Week 10 Riemann

November 21, 2024

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import math
import random
```

0.0.1 Activity 1

This codes the RIEMANN from p362 to include left, right, and midpoint versions.

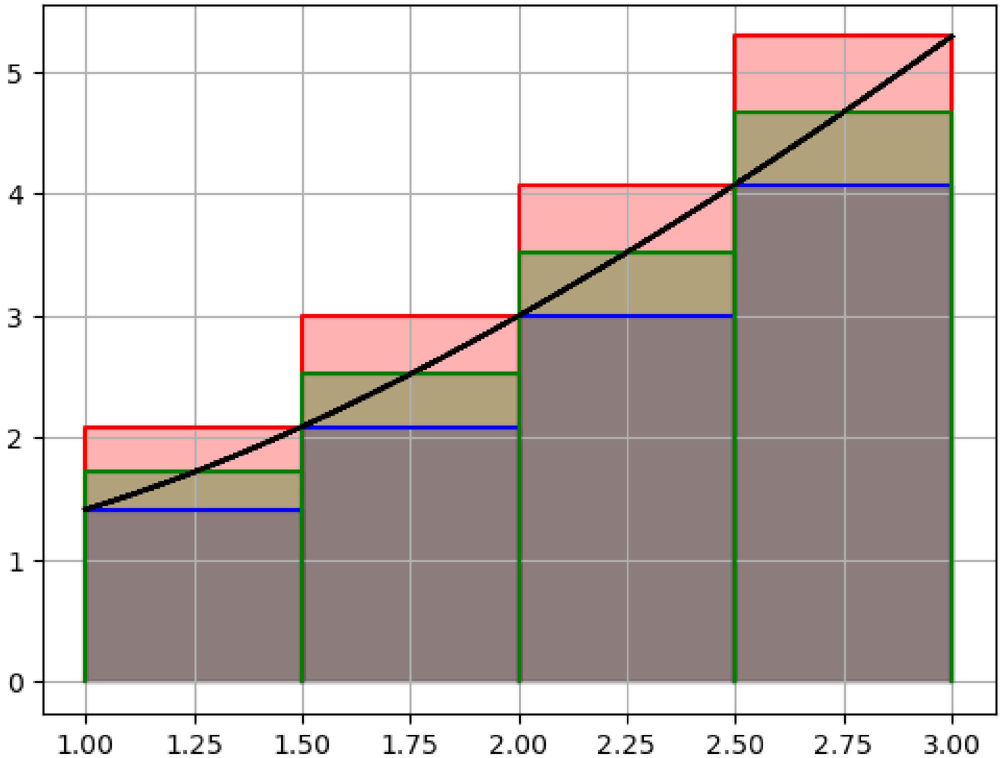
[2]:
```
def fnf(x):
    return np.sqrt(1 + x**3)


def left(func, a, b, numberofsteps, plot, display):
    print("Left Riemann Sum")
    deltax = (b-a)/numberofsteps
    x = a
    accumulation = 0
    for k in range (numberofsteps):
        deltaS = func(x)*deltax
        accumulation = accumulation + deltaS
        if display:
            print(x, deltaS, accumulation)

        if plot:#this plots the box of the Rieman sum boxes if requested
            xplot = [x, x, x+deltax, x+deltax]
            yplot = [0, func(x), func(x), 0]
            plt.plot(xplot, yplot,color="blue")
            plt.fill_between([x, x + deltax], 0, func(x), color="blue", alpha=0.
↪3)  # Shading

            xvals = np.linspace(a, b, 1000)
            plt.plot(xvals, func(xvals), color ="black" )
            plt.grid(True)
            plt.show
            x = x + deltax #set x to for the next interval
    return accumulation
```

```
def right(func, a, b, numberofsteps, plot, display):
    print("Right Riemann Sum")
    deltax = (b-a)/numberofsteps
    x = a + deltax #sets first x to the right point
    accumulation = 0
    for k in range (numberofsteps):

        deltaS = func(x)*deltax
        accumulation = accumulation + deltaS
        if display:
            print(x, deltaS, accumulation)

        if plot:#this plots the box of the Rieman sum boxes if requested
            xplot = [x-deltax, x-deltax, x, x]
            yplot = [0, func(x), func(x), 0]
            plt.plot(xplot, yplot,color="red")
            plt.fill_between([x - deltax, x], 0, func(x), color="red", alpha=0.
 ↪3)

             #plots the original function
            xvals = np.linspace(a, b, 1000)
            plt.plot(xvals, func(xvals), color ="black" )
            plt.grid(True)
            plt.show

        x = x + deltax
    return accumulation


def midpoint(func, a, b, numberofsteps, plot, display):
    print("Midpoint Riemann Sum")
    deltax = (b-a)/numberofsteps #sets first x to the midpoint
    x = a + deltax/2
    accumulation = 0
    for k in range (numberofsteps):
        deltaS = func(x)*deltax
        accumulation = accumulation + deltaS

        if display:
            print(x, deltaS, accumulation)

        if plot:#this plots the box of the Rieman sum boxes if requested
            xplot = [x-deltax/2, x-deltax/2, x+deltax/2, x+deltax/2]
            yplot = [0, func(x), func(x), 0]
            plt.plot(xplot, yplot,color="green")
```

```
            plt.fill_between([x - deltax / 2, x + deltax / 2], 0, func(x),␣
↪color="green", alpha=0.3)  # Shading
            xvals = np.linspace(a, b, 1000)
            plt.plot(xvals, func(xvals), color ="black" )
            plt.grid(True)
            plt.show

        x = x + deltax
    return accumulation

a = 1
b = 3
numberofsteps = 4

left (fnf, a, b, numberofsteps, True, True)
right (fnf, a, b, numberofsteps, True, True)
midpoint (fnf, a, b, numberofsteps, True, True)
```

