Muath Hamidi | PHYS-512 | Fall 2022 | PS1

Problem 1

(a)

The derivative from $x \pm \delta$ is

$$f'(x) = \frac{f(x+\delta) - f(x-\delta)}{2\delta} \tag{1}$$

Expanding using Taylor series,

$$f'(x) = \frac{\left(f + \delta f' + \frac{\delta^2}{2} f'' + \frac{\delta^3}{3!} f^{(3)} + \cdots\right) - \left(f - \delta f' + \frac{\delta^2}{2} f'' - \frac{\delta^3}{3!} f^{(3)} + \cdots\right)}{2\delta} \tag{2}$$

Clearly, we can cancel all even terms,

$$f'(x) = \left(f' + \frac{\delta^2}{3!} f^{(3)} + \frac{\delta^4}{5!} f^{(5)} + \cdots\right) \tag{3}$$

And the derivative from $x \pm 2\delta$ is

$$f'(x) = \frac{f(x+2\delta) - f(x-2\delta)}{4\delta} \tag{4}$$

Expanding using Taylor series,

$$f'(x) = \frac{\left(f + (2\delta) f' + \frac{(2\delta)^2}{2} f'' + \frac{(2\delta)^3}{3!} f^{(3)} + \cdots\right) - \left(f - (2\delta) f' + \frac{(2\delta)^2}{2} f'' - \frac{(2\delta)^3}{3!} f^{(3)} + \cdots\right)}{4\delta} \tag{5}$$

Clearly, all even terms cancel,

$$f'(x) = \left(f' + \frac{(2\delta)^2}{3!} f^{(3)} + \frac{(2\delta)^4}{5!} f^{(5)} + \cdots\right) \tag{6}$$

Now, we can get red of the second term of the expansion by multiplying the Eq(3) by 4 then substitute Eq(4). i.e.,

$$4f'(x) - f'(x) = 4\left(f' + \frac{\delta^2}{3!} f^{(3)} + \frac{\delta^4}{5!} f^{(5)} + \cdots\right) - \left(f' + \frac{(2\delta)^2}{3!} f^{(3)} + \frac{(2\delta)^4}{5!} f^{(5)} + \cdots\right) \tag{7}$$

Rearrange,

$$f'(x) = \frac{1}{3}\left(4\left(f' + \frac{\delta^4}{5!} f^{(5)} + \cdots\right) - \left(f' + \frac{(2\delta)^4}{5!} f^{(5)} + \cdots\right)\right) \tag{8}$$

I think this is a good estimation. So, we can write it again in terms of Eq(1) and Eq(4),

$$\boxed{f'(x) = \frac{8f(x+\delta) - 8f(x-\delta) - f(x+2\delta) + f(x-2\delta)}{12\delta}} \tag{9}$$

(b)

We have two sources of errors here. First, the roundoff error which is here

$$e_r \sim \epsilon |f/\delta| \tag{10}$$

Where ϵ is the machine precision. Second, the truncation error which is here (from Eq(8))

$$e_t \sim \left|\delta^4 f^{(5)}\right| \tag{11}$$

To minimize the sum of these two errors,

$$\frac{d}{d\delta}(e_r + e_t) = \frac{d}{d\delta}\left(\epsilon\left|\frac{f}{\delta}\right| + \left|\delta^4 f^{(5)}\right|\right) = -\epsilon\left|\frac{f}{\delta^2}\right| + \left|\delta^3 f^{(5)}\right| = 0 \tag{12}$$

So,

$$\delta = \left(\frac{\epsilon f}{f^{(5)}}\right)^{1/5} \tag{13}$$

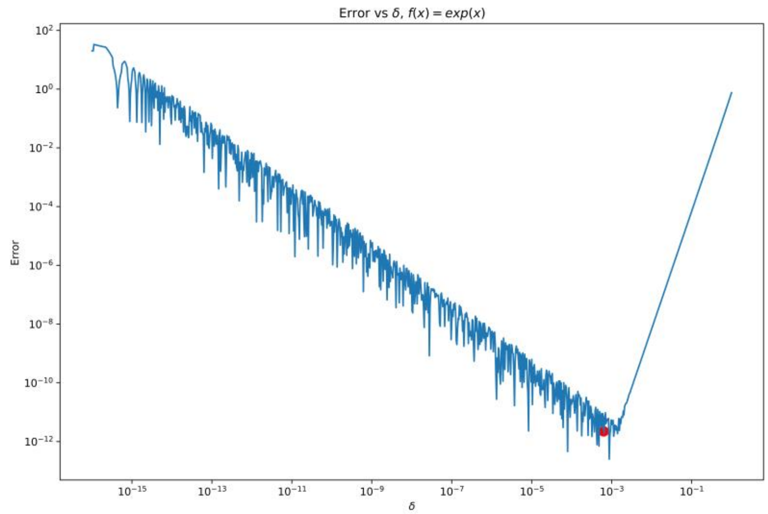
From this equation, the optimal δ for these functions is

