Faculty of Electrical Engineering and Informatics

Basics of Programming 2

Integrator2 C++ program

Developers doc

Student

Klevis Imeri

Budapest, May, 2023

Introduction and Problem Statement

Integration plays a major role in sciences. One of the problem aroused in this field is that many of them are unsolvable or very hard to solve, so approximation becomes a valid approach. These leads into development of programs that can approximate integrals using different methods. One of them which is valid is taking the Riemann Sum[1]. In other words summing up rectangles with width Δx and height $f(x)$. The smaller the Δx the smaller the error of approximation. This program makes this process easy and visual for the user. Even though the task may seem easy int first hand there are many difficult steps needed to be solved which mimic the problems the compilers solve trying to define a programming language.

Solution Design and Implementation

Even though there are many ways of solving the same problem, one of them being implemented in 'Integrator C program'[2], the design chosen for this program is structurally more compact and easier to expand for further development of the program in the future. The solutions implemented are similar to those needed to build modern compact compilers.
This section explains each classes purposes:

- Menu
- Lexer
- Parser
- Tree
- Node
- Tokens
- BmpImage
- Exceptions

Menu

```
class Menu{
  string dx;
  string X_0;
  string X_n;
```

```
    char input;
    isValidDouble();
    waitEnter();
  public:
    //constructor
    Menu();
    //setters|getters
    getStart();
    getEnd();
    getSize();
    getData();
    //methods
    start();
    print();
  };
```

The menu allows the user to input the limits of integration x_0, x_n and the width dx of the rectangle. The most important function is `void start()`. It starts the menu and takes care of user input. It deals with invalid user input in place and it also implements exception handling. After the menu is exited it checks if the class has remain consistent. If no it throws `InvalidDoubleInput()` exception.

Lexer

```
class Lexer{
    string str;
    vector<Token> tokenList;
    validArity();
  public:
    //constructor
    Lexer();
    //setters|getters
    setString();
    getStr();
    getTokenList();
    //methods
    askForFunction();
    tokenize();
    print();
    operator=(str);
};
//operator << overloading
operator<<(os, lexer);
```

The `void askForFunction()` asks user to input a function. It tries to tokenize it. If it succeeds than it returns, else tells the user the error in his input and ask for a new valid function. It uses exception handling with the `tokenize()` function to deal with unwanted user input.
The `void tokenize();` takes a the `str` where our function is stored to and loops throw the string to create the tokens. If it encounters an invalid input it throws `RuntimeError()` or `InvalidStringInput()` exception.
You can also `operator<<()` to print the lexer.
Example:

```
Lexer{
 Function: sin(2x)cos(4x-3)log(10x,e)
 Tokens: {
     {FUNCTION: 'sin'} {PAREN_LEFT: '('} {NUMBER: '2'}
     {OPERATOR: '*'} {VARIABLE: 'x'} {PAREN_RIGHT: ')'}
     {OPERATOR: '*'} {FUNCTION: 'cos'} {PAREN_LEFT: '('}
     {NUMBER: '4'} {OPERATOR: '*'} {VARIABLE: 'x'}
     {OPERATOR: '-'} {NUMBER: '3'} {PAREN_RIGHT: ')'}
     {OPERATOR: '*'} {FUNCTION: 'log'} {PAREN_LEFT: '('}
     {NUMBER: '10'} {OPERATOR: '*'} {VARIABLE: 'x'}
     {COMA: ','} {EULER: 'e'} {PAREN_RIGHT: ')'}
 }
}
```

Parser

```
class Parser{
   Tree tree;
   vector<Token> output;
  public:
   //constructor
   Parser();
   //geter
   getFunction();
   //methods
   shunting_yard();
   parse();
   integrate();
};
```

The `prefix shunting_yard(infix)` is the implementation of the Shunting Yard Algorithm[3]. Is takes tokens specifying a function in infix notation and outputs them in postfix notation or more widely knows as Reverse Polish Notation. It puts the output into the `vector<Token> output`.
The `bool parse(output)` takes the tokens prefix order and builds an expression tree from them.
The `double integrate(limits)`integrates the function according to the limits which the user entered at the menu. Returns the value of the Riemann Sum.

