Name: Abdulla Alseiari
PHYS 512
PS # 2
Sep/18/2020

problem 1:

a) ① $f(x\pm\delta) = f(x) \pm \delta f'(x) + \frac{\delta^2}{2!} f''(x) \pm \frac{\delta^3}{3!} f'''(x) + \cdots$

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

Manipulating the above 4 equations, we get

$f(x+\delta) - f(x-\delta) + f(x+2\delta) - f(x-2\delta)$

$= f(x) - f(x) + f(x) - f(x) + \delta f'(x) + \delta f'(x) + 2\delta f'(x) + 2\delta f'(x)$

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

$+ \frac{\delta^3}{3!} f'''(x) + \frac{8}{3!}\delta^3 f'''(x) + \frac{8}{3!}\delta^3 f'''(x) + \cdots$

$= 6\delta f'(x) + 3\delta^3 f'''(x) + \cdots$

thus, $\boxed{f'(x) = \frac{f(x+\delta) - f(x-\delta) + f(x+2\delta) - f(x-2\delta)}{6\delta} - \frac{\delta^2 f'''(x)}{2}}$

problem 1)

b) Notice that we can represent the total error by

(E) Total error = roundoff error + truncation error

- In the previous part we had the truncation error = $\frac{\delta^2}{2} f''(x)$
- We can represent it as $C_1 \delta$ where C_1 is some constant
- For the roundoff error, we can represent it by $C_2 \epsilon / \delta$ where ϵ is the machine precision and C_2 is some constant
- Therefore, we are trying to minimize the total error

$\frac{\partial E}{\partial \delta} = -\frac{C_2 \epsilon}{\delta^2} + C_1 \delta = 0 \Rightarrow \delta = \sqrt[3]{\frac{C_2 \epsilon}{C_1}}$ is the optimal

where C_1 and C_2 depends on function f.

Problem 2:

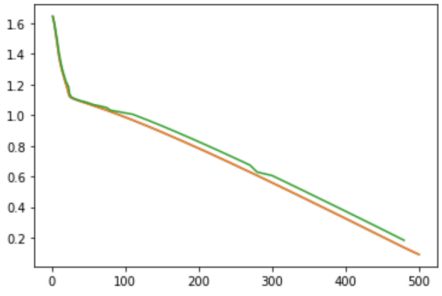
Code:

```
#the code is linear spline code that connect the points with lines for interpolation
#the code take the voltages and tempretures as an inputs to construct an interpolation
#after feeding the function with a desired voltage value within the range of the voltages
#The code will interpolate an approximate tempreture

def linear_interp(volt, temp, x):
  ind=np.max(np.where(volt>=x)[0])
  if x==volt[0]: #this to deal with the first entry since we need previous point to construct the interpolation
  #the code return the tempreture as it is for the first voltage entry
    return temp[0]
  else:
    x_use=volt[ind-1:ind+1]
    y_use=temp[ind-1:ind+1]
    dx=x_use[1]-x_use[0] #the change between the nearest two point from dataset
    t=(x-x_use[0])/dx
    return (1-t)*y_use[0]+t*y_use[1]
```

Results:
The following is three graphs of true values from data set blue color (under the interpolated graph of full dataset) the orange graph is the interpolation from the entire dataset. The green is the graph from using every other point from dataset then interpolating the points between them.

```
plt.plot(temp,volt)
plt.plot(temp_interp,volt)#when using the full point we can see that both interpolated and true values overlap
plt.plot(error[:-2],volt[1:-4:2])# I had to subtract the edges for beyound the limit error
#the blue and orange lines are the interpolation of the full dataset
#while the green is the interpolation of everyother point which has some error
```

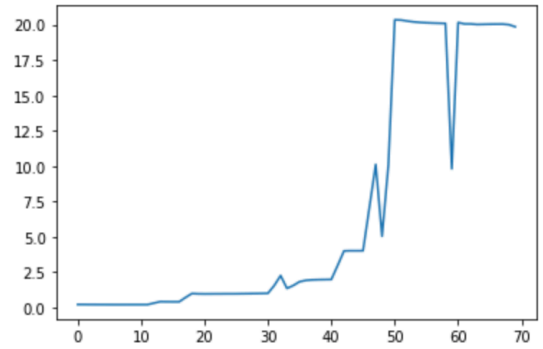
```
[<matplotlib.lines.Line2D at 0x7f7ccadd38d0>]
```



