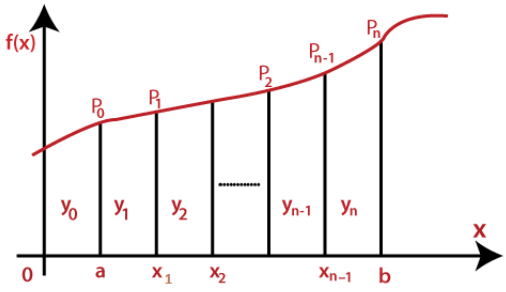
phys 222 HW1 report

Issar Amro

Approximating Integrals

1- Trapezoidal rule



Integration by the trapezoidal rule

For "curvy" functions, approximating the integral over an interval by summing over the area of trapezoids is more effective than the Riemann sum. We can approximate the area under the curve by:

$$\sum_{i=1}^{i=n-1} y_i \Delta x + (y_0 + y_n)\frac{\Delta x}{2}$$

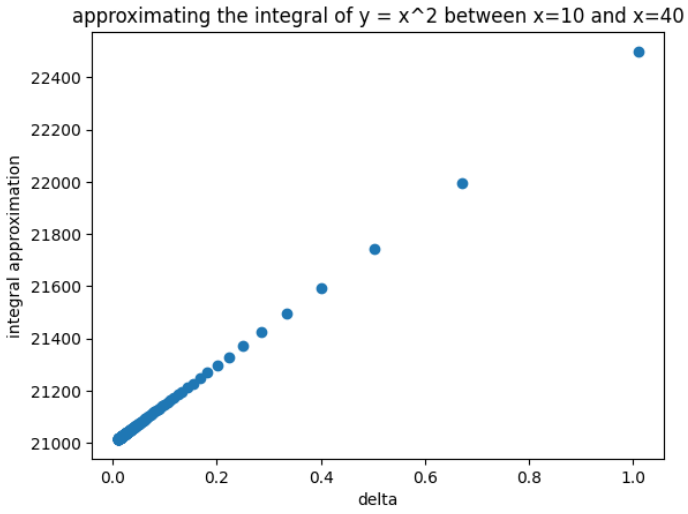
Where n is the number of data points with a given interval $[x_0, x_n]$.

• **First try:** as a first try to code the trapezoidal rule, I generated a data set from the function $y = x^2$ and interpolated over them using L2 optimization method. I fixed my interval using the coefficients I got from the Vandermonde matrix. Then, I started varying the number of data points generated, thus changing the width of the trapezoids, and plotted the outputted values of the approximated integral as a function of the decreasing width of the trapezoids.

code.

```
def TrapRule1(x, x_0, x_n, p_0, p_n, delta):
    approx_int = (p_0 + p_n) * (delta/2)
    for i in range(n):
        if x[i] > x_0 and x[i] < x_n:
            approx_int += y[i] * delta
    return approx_int
approx_int = []
```

```
delta_change = []

x_0 = 10
x_n = 40

for i in range(100, 10000, 50):
   x = np.linspace(-50,50,i)
   delta = x[1]-x[0]
   order = 2
   n = len(x)
   y = x**2
   v = (np.reshape(x, (n, 1)) * np.ones(order + 1)) ** [i for i in range(order, -1, -1)]
   v_new = np.transpose(v) @ v
   y_matrix = np.transpose(v) @ y
   coef_matrix = np.linalg.solve(v_new, y_matrix)
   p_0 = 0
   p_n = 0
   for j in range(order+1):
     p_0 += coef_matrix[len(coef_matrix) - (j+1)]* x_0**j
     p_n += coef_matrix[len(coef_matrix) - (j+1)]* x_n**j
   approx_int.append(TrapRule1(x, x_0, x_n, p_0, p_n, delta))
   delta_change.append(delta)

plt.scatter(delta_change, approx_int1)
plt.ylabel("integral approximation")
plt.xlabel("delta")
plt.title("approximating the integral of y = x^2 between x=10 and x=40")
plt.show()
```



