ChatGPT

3D Guillotine Cutting Algorithm Implementation

Overview of Guillotine Cutting

A **guillotine cut** is an orthogonal cut that goes from one side of a material block to the opposite side, always parallel to one of the block's faces. In a 3D guillotine cutting problem, we start with large cuboid stock blocks (also called *bins*) and cut them into smaller rectangular box-shaped parts (items) using only full-edge-to-edge straight cuts along X, Y, or Z directions. This constraint means we cannot make arbitrary interior cuts: every cut splits a block completely into two sub-blocks. The goal is to fulfill the demand for certain parts by cutting the stocks, while maximizing material usage (minimizing waste).

Cutting stock problems of this nature are NP-hard optimization problems [1], so an optimal solution is computationally difficult. We will implement a **heuristic algorithm** that fills each stock block with parts in a reasonable way. The approach uses a *first-fit decreasing* style strategy (commonly used in bin packing) combined with priority sorting [2], ensuring that high-priority parts are placed first. We allow parts to be rotated in any orientation (orthogonal rotations) – i.e. a part's (X, Y, Z) dimensions can be permuted in any of the six possible ways to try to fit into a block [3].

Each cut will account for a given **kerf (blade thickness)**, meaning the material lost to the cut is subtracted so that subsequent pieces are slightly smaller by that thickness. The algorithm will produce a sequential plan of cuts and the resulting pieces. We also construct a tree (or table) of cutting steps, where each row represents a piece produced by a cut (either a final part or a leftover piece), along with the cut information and parent-child relations to trace the cutting sequence.

Input Data Structures

Our implementation uses the following input structures:

- **Part List:** A list of part types, each with:
- Dimensions `(X, Y, Z)` – the required size of the part.
- Quantity – how many such parts are needed.
- Priority – `1` for high-priority parts (to cut first), or `0` for normal priority.
- **Stock Block List:** A list of available stock blocks, each with:
- Dimensions `(X, Y, Z)` – the size of the raw block.
- Kerf thickness – the saw cut thickness (material lost per cut).
- Material grade (steel grade) – (string or identifier, not used in cutting logic but retained for reference).

We will represent parts and blocks as Python classes for clarity. This makes the code modular and easier to integrate into a larger system or web interface.

Algorithm Design

Orientation Handling and Fit Checking

For each part type, before cutting, we consider all 6 possible orientations of its dimensions (since rotating the part can swap its length, width, height) [3]. A part can fit into a given block if *any* orientation of its `(x, y, z)` is less than or equal to the block's `(X, Y, Z)` dimensions (accounting for kerf allowances). For example, a part $20 \times 10 \times 5$ can fit in a block $10 \times 5 \times 20$ by appropriate rotation.

We implement a function to check if a part fits into a block and to determine the **best orientation** to use. The heuristic for choosing orientation is to minimize the number of cuts needed and the leftover volume: - We prefer orientations where the part uses the full length of the block in one or more dimensions, reducing the number of cuts and leftover pieces in those directions. Each dimension in which the part matches the block size means one less cut (and no leftover in that direction). - If multiple orientations are possible, we can compare the total leftover volume or number of cuts required. We choose the orientation that yields the smallest leftover volume (or equivalently, maximizes the part volume utilization in that block).

This orientation selection ensures efficient use of space and fewer waste pieces from each cut.

Priority-Based Part Selection

To satisfy part priorities, we first cut all **priority-1** parts before any priority-0 parts [2]. We sort the parts such that high-priority items are considered first. Within the same priority, we can further sort by volume (largest first) or another heuristic, so larger parts get placed earlier (this tends to reduce fragmentation and waste). This sorting by priority (and size) is a common approach in cutting stock problems [2]. The algorithm will iterate through parts in this order when looking for the next part to cut.

During the cutting process, when selecting a part to cut from a particular block, we scan the sorted part list for the first part (highest priority, then largest) that still has remaining quantity and can fit in the current block (in some orientation). If no remaining part fits, that block can no longer be used for any part and becomes a leftover scrap piece.

Sequential Guillotine Cutting Strategy

The core of the algorithm uses a **recursive, depth-first cutting strategy** similar to the stack-based approach used in 2D guillotine algorithms [4]. The process for each stock block is as follows:

1. **Find a Part to Cut:** If there is a required part that fits in the current block, choose the highest-priority fitting part. If no part fits (or no demand remains), we mark the entire block as unused leftover (waste).