```
Left Riemann Sum
1 0.7071067811865476 0.7071067811865476
1.5 1.0458250331675945 1.7529318143541421
2.0 1.5 3.252931814354142
2.5 2.038688303787511 5.291620118141653
Right Riemann Sum
1.5 1.0458250331675945 1.0458250331675945
2.0 1.5 2.5458250331675947
2.5 2.038688303787511 4.584513336955106
3.0 2.6457513110645907 7.230264648019697
Midpoint Riemann Sum
1.25 0.85923294280422 0.85923294280422
1.75 1.260890062614501 2.120123005418721
2.25 1.7600159800410904 3.880138985459811
2.75 2.3343561746228874 6.2144951600826985
```

[2]: 6.2144951600826985



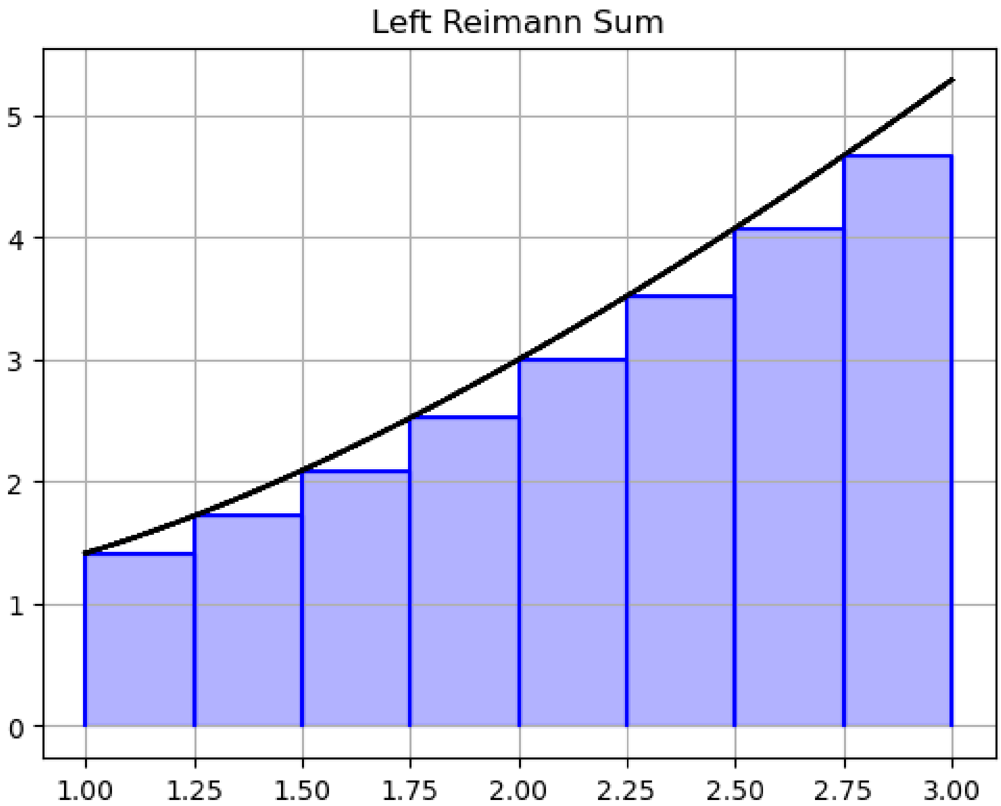
0.0.2 Activity 2

For each of the 3 versions of RIEMANN, create a single "hero" graphic that illustrates the difference between the 3 methods.

[3]:
```
left (fnf, a, b, numberofsteps*2, True, False)
plt.title('Left Reimann Sum')
```

```
Left Riemann Sum
```

[3]:
```
Text(0.5, 1.0, 'Left Reimann Sum')
```

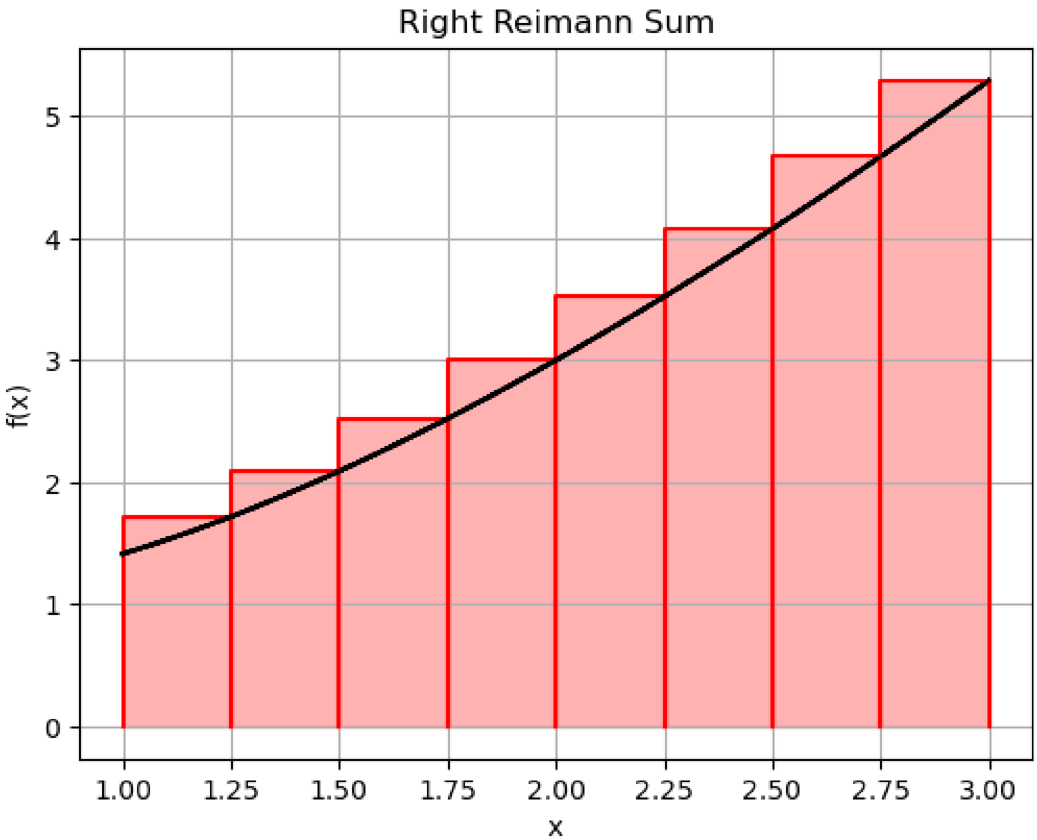


The Left Riemann Sum calculates based on the value of the left endpoint of each subinterval. In the case of an increasing function like above, this means the value will be slightly less than the actual function as demonstrated by the white space between the approximations and the function. If the function was decreasing, this would result in an over estimation.

[4]:
```
right (fnf, a, b, numberofsteps*2, True, False)
plt.title('Right Reimann Sum')
plt.ylabel("f(x)")
plt.xlabel("x")
```