Tree

```
class Tree{
  Treetype type;
  Node<Token> root;
  public:
  //constuctor
  Tree()
  //methods
  buildExpressionTree();
  evaluate();
  print();
  //operator << overloading
  friend operator<<();
};
```

The `void buildExpressionTree(tokens)` takes the prefix order of tokens and builds an expression tree. It also sets the type of the tree to `EXPRESSION`.
The `double evaluate(double x)` takes the value of the variable x and evaluates the expression for that x. More precisely, it transverses the tree and evaluates the nodes.
Also the `operator<<()` was overloaded so the developer can have a terminal representation of the tree.
Example:

```
Function: sin(2x)cos(4x-3)log(10x,e)
Tree:
└──{OPERATOR: '*'}
     ├──{FUNCTION: 'log'}
          ├──{EULER: 'e'}
          └──{OPERATOR: '*'}
               ├──{VARIABLE: 'x'}
```

```
            └──{NUMBER: '10'}
    └──{OPERATOR: '*'}
        ├──{FUNCTION: 'cos'}
            └──{OPERATOR: '-'}
                ├──{NUMBER: '3'}
                └──{OPERATOR: '*'}
                    ├──{VARIABLE: 'x'}
                    └──{NUMBER: '4'}
        └──{FUNCTION: 'sin'}
            └──{OPERATOR: '*'}
                ├──{VARIABLE: 'x'}
                └──{NUMBER: '2'}
```

Node

```cpp
template<class T>
class Node{
    T data;
    vector<Node<T>*> children;
    public:
    //constructors
    Node(){};
    Node(T data);
    //destructor
    ~Node();
    //seters|geters
    setData();
    getData();
    getChildren();
    getChildAtIndex();
    //methods
    numberOfChildren();
    hasChildren();
    addChild();
    print();
    //friends
    friend class Tree;
};
```

Node a class template which allows the developer to create nodes that can store different data types. The `addChild(data)` creates a ***dynamically allocated*** node and puts its pointer in the `children` pointer list.

Because we have dynamically allocated nodes we need a destructor to take care of memory leaks. `~Node()` is a recursive function which deletes the whole subtree with `this` node as root. It calls delete on all the children by iterating the `children` pointer array.

Tokens

```
// Define operator precedence
const unordered_map<char, int> PRECEDENCE{
    {'+', 1},
    {'-', 1},
    {'*', 2},
    {'/', 2},
    {'^', 3},
};
// Define supported functions and their arities
const unordered_map<string, int> FUNCTION_ARITY{
    {"sin", 1},
    {"cos", 1},
    {"tan", 1},
    {"log", 2},
};
// Token Types
enum TokenType{
    NONE,
    NUMBER,
    OPERATOR,
    PAREN_RIGHT,
    PAREN_LEFT,
    COMA,
    FUNCTION,
    VARIABLE,
    EULER,
    PI
};
class Token{
public:
    TokenType type;
    string value;
    //contructors
    Token();
    //oprator <<
    friend operator<<();
};
```

Here you can find the definition of the `order of precedence` of the operators and the `functions arity` (How many parameters a function has). Here the functions that can be used are defined.
`enum TokenType{}` defined all the types of tokens there can be in our program.
Every token has a `type` and a `value`.
Example `oprator<<()`:

```
├──{FUNCTION: 'cos'}
```

BmpImage

```
class BmpImage{
    string name;
    int width;          //widht of the image  (px)
    int height;         //height if the image (px)
    BitmapFileHeader fileHeader;
    BitmapInfoHeader imageHeader;
    int dpi;
    vector<rgb> pixels;
    int size;           //Numbers in number line
    int oneEntity;      //pixel/entity
    double onePixel;    //enity/pixel
public:
    //constructor
    BmpImage()
    //methods
    resize()
    pixel()
    point()
    backgroundcolor()
    rectangle()
    line()
    horizontalLine()
    verticalLine()
    plane()
    function()
    integral()
    create();
};
```

It takes care of all operations related to the graphics of the program. The `functoin()` takes the expression tree (function) and graphs it into the cartesian plane created by `plane()`. The `integral()` draws the rectangles of

the Riemann Sum according with the limits set by the user in the menu.
The `resize()` resizes the plane not the image dimensions. Those are set in the constructor `BmpImage()`. The `create()` function creates a file and dumps the `image/file Headers` and the `pixel` data in it to create the file located in the directory of the running program.
Everything function has default colors, but the developer can set every color using `struct rgb()`.

```
struct rgb{
    unsigned int r, g, b;
};
```

