DEPARTMENT OF ELECTRICAL
AND ELECTRONICS
ENGINEERING

EE492 PROJECT MIDTERM REPORT

# Recognition and Mathematical Operation System for Handwritten Expressions

Ender Efe ERYEGÜL – 280206062

Bartu ERDAL - 280206061

Mahmut Cenk EFELER

DATE: 08/05/2025

## I. PROJECT STATEMENT

The primary objective of the project is to execute mathematical operations on a given handwritten mathematical expression by recognizing the expression using a deep learning model. The main system of the project is designed as a pipeline of two main subsystems: The image processing and the mathematical processing system. These systems operate sequentially. The input of the entire system directly serves as the input for the image processing subsystem. After the processes in the first subsystem, the output of it enters the next subsystem, mathematical processing, as an input. Following that, the mathematical processing system gives an output as desired by the user that varies as symbolical and numerical differentiation, numerical integration and more. Afterwards, the subsystem's output serves as an output for the whole system.

In the following sections, the overall structure of the subsystems will be examined under several titles.

### A. The Image Processing System

The system begins by taking a handwritten expression image as an input and applying image processing techniques such as contour detection based on OpenCV to segment the characters. These characters are then passed through a trained convolutional neural network using that is implemented in PyTorch to classify each character. After each character is classified, an algorithm takes a list of the classified characters as an input where each node contains the coordinates and scale information of the characters. This algorithm uses several methods (e.g X, Y and Z) to convert the list to a string format, which becoming the output of the system.

#### a. Character Segmentation

The first step of image processing system is to segmentate the input image into individual character images. In the image processing system, each character is classified individually, thus, reliable segmentation is crucial for this project. For this task, we took advantage of OpenCV library.

OpenCV (Open Source Computer Vision Library) is a widely used open-source library designed for computer vision applications such as our project. OpenCV offers comprehensive tools for image and video processing. Due to its high performance and flexibility, OpenCV has become a popular choice in both academic and industrial applications. Although the library was written in C/C++, it provides its features for other languages like Python and Java. For these reasons, we decide0d to use OpenCV library for most of our image operations in our project.[1]

In our implementation, we start by converting input image to grayscale which reduces complexity of data and emphasizes structure at handwritten characters. Then we apply Gaussian Blur on to eliminate high-frequency noise that might mess with thresholding and contour detection. Next, we binarize the blurred image using Otsu's thresholding with inverse binary conversion. This turns the image into a clear foreground of characters and background white, making it ready for morphological processing.

After thresholding, we make dilation with a 4x4 rectangular kernel. This helps to close small gaps between lines that should be connected which is especially critical for handwritten inputs where strokes may not fully join.

Once pre-processing is done, we employ contour detection using cv2.findContours function. Each detected contour ideally corresponds to a single character. However, because handwritten input is inherently variable we implemented a robust bounding and padding strategy. For each contour:

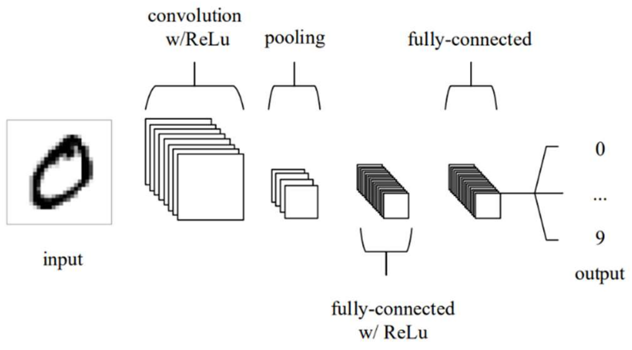
- A bounding box is calculated using cv2.boundingRect.
- A square blank canvas is prepared, ensuring each character fits uniformly regardless of its original aspect ratio.
- The character is centered in this canvas and resized to 64×64 pixels, which matches the expected input size of our convolutional neural network.