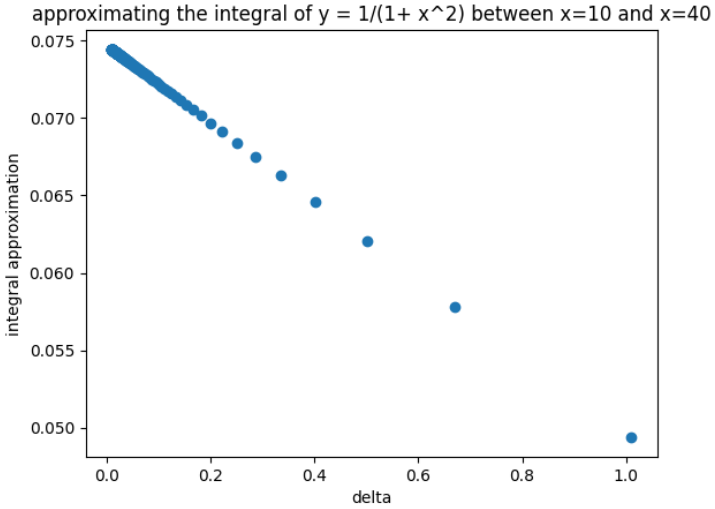
notes.

The integral is

$$\int_{10}^{40} x^2 dx = \frac{40^3 - 10^3}{3} = 21000$$

In the above graph, we see how the approximated value of the integral converges to 21000 as the width decreases (the convergence gets faster as I decrease the width), which is something expected.

Using the above code on the function $y = \frac{1}{1+x^2}$ (with order = 18), we get the below graph



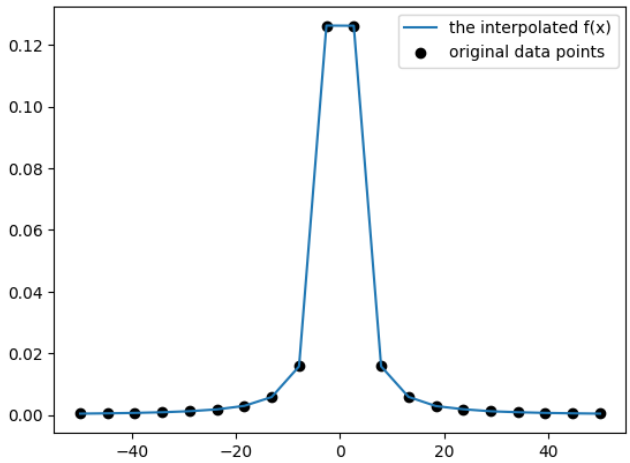
notes.

The integral is

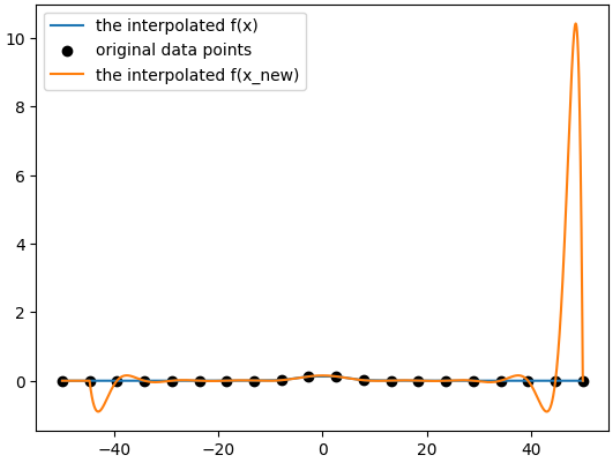
$$\int_{10}^{40} \frac{1}{1+x^2} dx = \arctan(40) - \arctan(10) = 0.07467385887$$

And we see in the above graph the convergence to this number as function of the decreasing width.