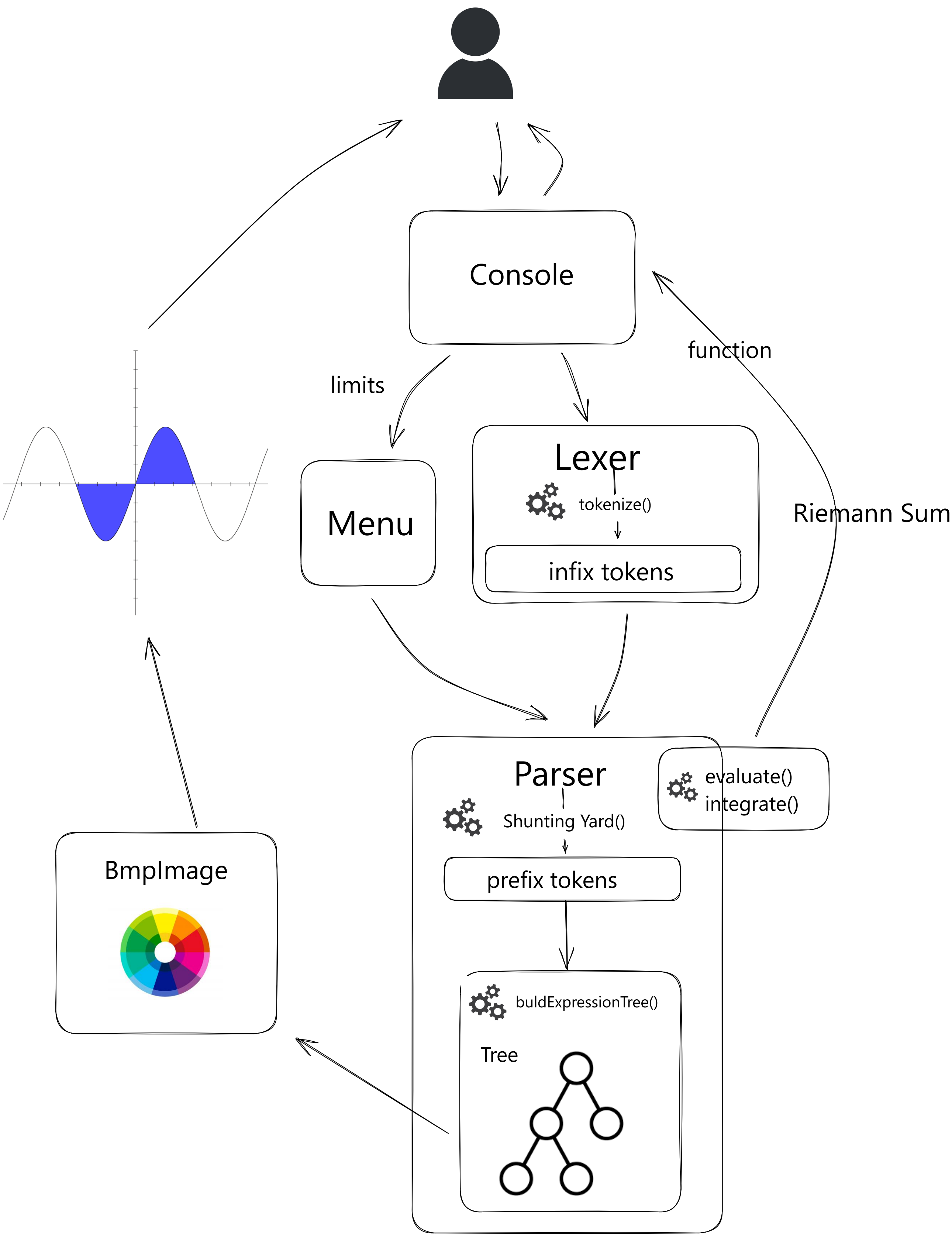
Exceptions

```
#include <exception>

class InvalidDoubleInput: public exception{};
class InvalidStringInput : public exception {};
class RuntimeError: public exception{};
class DivByZero: public exception{};
```

It includes the developer class defined exceptions. All the newly defined error types are derived classes form the base exception class. They all publicly inherit exception class.