This approach is effective not only because of its simplicity but also because of its strong empirical performance as shown in work like Shi et al. (2016) at successfully using contour-based segmentation for handwritten Chinese character recognition [2]. Additionally, our method follow the principles in region-based segmentation which is often favored in applications involving isolated symbol recognition [3].

### b. Character Classification

Character classification module is responsible for identifying each segmented character image and assigning it to correct class label. For this task, we use a Convolutional Neural Network, a deep learning architecture made for processing visual data.

A Convolutional Neural Network is a kind of artificial neural network that's really good at analyzing visual imagery. It works by applying a series of learnable filters—called kernels—across input image to pull out hierarchical features. These filters pick up simple patterns like edges in early layers and then progressively capture more complex structures deeper on the network. The use of shared weights and local connectivity lets CNNs be both more computationally efficient and good at capturing spatial hierarchies in images [4].



In many traditional character recognition systems like ones that use the MNIST dataset, the input size is usually 28×28 pixels. But in our project we tried to recognize not just alphanumeric characters but also different math operators like '+', '-', '√' and '∫'. These symbols often have small details that can get lost with lower resolution. So, we use 64×64 pixel inputs to help the CNN pick up finer features which makes it better at telling symbols apart. This choice is also supported by recent studies showing that higher resolution inputs can improve accuracy when the symbol set includes more complex shapes or a bigger number of classes such as ours [5].

Designing an effective CNN architecture means choosing things like how many convolutional layers to include what kernel sizes to pick and how many filters to use. From recent papers we found:

- A study on handwritten character recognition used a CNN with four convolutional layers each followed by max-pooling. It reached high accuracy on Kaggle and MNIST showing it can extract complex features [5].
- Another paper on non-native handwritten characters built a base model with four conv layers plus max-pooling. They put a ReLU activation after each conv layer to add non-linearity and help the network learn tougher representations [6].

Considering these findings, our CNN architecture includes:

- Input Layer: Accepts 64×64 grayscale images.
- Convolutional Layers: A series of 3 to 4 convolutional layers with increasing numbers of filters (e.g., 32, 64, 128), each using 3×3 kernels. This setup allows the network to learn a hierarchy of features, from simple edges to complex shapes.
- Activation Functions: ReLU (Rectified Linear Unit) activations are applied after each convolutional layer to introduce non-linearity and accelerate convergence.
- Pooling Layers: Max-pooling layers follow each convolutional layer to reduce spatial dimensions and control overfitting.
- Fully Connected Layers: One or more dense layers that interpret the extracted features and make classification.
- Output Layer: A SoftMax layer that outputs probabilities across the classes, corresponding to the various characters and symbols.

### c. Output Representation

After each character is segmented and classified, the system organizes the results into a structured output format to prepare them for mathematical processing. This step serves as the bridge between the image processing and mathematical analysis subsystems.

From the segmentation stage, the position and size of the character in the original image are recorded as a tuple (x,y) and (w,h). From the classification stage, the identified character with its classification confidence recorded as (char, conf). Char is the predicted character (such as “5”, “*”, “x” etc. ) and conf is the confidence score, a float between 0 and 1. The confidence of the character might be needed in some tests so it’s included as well.

This list is then passed into a higher-level parsing structure, whose responsibility is to convert the list of symbols into a mathematically meaningful string format. The parser leverages the positional data specifically the coordinates and bounding box dimensions of each symbol to determine the relational structure of the expression (e.g., fraction lines, term ordering).

## B. The Mathematical Processing System

The system takes the output of the previous subsystem as an input. The system has several of modules (e.g Tokenizer, PostFixExtractor, MathOps) and uses some data structures (e.g Binary tree, stack, queue). The postfix expression in a stack gets converted to a binary tree. The reason we chose binary tree is its capability of fragmenting and reconstructing its own structure. The leftchild and the rightchild of any node can be extracted, subjected to a mathematical operation and reinserted into the correct position within the structure.