```
Right Riemann Sum
```

[4]:
```
Text(0.5, 0, 'x')
```

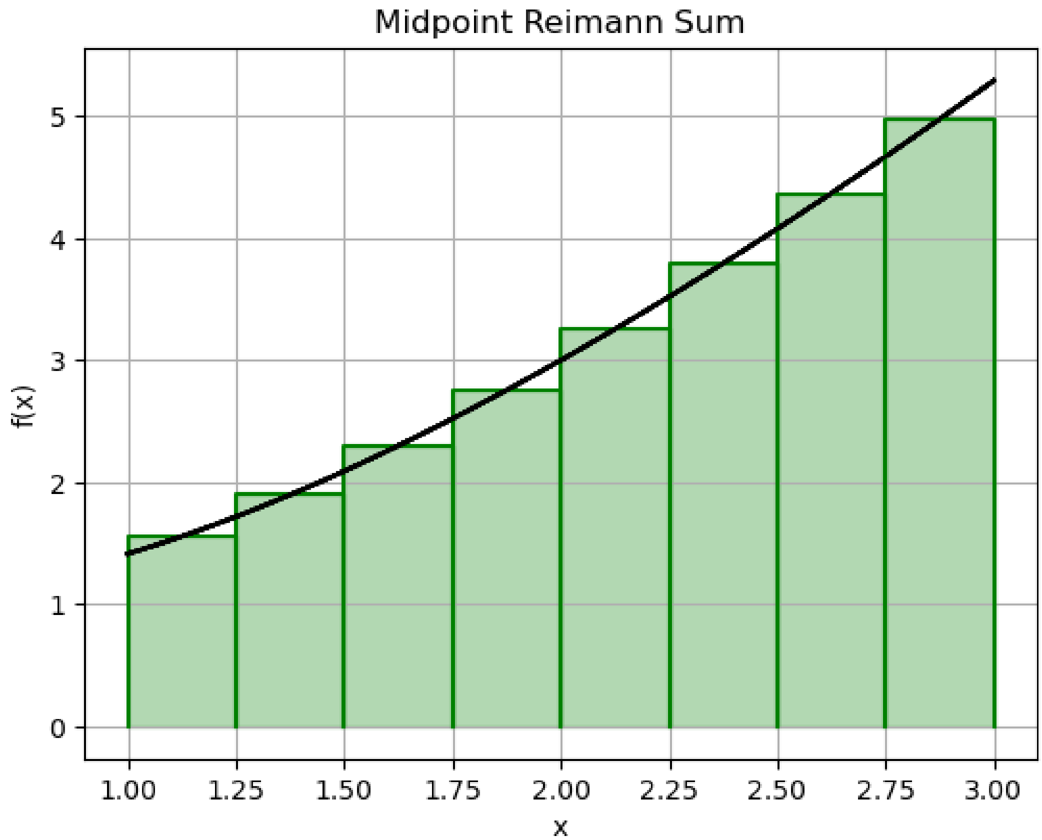


The Right Reimann Sum calculates using the right side of each subinterval. In a case of an increasing function, this results in an sum that is slightly larger than the actual area under the function. As with the Left Sum, this would be the opposite compared to a decreasing function.

[5]:
```
midpoint (fnf, a, b, numberofsteps*2, True, False)
plt.title('Midpoint Reimann Sum')
plt.ylabel("f(x)")
plt.xlabel("x")
```

```
Midpoint Riemann Sum
```

[5]:
```
Text(0.5, 0, 'x')
```



The midpoint Riemann Sum calculates at the midpoint of each subinterval and therefore has some error above and below the function which averages out much closer to the actual integral with less required subintervals. The error is also less dependent on whether the function is increasing or decreasing.

0.0.3 Activity 3

Experiment with questions 4 - 12 on pages 366 - 368, either or on your own, coding as much or as little as you need to show the point. Once you have the code this should be pretty quick. Add a summary of the advantages and disadvantages of the three methods.

[6]:
```
#4a) Calculate left endpoint Riemann sums for the function √(1 + x3) on
#the interval [1, 3] using 40, 400, 4000, and 40000 equally-spaced subintervals.
#How many ddecimal points in this sequence have stabilized?

numsteps = [40, 400, 4000, 40000, 400000]
a = 1
b = 3
for num in numsteps:
    print (num, left(fnf,a, b, num, False, False))
```

```
Left Riemann Sum
40 2.8284271247461894
Left Riemann Sum
400 2.828427124746173
Left Riemann Sum
4000 2.828427124746021
Left Riemann Sum
40000 2.828427124744823
Left Riemann Sum
400000 2.8284271247403834
```

The first two decimal places have stabilized in this sum, three when I increased the number of subintervals to 400000.

[7]:
```
# 4b)The left endpoint Riemann sums for √1 + x3 on the interval [1, 3] seem
# to be approaching a limit as the number of subintervals increases without
# bound. Give the numerical value of that limit, accurate to four decimal
# places.