Flowchart

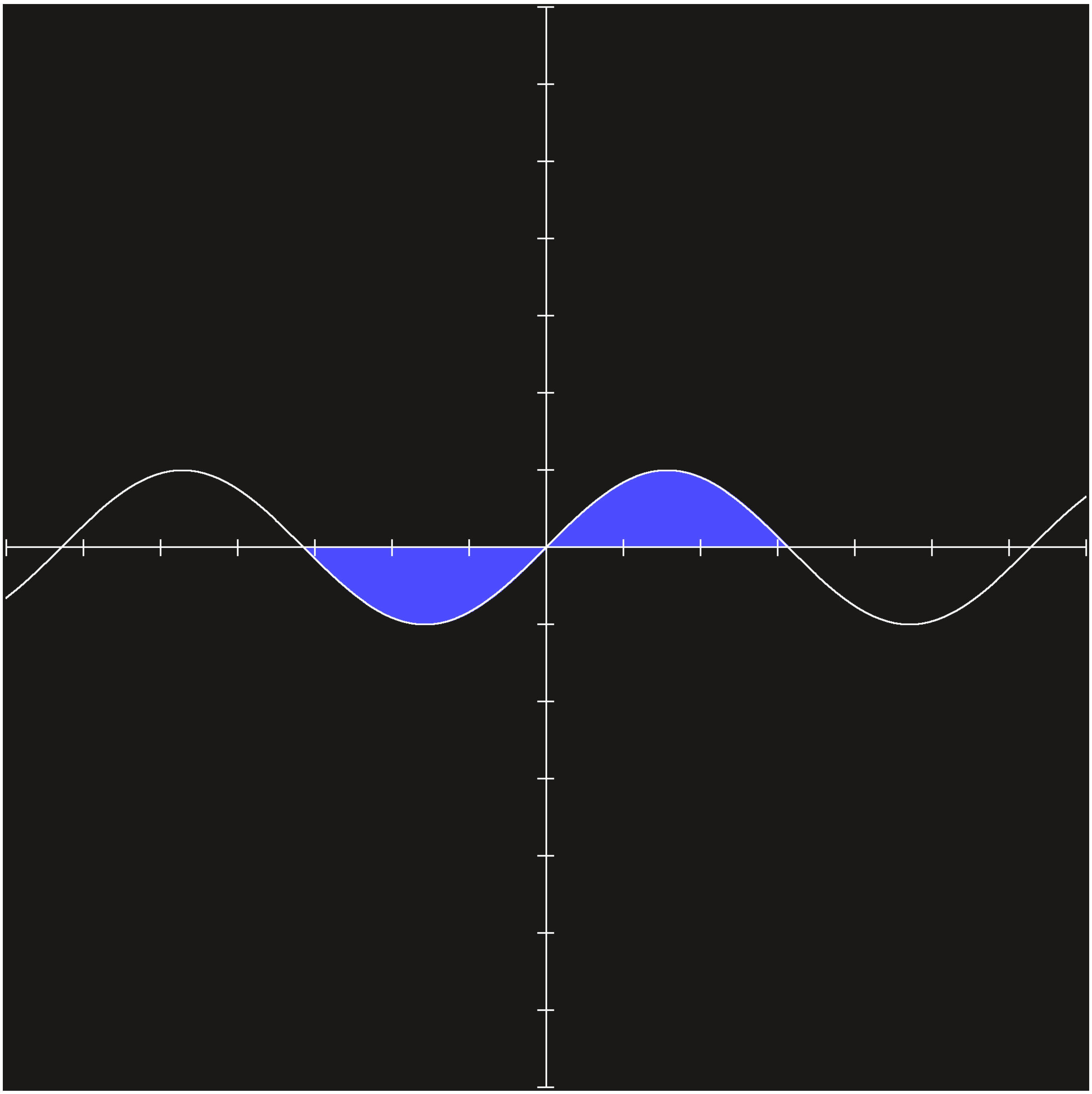


Console
limits
function
Menu
Lexer
tokenize()
infix tokens
Riemann Sum
Parser
Shunting Yard()
evaluate()
integrate()
prefix tokens
BmpImage
buldExpressionTree()
Tree

