Mathematics HL IA

Comparing different ways of numerical integration

Introduction

Usually in a school course, when we asked to find a definite integral, we have to solve it by hand and then evaluate it at upper and lower limit. Of course, there are many integrals which you can solve, but there are infinitely many of the ones, which you can't easily solve by hand. That's where people use numerical integration by computers or calculator.

During the introduction into Riemann sums our class had an assignment, where we calculated integrals of different functions using small and large amounts of rectangles under the graph and compare them. But to reach high accuracy using left and right Riemann sums you need to have a large amount of rectangles, even for a computer it micht be time consuming to process large amounts of data. That's why for example TI-84 calculator uses *Gauss-Kronrod* method of numerical integration[1].

In this internal assessment I was interested in finding an optimal method of numerical integration. I will create small scripts for each method on programming language Python 3 and then test them individually to find the method that best fits the criteria presented in the next section. During the test I will run each script 100 times, then calculate mean time to run it.

Formulation of the problem

In order for a method to be optimal it must meet several requirements:

Versatility - method should be able to find integrals of all, or at least of majority of the functions.

Efficiency - method should be as fast as possible.

Accuracy - systematic errors that come up using this method should be as low as possible.

So in order for a method to be effective - it should reach the highest accuracy with as less amount of steps as possible.

But first we need to define a continuous function $f(x)$, that is defined on $[a; b]$, integral of which we will find using different methods.

I picked $f(x) = 5x^4 - 36x^3 + 93x^2 - 102x + 42$ as a function that I will use, with upper and lower boundaries $a = 1$ and $b = 3$.

To understand why I picked a function like this, let's take a look at the plot of $f(x)$ within the domain - figure[1] 1.

First of all it is not monotonous, gradient of it changes greatly, therefore inaccurate methods would be easier to distinguish. Integral of it, evaluated by hand (equation 1), is a rational number - by comparing results, produced by other methods with it, particular error of each method can found easier, than if integral was an irrational number or a number with many decimals.

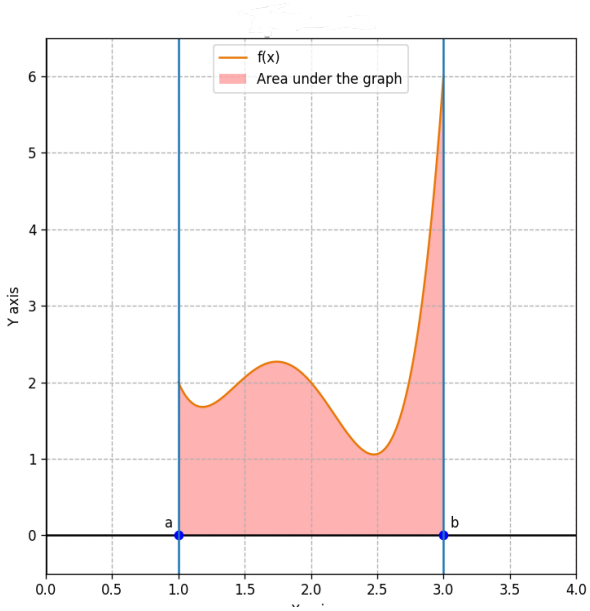

Figure 1: Plot of $f(x)$ defined on $[a, b]$

$\int_a^b f(x)\mathrm{d}x$ Can be solved analytically:

$$\int_a^b f(x)\mathrm{d}x = \int_1^3 5x^4 - 36x^3 + 93x^2 - 102x + 42\,\mathrm{d}x = \\ = \left[x^5 - 9x^4 + 31x^3 - 51x^2 + 42x\right]_1^3 = 18 - 14 = 4 \tag{1}$$

After introducing each method in this exploration I want to test them using a computer. I will write functions, on programming language Python 3, that numerically estimate integrals. I will be using `timeit` library (which is basically an instrument that measures time to execute a piece of code) to measure the time it takes to run this function, and then repeat this for 100 times using a loop and find average time to run this code, and the uncertainty of this value, using the formulae below. To ensure the consistency of the results and to make sure that they all were taken in the same conditions, all code runs are going to be done in one day and one computer.

$$\text{Average time: } t_{\text{avg}} = \frac{t_1 + t_2 + t_3 + \ldots + t_n}{n} = \frac{t_1 + t_2 + t_3 + \ldots + t_{100}}{100}$$

$$\text{Uncertainty of this average time: } \Delta t = \frac{t_{max} - t_{min}}{2 \times \sqrt{n}} = \frac{t_{max} - t_{min}}{20}$$

[1]All figures in this exploration, unless specified otherwise, were created by me using `matplotlib` graphing library for Python 3.