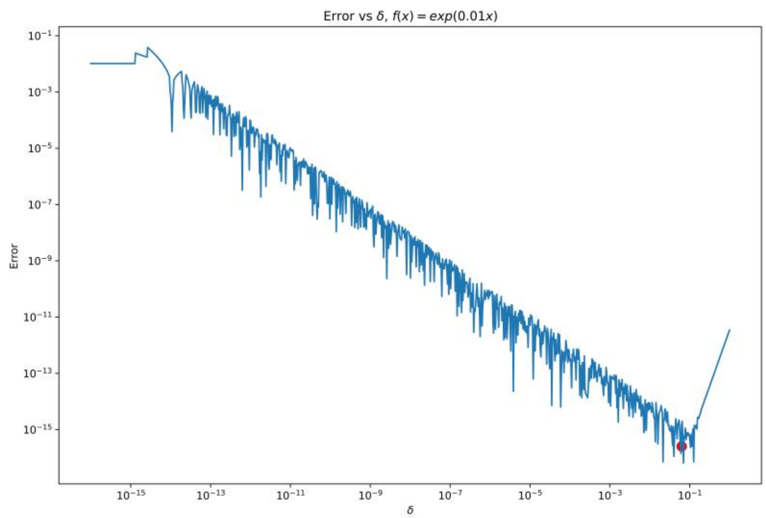
Function	Optimal δ
$\exp(x)$	$(\epsilon)^{1/5}$
$\exp(0.01x)$	$(\epsilon)^{1/5} \times 10^2$

To show this in practice,

```
#===================================================
# Course: PHYS 512 #
Problem: PS1 P1
#===================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
# Department of Physics, McGill University #
September 2022

#===================================================
# Libraries
#===================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs

#===================================================
# Numerical Differentiator
#===================================================
def NDiff(f,x,d): # f:function, x:variable, d:delta
    NDiff = (8 * (f(x + d) - f(x - d)) - f(x + 2*d) + f(x - 2*d)) / (12*d)
    return NDiff

#===================================================
# Optimal δ
#===================================================
E = 10**-16 # machine precision
```

```
# For exp(x)
d1 = E**(1/5)

# For exp(0.01x)
d2 = E**(1/5) * 100

#==================================================
# Functions & Errors
#==================================================
f = np.exp x
= 3
d = 10**np.linspace(-16, 0, 1000)

Error1 = np.abs(np.exp(x) - NDiff(f, x, d))
Error2 = np.abs(0.01*np.exp(0.01*x) - NDiff(f, 0.01*x, 0.01*d)/100) # This a

#==================================================
# Plot
#==================================================
# First Plot for exp(x)
plt.loglog(d, Error1)
plt.scatter(d1, np.abs(np.exp(x) - NDiff(f, x, d1)), c='red', s=80)
plt.title("Error vs $\delta$, $f(x) = exp(x)$")
plt.xlabel("$\delta$")
plt.ylabel("Error")
plt.savefig('Error1 vs delta.png', format='png', dpi=1200)
plt.show()

# Second Plot for exp(0.01x)
plt.close()

plt.loglog(d, Error2)

plt.scatter(d2, np.abs(0.01*np.exp(0.01*x) - NDiff(f, 0.01*x, 0.01*d2)/100),

plt.savefig('Error2 vs delta.png', format='png', dpi=1200)

plt.title("Error vs $\delta$, $f(x) = exp(0.01 x)$")

plt.xlabel("$\delta$")

plt.ylabel("Error")

plt.show()
```





Problem 2

To find the optimal dx, we can use the same method above. So, the roundoff error as before, and the truncation error from Eq(3) is $e_t \sim f'''(dx)^2$. So, to minimize the sum of these two errors

$$\frac{d}{d\delta}(e_r + e_t) = \frac{d}{d(dx)}\left(\epsilon\left|\frac{f}{(dx)}\right| + \left|f^{'''}(dx)^2\right|\right) = -\epsilon\left|\frac{f}{(dx)^2}\right| + \left|f^{'''}(dx)\right| = 0 \qquad (14)$$

So,

$$(dx) = \left(\frac{\epsilon f}{f^{(3)}}\right)^{1/3} \qquad (15)$$

Now, for choosing the function $f(x) = \exp(x)$, and choosing $x = np.linspace(-1, 1, 10)$, with (full=True) in this code:

```python
#==================================================
# Course: PHYS 512
# Problem: PS1 P2
#==================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
# Department of Physics, McGill University
# September 2022

#==================================================
# Libraries
#==================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs

#==================================================
# Function, Derivative, 3rd Derivative & Variable
#==================================================
fun = np.exp
funD = np.exp
funD3 = np.exp
x = np.linspace(-1, 1, 10)

#==================================================
# Numerical Differentiator
#==================================================
def ndiff(fun,x,full):
    dx = (10**-16 * fun(x) / funD3(x))**(1/3) # Optimal dx
    F = (fun(x + dx) - fun(x - dx)) / (2 * dx) # Differentiator
    error = np.abs(F - funD(x)) # Error
    if full == False:
        return F
    if full == True:
        return F, dx, error

#==================================================
# Numerical Differentiator Prototype
#==================================================
NDiff = ndiff(fun,x,full=True)

#==================================================
# Saving Results
#==================================================
result = open("result.out","w")
np.savetxt(result, np.c_[NDiff] )
result.close()
```