left (fnf,a, b, 10000000, False, False)
```

```
Left Riemann Sum
```

[7]:
```
2.8284271248021677
```

The limit I found is 6.2299 four decimal places.

[8]:
```
#4c) Calculate left endpoint Riemann sums for the function √1 + x3 on the
#interval [3, 7]. Construct a sequence of Riemann sums using more and more
#subintervals, until you can determine the limiting value of these sums, accu-
#rate to three decimal places. What is that limit?


numsteps = [40, 400, 4000, 40000, 400000]
a = 3
b = 7
for num in numsteps:
    print (num, left(fnf, a, b, num, False, False))
```

```
Left Riemann Sum
40 21.166010488516733
Left Riemann Sum
400 21.166010488516815
Left Riemann Sum
4000 21.166010488515166
Left Riemann Sum
40000 21.166010488531597
Left Riemann Sum
```

400000 21.16601048852531

The limit when applied between 3 and 7 approaches 45.819 accurate to three decimal places.

[9]:
```
#d) Calculate left endpoint Riemann sums for the function √1 + x3 on the
#interval [1, 7] in order to determine the limiting value of the sums to three
#decimal place accuracy. What is that value? How are the limiting values in
#parts (b), (c), and (d) related? How are the corresponding intervals related?

numsteps = [40, 400, 4000, 40000, 400000]
a = 1
b = 7
for num in numsteps:
    print (num, left(fnf, a, b, num, False, False))
```

```
Left Riemann Sum
40 8.485281374238562
Left Riemann Sum
400 8.485281374238628
Left Riemann Sum
4000 8.485281374238356
Left Riemann Sum
40000 8.485281374242264
Left Riemann Sum
400000 8.48528137424832
```

[10]:
```
6.2299+45.8199
```

[10]: 52.0498

The limit from 1 to 7 is 52.0498 which is the same as the limit from 1 to 3 added to the limit from 3 to 7 as demonstrated above.

5. Modify RIEMANN so it will calculate a Riemann sum by sampling the given function at the midpoint of each subinterval, instead of the left end- point. Describe exactly how you changed the program to do this.

The RIEMANN program was already modified above to use the midpoint of each subinterval. It was done by adding half the interval (deltax/2) to the starting value of x (a in this case) to get a point halfway through each interval. Then adding deltax to that value has a value of x midway between the next interval. The rest of the code did not need to be modified.

[11]:
```
#6. a) Calculate midpoint Riemann sums for the function √1 + x3 on the
#interval [1, 3] using 40, 400, 4000, and 40000 equally-spaced subintervals.
#How many decimal points in this sequence have stabilized?

numsteps = [40, 400, 4000, 40000]
a = 1
b = 3
for num in numsteps:
```

```
    print (num, midpoint(fnf,a, b, num, False, False))
```

```
Midpoint Riemann Sum
40 6.229804122734282
Midpoint Riemann Sum
400 6.229957835175769
Midpoint Riemann Sum
4000 6.229959372356611
Midpoint Riemann Sum
40000 6.229959387732263
```

It appears 7 decimal points have stabilized for this sequence.

6. b) Roughly how many subintervals are needed to make the midpoint Riemann sums for √1 + x3 on the interval [1, 3] stabilize out to the first four digits? What is the stable value? Compare this to the limiting value you found earlier for left endpoint Riemann sums. Is one value larger than the other; could they be the same?

It appears the midpoint was stabilized to four digits after 400 intervals at 6.2299. This is the same as the limiting value I found on for the left endpoint, though overall this is slightly bigger. This makes sense given where the interval is taken, however if you take the interval out to infinity they will converge to the same number.

6. c) Comment on the relative "efficiency" of midpoint Riemann sums versus left endpoint Riemann sums (at least for the function √1 + x3 on the interval [1, 3]). To get the same level of accuracy, an efficient calculation will take fewer steps than an inefficient one.

The midpoint Riemann sum was significantly more efficient than the left, to get to stabilized for decimals the midpoint only took 400 intervals while the left required more than 4000 and closer to 40000 to fully stabilize.

[12]:
```
#7. a) Modify RIEMANN to calculate right endpoint Riemann sums, and
#use it to calculate right endpoint Riemann sums for the function √1 + x3 on
#the interval [1, 3] using 40, 400, 4000, and 40000 equally-spaced subintervals.
#How many digits in this sequence have stabilized?

numsteps = [40, 400, 4000, 40000, 400000]
a = 1
b = 3
for num in numsteps:
    print (num, right(fnf,a, b, num, False, False))
```

```
Right Riemann Sum
40 6.327202149550815
Right Riemann Sum
400 6.239655715949005
Right Riemann Sum
4000 6.230928741202806
Right Riemann Sum
```

