Second Assignment - B. Halefoglu and I. Kurutepe

CENG 301 - Analysis and Design of Algorithms - 2024

Group xx

Full Name	**Student ID**
Buse Berfin Halefoglu	210401020
İrem Kurutepe	210401059

Tutor: Prof. Dr. Doğan Aydın

İzmir, December 4, 2024

Contents

1 | Introduction

In modern manufacturing, laser cutting technology is pivotal for precise material processing. Advanced laser cutting machines utilize high-energy CO2 lasers, automatic feeding systems, and camera controls to execute intricate designs efficiently. The design and programming phases are crucial to minimize material waste and optimize layout utilization.
This project focuses on developing a simulation for a laser cutting layout optimization system, emphasizing computational geometry and algorithmic efficiency. Central to this system is the Convex Hull Algorithm, implemented via the divide-and-conquer approach, which defines the geometric boundaries of user-defined shapes. The system also enhances material usage by strategically positioning and rotating parts within a specified cutting area, thereby reducing waste and improving efficiency.

1.1 | Project motivation and objectives

- Creating a user-friendly interface for part selection and layout visualization.
- Implementing an efficient convex hull generation algorithm.
- Optimizing material utilization through advanced placement strategies.

1.2 | Description of the problem

Efficient material utilization is a critical challenge in industrial laser cutting processes. Traditional methods often result in significant material waste due to suboptimal part placement and layout design. Additionally, existing systems may lack user-friendly interfaces for precise part selection and configuration. This project addresses these challenges by implementing a simulation that optimizes cutting layouts using computational geometry techniques, specifically the convex hull algorithm, to minimize waste and improve efficiency.

1.3 | Scope and Purpose:

The primary purpose of this project is to develop an interactive tool for **laser cutting layout optimization**. The tool allows users to select points interactively, generates a convex shape (Convex Hull) based on these points, and visualizes the layout within a defined cutting area. This approach minimizes material waste and enhances the efficiency of the cutting process.

2 | Background and Theoretical Framework

2.1 | Convex Hull Algorithm (Divide-and-Conquer Method)

2.1.1 | Explanation

The **Convex Hull** is the smallest convex polygon that can completely enclose a set of points. It is analogous to stretching a rubber band around the outermost points and letting it snap back to form a boundary. The concept of Convex Hull is widely used in computational geometry, such as shape analysis, image processing, and optimization problems like laser cutting.[1]

2.1.2 | Steps of the Convex Hull Algorithm

- **Concept of Convexity:** A polygon is considered *convex* if, for any two points inside the polygon, the line segment connecting them lies entirely within the polygon.[2] The Convex Hull algorithm aims to find the minimal convex boundary that encloses all the given points.
- **Input Data and Preprocessing:**
  - **Input:** A set of 2D points (x, y).
  - **Assumptions:**
    - The number of points is finite.
    - No duplicate points with identical coordinates exist.
- **Convex Hull Computation:** The Convex Hull can be calculated using various algorithms:
  - Graham's Scan Algorithm
  - Jarvis March (Gift Wrapping)
  - Quickhull
  - Divide and Conquer Algorithm

  In this implementation, the `ConvexHull` function from the `SciPy` library is used, which internally employs the Quickhull algorithm.

2.1.3 | Working Principle of the Quickhull Algorithm

- **Base Case:** Identify the two extreme points with the minimum and maximum x-coordinates. These two points form the initial line segment of the Convex Hull.
- **Partitioning:** Divide the remaining points into two subsets based on their position relative to the initial line segment: points above and below the line.
- **Recursive Construction::** For each subset, find the farthest point from the current hull segment. This point, along with the segment endpoints, forms a triangle, and its edges become new segments for further processing.
- **Pruning:**Exclude points inside the triangle. Repeat this process recursively for the new segments until no points are left outside the constructed hull.
- **Termination:** Combine the hull points from all recursive steps to form the final Convex Hull.

2.2 | Placement Algorithm

The convex hull shape is placed iteratively across the defined cutting area (1000x200 grid in this example) without overlap.