Riemann Sums

Riemann sums is a method of numerical integration, that is based on finding area under the graph by fitting geometrical figures under the graph and finding the areas of them. This is a simple and relatively fast method, in many cases approximation can even be easily done by hand.

If $x \in [a, b]$ and $x_i + \Delta x = x_{i+1}$, where $\Delta x = \dfrac{b-a}{n}$ and n is the number of figures you want to add up. Then the integral of $f(x)$, using these methods can be approximated using these formulas:

Left sum	Right sum	Trapezoids
$\sum_{i=0}^{n-1} f(x_i)(x_{i+1} - x_i)$	$\sum_{i=1}^{n} f(x_i)(x_i - x_{i-1})$	$\sum_{i=0}^{n-1} \dfrac{f(x_i) + f(x_{i+1})}{2} \Delta x$

By increasing number n, higher accuracy can be achieved.

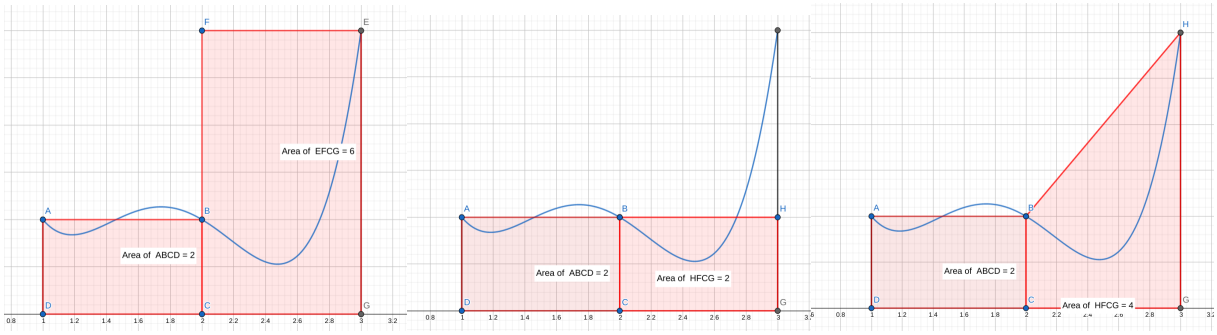

Figure 2: Comparison of areas produced using different methods, $\Delta x = 1$ (created in GeoGebra)

The formula for the approximation using trapezoids comes from the equation of the area of a trapezoid - half of the sum of bases times the height. In case with integrals, length of the bases will be values of $f(x)$ and $f(x_{i+1})$ and height is the step size $\Delta x = x_{i+1} - x_i$

The midpoint method is another way of approximation of integrals. It is similar to both of the rectangle and trapezoid method. However, instead of picking value with step size of Δx, a middle point in between of x_i and x_{i+1} is chosen. It can be represented by the following formula:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} f\left(\frac{x_i + x_{i+1}}{2}\right)(x_{i+1} - x_i) \tag{2}$$

I am interested in testing every method, to find how fast would it take for them to reach certain accuracy – 10%, 5%, 1% etc. (time is shown in microseconds). I will also record the lowest amount of figures n it would take for them to do so. I will leave all scripts that I have written for method testing in appendix, with some small comments alongside.

Relative error, %	Left rectangle, n	Time μs	Right rectangle, n	Time μs	Trapezoid method, n	Time μs	Midpoint method, n	Time μs
10	13	23.9 ± 10.0	4	23.5 ± 17.8	6	59.5 ± 14.4	5	25.8 ± 6.4
5	24	24.8 ± 7.5	19	25.8 ± 9.0	8	63.6 ± 14.7	6	29.2 ± 13.0
1	104	40.3 ± 3.7	99	37.9 ± 3.6	17	130.2 ± 22.2	12	42.8 ± 9.7
0.1	1004	187.1± 1.4	999	187.9 ± 1.3	50	352.7 ± 25.3	36	105.7 ± 7.8
0.01	10004	1481.0 ± 1.3	9999	1497.0 ±2.2	154	1102.0 ± 58.1	110	299.5 ± 7.9
0.001	-	-	-	-	487	3394.0 ± 67.1	343	888.9 ± 9.9

Table 1: Relative error of different methods

From table 1 it can be seen that left and right Riemann sums methods were slower than the midpoint method, and in fact they never could reach the accuracy of 0.001%, which makes them the least preferred methods for numerical integration on electronic devices. Although while testing I've noticed, that these methods produced exact results for first n (1,2,3) - this is is just a coincidence, and shall not be considered as reliable result.

