1 背景介绍

1.1 问题简述与分析

简述：货物装箱问题，将一定数量的物品放入有一定容量的容器中，使得每个容器中的物品之和不超过容器容量并获得某种最佳的效益。待放入的物品和使用的箱子均为长方体。设车厢为长方形，其长宽高分别为 L，W，H；共有 n 个箱子，箱子也为长方形，第 i 个箱子的长宽高为 l_i，w_i，h_i（n 个箱子的体积总和是要远远大于车厢的体积），做以下假设和要求：

1. 长方形的车厢共有 8 个角，并设靠近驾驶室并位于下端的一个角的坐标为（0,0,0），车厢共 6 个面，其中长的 4 个面，以及靠近驾驶室的面是封闭的，只有一个面是开着的，用于工人搬运箱子；

2. 需要计算出每个箱子在车厢中的坐标，即每个箱子摆放后，其和车厢坐标为（0,0,0）的角相对应的角在车厢中的坐标，并计算车厢的填充率。

问题分解为基础和高级部分。

基础部分：

1.所有的参数为整数；

2.静态装箱，即从 n 个箱子中选取 m 个箱子，并实现 m 个箱子在车厢中的摆放（无需考虑装箱的顺序，即不需要考虑箱子从内向外，从下向上这种在车厢中的装箱顺序）；

3.所有的箱子全部平放，即箱子的最大面朝下摆放；

4.算法时间不做严格要求，只要 1 天内得出结果都可。

高级部分：

1.参数考虑小数点后两位；

2.实现在线算法，也就是箱子是按照随机顺序到达，先到达先摆放；

3.需要考虑箱子的摆放顺序，即箱子是从内到外，从下向上的摆放顺序

4.因箱子共有 3 个不同的面，所有每个箱子有 3 种不同的摆放状态；

5.算法需要实时得出结果，即算法时间小于等于 2 秒。

1.2 问题分析

针对特定需求的装箱问题有很多约束，例如重心约束，承重能力约束等。本文主要考虑的是单容器异构三维装箱问题，可以形式化定义如下:

给定一个容器(其体积为 V)和一系列待装载的物品，容器和物品的形状都是长方体。问题的目标是要确定一个可行的物品放置方案使得在满足给定装载约束的情况下，容器中包含的物品总体积 S 尽可能大，填充率尽可能高。填充率定义为：

$$\rho = \frac{S}{V} * 100\%$$

可行的放置方案要求放置满足如下四个基本约束条件:

（1）任何两个被装载的物品不能互相重叠。

（2）被装载的物品必须完全被包含在容器中。

（3）所有被装载的物品不能斜放。

（4）静态装箱：从 n 个物品中选取 m 个物品，并且实现 m 个物品在容器内的摆放。

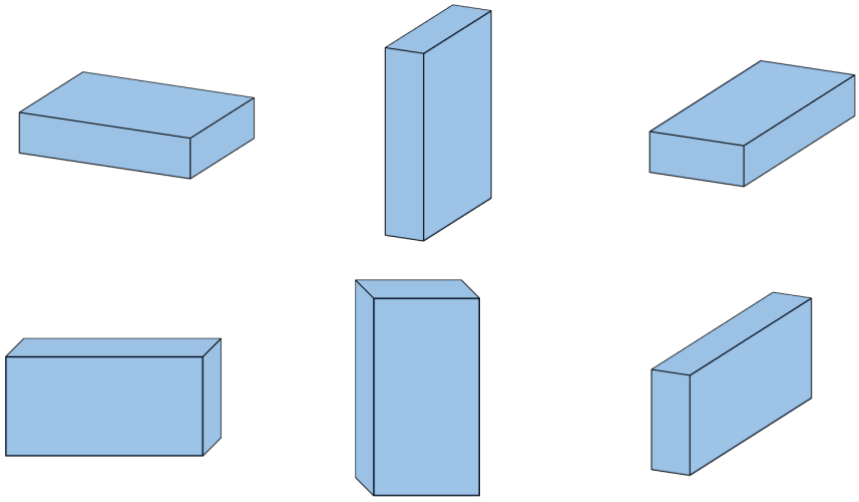
除了以上的基本约束以外，针对实际问题，本方案还考虑以下五个额外的约束。

（1）稳定约束：禁止被装载的物品悬空。

（2）浮点尺寸：物品的尺寸最多考虑到小数点后两位。

（3）顺序摆放：物品在容器内摆放时，按照从内到外或者是从下到上的顺序摆放。

（4）摆放姿态：对于一个 L×W×H 的物品而言，其在容器中有 6 种不同的摆放姿态。如下图所示。以某一种方式不能装载新的货物时，旋转货物，可能就可以装载。显然，不同的摆放方式会导致货物在车厢中的状态不同，导致不同的空间利用率。



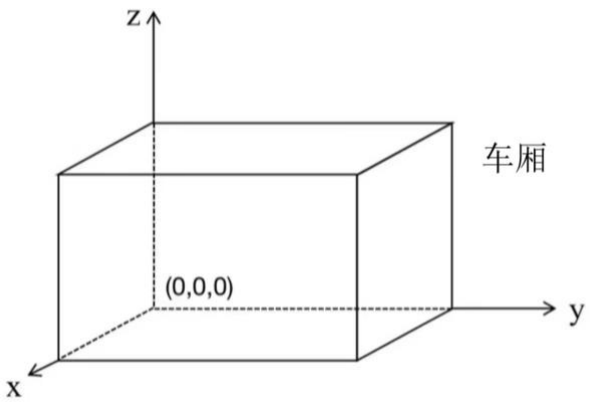
（5）实用性：装箱算法能够较快地实现，在 2 秒以内完成

2 算法介绍

2.1 模型说明

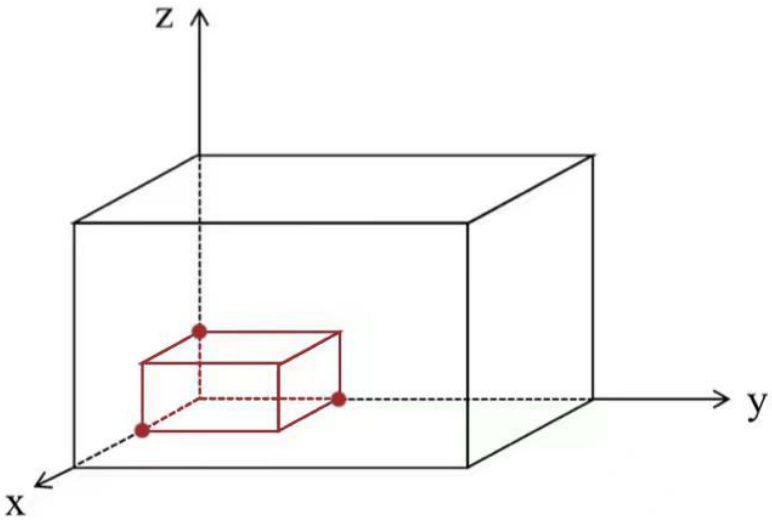
1. 车厢与货物

实验中考虑的车厢是给定长宽高的长方体，设靠近驾驶室并位于下端的一个 角的坐标为（0,0,0），建立空间直角坐标系,如下图所示。同理，货物也是若干给定长宽高的长方体，一共有八个角，以靠近坐标原点的角的坐标作为货物的坐标。



2. 可放置点

可放置点指的是车厢内可以用来放置货物的坐标点。初始化时，车厢为空， 可放置点只有一个，为坐标原点（0，0，0）。因此，第一个放入车厢的物品需 要放在坐标原点。放入第一个物体后，删除使用的可放置点，并添加新生成的可 放置点到列表中。放入第一个物体的状态如下图所示，其中可放置点在图中加粗显示。设第 i 个放入的物体的坐标为$(x_i, y_i.z_i)$，则新生成的可放置点有三个，分别是$(x_i + l_i, y_i.z_i)$，$(x_i, y_i + w_i.z_i)$，$(x_i, y_i.z_i + h_i)$。