2.2.1 | Steps

1. **Grid Initialization:**
   - The cutting area is defined with x and y limits (x_limit=1000, y_limit=200).
   - The starting position for placement is initialized at the origin (0, 0).
2. **Iterative Placement:**
   - The algorithm iterates along the x-axis in steps of the shape's width (x_step).
   - The convex hull points are translated (offset) by the current (i, j) position.
   - A check ensures that the translated shape does not exceed the cutting area's boundaries.
   - If valid, the shape is drawn on the cutting area, and its area is added to the total utilized area.

2.2.2 | Nesting Algorithm

- **Objective:** To fit a specific shape as efficiently as possible within a confined container.
- **Usage:** The shape is rotated to different angles and placed in the container area, testing various positions to find the most optimal arrangement.
- **Key Strategy:**
  - The shape is rotated at predefined angles (e.g., 0°, 90°, 180°, 270°) to explore placement possibilities.
  - A grid-based placement approach is used, where the algorithm iterates through potential positions at regular grid intervals (each x, y coordinate is tested).

2.2.3 | Greedy Approach

- **Objective:** To make the best placement decisions quickly and locally.
- **Usage:** When a rotated shape fits within the container and does not overlap with previously placed shapes, it is immediately placed. Other potential arrangements are not explored further.
- **Advantages:**
  - The algorithm is fast and computationally efficient, making quick decisions without exhaustive searches.
- **Disadvantages:**
  - It sacrifices global optimization, as local decisions may not contribute to the overall best solution. This can lead to suboptimal space utilization in the long run.

2.3 | Function Requirements

- **Required Libraries:**
  - `matplotlib.pyplot`: Used for creating plots and visualizing the cutting layout.
  - `scipy.spatial.ConvexHull`: Used for computing the convex hull of a set of points selected by the user.
  - `numpy`: Facilitates mathematical operations and array handling.
  - `tkinter`: Provides the graphical user interface (GUI) for user interaction.
  - `shapely.geometry.Polygon`:
    - Creates polygons from a set of points.
    - Calculates areas and other geometric properties of the shapes.
  - `shapely.affinity.rotate and shapely.affinity.translate`:
    - **rotate:** Rotates shapes around their centroid or a specified origin, using angles in degrees.
    - **translate:** Translates shapes by specified offsets along the x and y axes.

    - These transformations enable dynamic shape adjustments within the cutting area and help optimize material utilization.[3]

- **Point Selection:**
  - Allows the user to interactively select points on a 50x50 pixel grid.
  - Captures the selected points as a NumPy array for subsequent operations.
- **Shape Operations:**
  - **`create_shapely_polygon(points)`**:
    - Converts a set of points into a `Shapely` polygon.
    - Ensures compatibility with geometric transformations and calculations.
  - **`create_rotated_shapes(shape, angles)`**:
    - Generates rotated versions of a polygon for a list of specified angles (e.g., 0°, 90°, 180°, 270°).
    - Returns a dictionary of rotated shapes for efficient placement.
  - **`can_place(candidate, placed_shapes, bounds)`**:
    - Validates whether a candidate shape can be placed within the defined container.
    - Ensures that the shape does not overlap with already placed shapes or exceed the container boundaries.
  - **`place_shapes(container, shape, angles, grid_step)`**:
    - Iteratively places rotated versions of a shape within a container grid.
    - Ensures efficient utilization of the available cutting area by avoiding overlaps.
- **Cutting Area Visualization:**
  - Computes the convex hull based on user-selected points.
  - Dynamically places rotated shapes within the cutting area using a grid-based approach.
  - Displays the layout with filled shapes and calculates the cutting efficiency as a percentage.
- **Tkinter GUI:**
  - Provides a user-friendly interface with buttons for key actions.
  - Includes ”Select Points” and ”Show Cutting Area” functionalities.
  - Ensures smooth interaction by validating inputs and displaying error messages for insufficient points.
- **Program Workflow:**
  - Initializes the GUI for user interaction.
  - Links point selection and cutting area visualization functions.
  - Displays real-time results and supports dynamic shape placement and rotation.

