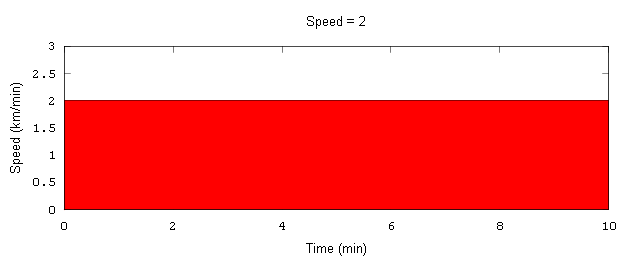
# 4 - Write programs for computing integrals of functions

In this learning outcome, we will cover part of the mathematical field called calculus: integration. But first, let’s start with a review of some simple calculations. See **“Integration Intro.docx”** for an exercise to start off with – the answers are given below.

If you are travelling at 2 km/min for 10 minutes, how far do you go?

*distance* = *rate* × *time* = 2 km/min \* 10 minutes = 20 km

We can represent this on a graph with a straight line function *f*(*x*) = 2 km/min over the time *x* from 0 to 10 minutes as follows – note that the area underneath the function from 0 to 10 is *area* = *height* \* *width* = 2 \* 10 = 20:



How far do you travel in minutes 7 through 10?

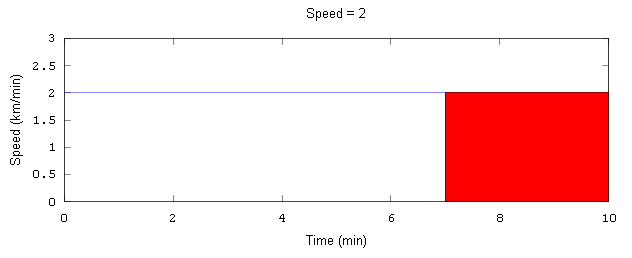
*distance* = *rate* × change in *time*

so *distance* = *rate* × (*time*2 - *time*1)

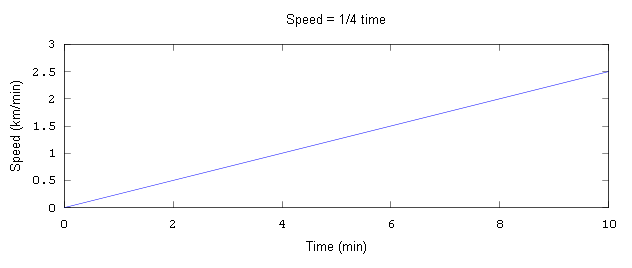
so *distance* = *rate* × ∆*time*, where ∆*time* is just a special notation for the change in time

so *distance* = 2 km/min \* (10 - 7) minutes = 2 \* 3 = 6 km

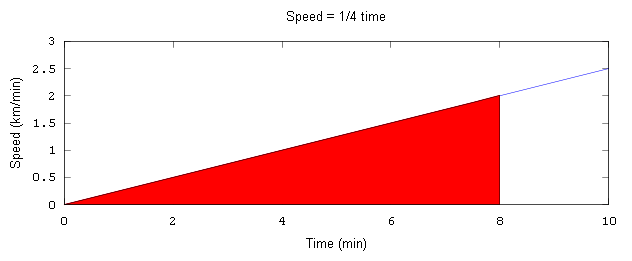
Graphically, this could be shown as follows – note that the area underneath the function from 7 to 10 is *area* = *height* \* *width* = 2 \* (10 - 7) = 2 \* 3 = 6:



This speed function isn’t particularly realistic – people don’t drive at a constant speed. Instead, you might start at a speed of 0 and increase their speed. For instance, your speed could be represented by the function *f*(*x*) = ¼ *x* (where *x* is the time). That could be shown by the following graph:



How far do you travel in the first 8 minutes? We can look at the area underneath the function between 0 and 8 minutes:

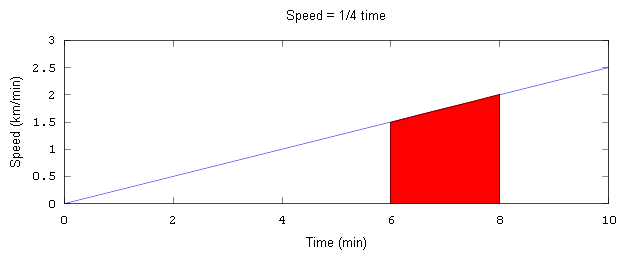


The triangle has a base of 8 and a height of 2 (since *f*(8) = ¼ \* 8 = 2).

