2.1)

1)

Barycentric interpolation

f(x) = (∑[i=0 to n] (w\_i / (x - x\_i)) \* y\_i) / (∑[i=0 to n] (w\_i / (x - x\_i)))

w\_i = 1 / (∏[j=0 to n, j≠i] (x\_i - x\_j))

Padé approximation

P(x) = ∑[j=0 to m] (a\_j \* x^j)

Q(x) = ∑[k=0 to n] (b\_k \* x^k)

2)

To compare these methods, there are 2 criterias

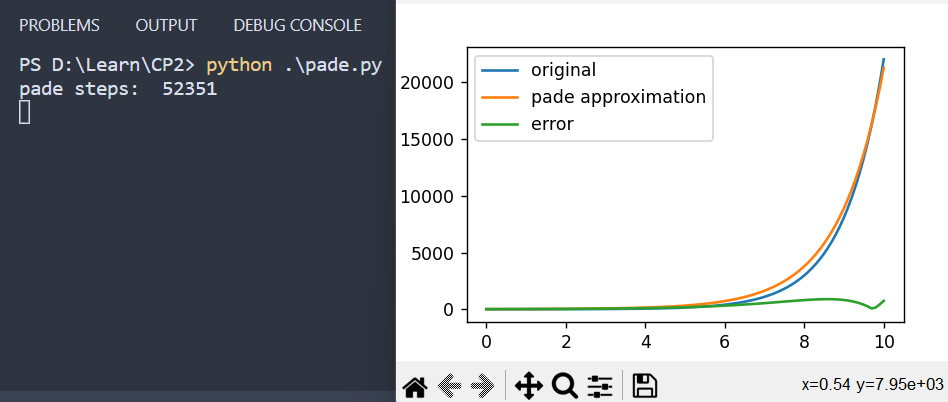
Accuracy: The accuracy criterion assesses how well each method approximates the true function. It evaluates the overall quality of the approximation by comparing the interpolated values to the actual values of the function at the given points. A higher accuracy implies a closer match between the interpolated values and the true function.

Computational efficiency: The computational efficiency criterion examines the efficiency of each method in terms of computational cost and speed. It evaluates the computational resources required to perform the interpolation, such as the number of operations or time complexity. A more efficient method requires fewer resources and is faster in generating the approximations.

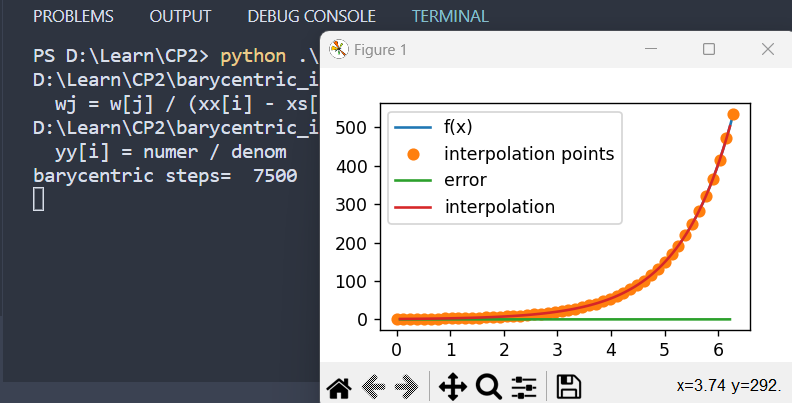
I will test with these 2 criterias and compare with results from code.

3)

This is pade approximation with steps that it needed



This is barycentric with a really low error



And steps.

From this I can conclude that barycentric ration interpolation is better in both ways.

4)

Start with the original image and define the desired scale factor. In this case, we want to make the width and height 2 times bigger.

Calculate the dimensions of the zoomed image by multiplying the original width and height by the 2.

Iterate over each pixel in the zoomed image. For each pixel, check if it falls within the bounds of the original image or outside the bounds (new pixels).