Testing and Verification

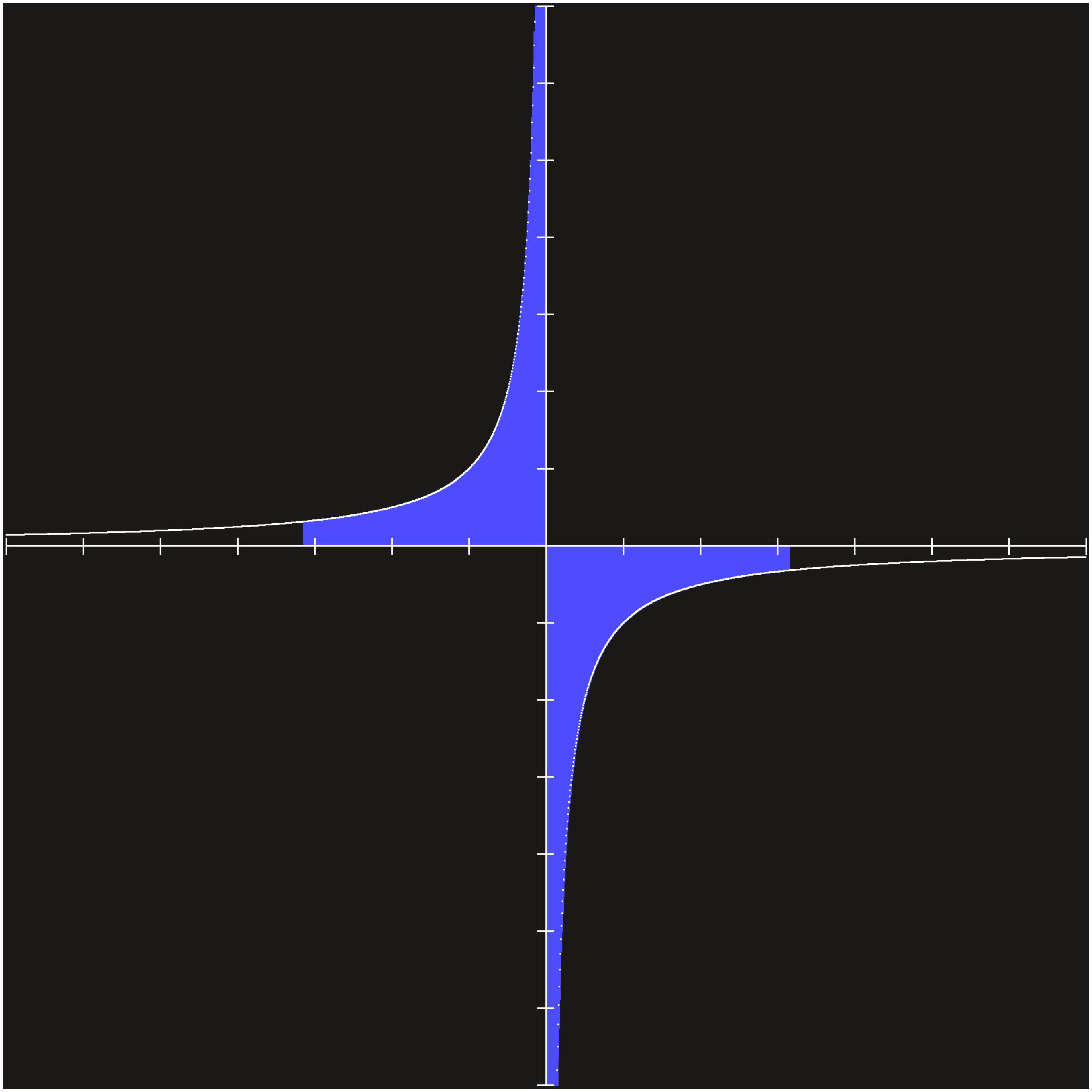
Now we will test different input to the program and test the result.

```
dx = 0.010000  X_0 =-3.140000  X_n = 3.140000
Type the function: sin(x)
Integral: -0.0000000000
```



Now we will test with in an interval where division by zero is prominent.
The program outputs nan because the Integral is $+\infty$ in this interval;

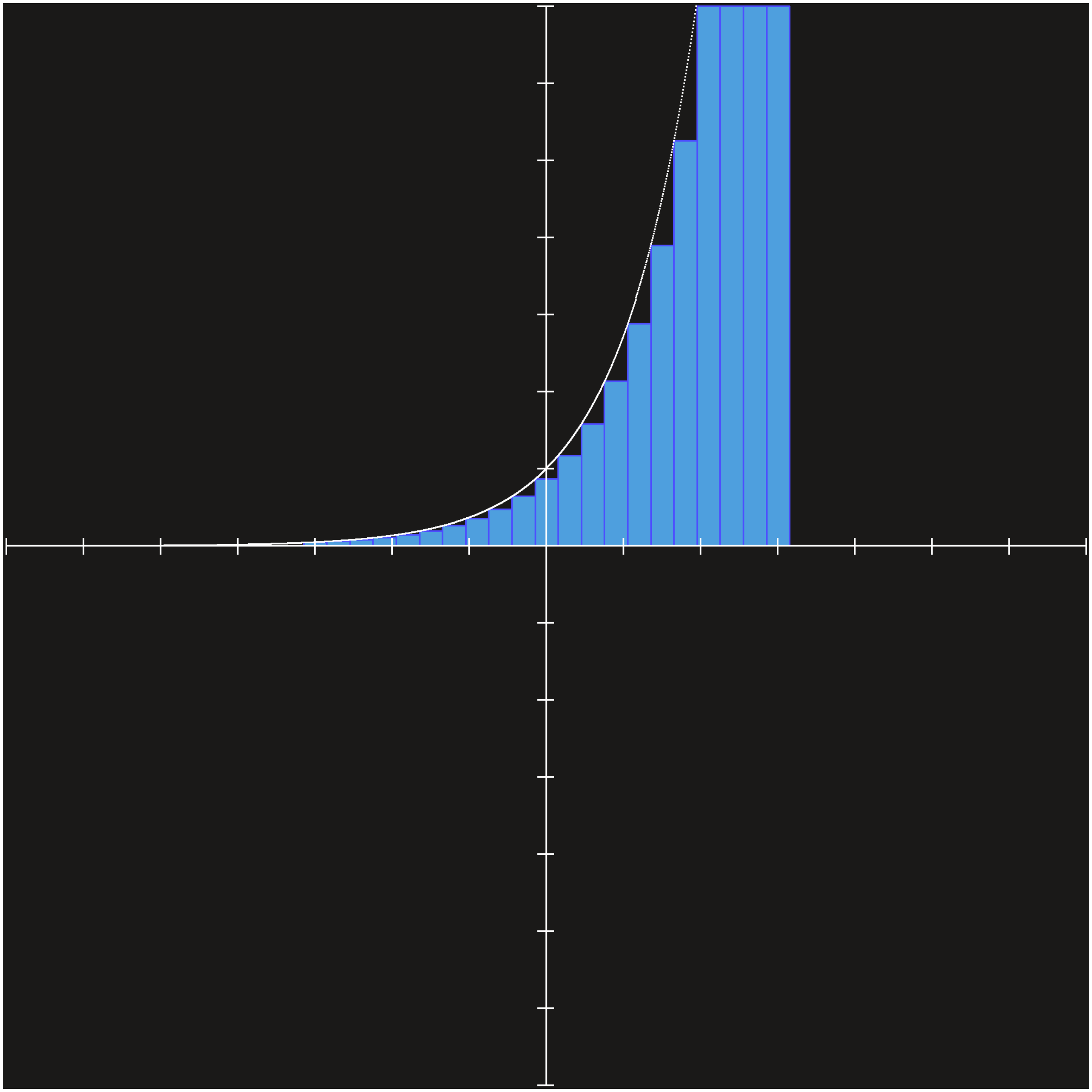
```
dx = 0.010000  X_0 =-3.140000  X_n = 3.140000
Type the function: 1/(-x)
Integral: nan
```