### a. Tokenization

The system takes the output of the previous subsystem as an input. The system has several modules (e.g Tokenizer, PostFixExtractor, MathOps) and uses some data structures (e.g Binary tree, stack, queue). The postfix expression in a stack gets converted to a binary tree. The reason we chose binary tree is its capability of fragmenting and reconstructing its own structure. The leftchild and the rightchild of any node can be extracted, subjected to a mathematical operation and reinserted into the correct position within the structure.

The base Token class and its three subclasses (FunctionToken, OperatorToken and NumberToken) have been successfully implemented. Each subclass contains specific attributes:

- FunctionToken represents mathematical functions such as sin, cos.
- OperatorToken includes metadata like operator type (prefix, suffix) and precedence level.
- NumberToken stores numerical values and flags whether the value is constant or not.

Each of the three classes passed independent testing and verification to guarantee proper operation, including content search, operator type and precedence checks and constant identification. This object-oriented token structure serves as the basis for efficient mathematical statement processing and analysis.

The tokenizer module has been implemented in Python. It can successfully parse mathematical expressions into structured tokens such as numbers, operators and function identifiers. Common mathematical functions (such as sin, cos, tan, log), constants, and conventional arithmetic operations are all supported by the implementation, which also makes the difference between various operator types (such as prefix, suffix, left-associative and right-associative).

Mathematical operations, trigonometric functions, negative numbers, variables, factorials, percentages, square roots and power operations have all been tested using a variety of expressions. Every tested scenario shows the tokenizer acting as it should.

We’re now working on better error handling for wrong inputs and refining how we parse functions, especially ones with multiple characters or variable names.

Also, we’ve added a basic parsing mechanism that understands spatially structured symbols like division bars by looking at their geometric relationships in the image. It uses the character list from our image processing pipeline.

### b. Postfix Conversion and Tree Construction

To convert infix expressions into postfix notation, the system employs a stack-based algorithm inspired by Dijkstra's Shunting Yard Algorithm. This method processes tokens sequentially, managing operator precedence and associativity to produce a postfix expression [7].

Once the postfix expression is generated, it is used to construct a binary expression tree. In this tree:

- Each operator becomes an internal node.
- Each operand (number or variable) becomes a leaf node.

For example the postfix expression 4 56 3.1 9 + - * corresponds to the infix expression (9 - 3.1) * (56 + 4).

In the resulting binary tree the root node is the multiplication operator * and its left and right subtrees represent the sub-expressions (9 - 3.1) and (56 + 4) respectively.

This tree structure makes evaluation and manipulation of mathematical expressions efficient and serves as foundation for later operations like differentiation and integration.

### c. Mathematical Operations Module

The Mathematical Operations Module (MathOps) is designed to perform advanced symbolic computations on the binary expression tree.

#### Symbolic Differentiation

Symbolic differentiation is implemented using a recursive traversal of the binary expression tree. The differentiation rules are applied based on the type of each node:

- Addition/Subtraction (+, -): The derivative of a sum/difference is the sum/difference of the derivatives of the operands.
- Multiplication (*): The product rule is applied:$(f * g)' = f' * g + f * g'$
- Division (/): The quotient rule is applied: $(f / g)' = (f' * g - f * g') / g\text{^}2$
- Exponentiation (^): The power rule is applied when appropriate.

Each node's derivative is computed by recursively applying these rules, constructing a new binary tree representing the derivative of the original expression.

**Economy:**

- The system requires no additional hardware beyond a standard personal computer with GPU support.
- All development tools used are open-source and free (e.g., Python, OpenCV, PyTorch).
- The only ongoing costs are electricity consumption and personal computer maintenance.

**Environment:**

- The system has minimal power consumption during operation, depending on CPU/GPU usage.
- No physical components are manufactured, hence there is no material waste or environmental hazard.
- The development process is fully digital and does not produce any physical noise or pollution.