```
40000 6.2300563204246
Right Riemann Sum
400000 6.2299690810849775
```

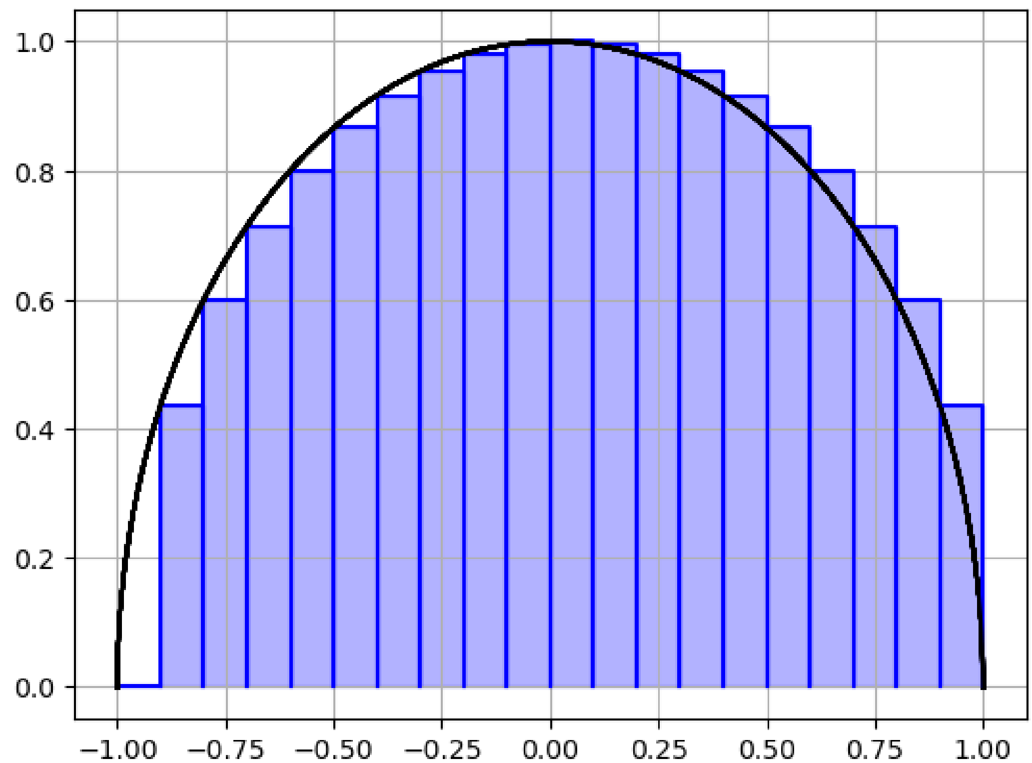
The right endpoint stabilized to three digits at 40000 subintervals, I bumped it up one more order of magnitude to get it to stabilize at the same value as left and midpoint.

7. b) Comment on the efficiency of right endpoint Riemann sums as compared to left endpoint and to midpoint Riemann sums—at least as far as the function √1 + x3 is concerned.

Right endpoint feels as efficient as left for stabilizing, but neither appears to be as efficient as midpoint. They also both converge from different directions (right from a higher value in this case).

[13]:
```
#8. Calculate left endpoint Riemann sums for the function
#f (x) = √1− x2 on the interval [−1, 1].
#Use 20 and 50 equally-spaced subintervals. Compare your values with the
#estimates for the area of a semicircle given on page 356.
def circle(x):
    return np.sqrt(1 - x**2) #redefine fnf for the new function

print("20 Intervals from -1 to 1: ", left(circle,-1,1,20,True, False))
print("50 Intervals from -1 to 1: ", left(circle,-1,1,50,False, False))
```

```
Left Riemann Sum
20 Intervals from -1 to 1:  1.5522591631241593
Left Riemann Sum
50 Intervals from -1 to 1:  0.0
```



I was able to get the same answers for both 20 and 50 intervals as on page 356.

[14]:
```
#9. a) Calculate left endpoint Riemann sums for the function
#f (x) = √1 + cos2 x on the interval [0, ].
#Use 4 and 20 equally-spaced subintervals. Compare your values with the
#estimates for the length of the graph of y = sin x between 0 and  , given on
#page 358.
def nine(x):
    return np.sqrt(1 + np.square(np.cos(x)))

a = 0
b = np.pi

print("4 Intervals from 0 to Pi: ", left(nine,a,b,4,False, False))
print("20 Intervals from 0 to Pi: ", left(nine,a,b,20,False, False))
```

```
Left Riemann Sum
4 Intervals from 0 to Pi:  4.442882938158366
Left Riemann Sum
20 Intervals from 0 to Pi:  4.442882938158367
```

These sums closely match the values for the lenght of sin(x) between 0 and Pi given on page 358 with only the 20 segment requiring some rounding to get the fourth decimal place.

[15]:
```
#9. b) What is the limiting value of the Riemann sums, as the number of subin-
#tervals becomes infinite? Find the limit to 11 decimal places accuracy.
#10. Calculate left endpoint Riemann sums for the function




for i in range(1,10):
    num = i**3
    print(num, " Intervals from 0 to Pi: ", left(nine,a,b,num,False, False))
```

```
Left Riemann Sum
1  Intervals from 0 to Pi:  4.442882938158366
Left Riemann Sum
8  Intervals from 0 to Pi:  4.442882938158366
Left Riemann Sum
27  Intervals from 0 to Pi:  4.442882938158366
Left Riemann Sum
64  Intervals from 0 to Pi:  4.442882938158366
Left Riemann Sum
125  Intervals from 0 to Pi:  4.442882938158371
Left Riemann Sum
216  Intervals from 0 to Pi:  4.442882938158375
Left Riemann Sum
343  Intervals from 0 to Pi:  4.442882938158371
Left Riemann Sum
512  Intervals from 0 to Pi:  4.442882938158365
Left Riemann Sum
729  Intervals from 0 to Pi:  4.442882938158346
```

The limit to 11 decimal places is 3.82019778902 which amazingly was reached after only 27 intervals.

[16]:
```
#10. Calculate left endpoint Riemann sums for the function
#f (x) = cos(x^2) on the interval [0, 4],
#using 100, 1000, and 10000 equally-spaced subintervals.

