} = 0$$

$$\sim \quad f^{(5)}(x)\,dx^5 - fg\varepsilon = 0$$

$$dx \sim \left(\frac{f\varepsilon}{\quad}\right)^{\frac{1}{5}}$$

$$dx \sim \left(\frac{f\varepsilon}{f^{(5)}(x)}\right)^{\frac{1}{5}}$$

$$\sim 10^{-3} \text{ for double}$$

$$\text{if } \frac{f}{f^{(5)}} \sim 1$$

we can naively use $dx = 10^{-3}$
to estimate f'''.
Of course this might be terribly
wrong, say if $f = \exp(0.00001x)$
or $f = \exp(100000x)$

But let's say in worst case, even
if our f''' is off by a factor of 10^9
(which would need our $f^{(5)}(x)$ or $f(x)$ to be
$\sim 10^{15}$)

$dx \cong \left(\frac{f\varepsilon}{f'''}\right)^{\frac{1}{3}}$ is off by 10^{-3}
which means

$$\text{err} \sim f\varepsilon/dx + f'''dx^2$$

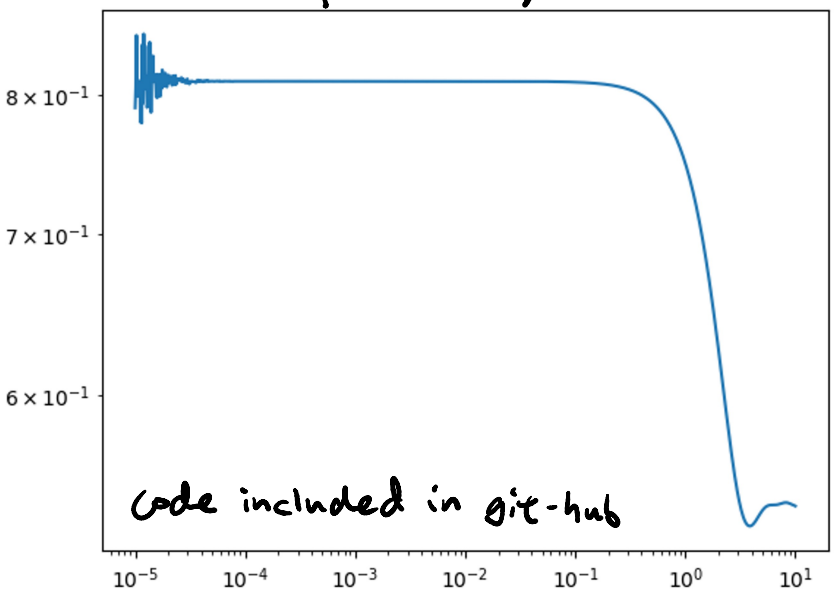
is off by $\sim 10^3$

which is good enough.

which is good enough.

(However, when I try to test the derivative error of f''' using a similar code in class, I was unable to get what I analyzed, Please tell me where I got wrong. Note we starte from 10^{-5} because $(10^{-5})^3 = 10^{-15}$.