*area* = ½ \* *base* \* *height* = ½ \* 8 \* 2 = 8

So we travel 8 km in the first 8 minutes. This can be verified by considering that the average speed is 1 km/minute, so 1 km/min \* 8 minutes = 8 km.

How far do you travel in minutes 6 through 8? Again, we can look at the area underneath the function between minutes 6 and 8, which in this case forms a trapezoid:



The trapezoid has a base with a width of 8 - 6 = 2

The height at the left edge, *h*1, is f(6) = ¼ \* 6 = 1.5

The height at the right edge, *h*2, is f(8) = ¼ \* 8 = 2

The area of a trapezoid of this type can be calculated by adding the area of the bottom rectangle to the area of the top triangle:

area of rectangle = base \* *h*1

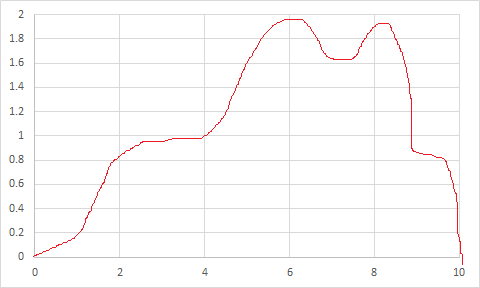
area of triangle = ½ \* base \* height = ½ \* base \* (*h*2 - *h*1) = ½ \* base \* *h*2 - ½ \* base \* *h*1

area of trapezoid = area of rectangle + area of triangle = base \* *h*1 + ½ \* base \* *h*2 - ½ \* base \* *h*1  
 = ½ base \* *h*1 + ½ \* base \* *h*2 = ½ base \* (*h*1 + *h*2)

area of example trapezoid = ½ base \* (*h*1 + *h*2) = ½ \* 2 \* (1.5 + 2) = 3.5

So we travel 3.5 km in minutes 6 through 8. This can again be verified by taking the average speed of 1.75 km/minute \* 2 minutes = 3.5 km.

But even this speed function isn’t particularly realistic. You might start at 0 km/h, then accelerate slowly onto the side road, then accelerate more onto the highway, then slow down at the speed trap, then speed up again, then decelerate onto another side road, and finally slow to a stop at 0 km/h. The speed function would look something like this:



How could we calculate the distance here? The distance is just the area under the speed function, so we could calculate that area. But this isn’t a simple shape like a rectangle or a trapezoid, so how can we calculate the area?

We can estimate the area under any curve by breaking it into pieces such as rectangles and then adding up the area of those pieces. The more pieces we use, the better our estimate will be, until we reach a desired level of precision. (If this sounds like an iterative method to you, you’re right!)

So we are sometimes interested in the area under the curve, or the *integral* of *f*(*x*). We can estimate the area using a bunch of rectangles and adding their areas:

Estimate of area = where ∆*x* is just the width of the rectangles

More, smaller rectangles give us a better result.

An “infinite” number of infinitely small rectangles will give us an exact answer. This is called the integral of the function. The notation for that is:

Area =

Notice how the Σ just becomes the integral symbol (elongated s) and ∆*x* becomes *dx* (over small changes in *x*).

As an example, for our first speed function, we could have:

This kind of integral is called an *indefinite integral*, because it just describes the interval without calculating any exact values.

The mathematical approach is to find a function *F*(*x*) that gives us the area under *f*(*x*). For instance, thanks to the magic of calculus:

*F*(*x*) = for some arbitrary constant *C* which is usually ignored, so

Indefinite integrals can be applied to specific areas by plugging the values for the boundaries of *x* into the formula for *F*(*x*). This produces a *definite integral* with a numerical answer such as:

Or

Note that the answers above match the answers we calculated for the speed function *f*(*x*) = ¼ *x* earlier.

As another example, consider the function , as shown in **“Sample function for integration.xlsx”**. We could look for the answer to or some similar integral.