Trapezoid method is rather controversial - it takes the most time to reach certain accuracy, but it does that in fewer steps than the left and right rectangle methods. Therefore I would leave it as a method that could be used for hand calculation, but less favourable than the midpoint method. It is not preferable for the approximation on a computer.

And as seen from table 1, method, that took the least time to reach the accuracy of 0.001% was the midpoint method, it also required the least amount of figures to reach this accuracy. Which makes this method the most suitable method for the numerical integration out the ones that were tested so far.

Simpson's method

Simpson's method or otherwise called parabola method can be defined as another case of Riemann sums, since it involves approximating integral by fitting parabolas under the graph of $f(x)$, however method is more complicated then the ones that were mentioned in the previous section, so I decided to explain it separately.

To start off with deriving the formula for this method let's look at a simple case, where the function will be fitted with one parabola.

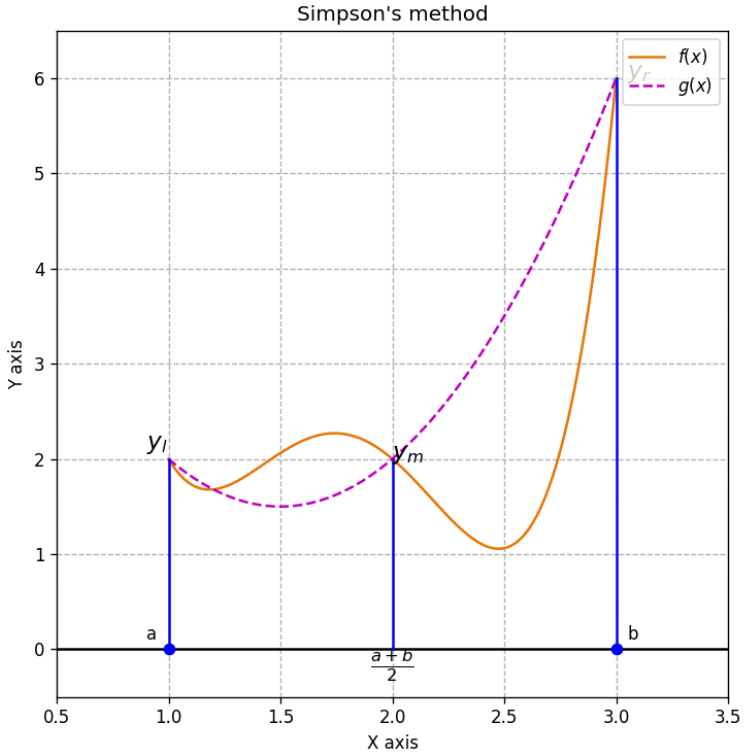

Figure 3: Visualisation of Simpson's method

Let $g(x)$ be parabola that is of the form $g(x) = Ax^2 + Bx + C$. The function can be integrated to find area under the curve:

$$\begin{aligned}
&\int_a^b Ax^2 + Bx + C \,\mathrm{d}x = \left[\frac{Ax^3}{3} + \frac{Bx^2}{2} + Cx\right]_a^b = \\
&= \frac{A\,(b-a)^3}{3} + \frac{B\,(b-a)^2}{2} + C\,(b-a) = \\
&= \frac{b-a}{6}(2A(b-a)^2 + 3B(b-a) + 6C) \qquad (3)
\end{aligned}$$

From Fig. 3: $y_l = f(a) = Aa^2 + Ba + C$, $y_r = f(b) = Ab^2 + Bb + C$, $y_m = f\left(\frac{a+b}{2}\right) = A\left(\frac{a+b}{2}\right)^2 + B\left(\frac{a+b}{2}\right) + C$

Knowing function values in a, b and $\frac{a+b}{2}$ allows for a system of simultaneous equations to be solved in order to find A, B and C:

$$\begin{cases} Aa^2 + Ba + C = f(a) \\ Ab^2 + Bb + C = f(b) \\ A(\frac{a+b}{2})^2 + B(\frac{a+b}{2}) + C = f(\frac{a+b}{2}) \end{cases} \Rightarrow \begin{cases} A + B + C = 2 \\ 9A + 3B + C = 6 \\ 4A + 2B + C = 2 \end{cases}$$

$$A = 2,\ B = -6,\ C = 6. \tag{4}$$

Three other systems of simultaneous equation has to be solved to obtain the final formula for the Simpsons method, that has a look of:

$$\int_a^b Ax^2 + Bx + C\,\mathrm{d}x = \frac{b-a}{6}(y_l + 4y_m + y_r) \tag{5}$$

However, approximation by fitting one parabola is going to be very inaccurate. To increase accuracy of approximation of integral $\int_a^b f(x)\mathrm{d}x$, $[a, b]$ needs too be divided into more pairs of segments and apply equation (5) to each of these pairs.