$f = \cos(x)$



But anyways, at least the third derivative is accurate enough, let's try to implement it.

```
import numpy as np
from matplotlib import pyplot as plt
def fun1(x):
    return np.exp(x) #define exp^(x)

def fun2(x):
    return np.exp(10*x) #define exp^(10*x)
```

```
def fun3(x):
    return np.sin(x)

def twosidederiv(fun,x0,dx): #define 2-sided derivative
    y1=fun(x0+dx)
    ym=fun(x0-dx)
    d2=(y1-ym)/(2*dx) #calculate the 2-sided derivative.
    return d2

def thirdderiv(fun,x0):
    dd = twosidederiv(fun,x0,10**-3)
    d2d = twosidederiv(fun,x0,2*10**-3)
    return (d2d - dd)/((10**-6)) # from what we calculated earlier, and by
setting dx to be 10^-3

def optimal_dx(fun,x0,d3):
    return 10**-5.3*((fun(x0)/d3)**1/3) # from what we learned in class, factored
10^-16 out so that it won't reach lower roundoff of double

def estimated_error(fun,x0,d3,dx):
    return fun(x0)*(10**-16)/dx + d3*dx**2

def ndiff(fun,x,full=False):
    d3 = thirdderiv(fun,x) #calculated estimated thrid deriv
    dx = optimal_dx(fun,x,d3) #calculated optimal dx
    d2 = twosidederiv(fun,x,dx) #calculated numerical deriv
    if full == False:
        return d2
    else:
        err = estimated_error(fun,x,d3,dx) #if required calculate estimated error
        return (d2,err)

tup1 = ndiff(fun1,1,full=True)
print ("numerical deriv of f(x)=exp(x) at x = 1 is: ", tup1[0], ", the estimated
error is: ", tup1[1],
    ", the real error is: ", np.abs(tup1[0]-fun1(1)))
tup2 = ndiff(fun2,10,full=True)
print ("numerical deriv of f(x)=exp(10*x) at x = 10 is: ", tup2[0], ", the
estimated error is: ", tup2[1],
    ", the real error is: ", np.abs(tup2[0]-10*fun2(10)))
tup3 = ndiff(fun3,1,full=True)
print ("numerical deriv of f(x)=sin(x) at x = 1 is: ", tup3[0], ", the estimated
error is: ", tup3[1],
    ", the real error is: ", np.abs(tup3[0]-np.cos(1)))
```

Output of the code is :

numerical deriv of f(x)=exp(x) at x = 1 is: 2.7182818284708903 , the estimated error is: 9.652867888703383e-11 , the real error is: 1.184519149433072e-11
numerical deriv of f(x)=exp(10*x) at x = 10 is: 2.6881164916009383e+44 , the estimated error is: 8.045451108858073e+35 , the real error is: 6.502151972137531e+37
numerical deriv of f(x)=sin(x) at x = 1 is: 0.540302305867916 , the estimated error is: -2.3485914195971102e-11 , the real error is: 2.2370993946196904e-13

Also note that this code is perfectly vectorized.

Q3:

First of all, by looking at dV/dT, we see that except when

V ~ 1.13, dV/dT is rather smooth.

This means that it is a good idea to use cubic spline as a interpolation method.

Although Rigel on Slacks says that we should write our own interpolation code, I really see no point in re writting a cubic spline interpolation and not use the one already there in SciPy.

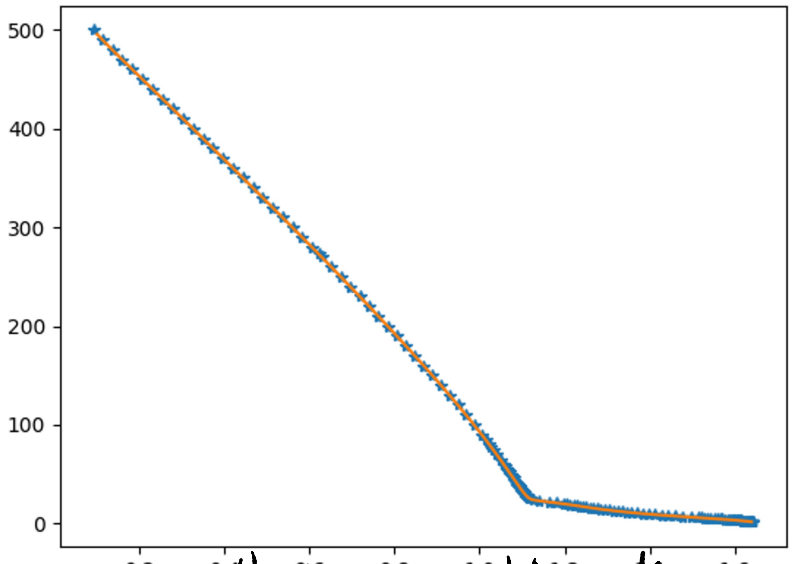
```
import numpy as np
import scipy.interpolate as intp
from matplotlib import pyplot as plt

dat = np.loadtxt('lakeshore.txt')

def lakeshore(V,data):
    temp_ys = data[:,0] #first column of data is temperature values
```

```
    v_xs = data[:,1] #second column of data is voltage values
    #using scipy.interpolate.interp1d to do a cubic spline interpolation
    #interp1d returns a function which can take in new points and return new
#values based on the interpolation
    spline = intp.interp1d(v_xs,temp_ys,'cubic')
    interp_temp = spline(V)
    return interp_temp

#test the interpolation scheme
plot_xs = np.linspace(0.1,1.64,1000)
plot_ys = lakeshore(plot_xs,dat)
plt.clf()
plt.plot(dat[:,1],dat[:,0],'*')
plt.plot(plot_xs,plot_ys)
plt.savefig('spine_interp.png')
plt.show()
```



we can see that our cubic spline does a pretty good job in interpolating.

what is the estimate error then?