The following is the error (difference) between the true values and the interpolated values of every other point. Please read comments in code for more information.

```
plt.plot(error_test)#plot of the error at every other point.
#As we can see as the tempreture increase the error increase since this is linear interpolation
#code that I have written take the nearest tempreture to the given voltage point
#the maximum error is about 20 degrees
```

```
[<matplotlib.lines.Line2D at 0x7f7cca7ba208>]
```

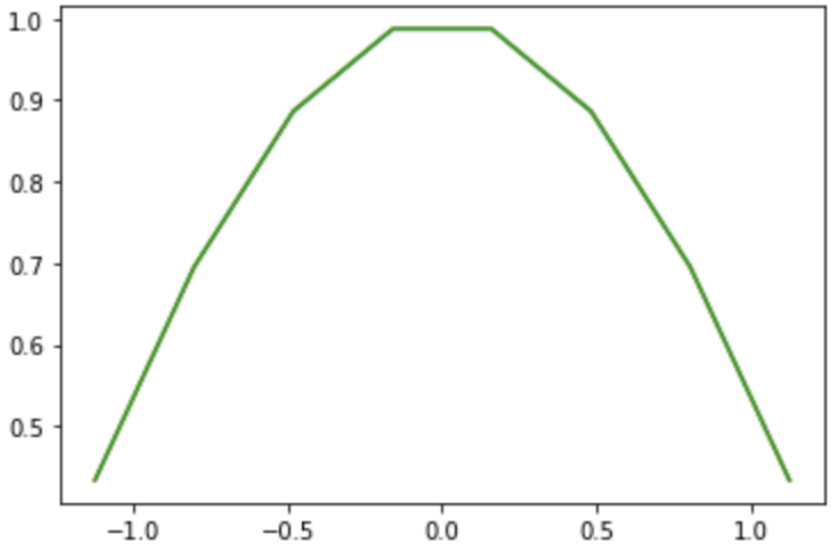


Problem 3:

Here you can look at the code for more coding details. The following is the result for the cosine function using the three mode with step size of 8 and the rms errors.

```
cubic_interp(x_c,y_c)
poly_interp(x_c,y_c,4,5)
rat_interp(x_c,y_c,4,5)
```

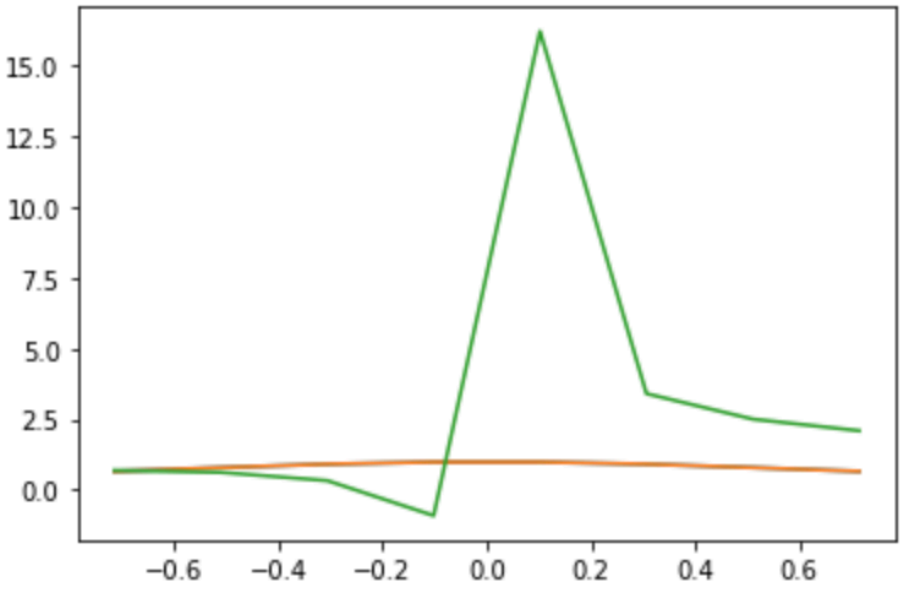
```
my rms error is  0.1605216973583836
polynomial rms error is  0.16026987871501078
rational function rms error is  0.1603088542932368
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.p
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.p

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.p
  This is separate from the ipykernel package so we can avo
```



Next, is the Lorentzian function with the original codes from the lecture notes. We can see that the rational function has some huge error compared to the other two.

```
[567] cubic_interp(x_l,y_l)
      poly_interp(x_l,y_l,4,5)
      rat_interp(x_l,y_l,4,5)
```