Final formula for the parabola method will have the following look:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} \int_{x_{2i-2}}^{x_{2i}} f(A_i x^2 + B_i x + C_i) = \\ = \frac{h}{3}\left(f(a) + 4\sum_{i=1}^{n} f(x_{2i-1}) + 2\sum_{i=1}^{n-1} f(x_{2i}) + f(b)\right) \tag{6}$$

Where $h = \dfrac{b-a}{2n}$ and n is the amount of segment pairs, x_{2i} are the even nodes and x_{2i-1} are the odd nodes.

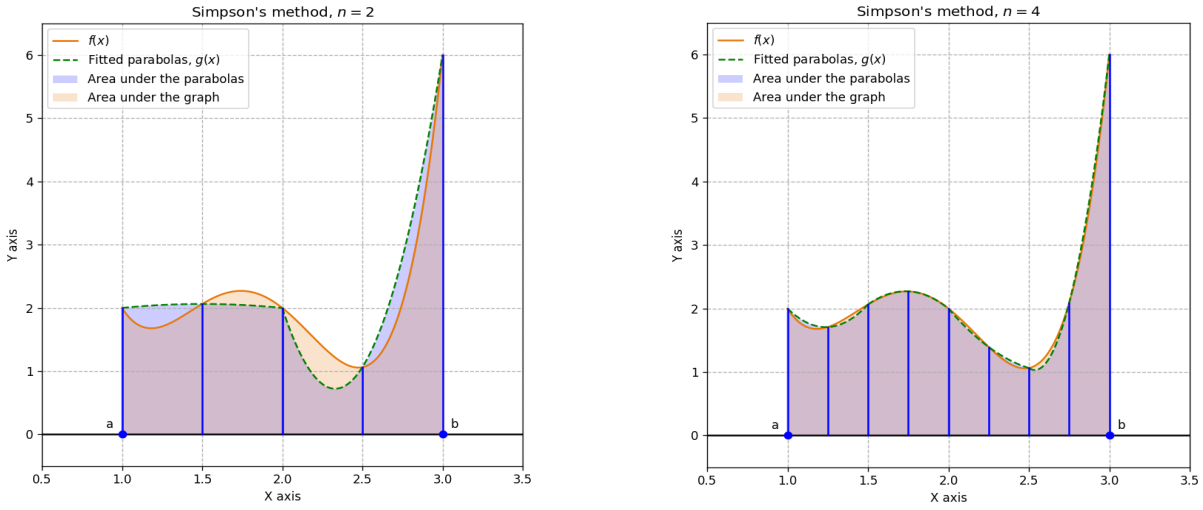

Figure 4: Use of parabola method by fitting $f(x)$ with 2 and 4 parabolas

To give a closer look I also fitted $f(x)$ with more parabolas - figure 4. It can be seen how close the fit is even by using 4 parabolas. This is significantly better than the fit that was produces by using Riemann sums.

I am going to use equation (6) to numerically evaluate integral of my function $f(x)$ on $[1, 3]$ for four pairs of segments:

$$a = 1;\ b = 3;\ n = 4;\ f(x) = 5x^4 - 36x^3 + 93x^2 - 102x + 42$$

First we need to find step size h:

$$h = \frac{b-a}{2n} = \frac{3-1}{2 \times 4} = 0.25$$

Then, it is required to calculate all the nodes and function values of them. Since I have 4 pairs - then I will need to find values of 8 nodes.

$$\begin{aligned}
i = 0:\ x_i = x_0 = a + ih = 1 + 0 \times 0.25 = 1;\ &\Rightarrow f(x_0) = f(1) = 2 \\
i = 1:\ x_i = x_1 = a + ih = 1 + 1 \times 0.25 = 1.25;\ &\Rightarrow f(x_1) = f(1.25) \approx 1.70703 \\
&\dots \\
i = 8:\ x_i = x_8 = a + ih = 1 + 8 \times 0.25 = 3;\ &\Rightarrow f(x_8) = f(3) = 6
\end{aligned}$$

It is easier to organise everything in a table:

Node number, i	0	1	2	3	4	5	6	7	8
x_i	1	1.25	1.5	1.75	2	2.25	2.5	2.75	3
$f(x_i)$	2	1.70703	2.0625	2.26953	2	1.39453	1.0625	2.08203	6