**Society:**

- The system can be used as an assistive educational tool, especially for students with handwriting input needs or accessibility challenges.
- There are no direct privacy concerns since all processing is performed locally unless future deployment requires online services.

**Politics:**

- The design is inclusive and neutral; it does not favor any demographic or geographical region.
- The tool can contribute to educational equity by enabling free access to handwritten mathematical processing.

**Ethics:**

- All third-party libraries are used in accordance with their open-source licenses.
- The project avoids collection of user data, maintaining user privacy and system transparency.

**Health and Safety:**

- Since the project is software-only, it poses no direct safety risks.
- All development and testing are done in safe computing environments.

**Manufacturability:**

- The project is purely software-based and can be run on any modern personal computer.
- The solution is easily distributable and installable on compatible systems.
- Sustainability:
- The software is modular, allowing for easy updates and integration of new features.
- The system is designed to be maintainable and extensible for future improvements, including additional mathematical features or handwriting styles.

## II. CURRENT STATUS

### A. The Image Processing System:

The image processing module is now functional and performs character segmentation on various handwritten expressions. It uses OpenCV techniques like thresholding dilation and contour detection to pull out individual characters with good accuracy under normal conditions. In early stages we tried a basic recognition system that extracted bitmap images of standard fonts and matched them directly to segmented characters. That method gave near-perfect results for typed text in known fonts but failed to work on handwritten input. After testing many reference fonts performance on actual handwriting stayed unreliable and not acceptable for our goals so we switched to a CNN-based classification system.

To train the CNN, we collected a custom dataset. Approximately 5 to 6 individuals contributed handwritten samples resulting in nearly 600 images per character. These images were collected in varying sizes and writing styles to increase diversity. To further augment the dataset, several preprocessing operations were applied using OpenCV:

- Random rotations in the range of ±1 to 5 degrees.
- Occasional application of Gaussian blur to simulate writing noise and focus variability.

After augmentation the dataset grew to around 1000 to 1300 images for each character giving the model a rich and varied set of training examples. It includes digits lowercase and uppercase letters and mathematical symbols.

Right now the system still can’t convert handwritten expressions straight into a structured string from start to finish. So, for testing, we created a couple of synthetic CharacterList structures that act like the output we’d expect from a working recognition pipeline. These lists include character labels along with info like position and bounding box size just like real output from segmentation and classification. We used these simulated lists to test the parser module. It was able to read the order and spatial relationships between characters and turn them into proper math expressions in string format. Division is handled well by spotting symbols above and below horizontal lines and treating them as numerator and denominator.

This simulation helped us test parser logic in a safe way even though it doesn't support exponent notation or more complex spatial structures yet. It provided us with useful ideas on how the system handles various layouts, stacked divisions and overlapping symbols. even though early findings are promising, the main objective remains full end-to-end recognition from actual handwritten input once the character classification component is completely linked.

### B. The Mathematical Processing System:

Token Class Implementation:

- The base Token class and its three subclasses (FunctionToken, OperatorToken and NumberToken) have been successfully implemented. Each subclass contains specific attributes:

- ➢ FunctionToken represents mathematical functions such as sin, cos.
- ➢ OperatorToken includes metadata like operator type (prefix, suffix) and precedence level.
- ➢ NumberToken stores numerical values and flags whether the value is constant or not.

- Each of the three classes passed independent testing and verification to guarantee proper operation, including content search, operator type and precedence checks and constant identification. This object-oriented token structure serves as the basis for efficient mathematical statement processing and analysis.

Tokenizer Module:

- The tokenizer module has been implemented in Python. It can successfully parse mathematical expressions into structured tokens such as numbers, operators and function identifiers. Common mathematical functions (such as sin, cos, tan, log), constants and conventional arithmetic operations are all supported by the implementation, which also makes the difference between various operator types (such as prefix, suffix, left-associative and right-associative).