• **Second try:** in the first try, I used the generating function to increase the number of my data points. If I do not know the generating function, I cannot generate new data points (which is the case generally). So instead, I tried using the L2 optimization on the data points and globally fit them. Then, I fixed my interval using the coefficients. I then looped over every 2 data points and found the middle point between them then added it to the data set. I did this a couple of times, each time doubling the number of my points and decreasing the spacing between them by half.
I generated a small number of data points initially (to avoid noisy behavior) and I interpolated the original set using L2 optimization (which is very good, no noise):



Then I added to the graph the interpolation of the new set (original data points + the repeatedly found middle points) also using L2 optimization:



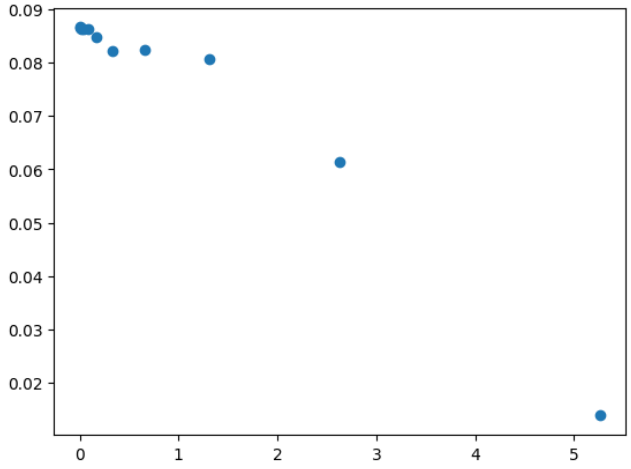
notes.

In the case of the interpolation of the new set, the testing set (the points added to the original set) is way larger than the training set (the original points), hence noise is expected. We see how the new set interpolation diverges sometimes from the original interpolation.
Now, I applied the trapezoidal rule on the new set each time I half the spacing, and plotted the outputted values of the integral as function of the spacing.

code.

```
def TrapRule(x, y,x_0, x_n, p_0, p_n, delta):
    l = len(x)
    approx_int = (p_0 + p_n) * (delta/2)
    for i in range(l):
        if x[i] > x_0 and x[i] < x_n:
```

```
            approx_int += y[i] * delta
    return approx_int
x = np.linspace(-50,50,20)
y = 1/(1+x**2)
delta = x[1] - x[0]
order = 18
n = len(x)
approx_int = []
delta_change = []
v = (np.reshape(x, (n, 1)) * np.ones(order + 1)) ** [i for i in range(order, -1, -1)]
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)

p = 0
for i in range(order+1):
    p += coef_matrix[len(coef_matrix) - (i+1)]* x**i

x_0 = 10
x_n = 30
p_0 = 0
p_n= 0
for j in range(order+1):
     p_0 += coef_matrix[len(coef_matrix) - (j+1)]* x_0**j
     p_n += coef_matrix[len(coef_matrix) - (j+1)]* x_n**j
approx_int.append(TrapRule(x, y,x_0, x_n, p_0, p_n, delta))
delta_change.append(delta)
x_new = x

for o in range(10):
   y_new = 0
   for i in range(1,len(x_new)-1):
     x_t = (x_new[i] + x_new[i+1])/2
     x_new = np.append(x_new, x_t)
   x_new.sort()
   for j in range(order+1):
     y_new += coef_matrix[len(coef_matrix) - (j+1)]* x_new**j
   delta = delta/2
   approx_int.append(TrapRule(x_new, y_new,x_0, x_n, p_0, p_n, delta))
   delta_change.append(delta)

plt.scatter(delta_change, approx_int)
plt.show()
plt.plot(x, p, label = "the interpolated f(x)")
plt.scatter(x,y, label = "original data points", color = "k")
plt.plot(x_new, y_new, label = "the interpolated f(x_new)")
plt.legend()
plt.show()
```



notes.