2. **Cut to Isolate the Part:** We perform a sequence of up to three cuts (along X, Y, and Z in some order) to extract the chosen part:

3. **First cut:** Split the block along one axis to separate a slab that will contain the part. We typically cut at a coordinate equal to the part's size in that axis (or the leftover size), accounting for kerf. This yields two pieces: one **sub-block** that will contain the part, and one **remainder** piece (the

leftover offcut) from the block. We set aside the remainder piece for later cutting (push it onto a stack or recursive call later) [4] , and continue working with the sub-block that contains the part.

4. **Second cut:** From the sub-block (which now has the part's size in one dimension), cut along the second axis at the part's size in that direction. Again, this produces a smaller sub-block containing the part, and a leftover piece. We set aside the new leftover and continue with the sub-block containing the part.
5. **Third cut:** Cut the final sub-block along the last axis (part's thickness) to separate the actual part. This yields the final **Part** piece and another small leftover piece.

Each cut is a full guillotine cut from one face to the opposite face of the current block. We subtract the kerf thickness from the remaining block dimension each time, so the leftover piece dimensions are reduced by the cut thickness. The sequence of cuts essentially carves the part out of one corner of the block. We always plan the cuts such that one side of the cut is exactly the part (or a slab leading to the part), ensuring the part is freed after the final cut.

1. **Record the Cut Step:** For each cut step, we record:
2. Step number (sequentially increasing for each cut).
3. Cut direction (X, Y, or Z) and the coordinate at which the cut was made (distance from the block's origin face).
4. The resulting piece from that cut, labeled either as a **Part** (if it exactly matches a requested part) or a **Leftover** (if it is a remainder piece). We capture the dimensions of that resulting piece.
5. A reference to the **previous step** (parent) from which this piece came. This links the piece to the block that was cut to produce it, allowing reconstruction of the cut tree. (For the two pieces produced by a single cut, both will reference the same parent step.)

The cutting plan will be a list (or DataFrame) of these step records. Each record essentially represents a node in the cut tree (except the initial stock, which has no parent). By following the parent links, one can trace how each part or leftover was obtained.

1. **Recursive Cutting of Leftovers:** After extracting a part, we now have up to three leftover pieces from the sequence of cuts (one from each cut). Those pieces may still be large enough to yield other parts. We then **recursively** apply the same procedure to each leftover piece: take a leftover, find the next part that fits, cut it out, etc. This depth-first recursion naturally follows one branch of the cut tree completely (one sequence of cuts to isolate one part) before moving on to the next leftover piece [4] . In practice, this means we always continue cutting the most recently produced sub-block (the one containing the part) until a part is obtained, then backtrack to earlier leftovers. This approach mirrors the idea of pushing unused remainder pieces onto a stack and continuing with the current piece in 2D algorithms [4] .

2. **Termination:** This process continues until either all required parts have been cut, or no remaining parts can fit in any leftover block. If all part demands are met and there are still leftover pieces that haven't been used, those are considered waste (unused material) [5] . We then stop cutting further. All remaining leftover blocks are recorded as scrap.

Throughout this process, we ensure **priority parts are cut first** (the algorithm always looks for any priority-1 part that fits before considering priority-0 parts for each block). This guarantees that if there is limited space or material, the high-priority parts will be obtained first [2] .

We also integrate the **kerf thickness** into each cut. For example, if a block of length L is cut at position p along X, and kerf k, then the two resulting pieces have lengths p and $(L - p - k)`. The kerf material is lost in the process (and counted as waste volume).

Data Output and Utilization Calculation

The output includes:

- **Cut Steps Table:** A list of steps (or pandas DataFrame) where each row corresponds to a piece resulting from a cut. The columns could include:
- **Step #** – the sequential cut step number.
- **Cut (Dir@Coord)** – the direction and coordinate of the cut (e.g., "X@500" meaning a cut along the X-axis at X=500mm from the origin face).
- **Result Type** – either "Part" or "Leftover", indicating if the resulting piece is a final part or a remainder piece.
- **Piece Dimensions** – the size of that piece after the cut, in the format X×Y×Z.
- **Parent Step** – the step number of the cut from which this piece was produced (or blank/None for initial stock pieces).

Using this table, one can identify the final parts (those marked "Part") and trace back their lineage through parent steps to the original stock. The leftover pieces (marked "Leftover") that have no further children in the table are the final scrap blocks.

- **Leftover Blocks List:** A summary of all remaining **unused** pieces of material after all parts are cut. For each leftover block, we list its dimensions and volume. These are essentially the leaf nodes of the cut tree that are marked "Leftover" (and were not subsequently used to cut any part). This allows calculation of the total waste volume.