For new pixels, perform interpolation to estimate their values. In this case, we'll use the following approach:

Interpolate on even rows: For pixels on even rows and columns within the original image bounds, use barycentric interpolation or any other desired interpolation method to estimate their values based on neighboring pixels.

Interpolate on even columns: For pixels on even columns and odd rows within the original image bounds, interpolate their values based on neighboring pixels using the same interpolation method.

For odd rows and columns outside the original image bounds, calculate the average of the four nearest pixels in the original image.

Assign the estimated values to the corresponding pixels in the zoomed image.

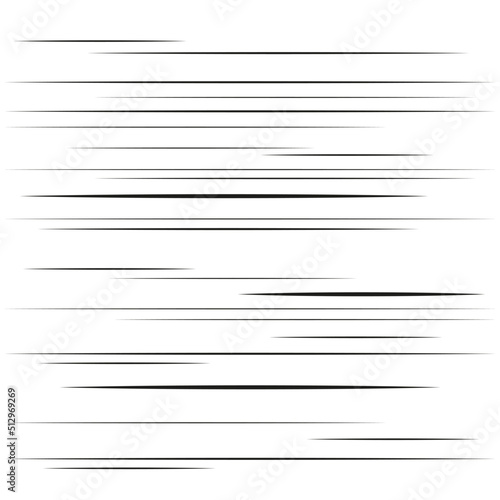
Display the small original image first, and after a delay of 15 seconds, show the zoomed image.

By applying this improved approach, the zoomed image will have interpolated values for the new pixels, blending them with the existing pixels from the original image.

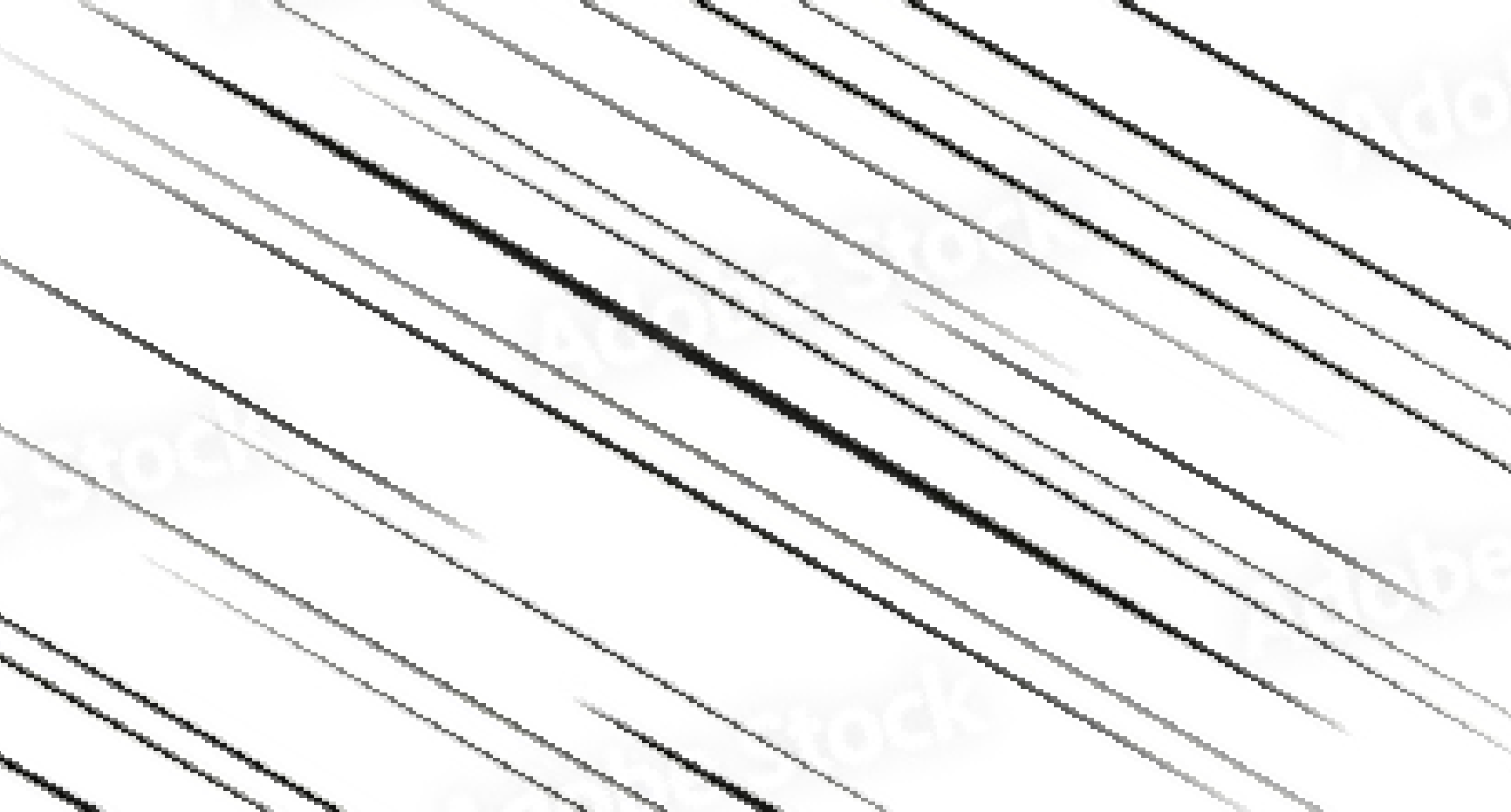
I use from “scipy.interpolate import barycentric\_interpolate” because of it works fast and is good for demonstration, I also have that function written in last exercise so I think it’s fine to use better version of it here.