Function	**Purpose**
`select_points_on_plot`	Interactive point selection for defining the initial shape.
`create_shapely_polygon`	Converts points into a Shapely polygon for geometric operations.
`create_rotated_shapes`	Generates rotated versions of a polygon for better placement flexibility.
`can_place`	Validates if a shape can be placed within the container.
`place_shapes`	Places as many rotated instances of a shape as possible in the container.
`draw_full_convex_hull`	Visualizes the cutting layout and calculates area utilization efficiency.
`start_gui`	Provides a user-friendly GUI for point selection and layout visualization.

Table 2.1: List of functions and their purposes.

3 | Development Concepts

3.1 | Libraries in Python

At the beginning of the code, the following libraries are needed:

3.1.1 | matplotlib.pyplot

Used to create plots and allow the user to select points visually.

3.1.2 | scipy.spatial.ConvexHull

Calculates the smallest convex polygon (Convex Hull) around a set of points.

3.1.3 | numpy

For mathematical operations and handling arrays.[4]

3.1.4 | tkinter

For creating a graphical user interface (GUI).

3.1.5 | shapely.geometry.Polygon

Used to calculate areas of geometric shapes.

3.2 | Methodology

3.2.1 | Programming Tools and Environment

- **Selected Programming Language:** The program was developed using Python, a versatile and powerful high-level programming language. Python is well-suited for data processing, mathematical computations, and graphical visualization, making it an excellent choice for this project. In addition to its simple syntax and vast ecosystem of libraries, Python provides the necessary tools to implement computational geometry and graphical user interfaces efficiently.
- **Integrated Development Environment (IDE):** The development was carried out using Visual Studio, a robust IDE that supports Python development through extensions like the Python workload. Visual Studio offers features such as intelligent code completion, debugging, and integration with version control systems, which greatly enhanced the development workflow. The IDE's ability to manage both GUI-based and computational code, along with its debugging tools, made it easier to iterate on the program and troubleshoot any issues.

3.2.2 | Implementation Steps

The following workflow diagram illustrates the step-by-step process implemented in the project. Starting with user input, the system computes the convex hull, applies the placement algorithm, visualizes the cutting layout, and calculates the efficiency of the solution. This diagram provides an overview of the system's architecture and functionality.

Key Steps:

- **Convex Hull Creation:** `ConvexHull` calculates the convex polygon around the given points.
- **Area Calculation:** The area of the convex shape is calculated using `shapely.geometry.Polygon`.
- **Shape Rotation:** Rotated versions of the shape are generated for specified angles (e.g., 0°, 90°, 180°, 270°) using the `create_rotated_shapes` function to maximize layout flexibility.
- **Shape Translation and Placement:** The `place_shapes` function iteratively translates and places the rotated shapes within the cutting area while avoiding overlaps and ensuring they remain within boundaries.
- **Efficiency Calculation:** The efficiency of the cutting area is calculated as the ratio of the utilized area to the total area.
- **Visualization:** The shapes are displayed in a plot, and the cutting area efficiency is shown as text.

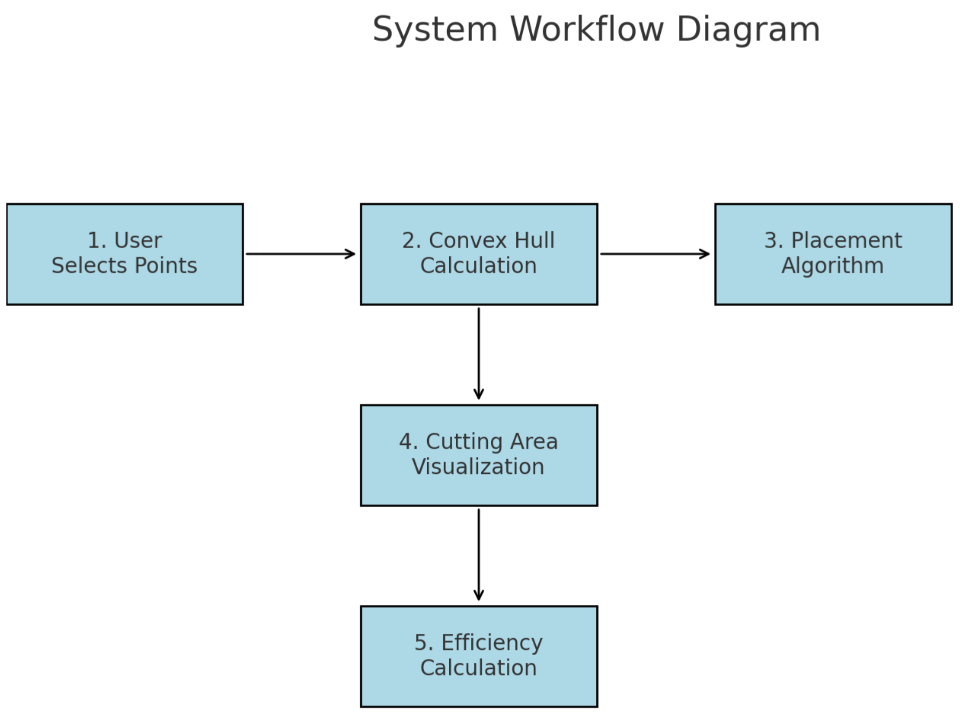
System Workflow Diagram
1. User Selects Points
2. Convex Hull Calculation
3. Placement Algorithm
4. Cutting Area Visualization
5. Efficiency Calculation