def ten(x):
    return np.cos(x**2)

a = 0
b = 4
intervals = [100, 1000, 10000]
for num in intervals:

    print(num, " Intervals from 0 to 4: ", left(ten,a,b,num,False, False))
```

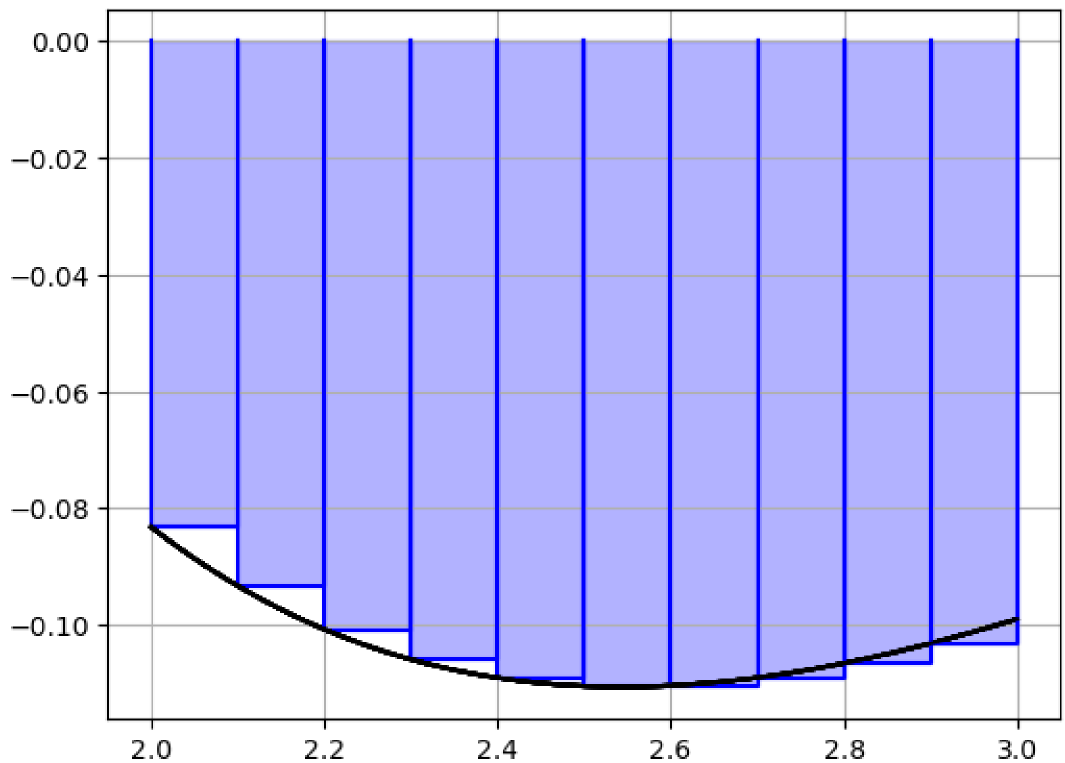
```
Left Riemann Sum
100  Intervals from 0 to 4:  4.000000000000003
Left Riemann Sum
```

```
1000  Intervals from 0 to 4:  4.000000000000003
Left Riemann Sum
10000  Intervals from 0 to 4:  3.9999999999996247
```

[17]:
```python
#11. Calculate left endpoint Riemann sums for the function
#f (x) = cos x /(1 + x^2) on the interval [2, 3],
#using 10, 100, and 1000 equally-spaced subintervals. The Riemann sums are
#all negative; why? (A suggestion: sketch the graph of f . What does that tell
#you about the signs of the terms in a Riemann sum for f ?)

def eleven(x):
    return np.cos(x)/(1+x**2)

a = 2
b = 3
intervals = [100, 1000, 10000]
for num in intervals:

    print(num, " Intervals from 2 to 3: ", left(eleven,a,b,num,False, False))

left(eleven,a,b,10,True, False)
```

```
Left Riemann Sum
100  Intervals from 2 to 3:  -0.0832293673094286
Left Riemann Sum
1000  Intervals from 2 to 3:  -0.08322936730942798
Left Riemann Sum
10000  Intervals from 2 to 3:  -0.08322936730943632
Left Riemann Sum
```

[17]: -0.10320774749944361

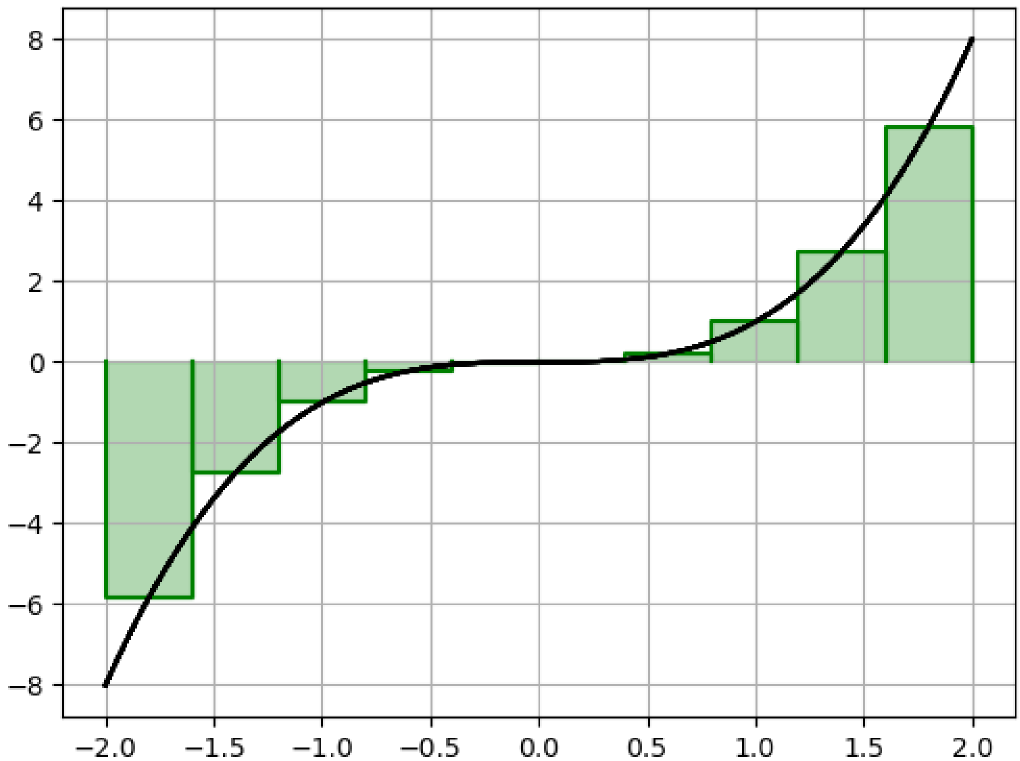


The sum is negative because all the values on that interval are negative. While negative area is impossible this would be the equivalent of driving in reverse on the "drive to grandma's house" example we use where you're moving closer to your starting point.

[18]:
```
#12 a) Calculate midpoint Riemann sums for the function
#H(z) = z3 on the interval [-2, 2],
#using 10, 100, and 1000 equally-spaced subintervals. The Riemann sums are
#all zero; why? (On some computers and calculators, you may find that there
#will be a nonzero digit in the fourteenth or fifteenth decimal place - this is
#due to "round-o  error".)