- **Utilization and Waste Percentage:** We compute the material utilization as the percentage of stock volume that ended up in final parts. Similarly, the waste percentage is the complement (including both leftover scrap and kerf losses). For example, if V_{stock} is the total volume of all stock blocks used and $\sum V_{\text{parts}}$ is the total volume of all produced parts, then:

- *Fill ratio* (utilization) $= \frac{\sum V_{\text{parts}}}{V_{\text{stock}}} \times 100\%$
- *Waste percentage* $= \frac{V_{\text{stock}} - \sum V_{\text{parts}}}{V_{\text{stock}}} \times 100\% $ [6] .

(All leftover block volumes plus kerf loss contribute to waste in this calculation.)

Any remaining stock pieces after fulfilling all demands are counted as waste by definition [5] (they might be reused for future orders, but for this computation we treat them as waste). The algorithm will output these percentages for analysis of cutting efficiency.

Modularity and Visualization Hooks

The implementation is organized into classes and functions to facilitate integration into other software or a web interface: - `Part` **class:** holds part dimensions, priority, and remaining quantity. - `Block` **class:** holds block dimensions, kerf, material grade, and possibly an identifier. - **Cutting functions (methods):** for example, a method to choose the next part for a block, a recursive function to cut a block, and helpers for orientation fitting and performing a cut.

We have included comments in the code (marked **"Visualization Hook"**) indicating where one could integrate visualization. For instance, after each cut, we could insert a call to a visualization function to render the current block subdivision or highlight the piece that was just cut (`# здесь можно` `отрисовать sub-block`). These hooks mark the points in the code where the state is ready to be visualized (e.g., right after a sub-block or part is obtained from a cut). The actual visualization (using matplotlib, plotly, or any 3D rendering library) is not implemented here, but the code is structured so that one can plug in those calls easily.

Next, we provide the Python implementation of this 3D guillotine cutting algorithm. The code is written in pure Python (using standard libraries and `numpy`/`pandas` where appropriate) with no CAD-specific dependencies, making it suitable for integration into various environments.