The integral is

$$\int_{10}^{30} \frac{1}{1+x^2} dx = \arctan(30) - \arctan(10) = 0.06634765661$$

We see in the graph, the approximated value of the integral started very far from the analytical value then it started oscillating about it as I decreased the spacing, but still the final outputted values were larger than the analytical value. Noting that the outputted values seem to stop growing when the spacing got so small (0,00..) hence the final approximated value of the integral was ≈ 0.0865.
The accuracy of the approximations of this method is affected by the noise/ error due to interpolation and the testing error.

2- Simpson's rule

Given an integral $\int_{x_0}^{x_2} f(x)dx$ such that $|x_2 - x_0| = 2h$, we can expand it about the middle point (x_1):

$$\int_{x_0}^{x_2} f(x_1) + f'(x_1)(x - x_1) + f''(x)\frac{(x-x_1)^2}{2!} + f'''(x_1)\frac{(x-x_1)^3}{3!} + \cdots dx$$

$$= f(x_1)(x_2 - x_0) + \left[f'(x)\frac{(x-x_1)^2}{2!} + f''(x_1)\frac{(x-x_1)^3}{3!} + \cdots \right]_{x_0}^{x_2}$$

only the odd powers survive, so we end up with

$$2hf(x_1) + \frac{2h^3}{3!}f''(x_1) + O(h^5) = 2hf(x_1) + \frac{h}{3}f(x_0) - \frac{2h}{3}f(x_1) + \frac{h}{3}f(x_2) + O(h^5)$$

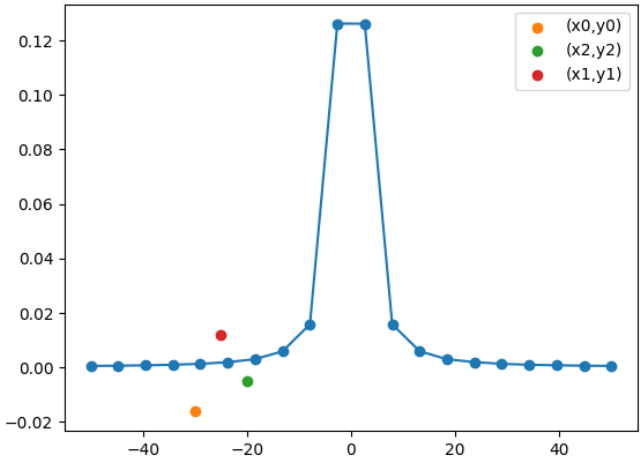
As h gets smaller, the more accurate the result becomes.

• To check how the value of h affects the accuracy of the approximation, I wrote a code to approximate the integrals over some intervals. I used L2 optimization to fix my interval and find the middle point. Then, I checked 2 cases where h was small in one and large in the other:

Small h: for the function $y = \frac{1}{1+x^2}$, I want to approximate

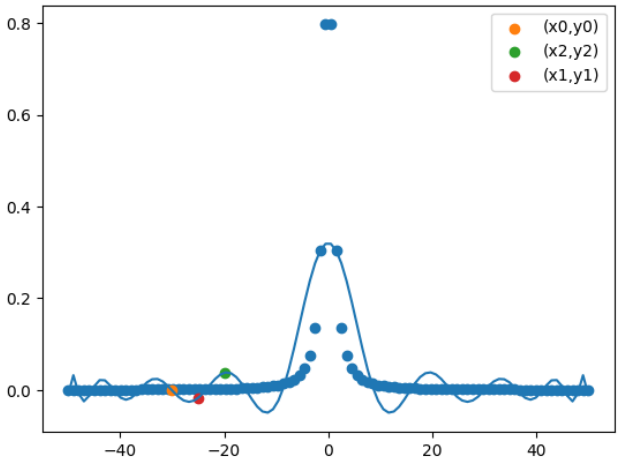
$$\int_{-30}^{-20} \frac{1}{1+x^2} dx = 0.01663739984$$

I used L2 optimization to fix this interval $(-20, -30)$ and fixed the middle point -25 such that $h = 5$. I plotted the interpolated function and these 3 points



notes.