Figure 3.1: System Workflow Diagram

3.2.3 | Error Handling and Debugging Process

Error Handling and Debugging Process highlights the mechanisms implemented to handle potential errors during runtime. For example, if a user selects fewer than three points, the system provides an appropriate warning and guides the user to select more points. Additionally, debugging practices were applied during the development phase to resolve issues such as overlapping shapes or performance bottlenecks.

3.2.4 | User Interface Design

This function provides a user-friendly interface.

The GUI includes two buttons:

- **Select Points:** Opens a plot window where the user can select points.
- **Show Cutting Area:** Displays the cutting area filled with repeated convex shapes based on the selected points and calculates the utilization efficiency.

3.2.5 | Code Modularity

The codebase was designed with modularity in mind. Each function serves a specific purpose, making the system easy to maintain and extend. For instance, the point selection process, Convex Hull generation, and cutting layout visualization are implemented as independent modules that can be reused in other projects or enhanced without impacting the core functionality.

4 | Results and Analysis

4.1 | Implementation

- Graphical representation of the cutting area

4.1.1 | Implemented Features:

The following features have been successfully implemented and are functional in the current version of the system:

- **Convex Hull Generation:** The convex hull algorithm has been implemented using the divide-and-conquer approach, enabling the efficient generation of geometric boundaries for user-defined shapes.[7]
- **Interactive Point Selection:** A user-friendly interface allows users to select points interactively on a 50x50 grid, which is then used to construct the convex hull of the selected shape.
- **Layout Visualization:** The system provides a clear graphical representation of the cutting layout, showing the placement of multiple shapes within a 200x1000 cutting area.
- **Advanced Placement Optimization with Rotations:** The placement algorithm has been enhanced to support rotations of shapes at angles of 0°, 90°, 180°, and 270°. This capability allows for better utilization of the cutting area by ensuring that shapes are optimally aligned to fit within the defined boundaries and avoid overlaps.
- **Tkinter-Based GUI:** A graphical user interface (GUI) is implemented using Tkinter, offering users easy access to key functionalities, such as point selection, layout visualization, and dynamic placement feedback.

4.1.2 | Non-Implemented Features:

The following features are planned but have not yet been implemented due to the limitations encountered during development:

- **Advanced Shape Optimization:** The current implementation does not effectively optimize the placement of irregular shapes, within the cutting area. This limitation reduces the potential efficiency of material usage, especially for shapes with non-convex boundaries.
- **Export Functionality:** A feature to export the cutting layout as an image or file is planned for future iterations to allow for practical usage beyond visualization.
- **User Customization:** Providing options for users to dynamically adjust parameters, such as shape colors, rotation angles, or layout configurations, is another planned improvement to make the tool more ergonomic and adaptable.
- **Multi-threading for Large-Scale Layouts:** To enhance performance, especially for large-scale layouts with a significant number of shapes, multi-threading could be implemented to parallelize computations and reduce processing time.

4.2 | Visualization of Cutting Layout

4.3 | Efficiency Indicators

The efficiency of the laser cutting layout optimization is evaluated based on the material utilization rate. This metric quantifies how effectively the available cutting area is utilized by the placed shapes. The calculations and comparisons presented here demonstrate the algorithm's ability to maximize utilization under different scenarios.