we get these results

```
3.678794411710746282e-01 4.641588833612781958e-06 3.677058657558518462e-13
4.594258240405856841e-01 4.641588833612781958e-06 4.659050922839469422e-12
5.737534207482182236e-01 4.641588833612781958e-06 1.078548361732600824e-11
7.165313105838169161e-01 4.641588833612781958e-06 1.002764538071687639e-11
8.948393168187788183e-01 4.641588833612781958e-06 4.409139719996346685e-12
1.117519068748799693e+00 4.641588833612781958e-06 6.936007324043202971e-12
1.395612425099417075e+00 4.641588833612781958e-06 1.332756127681022917e-11
1.742908998645224639e+00 4.641588833612781958e-06 1.176725383800203417e-11
2.176629931744358259e+00 4.641588833612781958e-06 2.811040289429911354e-11
2.718281828443793291e+00 4.641588833612781958e-06 1.525179982309055049e-11
```

As you see, the error (3^{rd} column) is small.

Problem 3

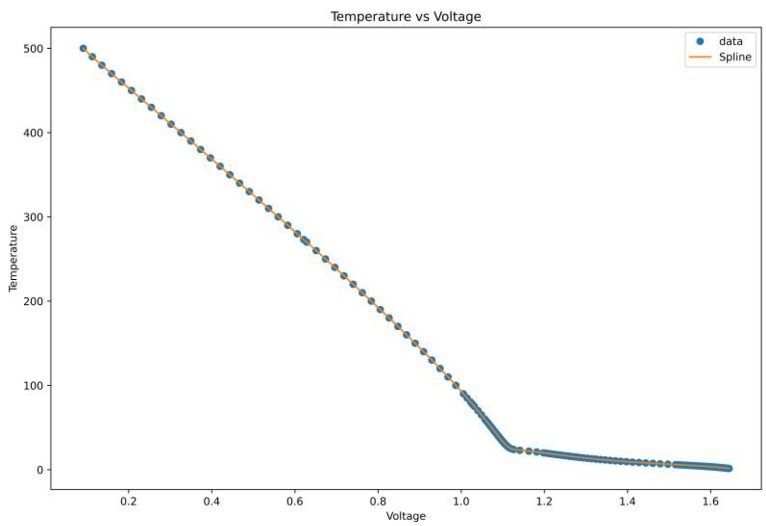
Python has variety of built-in functions for interpolation. In this problem's code, it is convenient to use the spline. To find the error, with considering the function we have, we might find the average of temperature between the nearest data points we have and compare it with the interpolated temperature that we calculated. The code is,

```python
#==================================================
# Course: PHYS 512
# Problem: PS1 P3
#==================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
# Department of Physics, McGill University
# September 2022

#==================================================
# Libraries
#==================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs
from scipy.interpolate import CubicSpline # Fot interpolation

#==================================================
# Function
#==================================================
data = open("lakeshore.txt", 'r')
v, t = [], [] # Voltage & Temperature in the Data
for line in data:
    values = [float(s) for s in line.split()]
    v.append(values[1])
```

```python
    t.append(values[0])

# Sort the data
v, t = zip(*sorted(zip(v, t)))

# Cubic Spline
cs = CubicSpline(v, t)
minv = 0.090681 # Minimum Voltage in Data
maxv = 1.64429 # Maximum Voltage in Data
vs = np.linspace(minv, maxv, 1000) # The period

#==================================================
# Plot the Data Set and the Cubic Spline
#==================================================
plt.plot(v, t, 'o', label='data')
plt.plot(vs, cs(vs), label="Spline")

plt.title("Temperature vs Voltage")
plt.xlabel("Voltage")
plt.ylabel("Temperature")
plt.legend()
plt.show()

#==================================================
# Voltage
#==================================================
V = np.linspace(0.2, 1, 5) # Change this

if type(V) == float:
    V = np.linspace(V, 10**10, 1)

#==================================================
# Prototype
#==================================================
def lakeshore(V,data):
    TT = [] # Temperature Array
    Error = [] # Errors Array
    for j in range(0, len(V)):
        T = cs(V[j]) # Temperature
        TT.append(T)

        # Error = interpolated temperature - average temperature of the 2 ne
        D = [i - V[j] for i in v]
        index = min([i for i in D if i > 0])
        t_avg = (t[D.index(index)-1] + t[D.index(index)]) / 2
        error = abs(T - t_avg)
        Error.append(error)

    TT = [float(i) for i in TT] # Make its elements float
```

```
    return TT, Error

#==================================================
# Results
#==================================================
print(lakeshore(V,data))
```

Here is the data and the interpolated function



First, for choosing number V = 0.5, we get

([325.7502867687234], [0.7502867687234129])

Second, for choosing V = np.linspace(0.2, 1, 5), we get 2 arrays, the first for the interpolated temperatures and the second is the errors.