Code Implementation

Below is the complete Python implementation, including data classes, the cutting algorithm, and the results compilation. This code is well-commented for clarity and debugging, and uses a step-by-step approach to mirror the described algorithm. You can adapt or extend this code for use in a web application or further optimization as needed.

```
import pandas as pd
import numpy as np

# Data classes for Part and Block
class Part:
    def __init__(self, part_id, x, y, z, quantity, priority=0):
        self.id = part_id        # Identifier (e.g., name or number)
        self.x = x
        self.y = y
        self.z = z
        self.quantity = quantity  # Remaining quantity to cut
        self.priority = priority
        self.volume = x * y * z

class Block:
    def __init__(self, block_id, x, y, z, kerf=0.0, grade=None):
        self.id = block_id
# Identifier for the block (could be index or label)
        self.x = x
        self.y = y
        self.z = z
        self.kerf = kerf        # kerf thickness for this block
        self.grade = grade
        self.volume = x * y * z

class GuillotineCutter:
    def __init__(self, parts, stocks):
        """
        Initialize the cutter with lists of Part and Block objects.
        """
        # Sort parts: priority first, then by volume (descending)
        self.parts = sorted(parts, key=lambda p: (-p.priority, -p.volume))
        self.stocks = stocks
        # Prepare result structures
        self.steps = []  # list of dict records for each piece/cut result
        self.step_counter = 0
        # Calculate total stock volume for utilization
        self.total_stock_volume = sum(b.volume for b in stocks)
        # Keep track of total parts volume cut
        self.total_parts_volume = 0.0

    def find_part_for_block(self, block):
        """
        Find the next part that fits into the given block (according to priority and size).
        Returns a tuple (Part, orientation) or (None, None) if no part fits.
        """
        for part in self.parts:
            if part.quantity <= 0:
                continue  # skip parts that are already fulfilled
            # Check all orientations for fit
            # We'll represent orientation as a tuple (px, py, pz)
            orientations = [
                (part.x, part.y, part.z),
                (part.x, part.z, part.y),
                (part.y, part.x, part.z),
                (part.y, part.z, part.x),
                (part.z, part.x, part.y),
                (part.z, part.y, part.x)
            ]
            # We can filter unique orientations to avoid duplicates if any dims equal
            orientations = list({orient for orient in orientations})  # remove duplicates
            # Try each orientation
            best_orientation = None
            best_cuts = 4  # more than max cuts (3)
            best_leftover_vol = None
            for (px, py, pz) in orientations:
                if px <= block.x and py <= block.y and pz <= block.z:
                    # Part fits in this orientation
                    # Determine how many cuts needed and leftover volume
                    cuts = 0
                    # Copy block dims to simulate cutting
                    bx, by, bz = block.x, block.y, block.z
                    kerf = block.kerf
                    leftover_vol = block.volume - (px * py * pz)
                    # Cuts in X
                    if px < bx:
                        cuts += 1
                        # Simulate removing kerf volume for X cut
                        leftover_vol -= (by * bz * kerf)
                        bx = px  # now block width reduced to part width
                    if py < by:
                        cuts += 1
                        leftover_vol -= (bx * bz * kerf)
                        by = py  # block height reduced to part height
                    if pz < bz:
                        cuts += 1
                        leftover_vol -= (bx * by * kerf)
                        bz = pz  # block thickness reduced to part thickness
                    # Choose orientation with fewer cuts, and then lower leftover volume
                    if cuts < best_cuts or (cuts == best_cuts and
                                             (best_leftover_vol is None or leftover_vol < best_leftover_vol)):
                        best_cuts = cuts
                        best_leftover_vol = leftover_vol
                        best_orientation = (px, py, pz)
            if best_orientation:
                # Return this part and the chosen orientation (as tuple of dims)
                return part, best_orientation
        return None, None

    def cut_block(self, block, parent_step=None):
        """
        Recursively cut the given block into parts using guillotine cuts.
        If parent_step is provided, it indicates the step number of the cut that produced this block.
        """
        # Find a part that fits in this block (priority-first)
        part, (px, py, pz) = self.find_part_for_block(block)
        if part is None:
            # No part fits - this block remains as leftover waste
            # Record leftover block (no further cuts)
            # We only record if this block itself was produced by a cut (parent exists) or if it’s an original stock unused.
            # For initial stock with no part cut, we consider it leftover as well.
            if parent_step is not None or True:
                # Mark this block as a leftover piece (final scrap)
                self.steps.append({
                    "Step": None,  # no new cut performed to create it
                    "Cut": None,
                    "Result": "Leftover",
                    "Dimensions": f"{block.x}x{block.y}x{block.z}",
                    "Parent": parent_step
                })
            return  # stop cutting this block

        # If a part is found, we need to cut it out of the block
        # Dimensions of part in this orientation