def H(z):
    return z**3

a = -2
b = 2
intervals = [10, 100, 1000]
for num in intervals:
    result = midpoint(H, a, b, num, False, False)
    print(f"{num} Intervals from {a} to {b}: {result:.9f}")

midpoint(H, a, b, 10, True, False)
```

```
Midpoint Riemann Sum
10 Intervals from -2 to 2: 0.000000000
Midpoint Riemann Sum
100 Intervals from -2 to 2: 0.000000000
Midpoint Riemann Sum
1000 Intervals from -2 to 2: 0.000000000
Midpoint Riemann Sum
```

[18]:
```
3.1086244689504383e-15
```

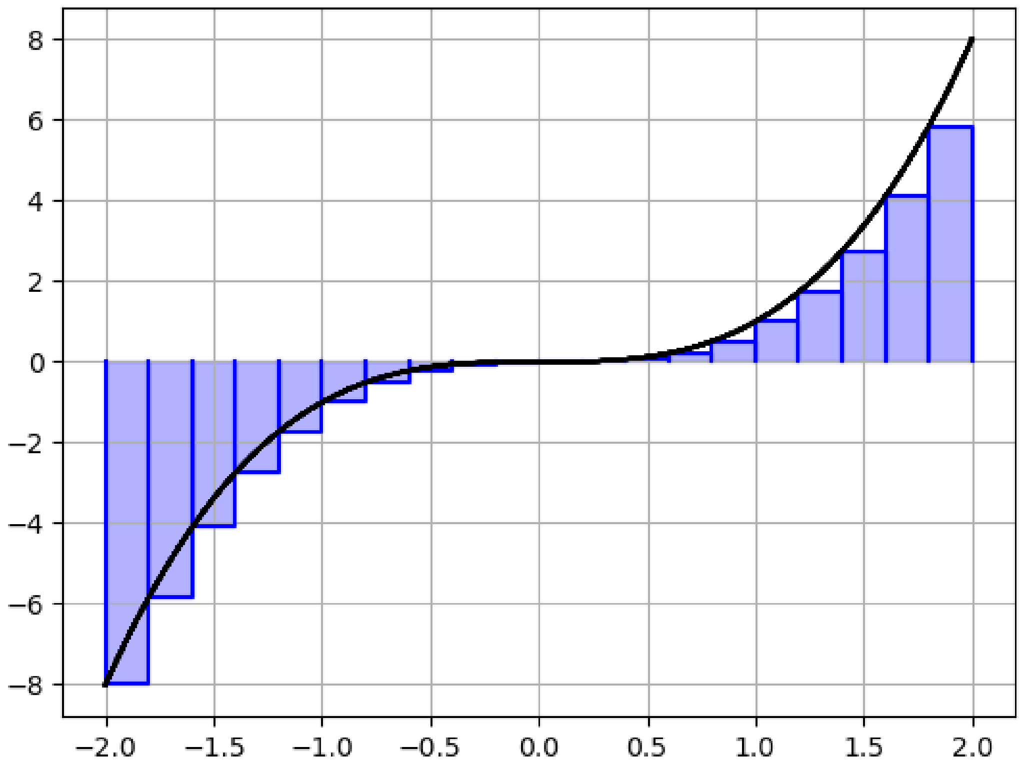


The sums are all zero because no matter how many points we use there is always a value below x=0 that corresponds with the value on the positive side as long as the interval is some combination of [-A, A]

[19]:
```python
#12 b) Repeat part (a) using left endpoint Riemann sums. Are the results still
#zero? Can you explain the di erence, if any, between these two results?

for num in intervals:
    result = left(H, a, b, num, False, False)
    print(f"{num} Intervals from {a} to {b}: {result:.9f}")