See **“IntegrationBackground MATH282-CST.docx”** for a further introduction to the topic of integration from a more theoretical standpoint and a discussion of the integral of the sample function given above.

The computational approach to finding the answers to definite integrals is to break the integral up into parts and add up the area of the parts. The first way to try this is the left rectangle rule.

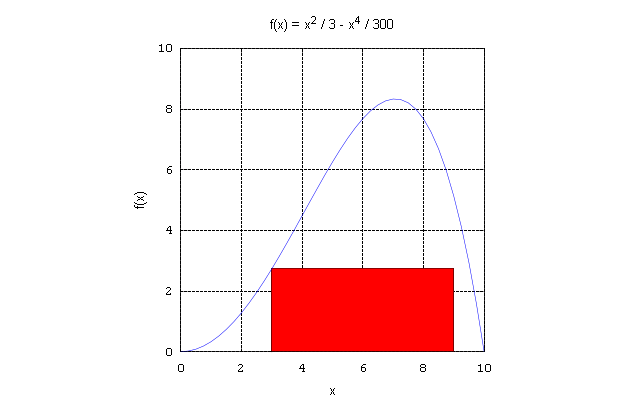
## Left Rectangle Rule

The left rectangle rule says that we should do the following:

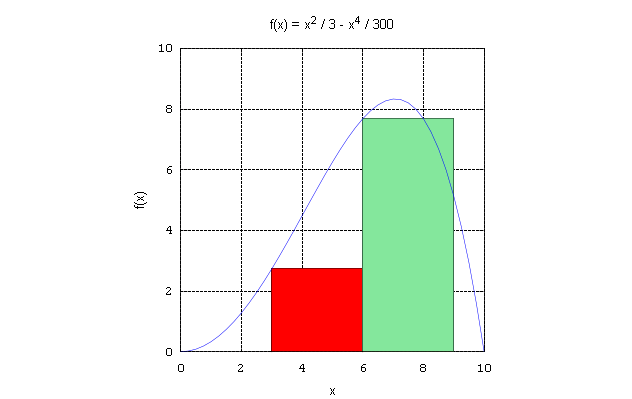
1. Take the interval being integrated (such as 3 to 9) and divide it into equal parts
   * so 1 part would go from 3 to 9 for a width of 6; 2 parts would go from 3 to 6 and 6 to 9 for a width of 3; 4 parts would go from 3 to 4.5, 4.5 to 6, 6 to 7.5, and 7.5 to 9 for a width of 1.5; etc.
2. For each subinterval, create a rectangle with the height based on the value of *f*(*x*) at the left edge of the subinterval
   * so for 1 part, the height would be *f*(3); for 2 parts, the heights would be *f*(3) and *f*(6); for 4 parts, the heights would be *f*(3), *f*(4.5), *f*(6), and *f*(7.5); etc.
3. Add up the areas of all of the rectangles created in step #2 to get an estimated answer
4. As long as your estimate isn’t close enough, use more (twice as many) rectangles and go through the process again.

See **“Sample function for integration.xlsx”** for some sample calculations. Here is what the graphs would look like:

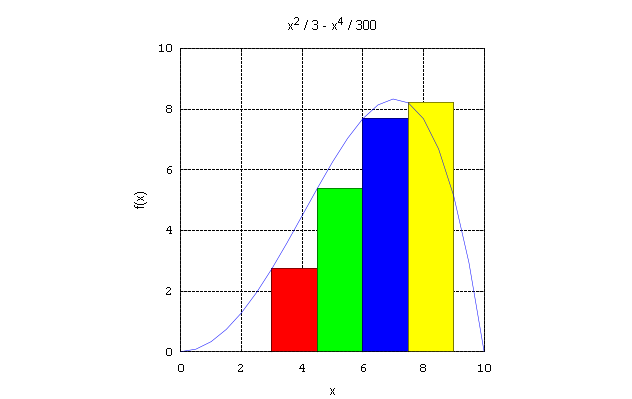
One rectangle:



Two rectangles:



Four rectangles:



To ensure that you understand how the left rectangle rule works, try looking at how the *right* rectangle works. The right rectangle rule is just like the left rectangle rule, but the height of the rectangle is calculated by taking the value of the function at the *right* edge of each interval. For practice, try **“RectRuleExercise MATH282-CST.docx”** and compare your results to the completed exercise in **“RectRuleExercise (answered) MATH282-CST.docx”**.