Making the width dx of the rectangles is also an option.

```
dx = 0.300000  X_0 =-3.140000  X_n = 3.140000
Type the function: e^x
```

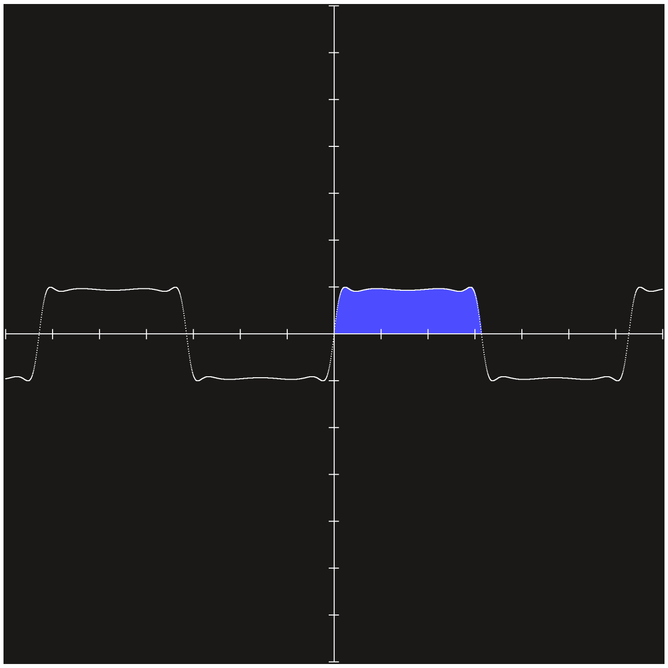
```
Integral: 20.1743795615
```

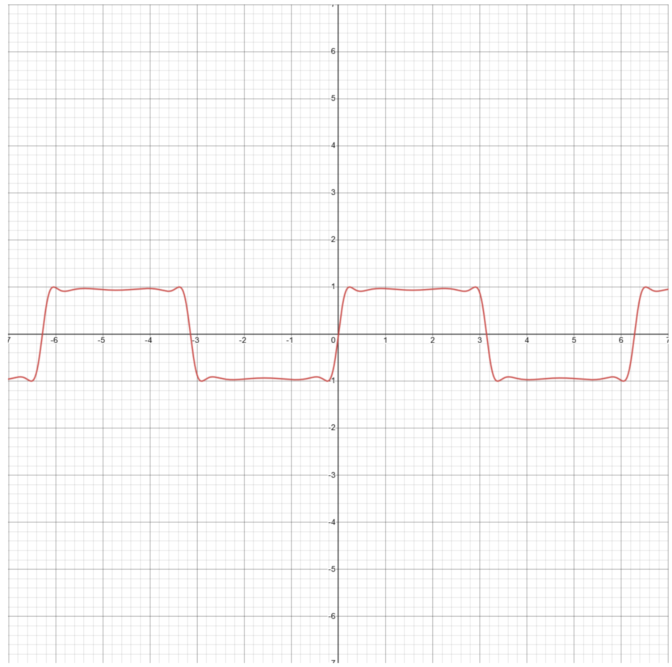


Let us test with more complicated functions and compare it with well established graphing calculators as Desmos[4]. The output of Desmos will be the integral and the graph.

```
dx = 0.010000  X_0 = 0  X_n = 3.140000
Type the function: sin(2sin(2sin(2sin(x))))
Integral: 2.8526723995
```