Table 2: Calculated values

Plugging all values into the Eq. (6) result in:

$$\begin{aligned}
\int_1^3 f(x)\mathrm{d}x &\approx \frac{h}{3}(f(x_0) + 4\sum_{i=1}^{n} f(x_{2i-1}) + 2\sum_{i=1}^{n-1} f(x_{2i}) + f(x_{2n})) = \\
&= \frac{0.25}{3}(2 + 4(1.70703 + 2.26953 + 1.39453 + 2.08203) + 2(2.0625 + 2 + 1.0625) + 6) = \\
&= 4.00521 \\
\text{error} &= 0.13\%
\end{aligned}$$

I have also tested this method and measured how long it takes for it to reached certain accuracy as well as amount of segment pairs it requires for it. Test results are presented in the table below (read table vertically):

Error, %	10	5	1	0.1	0.01	0.001
n	3	4	6	10	16	28
Time, μs	5.8± 0.2	6.5± 0.2	8.5± 0.2	11.9± 0.3	17.0± 0.6	37.7± 1.7

Table 3: Simpson's method test results

Compared to results shown in table 1 the test results of this method are significantly better than Riemann sums method. The conclusion for this method is that this method is much better for use in electronic devices, requiring less steps and time to reach higher accuracy than trapezoid, midpoint and rectangle methods. Even though hand calculations might take some time, parabola method is a better option to reach desired results than other aforementioned methods.

Gauss method

All previously motioned methods used an uniformly spaced grid. This sufficiently eases the calculations, but at the same time creates a lot of uncertainty, that quickly stacks up. To counter that, a less spaced grid can be used, but this increases the time to solve an integral (this can be seen in both Tab. 1 and Tab. 2). Gauss method can deal with these issues. It does not need to have a high degree to accurately approximate an integral.

Basically this method is an approximation of the definite integral of a function $f(x)$, that uses a weighted sum[1] of function values at specified points (nodes) within the domain of integration (usually $[-1, 1]$). Both weights and nodes values are table values, but they can still be found by hand.

The values of the n points of the Gauss method are the roots of n'th degree Legendre polynomial[2].

The weights are calculated using the formula $w_i = \frac{2}{(1 - x_i^2)\,[L_n'(x_i)]^2}$, where L_n' is the first derivative of the Legendre polynomial.

This method is somewhat similar to the parabola method, but now all terms in the sum are assigned weights and now to reach higher accuracy instead of adding more polynomials, the degree of polynomial just increases.

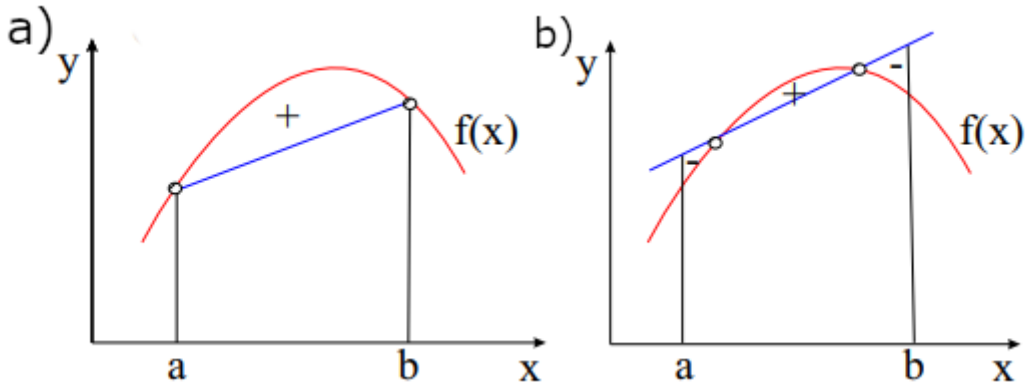

Figure 5: Illustration of the Gauss method[2]

On Fig 5(a) the function is approximated using a single trapezoid. It can be seen, that this approximation underestimates the area and produces some error. However on Fig 5(b)

[1] A **weight function** or weighted sum is used to give more "weight" or influence on the result than other elements in the same set. Example can be seen in equations (5) and (6), where some elements had an extra coefficient.[4]

[2] **Legendre polynomial** is an orthogonal polynomial that has lowest standard deviation from zero on the domain $[-1, 1]$. Integral of such polynomial on this domain equals 0.

function $f(x)$ is approximated using Gauss method, in it function area is underestimated and overestimated at the different places, which minimises the possible error.