([452.82275500901153, 368.5104515622143, 282.4640460364361, 192.44416255041457, 92.89964097755849], [2.17724499098847, 3.510451562214314, 2.535953963563884, 2.5558374495854252, 2.100359022441509])

The errors seem reasonable.

Problem 4

First, let's see the code

```
#==================================================
# Course: PHYS 512
# Problem: PS1 P4
#==================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
```

```python
# Department of Physics, McGill University
# September 2022

#==================================================
# Libraries
#==================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs
from scipy.interpolate import CubicSpline # Fot interpolation

#==================================================
# Function
#==================================================

fun = np.cos
x = np.linspace(-np.pi, np.pi, 1001)

order = 9
x_points = np.linspace(-np.pi, np.pi, order)
m = 4
n = 4

#==================================================
# Polynomial Fit
#==================================================
# We have optimized functions for the Polynomial Fit in Python
def PolynomialFit(fun, x, order):
    poly = np.polyval(np.polyfit(x_points, fun(x_points), order), x)
    return poly

#==================================================
# Cubic Spline
#==================================================
def Spline(fun, x):
    cs = CubicSpline(x_points, fun(x_points))
    return cs(x)

#==================================================
# Rational
#==================================================
def Rational(fun, x, order, m, n):
    # This part is totally/partially from Jon with change of variables to su
    pcols=[x_points**k for k in range(n+1)]
    pmat=np.vstack(pcols)

    qcols=[-x_points**k*fun(x_points) for k in range(1,m+1)]
    qmat=np.vstack(qcols)

    mat=np.hstack([pmat.T,qmat.T])
    coeffs=np.linalg.inv(mat)@fun(x_points)
```

```
    num=np.polyval(np.flipud(coeffs[:n+1]),x)
    denom=1+x*np.polyval(np.flipud(coeffs[n+1:]),x)
    rational=num/denom

    return rational

#==================================================
# Call
#==================================================
poly = PolynomialFit(fun, x, order)
cs = Spline(fun, x)
rational = Rational(fun, x, order, m, n)

#==================================================
# Errors
#==================================================
PolyError = fun(x) - poly
SplineError = fun(x) - cs
RationalError = fun(x) - rational

#==================================================
# Plot
#==================================================
# Interpolations Plot
plt.plot(x, fun(x), label='cos(x)')
plt.scatter(x_points, fun(x_points), label='points')
plt.plot(x, poly, label='PolyFit')
plt.plot(x, cs, label="Spline")
plt.plot(x, rational, label="Rational")

plt.title("Cos(x) interpolations, order={}".format(order))
plt.xlabel("x")
plt.ylabel("fun(x)")
plt.legend()
plt.show()
```

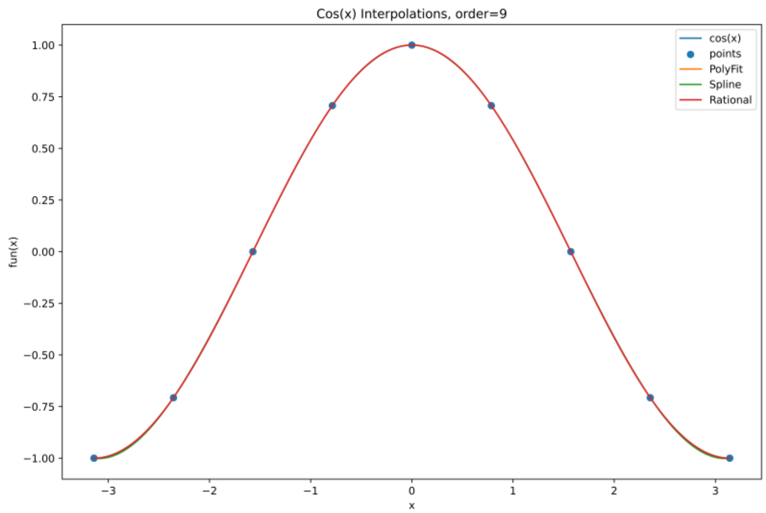
```
# Errors Plot
plt.close()
plt.plot(x, PolyError, label='PolyFit')
plt.plot(x, SplineError, label="Spline")
plt.plot(x, RationalError, label="Rational")

plt.title("Cos(x) interpolations Errors, order={}".format(order))
plt.xlabel("x")
plt.ylabel("Error")
plt.legend()
plt.show()

#==================================================
#==================================================
```