5)

This is the original image.



I tried to rate without interpolation and as you can see lines are not as smooth as they were in original image, that is because of when I try to rotate concrete coordinate sometimes it ends up float values and can not instantly be interpreted as one of the coordinates, I have to round it.

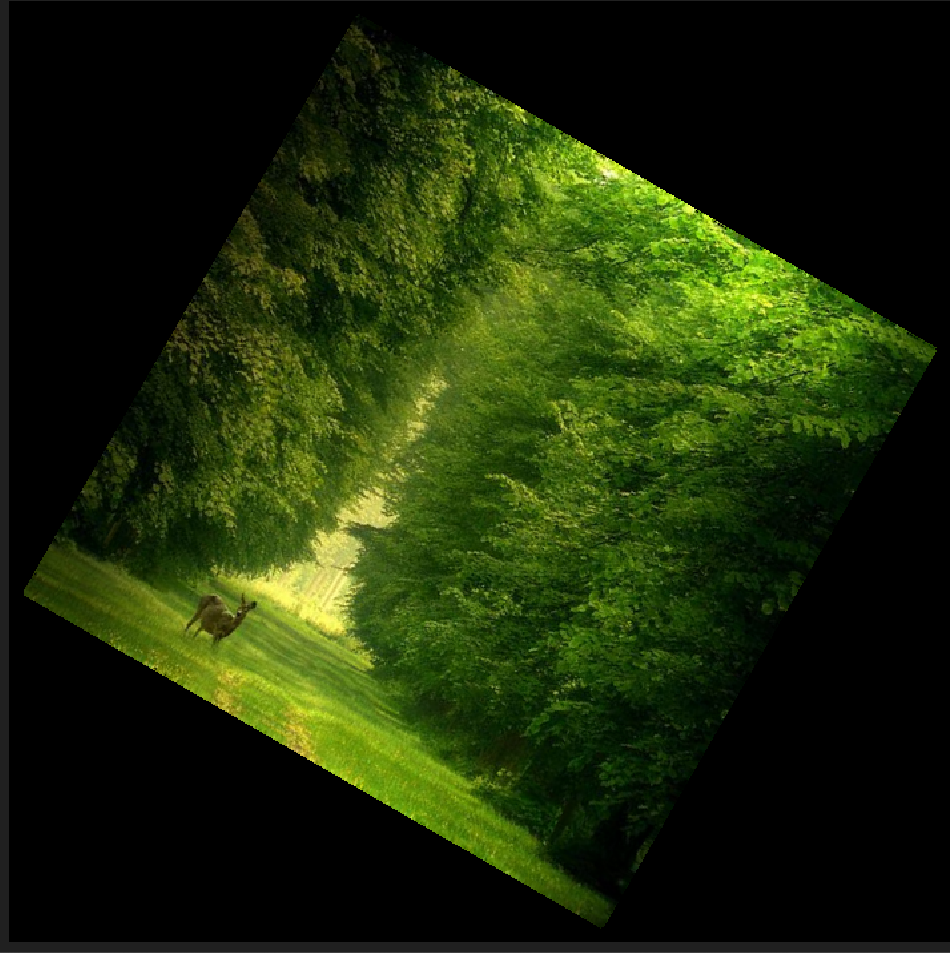