I used the coefficients from L2 optimization to find these 3 points, and it is obvious that they lie in the noisy area of the interpolation. I used a small number of data points (20 points), so the interpolation does not appear noisy but as I increase the number of initially generated data points, the interpolation gets noisy (something we witnessed while coding the L2 optimization method). For 100 points, I got this graph:



notes

In this case, the output value for the integral I am getting is negative so this is bad. I will stick to fewer number of data points.

code.

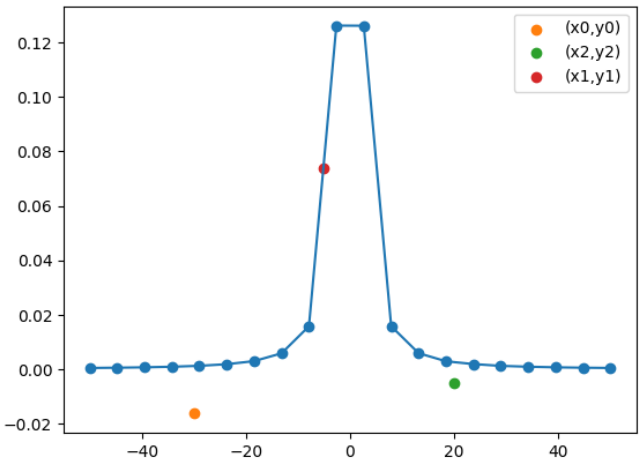
```
def SimpRule(p_0, p_1, p_2, h):
   approx_int = 0
   approx_int += ((4*h)/3)*p_1
   approx_int += (h/3) * p_0
   approx_int += (h/3)*p_2
   return approx_int

x = np.linspace(-50,50,20)
y = 1/(1+x**2)
delta = x[1] - x[0]
order = 18
n = len(x)

y = 1/(1+x**2)
v = (np.reshape(x, (n, 1)) * np.ones(order + 1)) ** [i for i in range(order, -1, -1)]
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)

p=0
x_0 = -30
x_2 = -20
x_1= (x_0 + x_2)/2
h = abs(x_2- x_1)
p_0 = 0
p_2= 0
p_1 = 0
for j in range(order+1):
     p_0 += coef_matrix[len(coef_matrix) - (j+1)]* x_0**j
     p_2 += coef_matrix[len(coef_matrix) - (j+1)]* x_2**j
     p_1 += coef_matrix[len(coef_matrix) - (j+1)]* x_1**j
     p += coef_matrix[len(coef_matrix) - (j+1)]* x**j


approx_int = SimpRule(p_0, p_1, p_2, h)
plt.plot(x,p)
plt.scatter(x,y)
plt.scatter(x_0,p_0, label = "(x0,y0)")
plt.scatter(x_2,p_2, label = "(x2,y2)")
plt.scatter(x_1,p_1, label = "(x1,y1)")
plt.legend()
plt.show()
```

Which outputted this value for the approximated integral: **0.04206470822323488.** So it is obvious that the noise affected the accuracy of the method.

Big h: here, I wanted to see what happens if the h was big, so I tried to approximate

$$\int_{-30}^{20} \frac{1}{1+x^2} dx = 3.058313262$$

I used L2 optimization to fix this interval $(-30, 20)$ and fixed the middle point -5 such that $h = 25$:



Using the above code, I got the value **2.2775349317548614**.

Comapring errors: the error/deviation due to the noise with the small h:

$$0.0420647082232348 - 0.01663739984 = 0.02542730838$$

and the error due to the noise with the big h:

$$3.058313262 - 2.2775349317548614 = 0.7807783302$$

Concluding that the choice of h does affect the approximation with the smaller the h the less the deviation from the analytical value.