## Test Results

1. Fundamental Arithmetic Functions

"2 + 3 * 4" was successfully processed by the tokenizer, which correctly identified the integers and operators while preserving the correct precedence. Accurate operator hierarchy handling was demonstrated by processing addition with precedence 2 after multiplication with precedence 3.

2. Functions of Trigonometry

The system properly identified both trigonometric functions and their arguments in the tokenized expression "sin(30) + cos(60)". For the addition between functions, the tokenizer kept the correct operator precedence and handled the parenthesis correctly.

3. Negative Values

Excellent handling of negative numbers in the equation "-5 + 3" was displayed by the tokenizer. It processed the negative integer correctly while preserving the right operator precedence for the next addition, correctly identifying the minus sign as a prefix operator.

4. Complicated Words

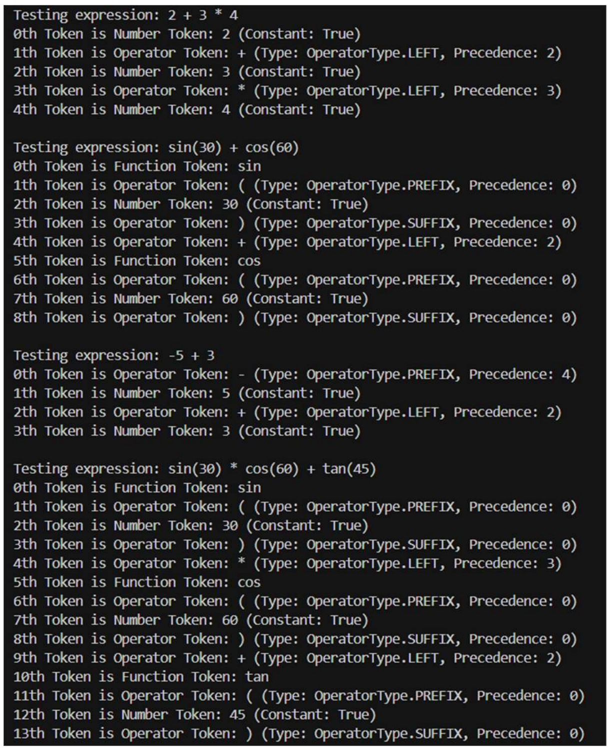
The complex equation "sin(30) * cos(60) + tan(45)" was successfully processed by the system, demonstrating its capacity to manage several trigonometric functions in a single expression.

The tokenizer processed all function inputs correctly while maintaining the correct operator precedence between addition and multiplication.

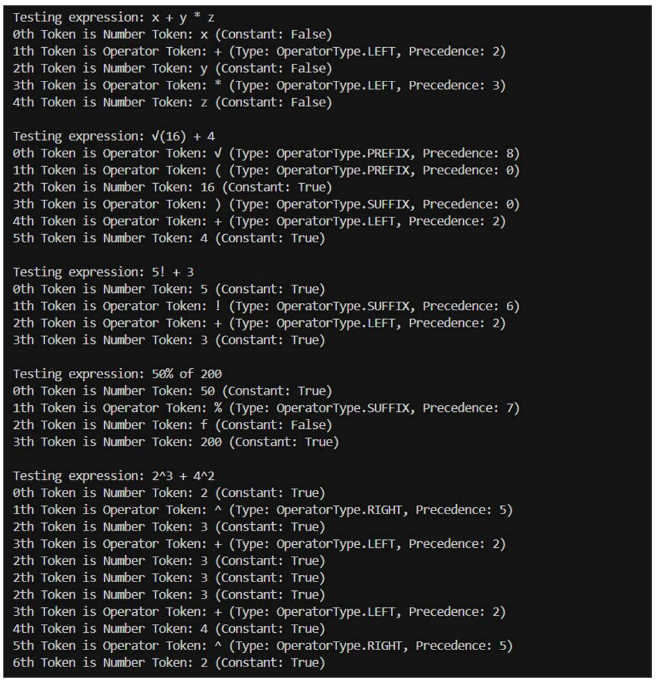
5. Elements

The tokenization of the expression "x + y * z" was accurate, and the algorithm correctly recognized variables as non-constant values. The tokenizer proved its capacity to handle algebraic expressions by correctly differentiating between variables and operators while maintaining accurate operator precedence.