Now I will try to not round and actually interpolate based on pixels around and show the result.

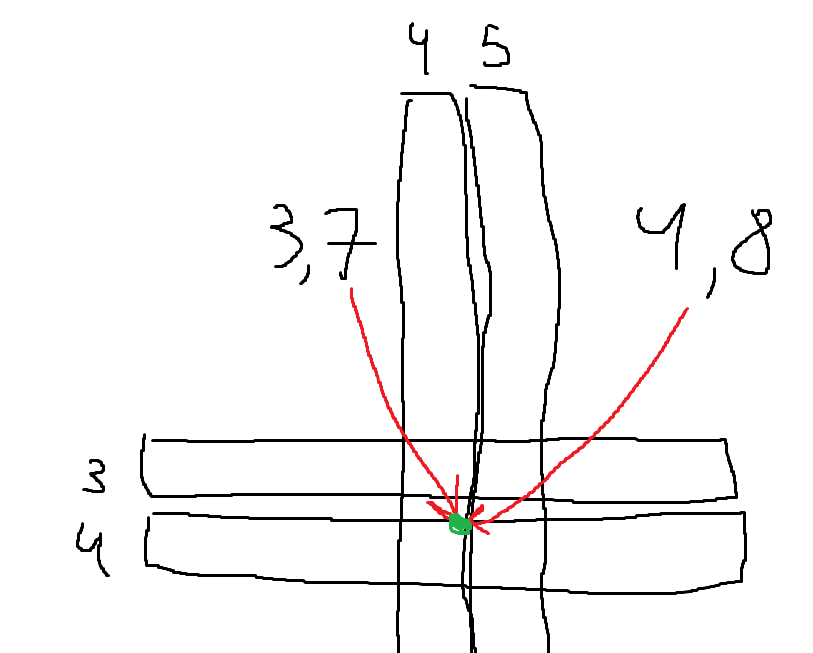
And now this is the interpolated result, it looks more smoother and less noisy.



Some other example,



The way I interpolate is intrepreted here:

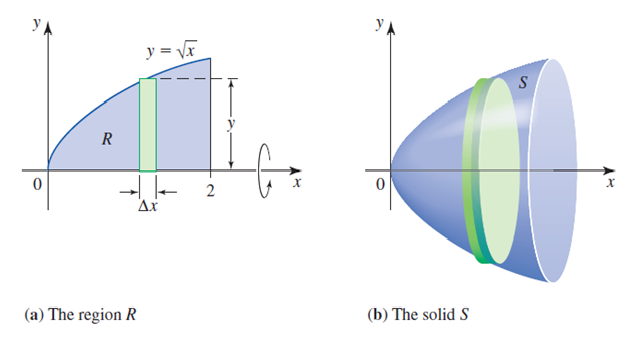


Lets say our coordinates are 3.7 and 4.8 and we want to know what are RGB values.