Below I will list some first Legendre polynomials[3]:

$$L_0 = 1$$
$$L_1 = x$$
$$L_2 = \frac{1}{2}(3x^2 - 1)$$
$$L_3 = \frac{1}{2}(5x^3 - 3x)$$
$$L_4 = \frac{1}{8}(35x^4 - 30x^2 + 3)$$

As previously mentioned, in Gauss method the function is integrated over the domain of $[-1, 1]$:

$$I = \int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{i=0}^{n-1} w_i f(x_i), \tag{7}$$

where w_i are the weights of the points x_i.

This process was never explained on any of the sources I've looked, so I decided to describe it myself, because it is not something that can be understood easily. To be able to integrate not only over $[-1, 1]$ but on any interval, following function transformation can be done:

$$\text{Let } \xi \text{ be } \xi(x) = \frac{a+b}{2} + \frac{b-a}{2}x. \tag{8}$$

$\frac{a+b}{2}$ component of the transformation shifts/centres $f(x)$ at the origin, $\frac{b-a}{2}x$ "squeezes" or "stretches" the function to fit on the $[-1, 1]$ interval. But since after the transformation of the function, the integral changes, it also required to multiply by the factor of $\frac{b-a}{2}$ to compensate for the transformation - equation (9) .
The process of transformation of $f(x)$ from interval $[1.5, 2.5]$ to $[-1, 1]$ is shown on Figure 6.

$$\int_a^b f(\xi)\mathrm{d}\xi = \frac{b-a}{2}\int_{-1}^{1} f(\xi(x))\mathrm{d}x \tag{9}$$

From equation (7) and (9):

$$I = \int_a^b f(\xi)\mathrm{d}\xi \approx \frac{b-a}{2}\sum_{i=0}^{n-1} w_i f(\xi_i) \tag{10}$$

Where $\xi_i = \frac{1}{2}(b+a) + \frac{1}{2}(b-a)x_i$

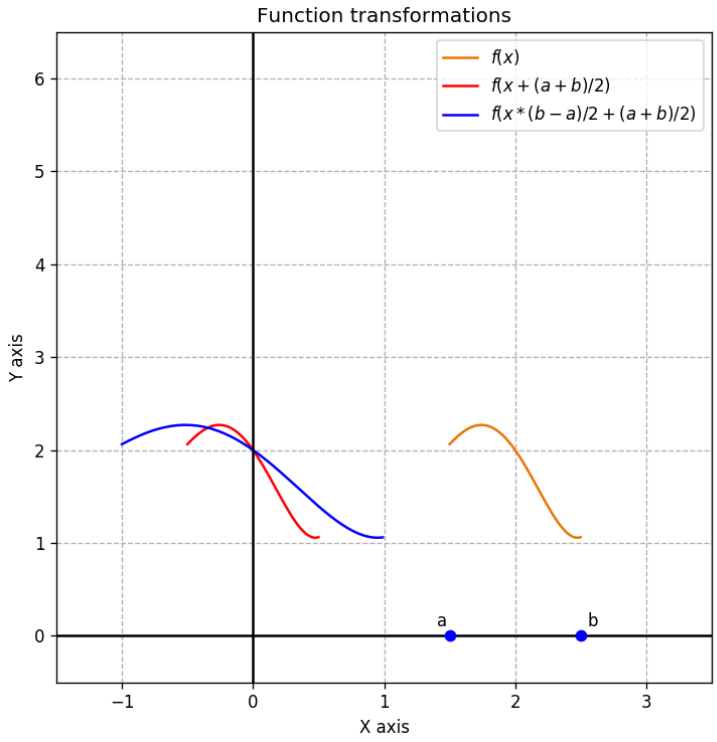

Figure 6: Transformation of $f(x)$

Below I worked out an example with approximation using second degree Legendre polynomial.

$$
\begin{aligned}
I &= \int_1^3 5x^4 - 36x^3 + 93x^2 - 102x + 42 \, \mathrm{d}x \\
I &= \frac{b-a}{2} \sum_{i=0}^{n-1} w_i f(\xi_i) = \frac{3-1}{2} \sum_{i=0}^{2-1} w_i f(\xi_i), \\
\xi_i &= \frac{3+1}{2} + \frac{3-1}{2} x_i = 2 + x_i, \\
w_0 &= 1, \; x_0 = -\frac{1}{\sqrt{3}}, \\
w_1 &= 1, \; x_1 = \frac{1}{\sqrt{3}}.
\end{aligned}
$$

$$
\begin{aligned}
I &= 1 \times f\big(2 - \frac{1}{\sqrt{3}}\big) + 1 \times f\big(2 + \frac{1}{\sqrt{3}}\big) = 1.94 + 1.171 = 3.11 \\
\text{error} &= \frac{4 - 3.111}{4} = 22.2\%
\end{aligned}
$$

Instead of looking at what degree of polynomial it will need to reach certain accuracy, I am measuring the error and time it takes to evaluate an integral using a polynomial of n'th degree. Results are presented in table 4.