        px, py, pz = px, py, pz  # oriented part dims (alias for clarity)
        kerf = block.kerf
        # *** First cut (along X-axis if needed) ***
        if px < block.x:
            # We will cut along X to isolate a slab of width = px for the
part
            # Determine cut coordinate from the origin face: we cut at px
(part width)
            cut_coord = px
            self.step_counter += 1
            step_num = self.step_counter
            # Perform the cut:
            left_piece_dims = (px, block.y, block.z)
# left slab containing the part (width px)
            right_piece_dims = (block.x - px - kerf, block.y, block.z)  #
right remainder
            # Create Block objects for the two new pieces
            slab_block = Block(block.id + "_L", *left_piece_dims, kerf=kerf,
grade=block.grade)
            remainder_block = Block(block.id + "_R", *right_piece_dims,
kerf=kerf, grade=block.grade)
            # Log the cut and results
            cut_info = f"X@{cut_coord}"
            # Record the remainder piece as a leftover (result of this cut)
            self.steps.append({
                "Step": step_num,
                "Cut": cut_info,
                "Result": "Leftover",
                "Dimensions": f"{remainder_block.x}x{remainder_block.y}
x{remainder_block.z}",
                "Parent": parent_step if parent_step is not None else
"Initial"
            })
            # Record the slab piece (this will be further cut, so treat as
intermediate leftover)
            self.steps.append({
                "Step": step_num,
                "Cut": cut_info,
                "Result": "Leftover",
                "Dimensions": f"{slab_block.x}x{slab_block.y}
x{slab_block.z}",
                "Parent": parent_step if parent_step is not None else
"Initial"
            })
            # Visualization hook: after first cut, slab_block and
remainder_block are obtained
            # здесь можно отрисовать sub-block (e.g., visualize slab_block
and remainder_block)
        else:
            # No X cut needed (part spans full X of block)
            slab_block = block  # the whole block is the slab for further
cutting
            step_num = parent_step  # no new step for this
            # (No logging here because no cut performed along X)

        # *** Second cut (along Y-axis if needed) ***
        if py < slab_block.y:
            cut_coord = py
            self.step_counter += 1
            step_num = self.step_counter
            # Perform cut along Y on the slab_block
            front_piece_dims = (slab_block.x, py, slab_block.z)  # piece
containing part footprint
            back_piece_dims = (slab_block.x, slab_block.y - py - kerf,
slab_block.z)  # back remainder
            part_slab_block = Block(slab_block.id + "_F", *front_piece_dims,
kerf=kerf, grade=slab_block.grade)
            remainder_block2 = Block(slab_block.id + "_B", *back_piece_dims,
kerf=kerf, grade=slab_block.grade)
            cut_info = f"Y@{cut_coord}"
            # Log the back remainder as leftover
            self.steps.append({
                "Step": step_num,
                "Cut": cut_info,
                "Result": "Leftover",
                "Dimensions": f"{remainder_block2.x}x{remainder_block2.y}
x{remainder_block2.z}",
                "Parent": slab_block.id  # using block id of the slab as
parent reference
            })
            # Log the front piece (to be further cut) as leftover
            self.steps.append({
                "Step": step_num,
                "Cut": cut_info,
                "Result": "Leftover",
                "Dimensions": f"{part_slab_block.x}x{part_slab_block.y}
x{part_slab_block.z}",
                "Parent": slab_block.id
            })
            # Update slab_block to the front part_slab for the final cut
            slab_block = part_slab_block
            # Visualization hook: after second cut, part_slab_block and
remainder_block2 are obtained

# здесь можно отрисовать sub-block (visualize part_slab_block and
remainder_block2)
        # else: no Y cut needed if py == slab_block.y

        # *** Third cut (along Z-axis if needed) ***
        if pz < slab_block.z:
            cut_coord = pz
            self.step_counter += 1
            step_num = self.step_counter
            # Perform cut along Z on the slab_block
            bottom_piece_dims = (slab_block.x, slab_block.y, pz)
# this will be the final part
            top_piece_dims = (slab_block.x, slab_block.y, slab_block.z - pz -
kerf)  # top remainder
            part_block = Block(slab_block.id + "_Bot", *bottom_piece_dims,
kerf=kerf, grade=slab_block.grade)
            remainder_block3 = Block(slab_block.id + "_Top", *top_piece_dims,
kerf=kerf, grade=slab_block.grade)
            cut_info = f"Z@{cut_coord}"
            # Log the top remainder as leftover
            self.steps.append({
                "Step": step_num,
                "Cut": cut_info,
                "Result": "Leftover",
                "Dimensions": f"{remainder_block3.x}x{remainder_block3.y}
x{remainder_block3.z}",
                "Parent": slab_block.id
            })
            # Log the bottom piece as the final part
            self.steps.append({
                "Step": step_num,
                "Cut": cut_info,
                "Result": "Part",
                "Dimensions": f"{part_block.x}x{part_block.y}
x{part_block.z}",
                "Parent": slab_block.id,
                "PartID": part.id  # include part identifier to know which
part it is
            })
            final_part_block = part_block
            # Visualization hook: after third cut, final_part_block and
remainder_block3 are obtained
            # здесь можно отрисовать sub-block (visualize final_part_block
and remainder_block3)
        else:
            # No Z cut needed; slab_block itself is exactly the part
            final_part_block = slab_block
            # Log this as a part (no cut performed in Z)
            self.steps.append({