The detailed algorithm for the left rectangle rule is as follows:

We want to calculate using rectangles, with the height of the rectangle given by the left edge, to some desired precision.

**Straightforward (simple but somewhat inefficient) version of the Left Rectangle Rule**

Given *left* (or *a* in the notation above), *right* (or *b* in the notation above), *f*(*x*), *precision, maxLoops*

// **Start with an initial approximation** - in this case, 1 rectangle

*numRectangles* 🡨 1

*totalWidth* 🡨 *right* - *left*

*height* 🡨 *f*(*left*)

*estimate* 🡨 *height* \* *totalWidth*

// setup for the loop:

*numLoops* 🡨 0

*keepGoing* 🡨 **true**

// **While we are not "close enough"**

While (*keepGoing*)

Add 1 to *numLoops*

*oldEstimate* 🡨 *estimate*

// **Find a better approximation** - use twice as many rectangles

Multiply *numRectangles* by 2

*currentWidth* 🡨 *totalWidth* / *numRectangles*

*estimate* 🡨 0

for each rectangle *i* starting at *i* = 0

// calculate area of current rectangle and add to total

*currentLeft* 🡨 *left* + *i* \* *currentWidth*

*currentHeight* 🡨 *f*(*currentLeft*)

*currentArea* 🡨 *currentHeight* \* *currentWidth*

Add *currentArea* to *estimate*

// **Check if we are "close enough"**

*error* 🡨 absolute value of *estimate* - *oldEstimate*

*relError* 🡨 absolute value of *error* / *estimate*

if *relError* ≤ *precision*

*keepGoing* 🡨 **false**

else if *numLoops* ≥ *maxLoops*

indicate error

*keepGoing* 🡨 **false**

return *estimate*

Let’s look at implementing the left rectangle rule in our function classes in Java. Here are some implementation issues that you should keep in mind:

* Make sure that all of your files (IFunction, ACFunction, TestFunction, and specific function classes) are in the same project!
* Make sure that your loop is doing the correct number of rectangles.
  + Since we start at *i* = 0, the correct code should be something like   
    for (i = 0; i < numRectangles; i++)  
    Make sure that you have the right initialization and condition!
* Make sure that the order of operations is correct in calculating error.
* If implementing as a method in the IFunction / ACFunction classes:
  + Don't forget to put the method definition in your interface (IFunction)!
  + If the integration rule is a method in ACFunction, you don't need to pass the function in as a parameter – it is the invoking object.
    - To calculate *f*(*x*), we will call this.calculate(x) – for instance, *f*(*left*) would be something like this.calculate(left), depending on your variable names.
* If implementing as a class of its own (such as Integral):
  + Consider what attributes you need and what values you want to pass into the constructor or constructors
  + Consider what results you want to use – might include:
    - Estimate
    - Boolean indication of convergence
    - Number of loops (or number of rectangles) used

The added code in IFunction should look something like:

**public** **double** leftRectangleRule( **double** left, **double** right, **double** precision, **int** maxLoops );

The added code in ACFunction should look something like:

**public** **double** leftRectangleRule( **double** left, **double** right, **double** precision, **int** maxLoops )

{

**long** numRectangles = 1;

**double** totalWidth = right - left;

**double** height = **this**.calculate( left );

**double** estimate = height \* totalWidth; // initial estimate with one rectangle

**int** numLoops = 0;

**boolean** keepGoing = **true**;

**while** ( keepGoing )

{

numLoops++;

**double** oldEstimate = estimate;

// calculate area for twice as many rectangles

numRectangles \*= 2;

**double** currentWidth = totalWidth / numRectangles;

estimate = 0;

**for** ( **long** i = 0; i < numRectangles; i++ )

{

**double** currentLeft = left + i \* currentWidth;

**double** currentHeight = **this**.calculate( currentLeft );

**double** currentArea = currentHeight \* currentWidth;

estimate += currentArea;

}

// check if we are close enough

**double** error = Math.*abs*( estimate - oldEstimate );

**double** relError = Math.*abs*( error / estimate );

**if** ( relError <= precision )

{

keepGoing = **false**;

