Numerical Integration

dr Szymon Murawski

December 11, 2019

1 Integration

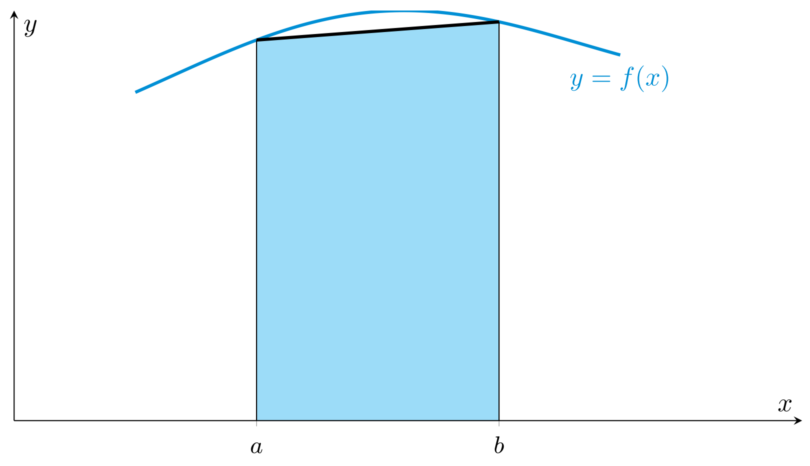
An **integral** of a function is the area below the curvature of the function in the given region. Depending on the order of the function this area can be multidimensional. For example for function $f(x)$, the integral in an interval $[a, b]$ is given by:

$$\int_a^b f(x)dx \tag{1}$$

Integration can be thought of as a reverse process to differentiation. Those two operations are the basis for calculus. In such a case, if the boundaries $[a, b]$ are not given, we use the term **antiderivative**.

2 Trapezoidal rule

We can approximate the value of defined integral by treating the are under the function as trapeze.

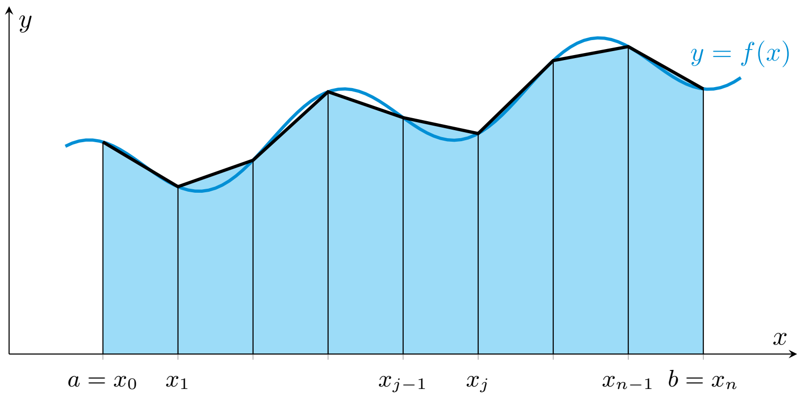


Based on the above picture we can then write:

$$I = \int_a^b f(x) \approx (b-a)\frac{f(a)+f(b)}{2} \tag{2}$$

where $b-a$ is the width h of the trapeze, and $\frac{f(a)+f(b)}{2}$ is the average height.

We can improve the accuracy of approximation, by partitioning the integral into smaller segments and applying the trapezoidal rule to each of the segment:



If we partition the domain into n points such that $h = \frac{b-a}{n}$ the formula for obtaining the approximation integral becomes:

$$I = \int_a^b f(x) = \int_{x_0}^{x_1} f(x) + \int_{x_1}^{x_2} f(x) + \cdots + \int_{x_{n-1}}^{x_n} f(x) \tag{3}$$

$$\approx h\frac{f(x_0)+f(x_1)}{2} + h\frac{f(x_1)+f(x_2)}{2} + \cdots + h\frac{f(x_{n-1})+f(x_n)}{2} \tag{4}$$

$$\approx \frac{h}{2}\left(f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right) \tag{5}$$

Where

$$x_i = a + i * h \tag{6}$$

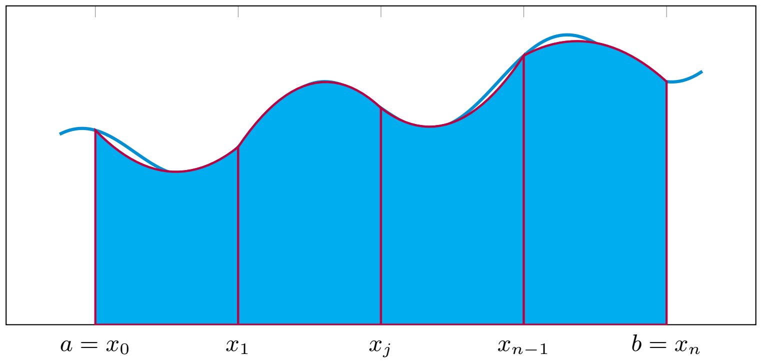
Pseudocode for algorithm that calculates the integral would then look more or less like this

```
double getIntegralTrapeze(double a, double b, int n){
  double h = (b-a) / n
  double I = h/2 * (getFunctionValue(a) + getFunctionValue(b))
  for (int i = 1; i < n; i++){
    I += h * getFunctionValue(a + i*h)
  }
  return I
}
```