6. Special Operators The tokenizer showed strong capabilities in handling special operators, successfully processing square roots (√(16) + 4), factorials (5! + 3), percentages (50% of 200), and power operations (2^3 + 4^2). Although the majority of special operators were handled successfully, there are a few small problems that require solving, such as the "of" keyword in percentage expressions and certain missing tokens in power operations.



```
Testing expression: 2 + 3 * 4
0th Token is Number Token: 2 (Constant: True)
1th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
2th Token is Number Token: 3 (Constant: True)
3th Token is Operator Token: * (Type: OperatorType.LEFT, Precedence: 3)
4th Token is Number Token: 4 (Constant: True)

Testing expression: sin(30) + cos(60)
0th Token is Function Token: sin
1th Token is Operator Token: ( (Type: OperatorType.PREFIX, Precedence: 0)
2th Token is Number Token: 30 (Constant: True)
3th Token is Operator Token: ) (Type: OperatorType.SUFFIX, Precedence: 0)
4th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
5th Token is Function Token: cos
6th Token is Operator Token: ( (Type: OperatorType.PREFIX, Precedence: 0)
7th Token is Number Token: 60 (Constant: True)
8th Token is Operator Token: ) (Type: OperatorType.SUFFIX, Precedence: 0)

Testing expression: -5 + 3
0th Token is Operator Token: - (Type: OperatorType.PREFIX, Precedence: 4)
1th Token is Number Token: 5 (Constant: True)
2th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
3th Token is Number Token: 3 (Constant: True)

Testing expression: sin(30) * cos(60) + tan(45)
0th Token is Function Token: sin
1th Token is Operator Token: ( (Type: OperatorType.PREFIX, Precedence: 0)
2th Token is Number Token: 30 (Constant: True)
3th Token is Operator Token: ) (Type: OperatorType.SUFFIX, Precedence: 0)
4th Token is Operator Token: * (Type: OperatorType.LEFT, Precedence: 3)
5th Token is Function Token: cos
6th Token is Operator Token: ( (Type: OperatorType.PREFIX, Precedence: 0)
7th Token is Number Token: 60 (Constant: True)
8th Token is Operator Token: ) (Type: OperatorType.SUFFIX, Precedence: 0)
9th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
10th Token is Function Token: tan
11th Token is Operator Token: ( (Type: OperatorType.PREFIX, Precedence: 0)
12th Token is Number Token: 45 (Constant: True)
13th Token is Operator Token: ) (Type: OperatorType.SUFFIX, Precedence: 0)
```