$$\int_0^{3.14} \sin(2\sin(2\sin(2\sin(x))))dx = 2.8527420$$

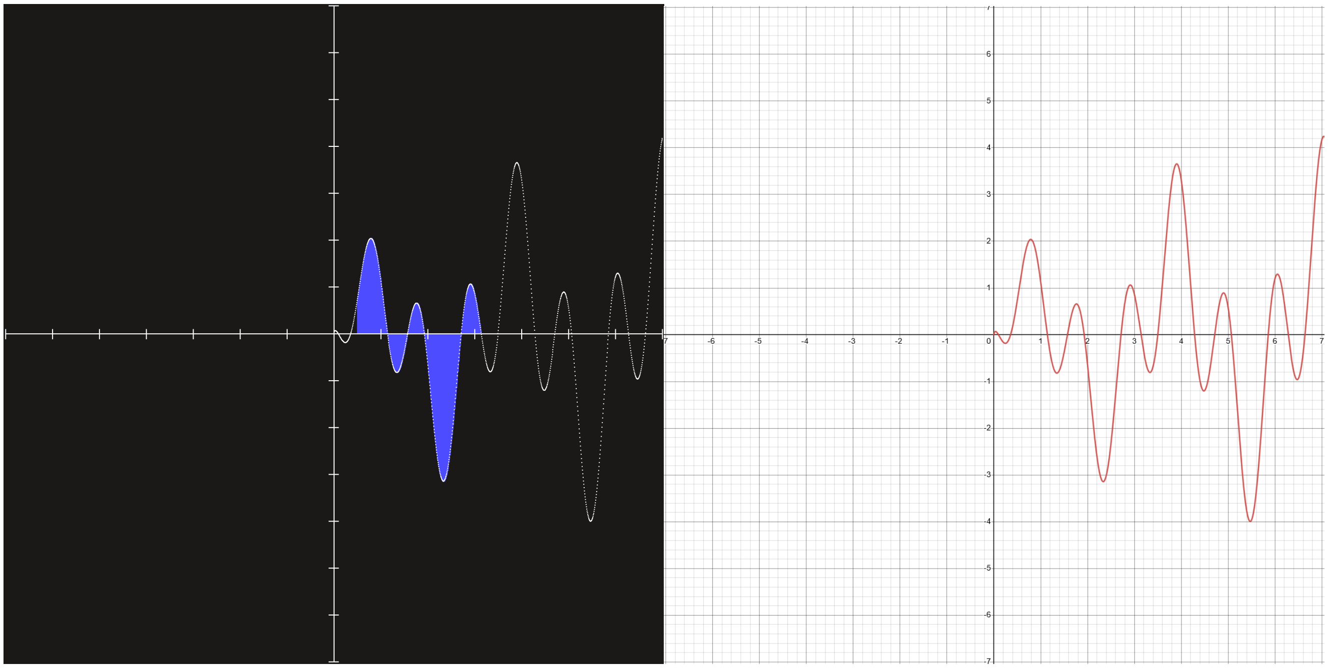




We make the precision larger but the computation time will increase.