For this first we fix one column low (floor(4.8) == 4) and find 3.7 on x interpolation so that this becomes linear, we do same for column high (ceil(4.8) = 5) and we find 3.7 on x interpolation there too, after we do same and fix rows and find 4.8 on y interpolation, after this we get 4 results and we take average and say that coordinate 3.7 and 4.7 has that RGB.

2.2)

1 and 3)



The formula for calculating the volume of the solid of revolution is based on the method of disks or the cylindrical shell method, depending on how the solid is obtained. In this case, the formula is based on the method of disks. Given a curve z = f(x) for x ∈ [a, b], the volume of the solid of revolution can be computed as follows:

Divide the interval [a, b] into small subintervals by choosing a sufficiently large value of n.

Compute the width of each subinterval as dx = (b - a) / n.

Initialize the current x-coordinate cur as a.

Initialize the total volume to zero.

Iterate over each subinterval:

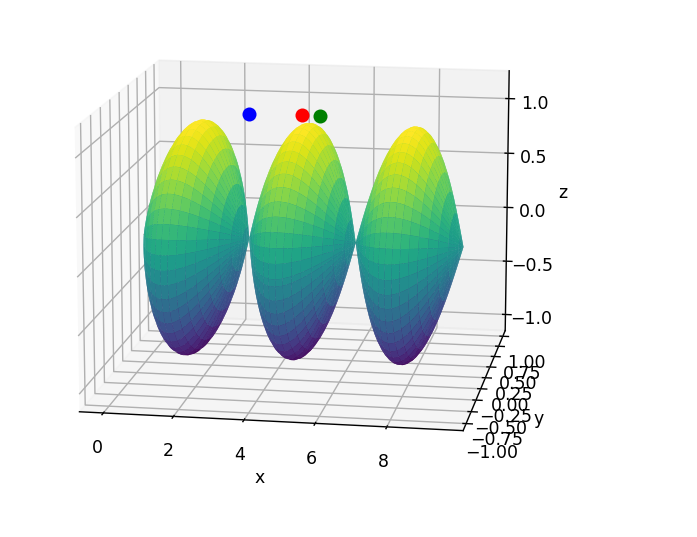
a. Compute the area of the disk at the current x-coordinate as π \* f(cur)^2.

b. Multiply the area by the width of the subinterval to obtain the volume contribution of the disk.

c. Add the volume contribution to the total volume.

d. Update the current x-coordinate by adding dx.

Return the total volume.

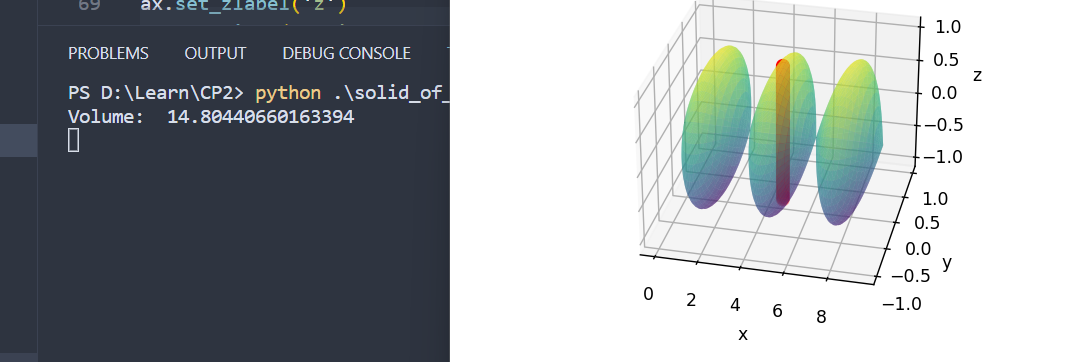


Red ½

Blue 1.3

Green 3/5

If you want to change accuracy u can increase n.



