Logic Challenge – Solution

-by Sandeep Patil

Problem Statement:

During the operation of picking up boxes from the pallet using a robot, decide the first object to be picked based on collision avoidance.

Given:

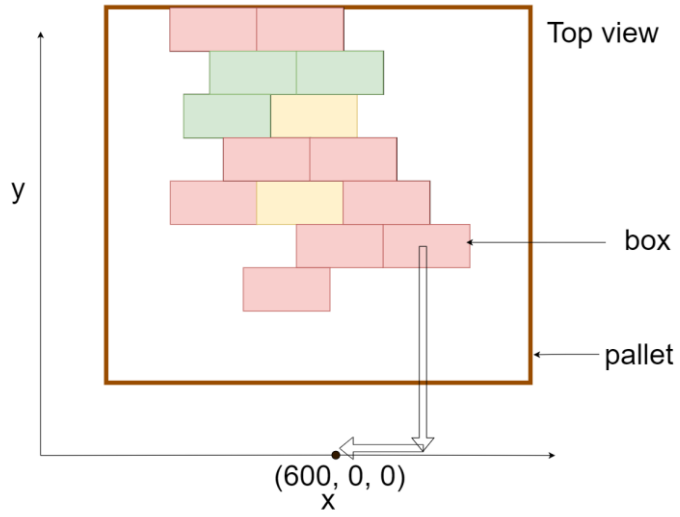
1. Position of Robot (X, Y, Z) = (0,0,0)
2. Height of tallest item = 400mm
3. All objects are right parallelogram prism
4. Object (box) properties:
    a. Center of the object's top surface = [object.center.x, object.center.y, object.center.z]
    b. X dimension = [object.x]
    c. Y dimension = [object.y]
    d. Maximum Z dimension: [object.z] ≤ 400mm
5. There is no restriction of number of boxes stacked.
6. Max height to which the robot can pick the object = up to the height of tallest item = 400mm
7. The dimensions of the flat pallet (width, breadth, height) = (1000mm, 1000mm, 0mm)
8. Items can slide on top of each other.
9. Robot will go on top of the box at a height = 100mm from surface of the selected box.

Object picking operation by robot

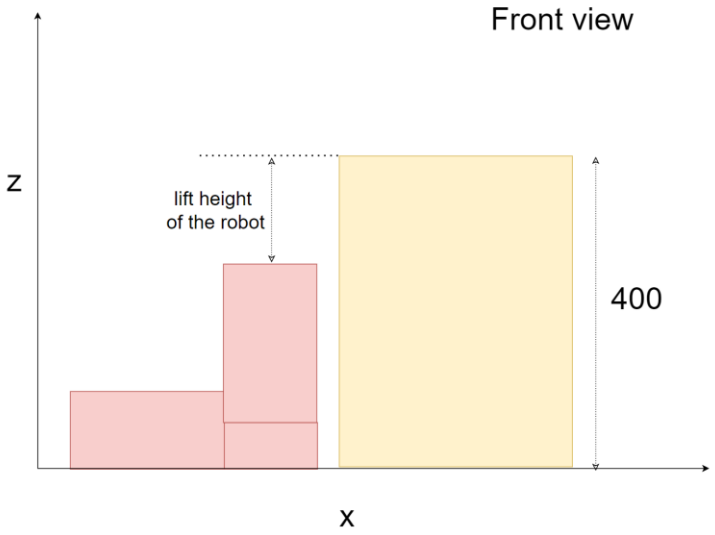
1. Go to the objects position and stop at 100mm from the surface.
2. Go vertically down.
3. Pick the object.
4. Go vertically up to the height of 400mm.
5. Move to the drop point of the item and drop the item.

Assumptions:

1. Object top surface centers [object.center.x, object.center.y, object.center.z] are given for all the objects.
2. Object centers [object.center.x, object.center.y, object.center.z] in mm are top surface centers of the object in the global coordinates (with (0,0,0) at the robot's center).
3. Vacuum gripper is used to pick up the boxes.
4. There is no collision/interference between robot and other boxes during the gripping action.
5. Boxes are assumed to be parallel to the coordinate axis.
6. The path of robot after picking the object will be as shown by the arrow marks in figure below:



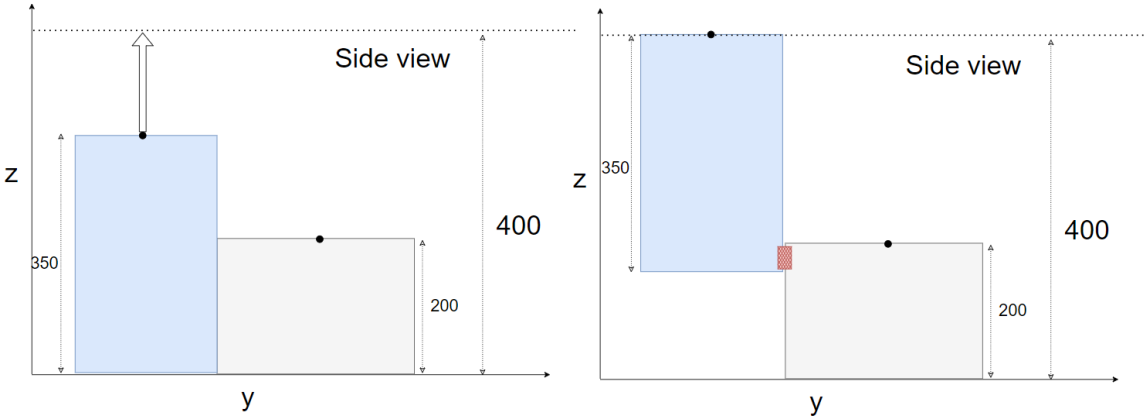
7. Max height to which the robot can pick the object = up to the height of tallest item = 400mm – this is defined as the lift height of the robot in the figure below:



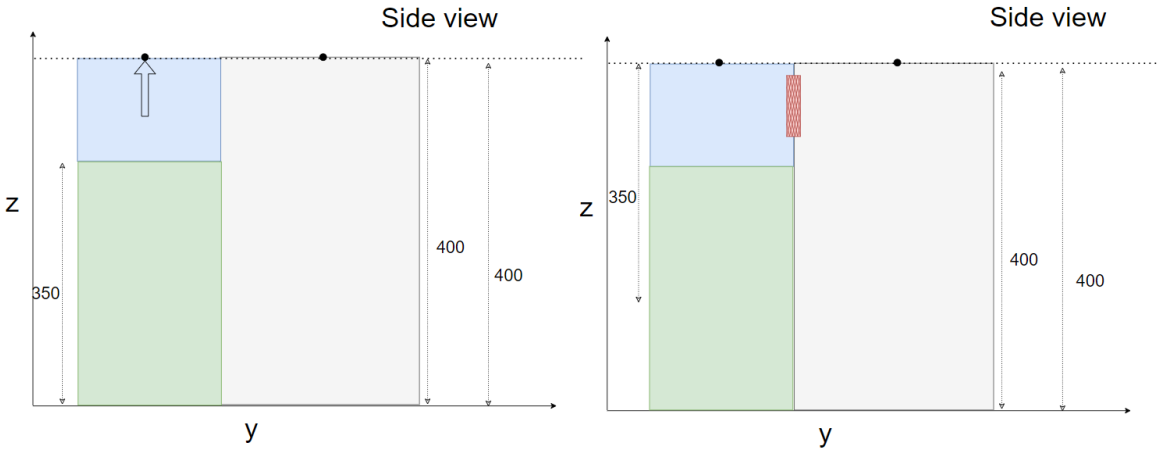
Methodology:

A simple solution could be given as first object in the sorted list of object.center.z of the objects. But there are edge cases during which picking box with highest object.center.z can cause collision between the robot and the box.

1. Consider the boxes as shown in the figure below. If the blue box is to be picked first based on its higher object.center.z to the height of 400mm. It can cause collision.

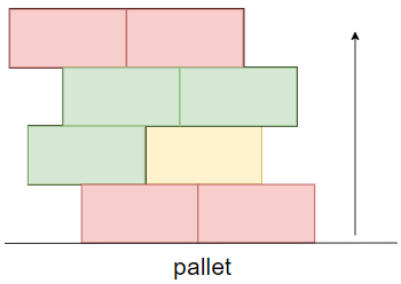


2. Consider three boxes as shown in the figure below, the tallest object's surface center has the object.center.z as 400mm. The object cannot be picked based on the given condition of lifting height limit of 400mm. Even if the robot picks the object, it will collide immediately with the next box as shown in the figure (right).



Hence, an optimal solution should be able to predict objects which are at the topmost position in their stack respectively. For all objects at the topmost position in their stack. If there is no collision between the selected object and other objects after virtually picking it, then pick the selected object.