I'll use a way similar to what is covered in tutorial.

I bootstrapped such that for each resampling, I fixed a set

each resampling, I fixed a set of V values and find the temperature interpolated value of each resampling of these V values. I then take the std as an estimation of error.

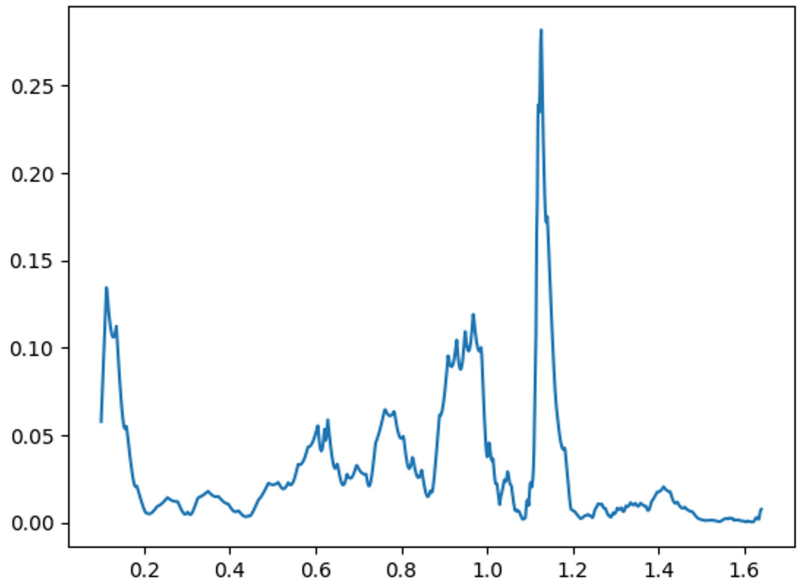
```
#Bootstrapping to find the estimated error. Mostly stolen from tutorial 2's
#bootstrap_interp.py
rng = np.random.default_rng(seed=12345) # Create a random number generator object
N_resamples = 20 # How many times to resamples
N_samples = 100 # How many points to resample with

temp_y = dat[:,0]
v_x = dat[:,1]
arr_size = temp_y.size

gen_pts = [] # To hold the interpolated y values for each run.
for i in range(N_resamples):
    # Making a list of all the indices for our x interpolation points.
    # Note!!!! This is different from tutorial code. I made sure that we keep the
#first and last point so we are indeed
    # interpolating, rather than possible extrapolating.
    indices = list(range(1,arr_size-1))
    to_interp = rng.choice(indices,size=N_samples-2,replace=False) # Choosing
#N_samples-2 (98) indices of values to use for new interp
    to_interp.sort() # Make sure x is increasing
    to_interp = np.insert(to_interp,0,0) # Add index to the first point
    to_interp = np.append(to_interp,arr_size-1) # Add index to the last point
    new_interpolation = intp.interp1d(v_x[to_interp],temp_y[to_interp]) #
#Choosing N_samples (100) indices of values to use for new interp
    interpolated_ys = new_interpolation(plot_xs) # Interpolate y-values at plot
points and save.
    gen_pts.append(interpolated_ys)

gen_pts = np.array(gen_pts) # Convert list of lists to 2D array
stds = np.std(gen_pts,axis=0) # Calculate std dev at each x across resamplings
plt.clf()
plt.plot(plot_xs,stds)
plt.show()


error2 = np.mean(stds) # Take mean of std for overall error
error2_std = np.std(stds) # Get std dev of that error.
print(f"{error2 = :.3e} +/- {error2_std:.3e}")
```



error2 = 3.143e-02 +/- 3.895e-02

we can see that overall, our interpolation scheme works pretty well, with an overall error of around $3.14 \times 10^{-2} \pm 3.89 \times 10^{-2}$.