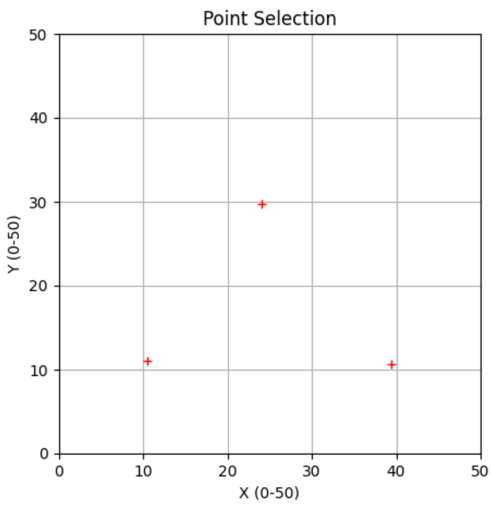

Figure 4.1: Figure 1

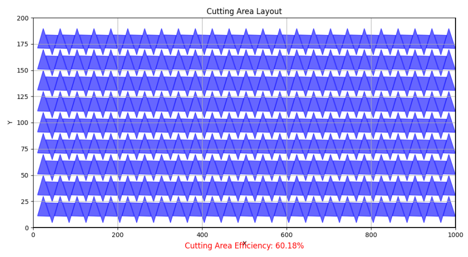

Figure 4.2: Figure 2

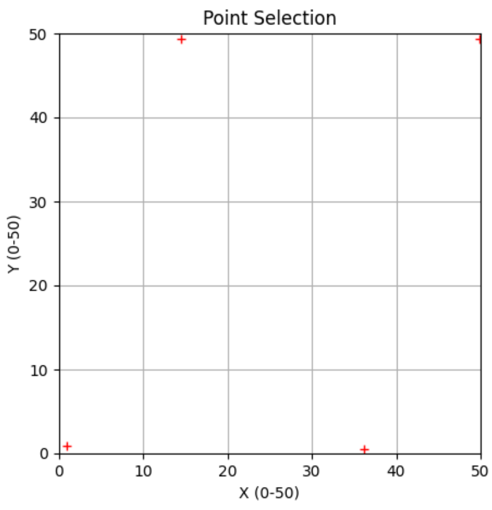

Figure 4.3: Figure 3

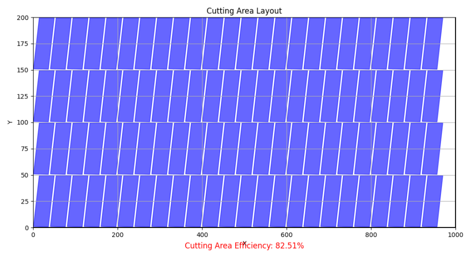

Figure 4.4: Figure 1k

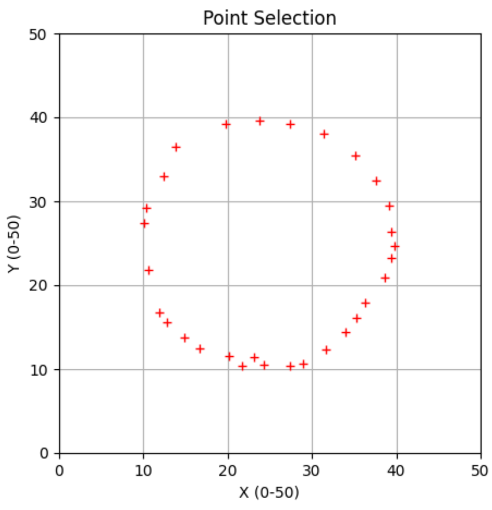

Figure 4.5: Figure 2k

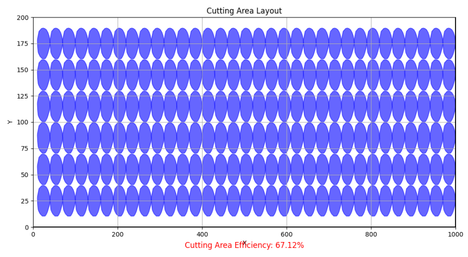

Figure 4.6: Figure 3k

Calculations for Material Utilization

Material utilization is calculated using the following formula:

$$\text{Utilization Percentage} = \left(\frac{\text{Total Shape Area}}{\text{Total Cutting Area}}\right) \times 100$$

- **Total Shape Area:** The sum of the areas of all Convex Hull shapes placed in the cutting area.
- **Total Cutting Area:** The product of the cutting area's width (x_{limit}) and height (y_{limit}).