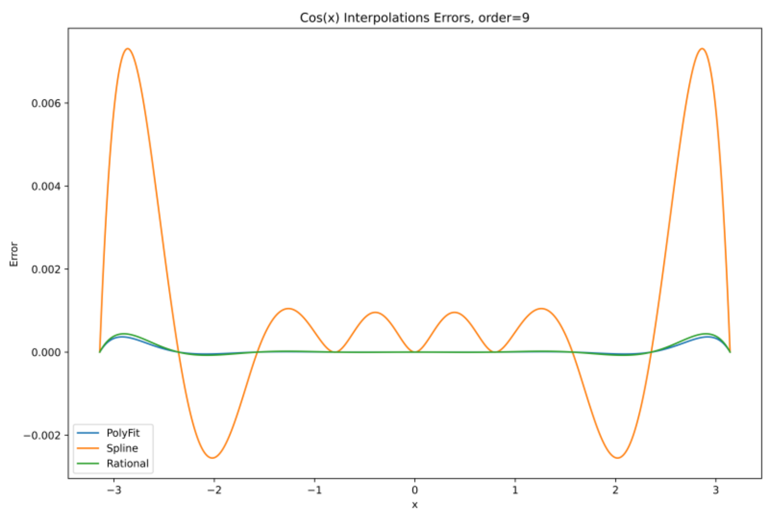
```
#==================================================
# Lorantzian
#==================================================
x_points = np.linspace(-1,1,order)
x = np.linspace(-1,1,1001)

def fun(x):
    fun = 1/(1 + x**2)
    return fun

#==================================================
# Call
#==================================================
poly = PolynomialFit(fun, x, order)
cs = Spline(fun, x)
rational = Rational(fun, x, order, m, n)

#==================================================
# Errors
#==================================================
PolyError = fun(x) - poly
SplineError = fun(x) - cs
RationalError = fun(x) - rational
```

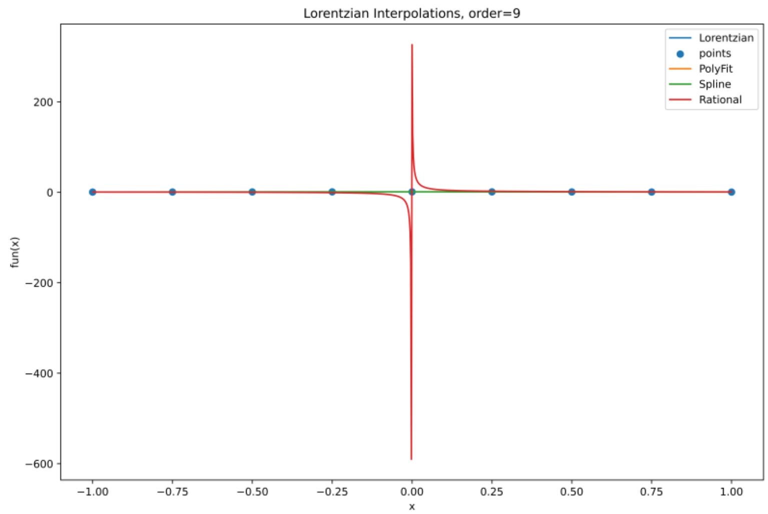
```
#==================================================
# Plot
#==================================================
# Interpolations Plot
plt.close()
plt.plot(x, fun(x), label='Lorentzian')
plt.scatter(x_points, fun(x_points), label='points')
plt.plot(x, poly, label='PolyFit')
plt.plot(x, cs, label="Spline")
plt.plot(x, rational, label="Rational")
plt.title("Lorentzian interpolations, order={}".format(order))
plt.xlabel("x")
plt.ylabel("fun(x)")
plt.legend()
plt.show()

# Errors Plot
plt.close()
plt.plot(x, PolyError, label='PolyFit')
plt.plot(x, SplineError, label="Spline")
plt.plot(x, RationalError, label="Rational")

plt.title("Lorentzian interpolations Errors, order={}".format(order))
plt.xlabel("x")
plt.ylabel("Error")
plt.legend()
```