We see a huge error peak at around 1.1 V, which is to be expected because around 1.13V is where our V-T curve takes a turn.

Any information loss around this area would cause our interpolation to be less accurate.

Q4

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

def lorentz(x):
    return 1/(1+x**2)

def rat_eval(p,q,x): #stolen from ratfit_exact.py from course github
    top=0
    for i in range(len(p)): #calculate nominator of rational fit
        top=top+p[i]*x**i
    bot=1
    for i in range(len(q)): #calculate denominator of rational fit
        bot=bot+q[i]*x**(i+1)
    return top/bot

def rat_fit(x,y,n,m):
    assert(len(x)==n+m-1)
    assert(len(y)==len(x)) #assert if number of fit points are bad
    mat=np.zeros([n+m-1,n+m-1]) #create zero square matrix
    #create matrix accroding to lecture ppt page 14
    for i in range(n):
        mat[:,i]=x**i
    for i in range(1,m):
        mat[:,i-1+n]=-y*x**i
    pars=np.dot(np.linalg.inv(mat),y) #inverse matrix
    #pars=np.dot(np.linalg.pinv(mat),y) #for second part of the question
    p=pars[:n] #p coeff is first n
    q=pars[n:] #q ceoff is after first n
    return p,q


fun=np.cos;x0=-np.pi/2;x1=np.pi/2; #comment this out to have cos
# fun=lorentz;x0=-1;x1=1 #comment this out to have lorentzian

n=3
m=3
# n=4 #for second part of the question
# m=5

#again, the following is stolen from ratfit.py from course github
x=np.linspace(x0,x1,n+m-1) #create linspace with right amount of points
y=fun(x)
p,q=rat_fit(x,y,n,m) #calculate rational fit coeff
xx=np.linspace(x[0],x[-1],1001)
y_true=fun(xx) #true value of cos
pred=rat_eval(p,q,xx) #rational fit value
fitp=np.polyfit(x,y,n+m-1) #np.polyfit returns vector of coefficients with degree of n+m-1
pred_poly=np.polyval(fitp,xx) #polynomial fit value
myfun=interpolate.interp1d(x,y,'cubic') #scipy.interpolate.interp1d returns a function
which cubic spine fit of the data
pred_spline=myfun(xx)

#print the standard deviation of the fitted values with true values
print('rat err ',np.std(pred-y_true))
print('poly err ',np.std(pred_poly-y_true))
```

```
print('spline err ',np.std(pred_spline-y_true))
```

f(x) = cos(x), n=3, m=3

rat err 0.001111531338043094
poly err 0.0007343476108126695
spline err 0.0034168785145555504

f(x) = lorentz(x), n=3, m=3

rat err 1.6389243820419108e-16
poly err 0.009607083314602723
spline err 0.002465087739602325

f(x) = lorentz(x), n=4, m=5

rat err 21.51833115012628
poly err 0.004138670720477468
spline err 0.0006751580857693977

f(x) = lorentz(x), n=4, m=5, use pinv(x)

rat err 2.106365296468666e-16
poly err 0.004138670720477468
spline err 0.0006751580857693977

We can see that for cos(x), poly interpolation seems to be the most accurate. This is to be expected because cos(x) is well represented by its taylor series.

For Lorentzian, when n=3, m=3, the fit is almost machine precision. This is also to be expected since Lorentzian is infact a rational function with m=2 therefore the interpolation

a rational function with
$m=3$, therefore the interpolation
is perfect.

However, when using $n=4$,
$m=5$, the interpolation
is very poor. We can
think of such rational
interpolation with order
higher than the true
rational function:
the original function

$\frac{P(x)}{1+qq(x)}$ is being
interpolated as

$\frac{P(x)\cdot(1+ax)}{(1+qq(x))(1+ax)}$

though we know they
cancel, but the
computer would still try to
fit. This means that we
are trying to solve
$\ldots$ where x is

are trying to solve

$Ax = b$ where x is

not unique. In this

case, a pseudo-inverse

(pinv) would be the

the best choice here

since it solves for

linear least square solution

for this system of equations.