                "Step": step_num,  # last step number (X or Y cut) carries
over
                "Cut": None,
                "Result": "Part",
                "Dimensions": f"{final_part_block.x}x{final_part_block.y}
x{final_part_block.z}",
                "Parent": slab_block.id,
                "PartID": part.id
            })

        # At this point, final_part_block is the extracted part
        # Update part demand count and total parts volume
        part.quantity -= 1
        self.total_parts_volume += (part.x * part.y * part.z)  # original part volume
        # If multiple of this part needed, it might be cut again later from another block or leftover.

        # Now we have possibly produced up to three leftover blocks: remainder_block (X cut), remainder_block2 (Y cut), remainder_block3 (Z cut).
        # We need to recursively cut each of those leftovers as needed.
        # Note: We must be careful to only cut pieces that have meaningful size (dimensions > 0).
        # We will handle each leftover if it exists and has non-zero volume.
        # We'll start with the last produced leftover (depth-first) for efficiency (DFS approach).

        # Process Z-cut leftover (remainder_block3) if it exists and has volume:
        if 'remainder_block3' in locals() and remainder_block3.x > 0 and remainder_block3.y > 0 and remainder_block3.z > 0:
            self.cut_block(remainder_block3, parent_step=step_num)
        # Process Y-cut leftover (remainder_block2):
        if 'remainder_block2' in locals() and remainder_block2.x > 0 and remainder_block2.y > 0 and remainder_block2.z > 0:
            self.cut_block(remainder_block2, parent_step=step_num if slab_block else None)
        # Process X-cut leftover (remainder_block):
        if 'remainder_block' in locals() and remainder_block.x > 0 and remainder_block.y > 0 and remainder_block.z > 0:
            self.cut_block(remainder_block, parent_step=step_num if parent_step is not None else step_num)

    def run(self):
        """
        Execute the cutting algorithm on all stock blocks.
        Returns the steps table (DataFrame) and a summary of leftover blocks with volumes and utilization stats.
        """
        # Cut each stock block
        for stock in self.stocks:
            # Start cutting this stock block
            self.cut_block(stock, parent_step=None)

        # Create DataFrame from steps list for nice formatting (if pandas is available)
        df = pd.DataFrame(self.steps)
        # Calculate leftover blocks from steps: those marked Leftover that never became a parent of a Part.
        # Actually, since we logged all leftover pieces, we can filter final leftovers as those that have no child entries in steps.
        # One way: find all Parent values that appear (these are parent pieces that were cut further).
        parent_ids = set(item["Parent"] for item in self.steps if item["Parent"])
        # A final leftover is a row with Result = "Leftover" and whose piece id is not listed as a Parent in any other row.
        # But we didn't assign unique piece IDs to leftover entries in the table explicitly (we used block.id and parent ids).
        # We can augment our logging to include a unique piece ID for each row to trace parent-child precisely.
        # For simplicity here, we'll just output all leftovers from steps as potential scraps (the user can derive final ones if needed).
        leftover_rows = [item for item in self.steps if item["Result"] == "Leftover"]
        # Compute total leftover volume from all final leftover blocks (approximately, using their dimensions)
        total_leftover_vol = 0.0
        leftover_list = []
        for row in leftover_rows:
            dims = row["Dimensions"]
            if dims:
                x, y, z = map(float, dims.split('x'))
                vol = x * y * z
                total_leftover_vol += vol
                leftover_list.append((dims, vol))
        # Utilization and waste
        utilization = (self.total_parts_volume / self.total_stock_volume * 100.0) if self.total_stock_volume > 0 else 0.0
        waste = 100.0 - utilization  # percentage

        return df, leftover_list, utilization, waste

# Example usage (for testing purposes, if needed):
if __name__ == "__main__":
    # Define parts and stock for a simple test case
    parts = [
        Part("PartA", 50, 40, 20, quantity=2, priority=1),
        Part("PartB", 30, 20, 10, quantity=1, priority=0)
    ]
    stocks = [
        Block("Stock1", 100, 50, 20, kerf=0.5, grade="Steel_A")
    ]
    cutter = GuillotineCutter(parts, stocks)
    steps_df, leftovers, utilization, waste = cutter.run()
    print("Cut Steps:\n", steps_df)
    print("\nRemaining Leftover Blocks:", leftovers)
    print(f"Utilization: {utilization:.2f}%, Waste: {waste:.2f}%")
```

Explanation: The code above defines the necessary classes and the `GuillotineCutter` which encapsulates the algorithm. The `cut_block` method uses recursion to cut a block into a part and leftover pieces. It logs each cut step in the `self.steps` list as a dictionary (later converted to a DataFrame for convenience). We inserted comments like `# здесь можно отрисовать sub-block` to indicate where a visualization call could be placed to show the result of a cut. After running `cutter.run()`, you get the `steps_df` (table of cut steps), a list of leftover blocks with their volumes, and the utilization/waste metrics. This modular design allows easy integration into a web interface or further extension (for example, adding a GUI to visualize each cut step or optimizing the packing strategy).

---

(1) (2) (4) (5) (6) A Practical Solution for the 2D Guillotine Cutting Stock Problem - Studocu
https://www.studocu.vn/vn/document/ho-chi-minh-university-of-technology/discrete-math/a-simple-approach-to-the-two-dimensional-guillotin/114507976

(3) Algorithms for 3D guillotine cutting problems: Unbounded knapsack ...
https://www.sciencedirect.com/science/article/pii/S0305054811000840