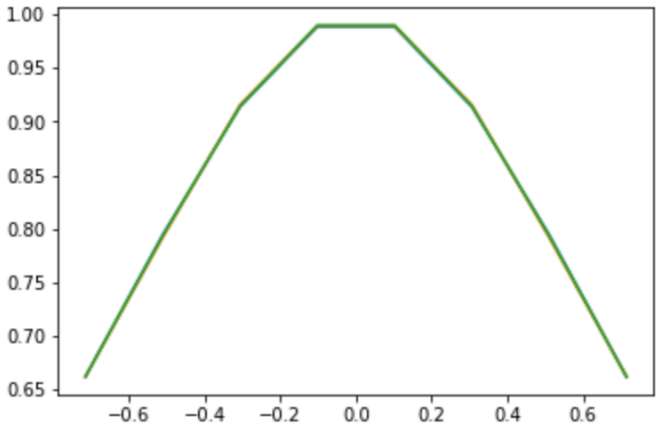
```
/usr/local/lib/python3.6/dist-packages/ipykernel_launch
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.6/dist-packages/ipykernel_launch

/usr/local/lib/python3.6/dist-packages/ipykernel_launch
  This is separate from the ipykernel package so we ca
my rms error is  0.059356128367793644
polynomial rms error is  0.05838745226574216
rational function rms error is  5.046997601482014
```



By fixing the np.linalg.inv code to be np.linalg.pinv, we got the following result:

```
cubic_interp(x_1,y_1)
poly_interp(x_1,y_1,4,5)
rat_interp(x_1,y_1,4,5)
```

```
my rms error is  0.059356128367793644
polynomial rms error is  0.05838745226574216
rational function rms error is  0.058613534187128585
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: RankWarning:
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: RankWarning:

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RankWarning:
  This is separate from the ipykernel package so we can avoid doing imports u
```



Since the rational function is dividing to polynomials p and q, the first graph show a singularity at the midpoint i.e q=0. np.linalg.pinv compute the (Moore-Penrose) pseudo-inverse of a matrix which deals with the singular matric and resolve the issue from the first graph. Comparing the rms error for rational function between the first and second graph we can see that the error decreased a lot. Comparing the performance of the three methods we can see that the polynomial one outperformed the other two in the two functions while the fixed rational function came next.

Problem 4:

By looking at the griffiths's 2.7 solution I set the integral as follows:

```
[579] #setting the integral from Griffiths 2.7 solution
      #the function is some constant of q, sigma and epsilon note  which we can ignore for simplisity


      def E_fun(u,z,r):
          return (z-r*u)/((r**2)+(z**2)-(2*r*z*u))**(3/2) # I got to this final equation by looking at the griffith 2.7 solution
```

I tried to use the integrator from class notes but I got stuck with a warning: maximum recursion depth exceeded. I think this due to singularity in the integral at point z=R.

```
#testing the simple integration from lecturenote


simp_integ=[]

for z in range(300):
  simp_integ.append(simple_integrate(E_fun,-1,1,0.01))
```

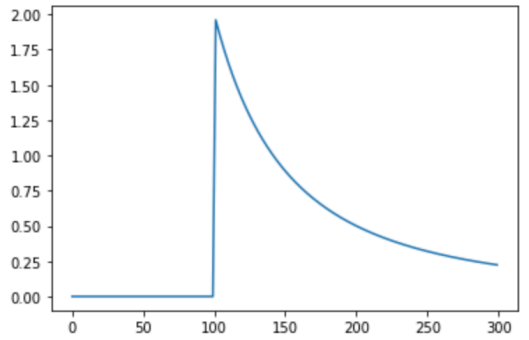
While using the scipy.integrate.quad, I got a nice graph that matches the analytical solution for this problem.

```
[571] #testing the integrate.quad for electric field

      quad_integ=[]

      for z in range(300):
        quad_integ.append(integrate.quad(E_fun, -1, 1,args=(z/100,1)))

      arrinteg=np.asarray(quad_integ)[:,0]
```

The result is the following:

```
plt.plot(arrinteg)
#we can see at the plot at z=100 which is equavilant to z = radius R, the E field is maximum
#and for z<R it is zero and at z>R it is decreasing by the form 1/z^2
```

```
[<matplotlib.lines.Line2D at 0x7f7cc7d34828>]
```




```
[ ]
```

In this graph the x-axis is the z variable and z=100 is when z=R. we can see from the graph that for z<R the E field is zero and for z=R it reached maximum and after that it decreased by a factor of 1/z^2.

Conclusion: we can see that the built in scipy code deals with singularities in the function while our code does not except with some modification.