Polynomial degree n	2	3	4	5	6
Integral value	3.11111...	4.00000...	3.99999...	4.00000...	3.99999...
Error, percent	22.2	$\ll 0.001$	$\ll 0.001$	$\ll 0.001$	$\ll 0.001$
Time, μs	3.96± 0.79	4.26± 0.51	5.89± 0.66	6.60± 0.75	8.80± 1.40

Table 4: Table showing integral approximation and timings for n'th degree Legendre polynomials

Looking at table 4 and comparing the results to the results of the other methods, a very significant improvement can be seen. Not only it took less then 5 microseconds to reach accuracy this high, it also required a low polynomial degree to do so. Which makes this method the most suitable for the numerical integration.

Conclusion

Looking at all methods, that were explored in the investigation and underlining everything that was previously said, it can be seen, that the most appropriate method of numerical integral approximation in the Gauss method. It is the fastest and the most accurate one. However, with very complicated concepts standing behind method, the maths formulae are not very difficult, which is a huge advantage of this method.

The next-best method was seen to be Simpson's method. It was slower than Gauss method and it took more steps for it to reach given accuracy. But the method is by far more accessible (in terms of learning) than Gauss method. To use this method you also don't need to know the roots of polynomials, weights etc., and you can quickly adapt this method for use on any machine or any programming language. So I would still consider this method as a reliable one and personally, I would prefer it to the Gauss method.

The third best method was one from the Riemann sums family, which is the midpoint method. It has proven itself to be the fastest method (compared to trapezoid and rectangle) and being simple at the same time. I would not prefer it to the Simpson's or Gauss methods for numerical integration on numerical devices, but due to its simplicity it can be a valid method for rough estimations of integrals by hand.

My method of comparing integrals seems relevant but at the same time has some issues: even though I tested all methods on the same computer, I still can not ensure, that testing conditions were exactly same all the time - this random uncertainty can affect my results.

Besides what was explored throughout this investigation, I would have still liked to do additional things. For example exploration can be extended to see how these methods would preform while approximating an integral like $\frac{\sin x}{x}$, that can not be evaluated by hand. Additionally, it could be interesting to write scripts for the TI-84 calculator and run tests on it.

Numerical integration is an important concept, that was developed by people. It is not only useful for calculating integrals, that can not be evaluated by hand, but also it is a way for integration on machines, in a sense that it is easier to make machine follow one algorithm, instead of teaching a computer all rules of integration (which is possible nowadays with use of artificial intelligence). Overall this was an interesting exploration to work on. I have developed my skills in use of mathematics as well as broadened my knowledge in the area of calculus. I also did a lot of coding practice while writing this IA, which is useful, because programming is a very important skill today.

Bibliography

[1] Tim Oftwar. *How does the TI-84 produce definite integral decimal approximations?* URL: `https://www.quora.com/How-does-the-TI-84-produce-definite-integral-decimal-approximations` (visited on Dec. 1, 2018).

[2] Задков В.Н. [Zadkov Viktor]. *Вычисление определенных интегралов. (Russian)[Computation of defined integrals].* Chap. 2. URL: `http://qilab.phys.msu.ru/people/zadkov/teaching/seminar1/semnumer2.pdf` (visited on Dec. 21, 2018).

[3] Eric W. Weisstein. *Legendre Polynomial.* URL: `http://mathworld.wolfram.com/LegendrePolynomial.html` (visited on Feb. 9, 2018).

[4] Eric W. Weisstein. *Weight.* URL: `http://mathworld.wolfram.com/Weight.html` (visited on Feb. 9, 2018).