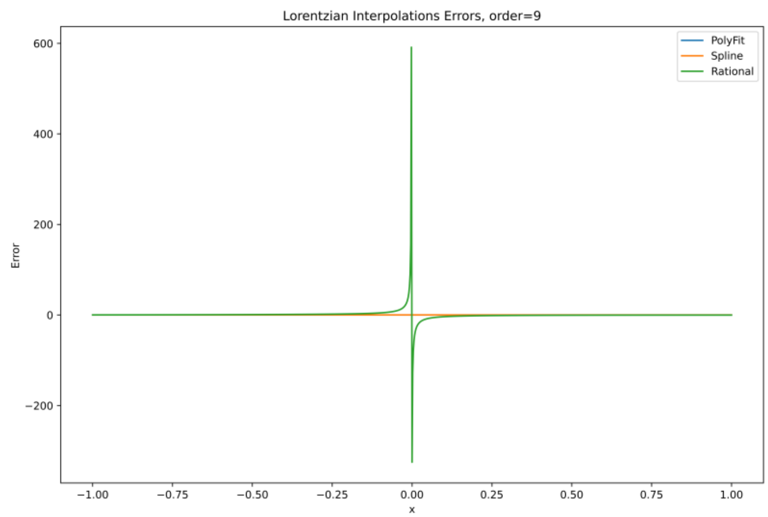
```
plt.show()
```

The produced plots are,

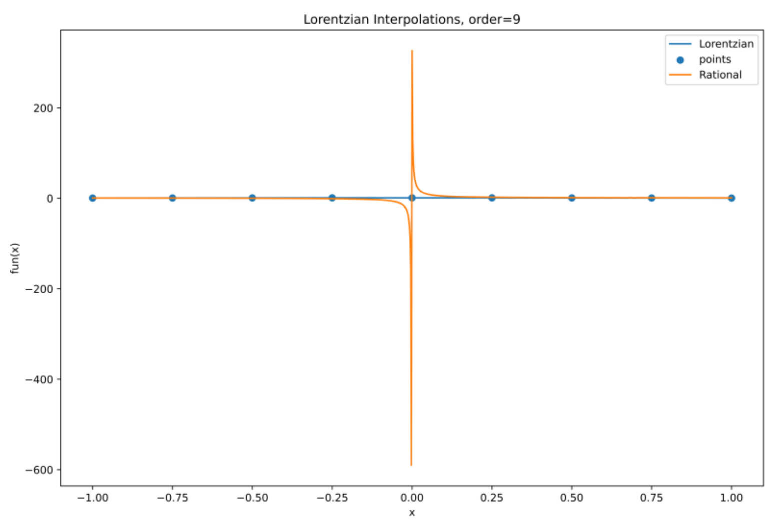


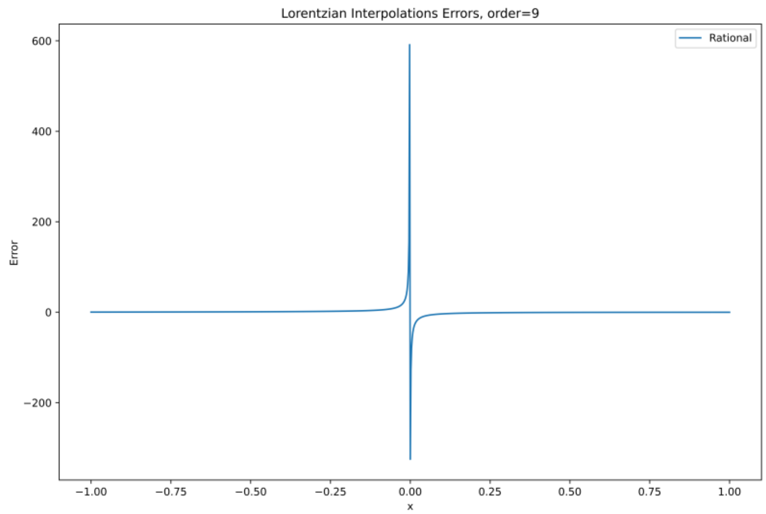




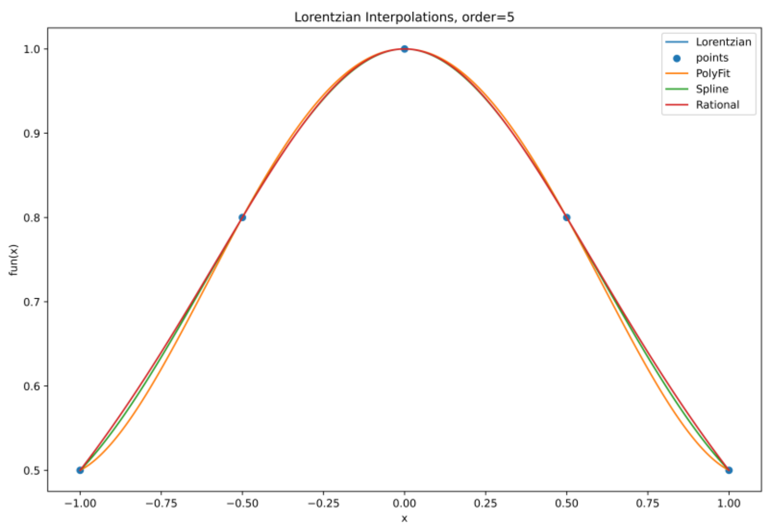