In the implemented code, the area of each Convex Hull shape is computed using the `shapely.geometry.Polygon` library. The iterative placement process ensures that overlapping or out-of-bound shapes are avoided, contributing to the efficiency of material usage.

4.4 | Validation and Performance Evaluation

The analysis reveals that while the Convex Hull algorithm performs efficiently with randomly distributed points, its effectiveness decreases when points are arranged in specific geometric patterns, such as equilateral triangles or rhombuses. These patterns often lead to inefficient utilization of the cutting area, as the algorithm struggles to optimize the layout due to overlapping shapes or unutilized spaces. Furthermore, the system may become less functional in such cases, highlighting the need for enhanced algorithms capable of handling irregular point distributions and improving the overall efficiency and adaptability of the tool.

4.5 | Time Complexity Analysis

This section provides a detailed time complexity analysis of the implemented functions in the laser cutting layout optimization system.[8]

- **Computational analysis of implemented algorithms.**

Table 4.1: Time Complexity Analysis

Component	Best Case	Worst Case	Average Case
Convex Hull	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Placement	$O(1)$	$O(m \cdot v)$	$O(m \cdot v)$
Total	$O(n \log n)$	$O(n^2 + m \cdot v)$	$O(n \log n + m \cdot v)$

- n: Total number of selected points.
- m: Total number of placement steps.
- v: Number of vertices on the Convex Hull.

4.5.1 | select_points_on_plot Function

Purpose: This function allows the user to select points interactively via mouse clicks on a 50x50 grid.
Time Complexity:

- User interaction time is not deterministic, and the algorithmic complexity is dependent on the number of selected points.
- For n selected points, the storage and processing time is $O(n)$.

4.5.2 | draw_full_convex_hull Function

The complexity of this function is based on several subcomponents:

Convex Hull Calculation

- **Algorithm Used:** `scipy.spatial.ConvexHull`, which uses the Quickhull algorithm.
- **Time Complexity:** $O(n \log n)$, where n is the number of selected points.

Polygon Area Calculation

- **Library Used:** `shapely.geometry.Polygon`.
- **Time Complexity:** $O(m)$, where m is the number of vertices of the polygon.

Shape Placement and Loops This involves filling the cutting area by replicating the Convex Hull shape:

- **Outer Loop (X-axis):** Iterates through the x_limit in steps of x_step. Total iterations: $\frac{x\_limit}{x\_step}$.
- **Inner Loop (Y-axis):** Iterates through the y_limit in steps of y_step. Total iterations: $\frac{y\_limit}{y\_step}$.
- For each iteration, area calculation has a complexity of $O(m)$.
- **Total Complexity:**

$$O\left(\frac{x\_limit \cdot y\_limit}{x\_step \cdot y\_step} \cdot m\right)$$

4.5.3 | start_gui Function

Purpose: Initializes the Tkinter GUI and manages user interactions. **Time Complexity:** Event-driven and dependent on user inputs, thus not algorithmically measurable. Function calls depend on the complexities outlined above.

4.5.4 | Overall Complexity

The overall complexity of the system combines the Convex Hull calculation and shape placement:

$$O(n \log n) + O\left(\frac{x\_limit \cdot y\_limit}{x\_step \cdot y\_step} \cdot m\right)$$

In the worst case:

$$O(n^2 + m \cdot v)$$

4.6 | Efficiency Considerations

- Smaller step sizes (x_step, y_step) improve precision but increase computational overhead.
- Optimizing placement algorithms and leveraging parallel computation can enhance efficiency.[9]

4.7 | Advantages of the Proposed Solution

- **Improved Material Utilization:** The proposed solution minimizes material waste by optimizing the placement of shapes in the cutting area.
- **Algorithmic Efficiency:** The divide-and-conquer implementation of the convex hull algorithm ensures computational efficiency, making it suitable for real-time applications.
- **User-Friendly Interface:** The GUI provides an intuitive and interactive environment for point selection and layout visualization.