Note: stack is defined as objects which are overlapping in their area and have increasing object.center.z being on top of each other. An example is shown below:



General steps of solution:

1. Get objects from computeObjects()
2. Obtain a list of objects which are stacked on top each other.
3. Calculate the height of each box in its stack.
4. Sort the objects at the topmost position in the stack by their surface center heights.
5. For each object in the sorted list of objects, check if there is any collision in y-direction only by virtually picking it.
6. Pick the first object with no collision. Continue.

Pseudo-code:

```
Objects = compute_objects()

Where
objects -> {'center_x': object_center_x, 'center_y': object_center_y,
'center_z': object_center_z, 'x': object_x,
            'y': object_y}
pallet_size = (1000,1000)
If no objects:
    Return none
# Obtain a list of objects which are stacked on top each other
for i in objects
    # store the bottom most objects in list_base
    list_base < - i

    Store bottom most object as key in dict list_objects
    list_objects = [i]

    for j in objects['center_x']:
        # Check if center_height is lower—to stack the objects as they-
                                                          -appear visually
        if i not equal to j and objects['center_z'][i] < ob-
jects['center_z'][j]:
        # Check if there is overlap
        if overlap(objects['center_x'][i], objects['center_y'][i],
objects['x'][i], objects['y'][i],
                   objects['center_x'][j], objects['center_y'][j],
                   objects['x'][j], objects['y'][j])
            list_objects < - j
            dict_objects[“i”] = list_objects
dict_base['base'] = list_base
dict_objects < - dict_objects + dict_base

# For the stack of objects, calculate the height of the box with respect to
surface
for key, value in dict_objects
    if key not 'base'
    repeat
    if h_init = 0
    h_init = objects['center_z'][value[i]]
else:
    h_init = objects['center_z'][value[i]] - h_init
until length(value)
min_h_object[“key”] = h_init
else
for i in value
    h_in < - objects['center_z'][value[i]]
    value_list < - value[i]
    # For the objects which are at the topmost in their stack, obtain their
heights
    min_h_object[“value”] = max(h_in)

# sort to topmost objects by their surface center heights in descending
order
sorted_objects = sorted(dict_objects.values[-1], key=lambda x:
max(objects['center_z'][x]))
```

```
for k in sorted_objects:
    for i in objects:
        l1 = Point(objects['center_x'][i] - (objects['x'][i] / 2),
          objects['center_y'][i] + (objects['y'][i] / 2))
        r1 = Point(objects['center_x'][i] + (objects['x'][i] / 2),
          objects['center_y'][i] - (objects['y'][i] / 2))
        l2 = Point(objects['center_x'][k] - (objects['x'][k] / 2),
          objects['center_y'][k] + (objects['y'][k] / 2))
        r2 = Point(objects['center_x'][k] + (objects['x'][k] / 2),
         objects['center_y'][k] - (objects['y'][k] / 2))
       # Check if there is no collision when the box is picked based on
its height.

        if l1[0] > l2[0] and r1[0] < r2[0] and
                       400 - min_h_object[k] > objects['center_z'][i]:
              pick_object = object[i]
              print('pick object:', object[i])
        return pick_object

# Required utility class and functions
class Point:
    """
    Creates global coordinate points objects given (x,y)
    """
    def __init__(self, x, y):
        self.x = x
        self.y = y


def overlap(cx_1, cy_1, x_1, y_1, cx_2, cy_2, x_2, y_2):
    """
    Checks whether two rectangles overlap
    Parameters
    ----------
    cx_1: first object center (x)
    cy_1: first object center (y)
    x_1: first object dimension (x)
    y_1: first object dimension (y)
    cx_2: first object center (x)
    cy_2: second object center (y)
    x_2: second object dimension (x)
    y_2: second object dimension (y)

    Returns bool
    -------

    """
    l1 = Point(cx_1 - (x_1 / 2), cy_1 + (y_1 / 2))
    r1 = Point(cx_1 + (x_1 / 2), cy_1 - (y_1 / 2))
    l2 = Point(cx_2 - (x_2 / 2), cy_2 + (y_2 / 2))
    r2 = Point(cx_2 + (x_2 / 2), cy_2 - (y_2 / 2))
    # To check if either rectangle is actually a line
    if l1.x == r1.x or l1.y == r1.y or l2.x == r2.x or l2.y == r2.y:
        return False

    # If one rectangle is on left side of other
    if l1.x >= r2.x or l2.x >= r1.x:
        return False
```

```
    # If one rectangle is above other
    if r1.y >= l2.y or r2.y >= l1.y:
        return False

    return True
```