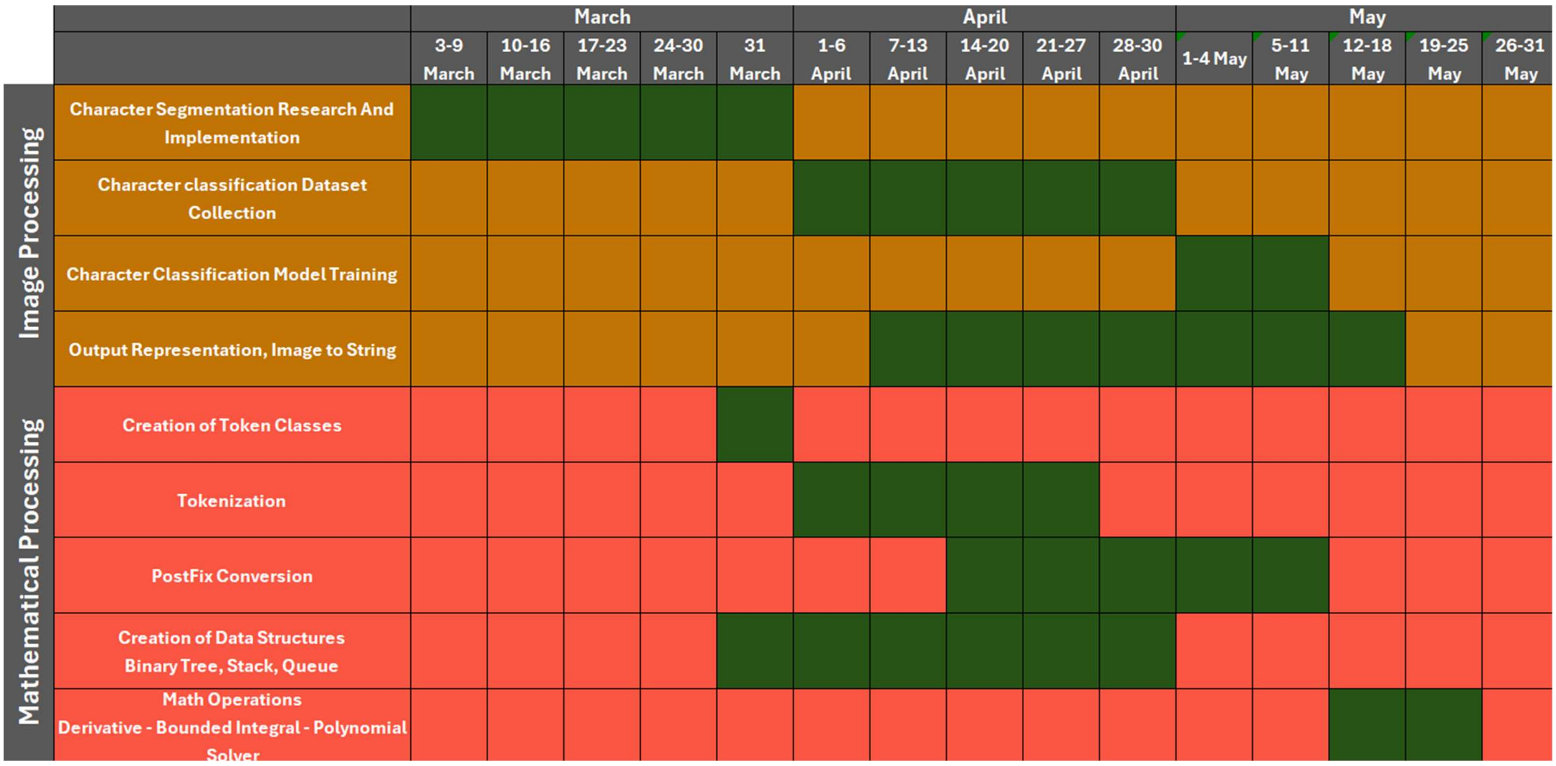
```
Testing expression: x + y * z
0th Token is Number Token: x (Constant: False)
1th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
2th Token is Number Token: y (Constant: False)
3th Token is Operator Token: * (Type: OperatorType.LEFT, Precedence: 3)
4th Token is Number Token: z (Constant: False)

Testing expression: √(16) + 4
0th Token is Operator Token: √ (Type: OperatorType.PREFIX, Precedence: 8)
1th Token is Operator Token: ( (Type: OperatorType.PREFIX, Precedence: 0)
2th Token is Number Token: 16 (Constant: True)
3th Token is Operator Token: ) (Type: OperatorType.SUFFIX, Precedence: 0)
4th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
5th Token is Number Token: 4 (Constant: True)

Testing expression: 5! + 3
0th Token is Number Token: 5 (Constant: True)
1th Token is Operator Token: ! (Type: OperatorType.SUFFIX, Precedence: 6)
2th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
3th Token is Number Token: 3 (Constant: True)

Testing expression: 50% of 200
0th Token is Number Token: 50 (Constant: True)
1th Token is Operator Token: % (Type: OperatorType.SUFFIX, Precedence: 7)
2th Token is Number Token: f (Constant: False)
3th Token is Number Token: 200 (Constant: True)

Testing expression: 2^3 + 4^2
0th Token is Number Token: 2 (Constant: True)
1th Token is Operator Token: ^ (Type: OperatorType.RIGHT, Precedence: 5)
2th Token is Number Token: 3 (Constant: True)
3th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
2th Token is Number Token: 3 (Constant: True)
2th Token is Number Token: 3 (Constant: True)
2th Token is Number Token: 3 (Constant: True)
3th Token is Operator Token: + (Type: OperatorType.LEFT, Precedence: 2)
4th Token is Number Token: 4 (Constant: True)
5th Token is Operator Token: ^ (Type: OperatorType.RIGHT, Precedence: 5)
6th Token is Number Token: 2 (Constant: True)
```