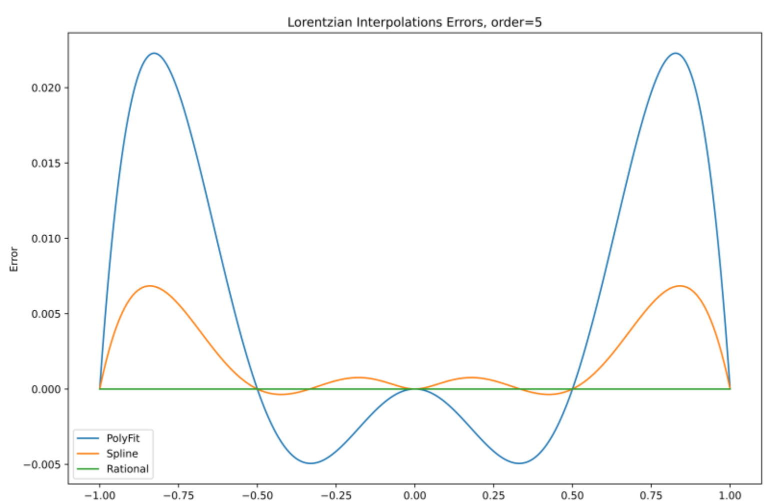
It seems like the rational interpolation gets off! Let’s observe it alone.



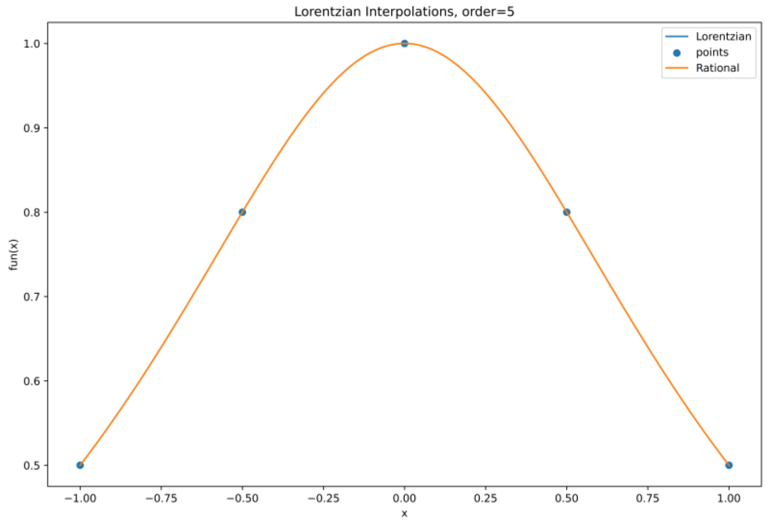


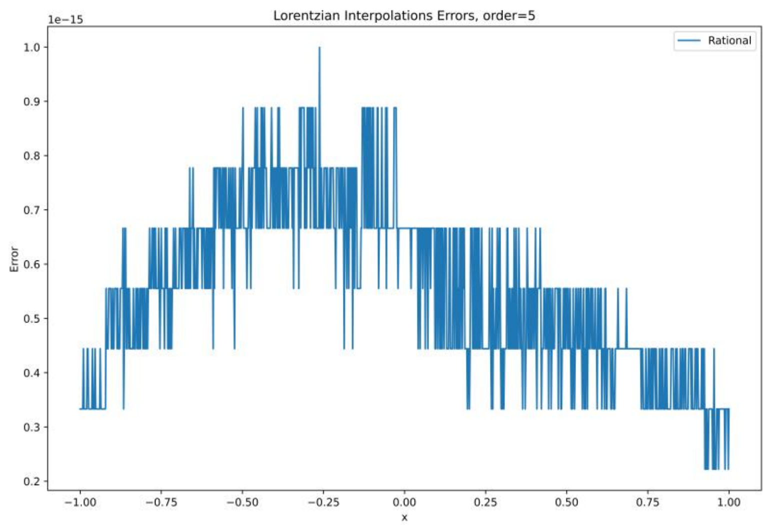
Reducing the order to 5, all interpolations;