4.8 | Limitations and Challenges

- **Lack of Advanced Shape Optimization:** The system struggles with optimal placement of irregular shapes, leading to potential inefficiencies.
- **Performance Constraints:** For large-scale layouts, the system experiences performance bottlenecks due to its single-threaded implementation.

4.9 | Upcoming Developments

- **Advanced Rotation Support:** Incorporating shape rotations at various angles to enhance material utilization.
- **Export Functionality:** Adding features to export layouts as images or files for practical use.
- **Multi-Threading and Performance Optimization:** Implementing multi-threading to improve the system's scalability and performance for larger datasets.
- **Enhanced User Customization:** Allowing users to adjust parameters dynamically, such as layout configurations and shape properties, to make the tool more adaptable.

5 | Conclusion

The development and implementation of the cutting layout optimization system provided valuable insights into computational geometry and GUI design using Python. The project successfully achieved several key milestones, including:

- Interactive convex hull generation.
- Efficient material utilization visualization.
- A user-friendly interface.
- Integration of rotation capabilities to improve material utilization.

The proposed solution demonstrated its effectiveness in improving material usage through optimal placement of convex hull shapes within defined boundaries. The addition of rotation support for predefined angles (e.g., 0°, 90°, 180°, 270°) further enhanced the flexibility of shape placement and led to better utilization of the cutting area. While the implemented system achieved a significant degree of functionality, there are still areas for enhancement, including:

- Lack of advanced shape optimization for irregular shapes.
- Limited support for arbitrary rotation angles.
- Performance scalability issues for larger datasets.

Future work aims to overcome these limitations by:

- Developing advanced algorithms for irregular shape placement.
- Supporting arbitrary rotation angles to maximize efficiency.
- Adopting multi-threading to handle larger datasets and improve performance.
- Incorporating export functionalities for practical usage.
- Providing enhanced customization options, such as adjustable layout parameters.

Overall, this project represents a strong foundation for solving real-world optimization problems in material cutting, with potential applications in various industries such as manufacturing and design. The integration of rotation capabilities marks a significant step towards better efficiency, and the insights and outcomes of this work can serve as a stepping stone for further innovation and development.[10]

6 | References

[1] GeeksforGeeks - Convex Hull Algorithm

[2] GeeksforGeeks - Divide and Conquer Convex Hull Algorithm

[3] KU Leuven - Convex Hull Research

[4] Stack Overflow - Cutting Patterns Algorithm

[5] Curve Optimization Problems

[6] GeeksforGeeks - Convex Hull Using Jarvis Algorithm

[7] Illinois - Convex Hull Lecture Notes

[8] Convex Hulls - Comparing Runtimes

[9] Jeff Erickson's Computational Geometry Notes

[10] ChatGPT - OpenAI Conversational Assistant

A | Laser Cutting Optimization Code

This appendix provides a breakdown of the Python code used for laser cutting optimization. The code is divided into key sections for clarity.

A.1 | Importing Required Libraries

The first step is to import the necessary libraries for the program. These include tools for visualization, geometry manipulation, and user interaction.

Listing 1: Importing Required Libraries

```
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull
import numpy as np
import tkinter as tk
from tkinter import messagebox
from shapely.geometry import Polygon
from shapely.affinity import rotate, translate
```

A.2 | Point Selection Function

This function allows the user to interactively select points on a plot. The points are then used to calculate the layout optimization.

Listing 2: Point Selection Function

```
def select_points_on_plot():
    """
    Allows the user to select points on a plot.
    """
    print("Click␣on␣points␣in␣the␣plot.␣Right-click␣or␣press␣'Enter'␣to␣finish."
        )
    fig, ax = plt.subplots(figsize=(5, 5))
    plt.title("Point␣Selection")
    plt.xlabel("X␣(0-50)")
    plt.ylabel("Y␣(0-50)")
    plt.grid(True)
    plt.xlim(0, 50)
    plt.ylim(0, 50)

    # User selects points
    points = plt.ginput(n=-1, timeout=0)
    plt.close()
    return np.array(points)
```

A.3 | Shape Operations

This section contains functions to create shapes, rotate them, and check placement conditions to avoid overlaps or boundary violations.