3. 参考面

本实验中通过引入水平和垂直方向上的参考面，来引导物体的装箱过程。在装箱过程中，优先在参考面的约束范围内进行货物装填。

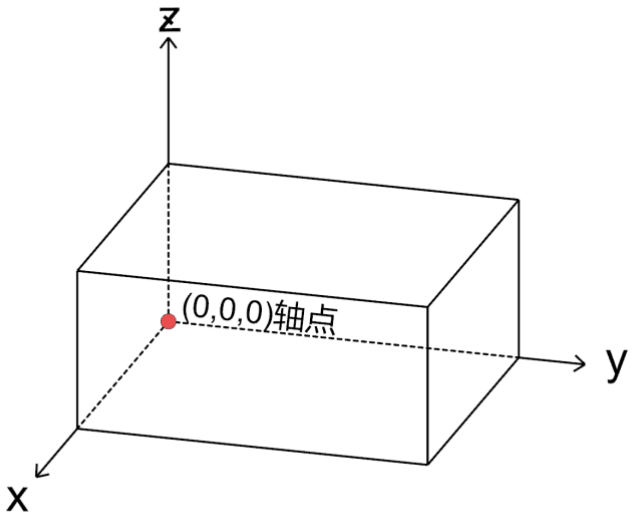
具体来说，两个参考面分别与 yoz 面和 xoy 面平行，当放入新货物时，需要检测是否在参考面的约束范围内。因此，新放入的货物优先使用 y 轴上的可放置点。当货物可以装入容器，但不满足参考面约束时，需要调整参考面，如果无论怎么调整，货物都不能装入容器，则认为该货物无法装入，考虑下一个货物。

4. 挪动

由于参考面的引入，可能出现先装入小货物，再装入大货物，从而出现悬空的情况。为了进一步提高空间利用率，考虑增加挪动的操作，即把货物挪到贴边放置，尽可能消除空隙。在实验中，挪动可以分为沿 x 轴、y 轴、z 轴三个方向，为了保证货物的状态不是悬空的，需要优先沿着 x 轴和 y 轴挪动，再考虑沿 z 轴方向挪动，使货物尽可能靠边放置。

2.2 匹配算法

在将物品放置入容器的过程中，需要考虑三个方向上的扩展来放置，即 x 轴方向，y 轴方向和 z 轴方向。在放置物品之前，我们首先选择一个“轴点”，这个轴点是物体左侧后方下侧的顶点，这个轴点坐标作为物品在容器内位置的记录。



如果当前的轴点位置无法被放置进入容器中的话，我们就将物体采取 6 个方式来旋转，直至物体能够被放置入容器中的当前轴点位置上。如果在尝试了 6 中不同旋转方式的情况下，物体依然无法放入，那么就先放入另一个物体，并将当前的物体添加到未放置列表中。在所有的物体放入完毕后，我们会尝试着放入未放置列表中的物体。在算法开始执行之前，一个空容器的首个轴点会被初始化 为（0,0,0）

2.3 算法简述

（1）初始化轴点为（0,0,0），对所有要放入的物体按照体积大小进行排序，在放置过程中，优先放置体积大的物体。
（2）在第一个物体放置完毕后，之后每放置一个物体之前，首先生成一个新的轴点。这个轴点是按照上一个物体的轴点在依次按照 x 轴，y 轴，z 轴三个方向上生成的三个可放置轴点。
（3）在新的放置点上，尝试 6 种不同的物体旋转方式，直到物品可以被放进容器之中。由于并没有回溯机制，因此在方案中，物体按照一种方式放进了某 个可放置轴点之中，就会再开始尝试放入下一个物体。
（4）如果尝试了 3 个可放置轴点，每个轴点上都尝试了 6 种不同的旋转方式，都无法将物品放入容器中，那么就将物品放入未放置列表中，跳过该物品处 理下一个。
（5）重复 2-4 步，处理完所有的物品之后，所有可放置的物品已经被放入 箱子中，同时也有一些物品保存在了未放置列表中，此时再来尝试处理这些未放置的物品。