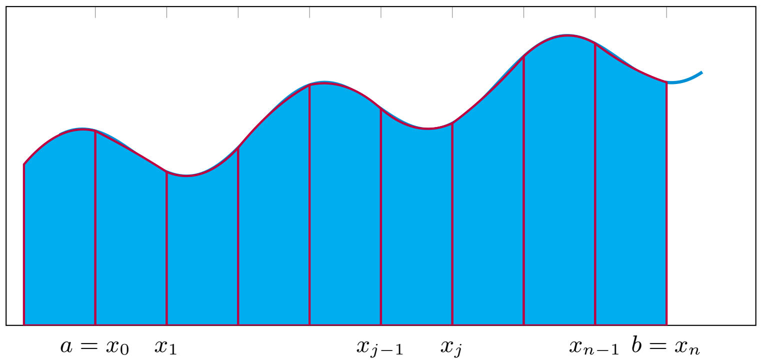
In the algorithm above n is the number of trapezes to which we divide our original integral, so the bigger number we choose the better approximation we get. `getFunctionValue` is a function that will return the value of $f(x)$.

3 Simpsons rule

Another way of obtaining the approximate value of an integral is the **Simpson's rule**. It is very similar to the previous, trapezoidal rule, but instead of approximating the integral by trapezes, quadratic functions are used instead.



Simpson's method provides much better approximation, just look at how closely the original functions is traced, when we use the same number of points as in the Trapezoidal rule:



3.1 Algorithm

It can be proven [1], that the area below the parabola can be computed as:

$$I = \int_a^b f(x)dx = h/3 \left(f(a) + 4f(\frac{a+b}{2}) + f(b) \right) \tag{7}$$

where $h = (b - a)/2$

Similar to Trapezoidal method we now divide the are into n segments. One important note: in Simpson's rule n **must be even**.

[1] `https://www.intmath.com/integration/6-simpsons-rule.php`

We now obtain the following formula for calculating the area:

$$I = \int_a^b f(x)dx \approx \frac{h}{3}\left(x_0 + 4x_1 + 2x_2 + 4x_3 + 2x_4 + \cdots + 4x_{n-1} + x_n\right)$$
$$\approx \frac{h}{3}\left[x_0 + 4(x_1 + x_3 + x_5 + \cdots + x_{n-1}) + 2(x_2 + x_4 + x_6 + \cdots + x_{n-2}) + x_n\right]$$

Where

$$x_i = a + i * h \tag{8}$$

This allows for very easy way of remembering Simpson's Rule:

$$I \approx \frac{h}{3}\left[FIRST + 4(sum\,of\,ODDs) + 2(sum\,of\,EVENs) + LAST\right]$$

```
double getIntegralSimpson(double a, double b, int n){
  double h = (b-a) / n
  double I = h/2 * (getFunctionValue(a) + getFunctionValue(b))
  for (int i = 1; i < n; i++){
    // we calculate whether we have odd or even point
    int coefficient = i / 2 == 0 ? 2 : 4;
    I += coefficient * getFunctionValue(a + i*h)
  }
  return I
}
```

4 Parsing any function

There are two main ways on how we can write mathematical equations:

- **Infix notation**, that we use everyday: $2 + 3$, $4 * (2 + 3) * 3^2$
- **Postfix notation**, that is used by computers: $2\,3\,+$, $4\,2\,3\;+\;*\,3\,2\;\wedge\;*$

Postfix notation, while strange looking, is very easy for computers to parse. We've learned about this during *Algorithms and Data Structures* classes, as the main component of the evaluator is the Stack. Stack, for a quick review, is a data structure where we push new elements on the top, and remove elements also from the top - just like a stack of plates.

Algorithm for evaluating postfix expression works as follows:

1. Read tokens from input one by one
2. If the current token is a number, push it onto the stack
3. If the current token is an operator $(+, -, *, /, \wedge)$ take two numbers from the top of the stack, perform the operation and push the result onto the stack
4. At the end there should be only one number on the stack - our result

Pseudocode:

```
postfixEvaluator(inputs)
  create stack
  for input in inputs
    if input is a number
      push input onto stack
    else
      op2 = stack.pop()
      op1 = stack.pop()
      output = evaluate(op1 input op2)
      push output onto stack
  return stack.pop
```

In the case of evaluating the function $f(x)$ using this algorithm one slight change needs to be made: if the token is x, the treat it as a number and push the value of x onto the stack

Now the question is how to obtain equation written in postfix notation. Thankfully there is another rather simple, albeit a little longer, algorithm: Shunting yard algorithm. This algorithm takes as an input an equation written in infix notation (the standard notation) and transforms it into the postfix notation. It works as follows:

1. Read a token from the input
2. If it's a number print it
3. If it's an operator:
    (a) While there's an operator on the top of the stack with greater precedence pop operators from the stack and print them
    (b) Push the current operator onto the stack
4. If it's a left bracket push it onto the stack
5. If it's a right bracket
    (a) While there's not a left bracket at the top of the stack pop operators from the stack and print them
    (b) Pop the left bracket from the stack and discard it
6. While there are operators on the stack pop them and print them

Pseudocode:

```
InfixToPostfix(expression)
  create stack
  foreach token in expression
    if token is a number
      print token
    else if token is an operator
      while Precedence(stack.peek()) > Precedence(token)
          OR (Precedence(stack.peek()) == Precedence(token)
            AND Associativity(token) = 'left')
```

```
10        print stack.pop()
11      stack.push(token)
12    else if token == '('
13      stack.push(token)
14    else if token == ')'
15      while stack.peek() != '('
16        print stack.pop()
17      stack.pop() //To remove ( from top of the stack
18  while stack.Count > 0
19    print stack.pop()
```

Only one change needs to be made to the above pseudocode. In step 2, apart from printing a number, also print an x if it is current token.