}

**else** **if** ( numLoops >= maxLoops)

{

System.***out***.println( "Did not converge with loops " + numLoops );

keepGoing = **false**;

}

}

**return** estimate;

}

The added code in TestFunction should look something like the following for a function class FunctionA containing the calculate method for :

IFunction fA = **new** FunctionA();

System.***out***.println( "\nTable of function A, f(x) = 1/3 x^2 - 1/300 x^4" );

fA.printTable( 0.0, 10.0, 1.0 );

System.***out***.println( "Integral of function A from 3 to 9: "

+ fA.leftRectangleRule( 3.0, 9.0, 0.000001, 20) );

System.***out***.println( "Integral of function A from 0 to 10: "

+ fA.leftRectangleRule(0.0, 10.0, 0.0000000001, 35) );

Try implementing this. Then start the exercises given in **“Integration Exercises MATH282-CST.docx”** (don’t worry about the trapezoid rule and Simpson’s rule yet). Ensure that your answers are the same (within the desired precision) to the correct answers. Add code so that each time that you go through the loop, you print out the loop number, the number of rectangles used in that iteration, and the estimate after that iteration. Compare your results to the results in **“Sample output - integration rules.txt”** (again, don’t worry about the trapezoid rule yet).

Since we are calculating thousands or millions or billions of rectangles, we would like to make the calculations of the rectangle rule as efficient as possible. Here are some ways that you can do so. Note that your answers and number of loops should be the same (with slight differences based on representation/propagation/round-off error), just calculated faster!

* Take common steps (like multiplication of the width) out of the loop (may decrease clarity, but increases efficiency):

for each rectangle *i* starting at *i* = 0

*currentLeft* 🡨 *left* + *i* \* *currentWidth*

*currentHeight* 🡨 *f*(*currentLeft*)

*currentArea* 🡨 *currentHeight* ~~\*~~ *~~currentWidth~~*

Add *currentArea* to *estimate*

Multiply *estimate* by *currentWidth*

* Combine steps in calculations in loop:

for each rectangle *i* starting at *i* = 0

Add *f*(*left* + *i* \* *currentWidth*) to *estimate*

Multiply *estimate* by *currentWidth*

* Store previously calculated heights so we don't have to call the function to calculate them again – calculate heights only for the odd-numbered rectangles (new heights)
  + Could use previous estimate as a starting point – must multiply by 0.5 to account for the smaller width; OR
  + Keep track of all heights calculated separately and then multiply by the width:

// in the initial estimate

*heights* 🡨 *f*(*left*)

*estimate* 🡨 *heights* \* *totalWidth*

...

// *estimate* 🡨 0 - remove this line, since we don't need to reset any more

...

// Change the for loop so it only looks at the new odd-numbered rectangles

For every new rectangle starting at *i* = 1 up to *numRectangles* by 2

Add *f*(*left* + *i* \* *currentWidth*) to *heights*

*estimate* 🡨 *heights* \* *currentWidth*

These efficiencies are also why we double the number of rectangles each time we go through the loop rather than multiplying by 5 or 10 or 20 – first, we calculate fewer rectangles to get to our desired precision, and second, we are never repeating any calculations for the rectangles.

The efficient version of the left rectangle rule algorithm, along with the notes about the left rectangle rule algorithm, can be found in **“Left Rectangle Rule Algorithm.docx”**.

Try to write the code for the efficient left rectangle rule, testing your results against the results that you got for the simple left rectangle rule. Again, you should find that the results are (nearly) the same, but if you calculate the time taken for the process, it will be cut roughly in half.