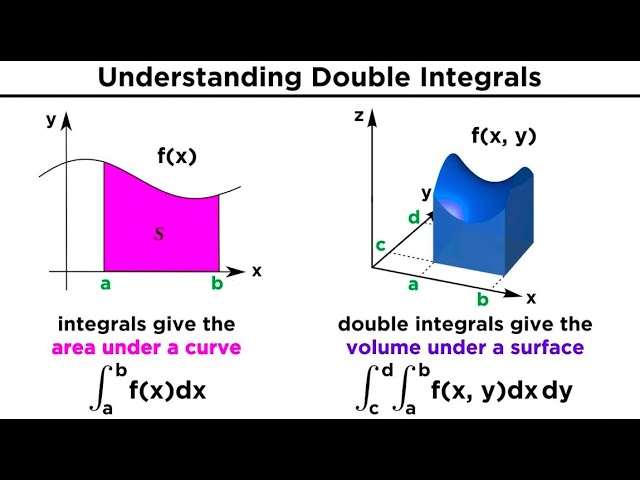
I changed marker to be a line so that it will be seen for any function(even if it has high values)

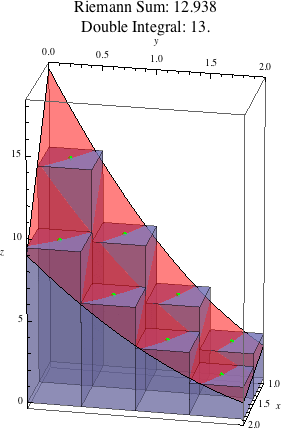
If u want to change and see where will markers be for 1/3 or 3/5, fraction parameter is controlling that.

Now it is 1/2

2 and 3)

Neccesary formulas are just double integral over the surface





The volume(f, a, b, n\_x, n\_y) function calculates the volume of a surface defined by the function f(x, y) over a specified range of integration. It does this by dividing the range of integration into subdivisions along the x and y axes, and then summing up the volume contributions of each subdivision.

The algorithm works as follows:

Compute the width of each subdivision along the x and y axes.

Initialize the total volume variable to 0.

Iterate over the subdivisions along the x axis.

Within the x loop, iterate over the subdivisions along the y axis.

Evaluate the function f(x, y) at each subdivision to obtain the height z.

Compute the base level at the bottom boundary of the surface.

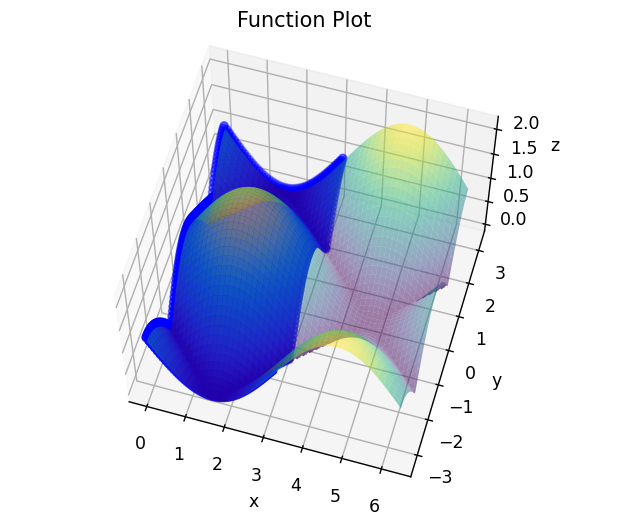
Calculate the volume contribution of each subdivision and accumulate it to the total volume.

Repeat steps 3-7 for all subdivisions.

Return the total volume, representing the approximate volume of the surface.

The function uses the concept of Riemann sums to approximate the volume by summing up the volume contributions of rectangular prisms with infinitesimal height and base areas defined by the function f(x, y) over the specified range of integration.

Please note that the accuracy of the volume approximation depends on the number of subdivisions (n\_x and n\_y), with a higher number of subdivisions resulting in a more accurate approximation.



This is function itself and also blue points which shows some fraction of the volume(in this case 1/2), it can be changed.

In my approach volume can also be negative or zero if the function goes under x and y plane, algorithm basically calculated double integral.

I checked answer with double integral calculator and it is correct.

If you want to also change functions it is defined with the name of ‘f’