3 实验环境

Window 11
Python 3.11

4 代码说明

1.定义坐标点，建立空间直角坐标系，设置合法化属性和序列化属性

```python
class Point(object):
    def __init__(self, x: int, y: int, z: int) -> None:
        self.x = x
        self.y = y
        self.z = z

    def __repr__(self) -> str:
        return f"({self.x},{self.y},{self.z})"

    def __eq__(self, __o: object) -> bool:
        return self.x == __o.x and self.y == __o.y and self.z == __o.z

    @property
    def is_valid(self) -> bool:
        return self.x >= 0 and self.y >=0 and self.z>= 0

    @property
    def tuple(self) -> tuple:
        return (self.x, self.y, self.z)
```

2.定义货物的六种摆放方式。初始化时，应提供相应的尺寸信息。由于摆放方式不同，货物的长宽高也不同，所以根据设置的摆放方式，设置长、宽、高属性，返回对应的值。

```python
class CargoPose(Enum):
    # 最长的那条边作为高 第二长的那条边作为宽
    tall_wide = 0
    # 最长的那条边作为高 最短的那条边作为宽
    tall_thin = 1
    # 第二长的那条边作为高 最长的那条边作为宽
    mid_wide = 2
    # 第二长的那条边作为高 最短的那条边作为宽
    mid_thin = 3
    # 最短的那条边作为高 最长的那条边作为宽
    short_wide = 4
    # 最短的那条边作为高 第二长的那条边作为宽
    short_thin = 5
```

```python
class Box(object):
    def __init__(self, length: int, width: int, height: int) -> None:
        # 初始化数据 车厢的长宽高
        self._length = length
        self._width = width
        self._height = height
        # 更新车厢状态
        self._refresh()

    # repr函数将对象转换为供解释器读取的形式
    # 返回一个对象的str格式 这里返回车厢的长，宽，高
    def __repr__(self) -> str:
        return f"{self._length}, {self._width}, {self._height}"

    # 更新车厢状态函数
    def _refresh(self):
        self._horizontal_planar = 0  # 水平放置参考面
        self._vertical_planar = 0  # 垂直放置参考面
        self._available_points = [Point(0, 0, 0)]  # 可放置点有序集合
        # 已放置的货物信息 货物信息由一个有序列表组成 可对其进行增删查操作
        self._setted_cargos : List[Cargo] = []
```

```python
# 货物具体摆放方式
@property
def _shape_swiche(self) -> dict:
    # 对货物长宽高做一个排序 从小到大
    edges = sorted(self._shape)
    return {
        # 最长的那条边作为高 最短的那条边作为长
        CargoPose.tall_thin: (edges[1], edges[0], edges[-1]),
        # 最长的那条边作为高 第二长的那条边作为长
        CargoPose.tall_wide: (edges[0], edges[1], edges[-1]),
        # 第二长的那条边作为高 最短的那条边作为长
        CargoPose.mid_thin: (edges[-1], edges[0], edges[1]),
        # 第二长的那条边作为高 最长的那条边作为长
        CargoPose.mid_wide: (edges[0], edges[-1], edges[-1]),
        # 最短的那条边作为高 第二长的那条边作为长
        CargoPose.short_thin: (edges[-1], edges[1], edges[0]),
        # 最短的那条边作为高 最长的那条边作为长
        CargoPose.short_wide: (edges[1], edges[-1], edges[0])
    }
```