## III. PLANNED PROJECT TASKS

In this section the work which is expected to be done until the end of the project study should be written.

GANTT CHART

*Also, you* **must** *provide the work plan* by filling out the table below with itemized project activities and mark their duration on the timeline to the right by shading the corresponding weeks. Add rows as needed.*



| | | March | | | | | April | | | | | May | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 3-9 March | 10-16 March | 17-23 March | 24-30 March | 31 March | 1-6 April | 7-13 April | 14-20 April | 21-27 April | 28-30 April | 1-4 May | 5-11 May | 12-18 May | 19-25 May | 26-31 May |
| Image Processing | Character Segmentation Research And Implementation | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | | | | | | | | | | |
| Image Processing | Character classification Dataset Collection | | | | | | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | | | | | |
| Image Processing | Character Classification Model Training | | | | | | | | | | | 🟩 | 🟩 | | | |
| Image Processing | Output Representation, Image to String | | | | | | | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | | |
| Mathematical Processing | Creation of Token Classes | | | | | 🟩 | | | | | | | | | | |
| Mathematical Processing | Tokenization | | | | | | 🟩 | 🟩 | 🟩 | 🟩 | | | | | | |
| Mathematical Processing | PostFix Conversion | | | | | | | | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | | | |
| Mathematical Processing | Creation of Data Structures Binary Tree, Stack, Queue | | | | | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | | | | | |
| Mathematical Processing | Math Operations Derivative - Bounded Integral - Polynomial Solver | | | | | | | | | | | | | 🟩 | 🟩 | |

*As of 25 April 2025, we are in the 9th week. Also, fill out the tasks that have been accomplished in the past weeks.

## REFERENCES

[1] I. Culjak, D. Abram, T. Pribanić, H. Dzapo, and M. Cifrek, “A brief introduction to OpenCV,” presented at the MIPRO 2012, May 21-25, 2012, Opatija, Croatia.

[2] Shi, B., Bai, X., & Yao, C. (2016). An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(11), 2298–2304.

[2] Shi, B., Bai, X., & Yao, C. (2016). An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(11), 2298–2304.

[3] Rani, S., & Kumar, A. (2021). Character segmentation techniques for handwritten text: A review. *International Journal of Computer Applications*, 183(34), 12–18.

[4] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444. https://doi.org/10.1038/nature14539
(Used in the "What is a CNN?" section.)

[5] Zhang, Z., Bengio, S., & LeCun, Y. (2017). Character-level convolutional networks for text classification. *Advances in Neural Information Processing Systems*, 29. https://papers.nips.cc/paper_files/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html
(Used in the “Input Size” and “CNN Architecture Design” sections.)

[6] Islam, M. N., Shuvo, S. B., Rahman, M. A., & Hossain, M. A. (2021). Handwritten character recognition using deep learning: A comprehensive review and benchmark evaluation. *Applied Sciences*, 11(10), 4614. https://doi.org/10.3390/app11104614

[7] “Parsing infix notation,” chris-j.co.uk. [Online]. Available: https://www.chris-j.co.uk/parsing.php