The code for the efficient left rectangle rule is as follows:

**public** **double** efficientLeftRectangle( **double** left, **double** right, **double** precision, **int** maxLoops )

{

**long** numRectangles = 1;

**double** totalWidth = right - left;

**double** heights = **this**.calculate( left );

**double** estimate = heights \* totalWidth; // initial estimate with 1 rectangle

**int** numLoops = 0;

**boolean** keepGoing = **true**;

**while** ( keepGoing )

{

numLoops++;

**double** oldEstimate = estimate;

// calculate area for twice as many rectangles

numRectangles \*= 2;

**double** currentWidth = totalWidth / numRectangles;

**for** ( **long** i = 1; i < numRectangles; i += 2 )

{

// Add up the heights for all new rectangles

heights += **this**.calculate( left + i \* currentWidth );

}

// Multiply all rectangles by the same width

estimate = heights \* currentWidth;

System.***out***.println( numLoops + "\t" + numRectangles + "\t" + currentWidth + "\t" + estimate );

// check if we are close enough

**if** ( Math.*abs*( (estimate - oldEstimate) / estimate ) <= precision )

{

keepGoing = **false**;

}

**else** **if** ( numLoops >= maxLoops)

{

System.***out***.println( "Did not converge with loops " + numLoops );

keepGoing = **false**;

}

}

**return** estimate;

}

But you may have noticed that the left rectangle rule uses rectangles that don’t always do a good job of matching the shape of the integral, particularly for intervals that are increasing or decreasing throughout the interval. This leads to our next improvement – use trapezoids instead of rectangles to calculate the width of each of the small intervals.

## Trapezoid Rule

The trapezoid rule is exactly the same as the left rectangle rule, except that you replace “rectangle” with “trapezoid” in the algorithm. Recall that the area of a trapezoid is given by *area* = ½ (*hleft* + *hright*) \* *width*.

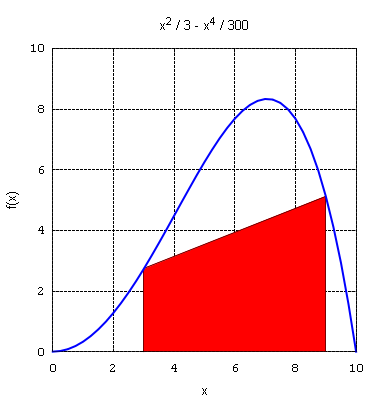
So to calculate the area of one trapezoid within the algorithm/code, we will need to figure out the value of *x* at right edge of each trapezoid and calculate the height *f*(*x*) for that value as follows:

*area* = ½ (*f*(*left*) + *f*(*right*)) \* *width*

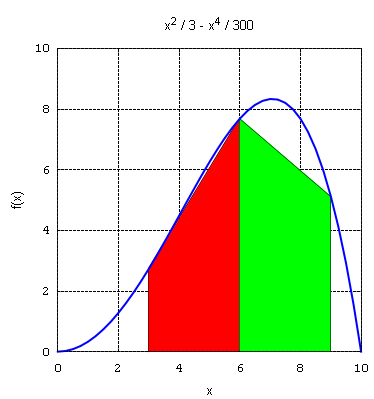
This is done both when we calculate the first estimate (one trapezoid) and when we calculate every subsequent estimate (double the number of trapezoids). Otherwise, our algorithm remains the same.

See the **Trapezoid Rule** sheet in **“Sample function for integration.xlsx”** for sample calculations. Here is what the graphs would look like:

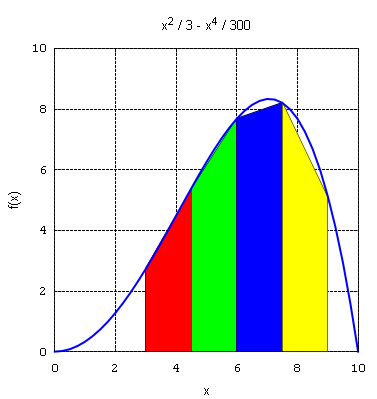
One trapezoid:



Two trapezoids:



Four trapezoids:



See **“Trapezoid Algorithm MATH282-CST.docx”** for the updated algorithm. Coding the function is left as an exercise for you. One note: when calculating the current right edge of the current trapezoid, you do *not* need to start at the initial right edge of the entire interval. Instead, you just need to add one width to the left edge of the current interval.

The trapezoid rule can also be made more efficient in similar ways to the rectangle rule. You can take the multiplication by ½ *currentWidth* outside of the loop to calculate twice as many trapezoids, since each trapezoid is multiplied by ½ and the *currentWidth*. You can also keep track of the previously used function heights – in this case, the heights at the left edge and right edge of the entire interval are only used once, but the heights of the new inner trapezoids are used twice (since the right edge of one trapezoid becomes the left edge of the next trapezoid). Again, you only need to add the new heights if you keep track of the heights previously used in the calculation. See **“Trapezoid Algorithm MATH282-CST.docx”** for a discussion of these efficiencies.