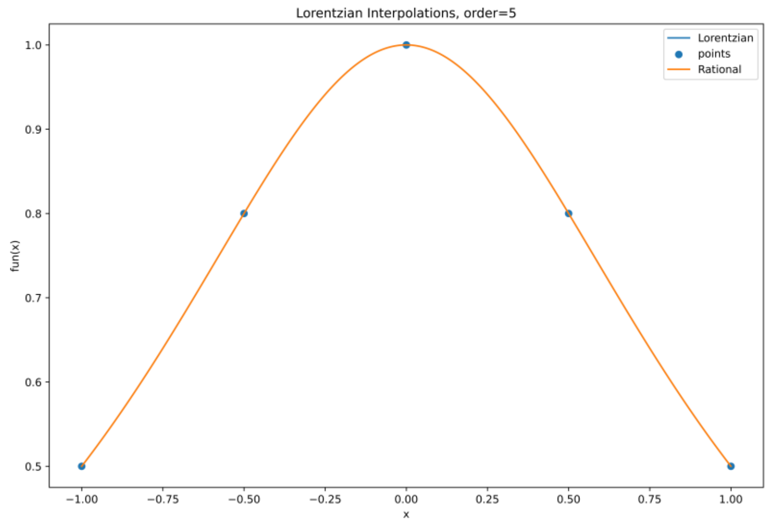


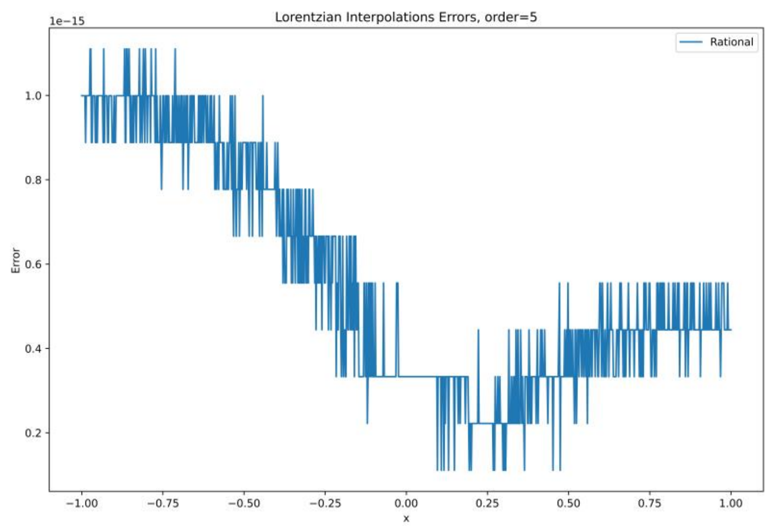
Just the rational interpolation with order 5; m=2, n=2;



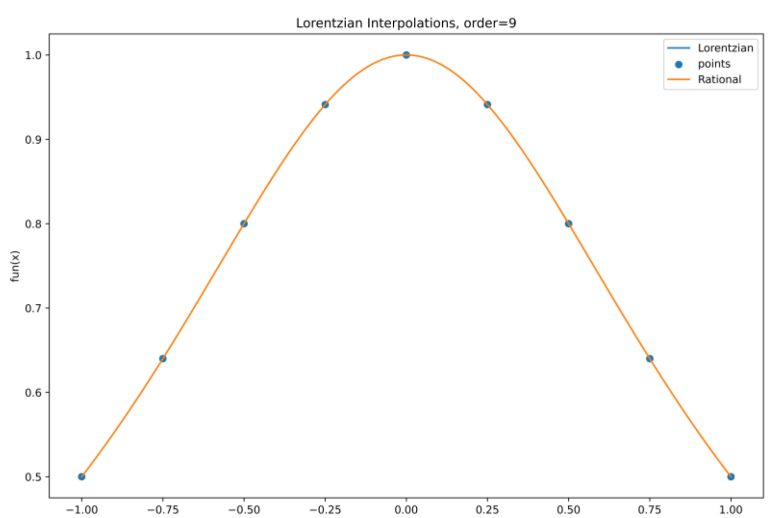


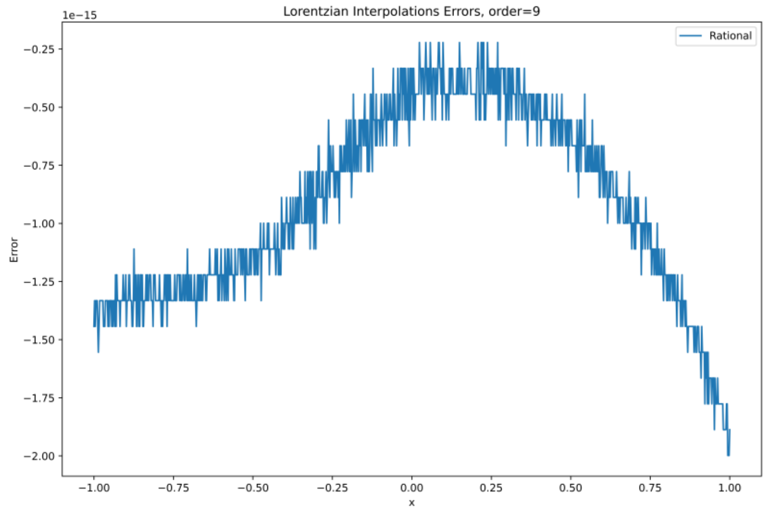
Wow! Its precision near the computer precision! Switch from np.linalg.inv to np.linalg.pinv. The rational interpolation with order 5; m=2, n=2;





Still very precise! Let's see what happened when the order is 9; m=4, n=4;





It return to be super precise… well, my guess is the difference between np.linalg.inv and np.linalg.pinv comes from how they do the calculations. In many cases, when we have calculations around the zero and sometimes, we take the inverse, while it might be both numerator and the dominator are going to zero, and this is usual when we choose higher order interpolations. In other words, np.linalg.pinv is getting nearly exact values, while np.linalg.inv is not due to the machine way in doing calculations!