Appendix

Listing 1: Script with all functions and testing environment

```
1  import numpy as np
2  import timeit
3  import math
4  import scipy.integrate
5  array=[]
6  def f(x):
7    return 5*pow(x,4)-36*pow(x,3)+93*pow(x,2)-102*x+42 #defining the function
         f(x)
8
9  def LeftSum(a,b,n):                    #function for left Reimann sum
10   dx = (b-a)/(n-1)
11   xvalues = np.linspace(a, b, n)
12   return sum(f(xvalues[1:]))*dx
13
14 def RightSum(a,b,n):                    #function for right reimann sum
15   dx = (b-a)/(n-1)
16   xvalues = np.linspace(a, b, n)
17
18   return sum(f(xvalues[:n-1]))*dx
19
20 def trapmethod(a,b,n,z=0):
21   dx = (b-a)/(n-1)
22   xvalues = np.linspace(a, b, n)                              #function for
       trapezoid method
23   for i in range(n-1):
24     array.append((f(xvalues[z])+ f(xvalues[z+1]))*dx / 2)
25     z+=1
26   return(sum(array))
27 def midpoint(a,b,n,z=0): #function for midpoint method
28   area=[]
29   dx = (b-a)/(n-1)
30   xvalues = np.linspace(a, b, n)
31   for i in range(n-1):
32     area.append(f((xvalues[z] + xvalues[z+1])/2)*dx)
33     z+=1
34   return sum(area)
35 def simpson(f, a, b, n):                #function for simpsons method
36     h=(b-a)/n
37     k=0.0
```

```
38        x=a + h
39        for i in range(1,int(n/2) + 1):
40            k += 4*f(x)
41            x += 2*h
42
43        x = a + 2*h
44        for i in range(1,int(n/2)):
45            k += 2*f(x)
46            x += 2*h
47        return (h/3)*(f(a)+f(b)+k)
48
49  a,b,n=1,3,4
50  #print("Left: ",LeftSum(a,b,n))
51  #print("Right: ",RightSum(a,b,n))
52  #print("Trap: ",trapmethod(a,b,n,z=0))
53  #print("Mid: " ,midpoint(a,b,n,z=0))
54  print(simpson(f,1,3,2))
55  #print(scipy.integrate.simps(f(np.linspace(a,b,5)),np.linspace(a,b,5),dx
        =0.5))
56  c='''
57  n=1
58  z=0
59  d=0
60
61  ns=[]
62  area=[]
63  errors=[0.1,0.05,0.01,0.001,0.0001,0.00001]
64  for i in range(len(errors)):
65      while (abs(4-simpson(f,1,3,n))/4)>errors[d]:
66          simpson(f,1,3,n)
67          n+=1
68      else: print(n) #subtract 1 for the actual n
69      ns.append(n)
70      d+=1
71  print(ns)
72
73  #[3, 4, 6, 10, 16, 28]
74  n=486
75  reslts=timeit.repeat("simpson(f,1,3,3)","from __main__ import simpson,f, a,
        b,n",number=1, repeat=100)
76  a=sum(reslts)/100
77  print(format(a,".4G"))
78  error=float(format(((max(reslts)-min(reslts))/(2*math.sqrt(n-1))), ".3G"))
```

```
79  print("error: ", error)
80  '''
81
82  def gauss(a,b,n): #function for gauss method
83      I=0
84      w2=[1,1]
85      x2=[-0.577350269, 0.577350269]
86      x3=[-0.774596669, 0, 0.774596669]
87      w3=[0.555555556, 0.888888889, 0.555555556]
88      w4=[0.347854845, 0.652145155, 0.652145155, 0.347854845 ]
89      x4=[-0.861136312, -0.339981044, 0.339981044, 0.861136312]
90      w5=[0.236926885, 0.478628670, 0.568888889, 0.478628670, 0.236926885]
91      x5=[-0.906179846, -0.538469310, 0         , 0.538469310, 0.906179846]
92      w6=[0.171324492, 0.360761573, 0.467913935, 0.467913935, 0.360761573,
            0.171324492]
93      x6=[-0.932469514, -0.661209386, -0.238619186, 0.238619186, 0.661209386,
             0.932469514] #weight and node values
94
95      if n==2:
96          for i in range (n):
97              xi= (b+a)/2 + x2[i]*(b-a)/2
98              I+=w2[i]*f(xi)
99      elif n==3:
100         for i in range (n):
101             xi= (b+a)/2 + x3[i]*(b-a)/2
102             I+=w3[i]*f(xi)
103     elif n==4:
104         for i in range (n):
105             xi= (b+a)/2 + x4[i]*(b-a)/2
106             I+=w4[i]*f(xi)
107     elif n==5:
108         for i in range (n):
109             xi= (b+a)/2 + x5[i]*(b-a)/2
110             I+=w5[i]*f(xi)
111     elif n==6:
112         for i in range (n):
113             xi= (b+a)/2 + x6[i]*(b-a)/2
114             I+=w6[i]*f(xi)
115     return I
```