## Simpson’s Rule

Simpson’s rule is the same as the rectangle/trapezoid rules, except that you replace “rectangle” (or “trapezoid”) with “area under the curve” where “curve” is the 2nd-degree polynomial matching the function values at the left edge, right edge, and midpoint of each interval.

Try out the exercise in **“Shapes for integrals MATH282-CST.docx”** to see if you understand which points are used for each rule. Compare your results to **“Shapes for integrals (answered) MATH282-CST.docx”**.

The idea of finding the 2nd-degree polynomial can be shown in Excel. First, we find the three points at the left, right, and middle. Then we calculate a 2nd-degree polynomial that fits those point using the least-squares method (which we will study later in the course) – use Excel trendlines to find the curve and display the equation of the curve. Then we need to find the area underneath the 2nd-degree polynomial curve.

That may seem like a lot of extra work, but fortunately, someone (presumably named Simpson!) did all of the necessary calculations, and found that the area under such a 2nd-degree polynomial curve is just:

While we won’t derive this formula, note how similar this is to the calculation of the area of the trapezoids. It just uses 1/6 instead of 1/2, and adds 4 \* *f*(*middle*) to the calculation. In a way, this makes sense. The middle is “closer” to most of the curve, so it has more weight or emphasis. The 1/6 just takes an average of the six heights used (one left, 4 middle, and one right). Why add exactly 4 times the middle height? Well, that’s where the derivation of the formula tells us the right answer.

A sample of the polynomial calculated to match three points is given in **“Simpson's Rule example polynomial.xlsx”**. A further example of the calculations for both one and two subintervals for can be found in **“Simpson's rule example.xlsx”**.

A recap of the theory behind Simpson’s rule and some sample calculations are as follows:

* Take three points on an interval (xleft, xright, and xmid, where xmid = (xleft+xright)/2)
* Use these three points to find a second degree polynomial to fit these points. (We will learn how to do this as part of learning outcome 3.)
* Integrate the area under the polynomial curve to estimate the area of the original function. (Likely easier to determine the antiderivative of the polynomial than the original function.)

Fortunately, we don’t have to do all these steps. The area under the curve defined by the three points (the value of the function at xleft, xright, and xmid) can be found using the following formula:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| One parabola: | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | # parts | Int size | xl | f(xl) | xr | f(xr) | xm | f(xm) | Est  para area | | 1 | 6 | 3 | 2.73 | 9 | 5.13 | 6 | 7.68 | 38.58 |   Est Total: 38.58 |
| Two parabolas: | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | # parts | Int size | xl | f(xl) | xr | f(xr) | xm | f(xm) | Est  para area | | 2 | 3 | 3 | 2.73 | 6 | 7.68 | 4.5 | 5.38 | 15.97 | |  |  | 6 | 7.68 | 9 | 5.13 | 7.5 | 8.20 | 22.81 |   Est Total: 38.78 |
| Four parabolas: | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | # parts | Int size | xl | f(xl) | xr | f(xr) | xm | f(xm) | Est  para area | | 4 | 1.5 | 3 | 2.73 | 4.5 | 5.38 | 3.75 | 4.02 | 6.06 | |  |  | 4.5 | 5.38 | 6 | 7.68 | 5.25 | 6.66 | 9.92 | |  |  | 6 | 7.68 | 7.5 | 8.20 | 6.75 | 8.27 | 12.24 | |  |  | 7.5 | 8.2 | 9 | 5.13 | 8.25 | 7.25 | 10.58 |   Est Total: 38.80 |

The implementation of Simpson’s rule, as well as the efficiencies that can be used when calculating Simpson’s rule, are left as an exercise for you. Make sure that you try all of the examples in **“Integration Exercises MATH282-CST.docx”** with both the trapezoid rule and Simpson’s rule to see that you are getting the same results, only with fewer loops. Examine the results to see when the trapezoid rule and Simpson’s rule are getting exact results – can you explain those cases?

See further notes and examples in the **oldNotes** folder.