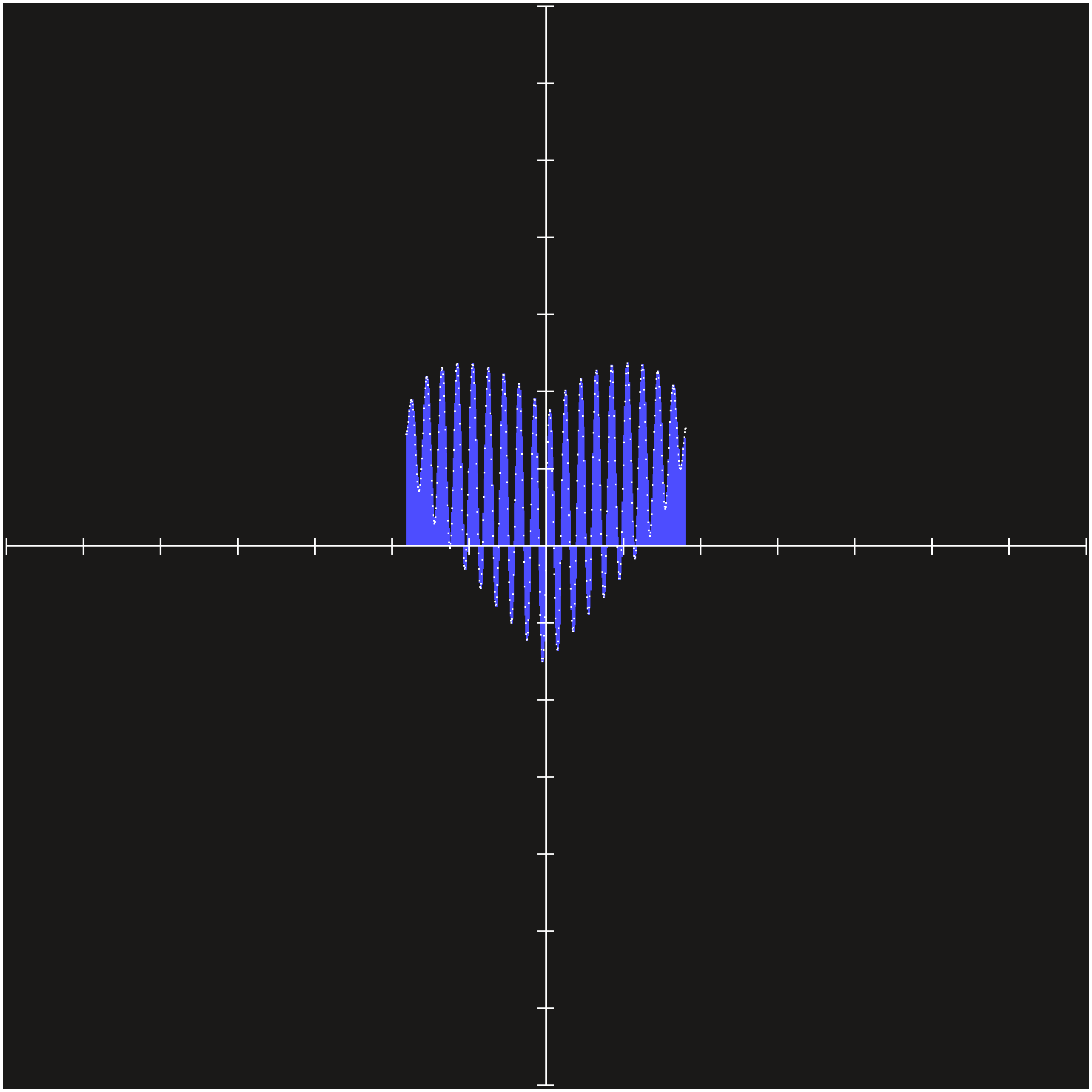
```
dx = 0.0001   X_0 = 0.5   X_n = 3.140000
Type the function: sin(2x)cos(4x-3)log(10x,e)
Integral: -0.3667504457
```

$$\int_{0.5}^{3.14} \sin(2x)\cos(4x-3)\ln(10x)dx = -0.366786230428$$



Next case is using NAN to cut of the domain of the function.

```
dx = 0.010000  X_0 =-1.80000  X_n = 1.80000
Type the function: x^(2/3)+0.9(3.3-x^2)^(1/2)sin(10*pi*x)
Integral: 3.1960759104
```



Discussion and Future Enhancements

The program in its present form can manage many types of functions with very high precision and in reasonable running time. Error may appear but they are easy to find and fix because of the structure and compactness of the program. Error handling is exceptional but in the future there is more work to be done there. The program is very stable and it can help the user calculate a large variety of integrals.
The biggest challenges faced were:

- Application of Shunting Yard Algorithm
- Building the Expression Tree and Evaluating it
- User input error handling

Even though the program in his current state is simple and usable there are many spaces left for improvement in the future. Some of them are:

- Bigger menu with more options
- Graphical user interface
- Runtime resizing
- More support for types of output images
- Optimization
- More Complete Classes
- Better user input error handling
- User option to create and define functions

Conclusions and References

In conclusion, even though many Integrals are unsolvable, their approximation is assisted by computational methods. The program achieves this while keeping the speed, simplicity, and visual output.

---

1. https://en.wikipedia.org/wiki/Riemann_sum - How Riemann sum is used to find integrals.↩
2. https://github.com/KlevisImeri/Integratior - The first release of the program in c.↩
3. https://en.wikipedia.org/wiki/Shunting_yard_algorithm - Pseudocode and Explanation of Shunting Yard Algorithm.↩
4. https://www.desmos.com/calculator - Desmos the graphing calculator↩