Listing 3: Shape Operations

```
def create_shapely_polygon(points):
    """
    Create a Shapely polygon from given points.
    """
    return Polygon(points)

def create_rotated_shapes(shape, angles):
    """
    Create rotated versions of the given shape for each angle in the list.
    """
```

```python
    rotated_shapes = {}
    for angle in angles:
        rotated_shapes[angle] = rotate(shape, angle, origin='centroid',
            use_radians=False)
    return rotated_shapes

def can_place(candidate, placed_shapes, bounds):
    """
    Check if the candidate shape can be placed without overlapping or going out
        of bounds.
    """
    if not bounds.contains(candidate):
        return False
    for existing_shape in placed_shapes:
        if candidate.intersects(existing_shape):
            return False
    return True
```

A.4 | Displaying the Cutting Area

This function visualizes the placement of shapes in the cutting area. It also calculates the efficiency of material usage.

Listing 4: Displaying the Cutting Area

```python
def draw_full_convex_hull(points, x_limit, y_limit):
    """
    Fills the cutting area with rotated Convex Hull shapes using the selected
        points.
    """
    if len(points) < 3:
        messagebox.showerror("Error", "At␣least␣3␣points␣must␣be␣selected!")
        return

    # Calculate the Convex Hull
    convex_hull = ConvexHull(points)
    hull_points = points[convex_hull.vertices]

    # Create the original shape
    original_shape = create_shapely_polygon(hull_points)

    # Define the container area
    container = Polygon([(0, 0), (x_limit, 0), (x_limit, y_limit), (0, y_limit)
        ])

    # Place shapes within the container
    angles = [0, 90, 180, 270]  # Allowed rotation angles
    placed_shapes = place_shapes(container, original_shape, angles, grid_step
        =10)

    # Visualization
    plt.figure(figsize=(12, 6))
    plt.xlim(0, x_limit)
    plt.ylim(0, y_limit)
    plt.grid(True)
    plt.title("Cutting␣Area␣Layout")
    plt.xlabel("X")
    plt.ylabel("Y")

    # Draw the container
    plt.plot(*container.exterior.xy, color="black")

    # Draw placed shapes
    for shape in placed_shapes:
        x, y = shape.exterior.xy
        plt.fill(x, y, color="blue", alpha=0.6)

    # Efficiency calculation
    total_shape_area = sum(shape.area for shape in placed_shapes)
    total_cutting_area = container.area
    utilization_percentage = (total_shape_area / total_cutting_area) * 100
    plt.text(x_limit / 2, -20,
             f"Cutting Area Efficiency: {utilization_percentage:.2f}%",
             fontsize=12, ha="center", color="red")
    plt.show()
```

A.5 | GUI Operations

The program provides a user-friendly GUI to interact with the cutting optimization system.

Listing 5: GUI Operations

```python
def start_gui():
    """
    Starts the Tkinter GUI and binds operations.
    """
    def on_select_points():
        global points
        points = select_points_on_plot()
        if len(points) < 3:
            points = None
            messagebox.showwarning("Warning", "At least 3 points must be 
                selected!")
        else:
            messagebox.showinfo("Success", "Points have been successfully 
                selected!")

    def on_display_cutting_area():
        if points is None or len(points) < 3:
            messagebox.showerror("Error", "Please select points first!")
            return
        draw_full_convex_hull(points, x_limit=1000, y_limit=200)
    root = tk.Tk()
    root.title("Laser Cutting Layout Optimization")
    root.geometry("400x200")
    # Buttons
    select_points_button = tk.Button(root, text="Select Points", command=
       on_select_points, height=2, width=20)
    select_points_button.pack(pady=10)

    display_cutting_button = tk.Button(root, text="Show Cutting Area", command=
       on_display_cutting_area, height=2, width=20)
    display_cutting_button.pack(pady=10)

    root.mainloop()
```

A.6 | Main Program

Finally, the main part of the code initializes the program.

Listing 6: Main Program

```python
if __name__ == "__main__":
    points = None  # Global variable
    start_gui()
```