left(H, a, b, 20, True, False)
```

```
Left Riemann Sum
10 Intervals from -2 to 2: -32.000000000
Left Riemann Sum
100 Intervals from -2 to 2: -32.000000000
Left Riemann Sum
1000 Intervals from -2 to 2: -32.000000000
Left Riemann Sum
```

[19]: -1.600000000000002



As demonstrated in the graph above, the left endpoint doesn't always match up with above and below the x=0 point compared to the midpoint sum. In this case the lack of symmetry ends up with a negative total Riemann sum. If this was run again with right endpoint it would result in a positive sum when the true value is zero.

0.0.4 Activity 4

Modify RIEMANN again with either the Trapezoid rule or the Simpsons Rule. What are the advantages of these methods.

[20]:
```
def trapezoid(func, a, b, numberofsteps, plot, display):
    print("Trapezoid Riemann Sum")
    deltax = (b-a)/numberofsteps
```

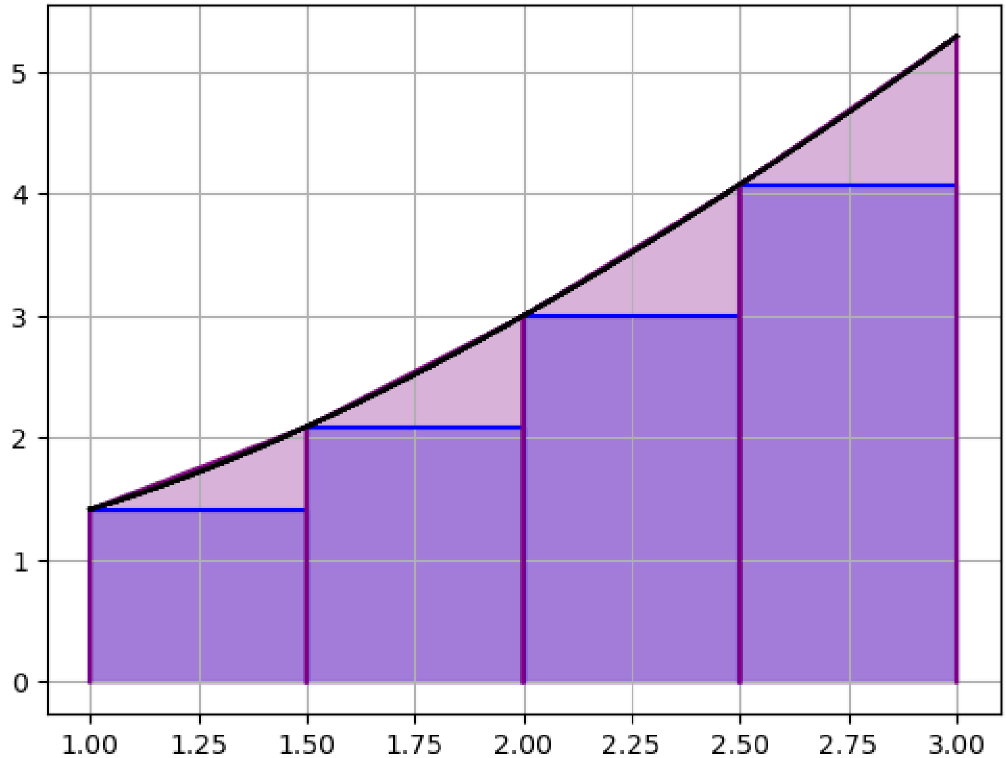
```
    x = a
    accumulation = 0
    for k in range (numberofsteps):

        # Calculate left side
        Ldelta = func(x)*deltax
        # Calculate right side
        Rdelta = func(x+deltax)*deltax

        #Average the two to determine trapezoid area
        deltaS = (Ldelta+Rdelta)/2
        accumulation = accumulation + deltaS
        if display:
            print(x, deltaS, accumulation)

        if plot:#this plots the box of the Rieman sum boxes if requested
            xplot = [x, x, x+deltax, x+deltax]
            yplot = [0, func(x), func(x+deltax), 0]
            plt.plot(xplot, yplot,color="purple")
            plt.fill_between([x, x + deltax], [func(x), func(x+deltax)], 
 ↪color="purple", alpha=0.3)
            xvals = np.linspace(a, b, 1000)
            plt.plot(xvals, func(xvals), color ="black" )
            plt.grid(True)
            plt.show
        x = x + deltax #set x to for the next interval
    return accumulation

def simpsons(func, a, b, numberofsteps, plot, display):
    print("Simpson's Riemann Sum")
    deltax = (b-a)/numberofsteps
    x = a
    accumulation = 0
    for k in range (numberofsteps):

        # Calculate left side
        Ldelta = func(x)*deltax
        # Calculate right side
        Rdelta = func(x+deltax)*deltax
        #Calculate midpoint
        Mdelta = func(x+(deltax/2))*deltax

        #Apply Simpson's Rule
        deltaS = (Ldelta+Rdelta+4*Mdelta)/6
        accumulation = accumulation + deltaS
        if display:
            print(x, deltaS, accumulation)
```

```
        if plot:#this plots the box of the Rieman sum boxes if requested
            xplot = [x, x, x+deltax, x+deltax]
            yplot = [0, func(x), func(x+deltax), 0]
            plt.plot(xplot, yplot,color="purple")
            xvals = np.linspace(a, b, 1000)
            plt.plot(xvals, func(xvals), color ="black" )
            plt.grid(True)
            plt.show
        x = x + deltax #set x to for the next interval
    return accumulation
```

[21]:
```
a = 1
b = 3
numberofsteps = 4
left(fnf, a, b, numberofsteps, True, True)
trapezoid(fnf, a, b, numberofsteps, True, True)
simpsons(fnf, a, b, numberofsteps, False, True) #plotting for Simpson's is not␣
 ↪accurate yet
```

```
Left Riemann Sum
1 0.7071067811865476 0.7071067811865476
1.5 1.0458250331675945 1.7529318143541421
2.0 1.5 3.252931814354142
2.5 2.038688303787511 5.291620118141653
Trapezoid Riemann Sum
1 0.8764659071770711 0.8764659071770711
1.5 1.2729125165837973 2.1493784237608686
2.0 1.7693441518937556 3.9187225756546242
2.5 2.342219807426051 6.260942383080675
Simpson's Riemann Sum
1 0.864977264261837 0.864977264261837
1.5 1.2648975472709332 2.12987481153277
2.0 1.7631253706586456 3.8930001821914155
2.5 2.3369773855572755 6.229977567748691
```

[21]: 6.229977567748691



By averaging the values of the left and right Rieman sum to get the trapezoid, we achieve a much more accurate value for the area. I also appreciate the simple math that allows for the calculation of the box and the triangle without having to decide if Left or Right makes the most sense for that calculation. Simpson's takes that even further by combining midpoint and trapezoidal to cancel out errors.

[ ]:

```
```