3. 开始货物装箱。测试数据以 TXT 文件的格式存储，首先获取货物的信息，包括货物的三个尺寸信息和同一规格的货物数量。接着利用贪心算法将货物按照体积大小从大到小排序，返回排序后的货物列表。然后，依次考虑在不同的摆放方式下，货物是否能装入车厢，是否改变了参考面，随后逐一装载货物到车厢中，直至车厢不能再装入任何货物。

```python
if __name__ == "__main__":
    paths = 'data/E1-3.txt'  #数据所在的位置
    filename = os.path.basename(paths)  # 从文件路径中读取最后一个文件夹的名字
    filename = os.path.splitext(filename)[0] #去掉文件名后缀
    f = open(paths)  #打开文件
    lines = f.readline() #读取文件第一行数据
    lines = lines.replace(','," ")
    lines = lines.split()
    int_list = [int(x) for x in lines]

    case = Box(int_list[0],int_list[1],int_list[2])  #设置箱子的属性

    cargos= [Cargo(0,0,0) for _ in range(0)]
    while True:
        lines = f.readline()
        if(lines):
            lines = lines.replace(','," ")
            lines = lines.split()
            lines.pop(0)
            lines = [int(x) for x in lines]
            cargos.extend([Cargo(lines[0],lines[1],lines[2]) for _ in range(lines[3])])
        else:
            break
```

```python
def encase_cargos_into_container(
    cargos:Iterable,
    container:Box,
    strategy:type
) -> float:
    # 根据体积大小排列好了货物
    sorted_cargos:List[Cargo] = strategy.encasement_sequence(cargos)
    i = 0 # 记录发当前货物
    while i < len(sorted_cargos):
        j = 0 # 记录当前摆放方式
        cargo = sorted_cargos[i]
        poses = strategy.choose_cargo_poses(cargo, container)
        while j < len(poses):
            cargo.pose = poses[j]
            is_encased = container._encase(cargo)
            if is_encased.is_valid:
                break # 可以装入 不在考虑后续摆放方式
            j += 1  # 不可装入 查看下一个摆放方式
        if is_encased.is_valid:
            i += 1 # 成功放入 继续装箱
        elif is_encased == Point(-1,-1,0):
            continue # 没放进去但是修改了参考面位置 重装
        else :
            i += 1 # 纯纯没放进去 跳过看下一个箱子
    return sum(list(map(
            lambda cargo:cargo.volume,container._setted_cargos
        ))) / container.volume
```

```python
# 装箱
def _encase(self, cargo: Cargo) -> Point:
    # flag存储放置位置, (-1, -1, 0)放置失败并调整参考面, (-1, -1, -1)放置失败.
    flag = Point(-1, -1, -1)
    # 用于记录执行前的参考面位置, 便于后续比较
    history = [self._horizontal_planar, self._vertical_planar]
    # 参考面是否改变
    def __is_planar_changed() -> bool:
        return (
            not flag.is_valid and# 防止破坏已经确定可放置的点位, 即只能在(-1, -1, -1)基础上改
            self._horizontal_planar == history[0] and
            self._vertical_planar == history[-1]
        )
    # 装箱实现 依次取出可放置点 尝试放置货物
    for point in self._available_points:
        # 正常放置  如果没有碰撞冲突且放置的货物没有超过参考面
        if (
            self.is_encasable(point, cargo) and
            point.x + cargo.length < self._horizontal_planar and
            point.z + cargo.height < self._vertical_planar
        ):
            # 存储放置点 跳出循环
            flag = point
            break
```

```python
if not flag.is_valid:
    if (
        # 查看yz参考面的位置 如果为0或者是达到容器边缘
        # 为0说明第一个货物还没放 到容器边缘说明yz参考面已经达到上限 该考虑叠放了
        # 就比如第一层放满了 开始放第二层
        self._horizontal_planar == 0 or
        self._horizontal_planar == self.length
    ):
        # 尝试从z轴的某一点开始放置 开始叠放货物 没有冲突
        if self.is_encasable(Point(0, 0, self._vertical_planar), cargo):
            # 能放 更新放置点
            flag = Point(0, 0, self._vertical_planar)
            # 更新参考面的值
            self._vertical_planar += cargo.height
            self._horizontal_planar = cargo.length
            # 放置了货物 不检测参考面改变
        # 有冲突 说明该货物的高度叠加到当前层后 超出了车厢高度
        elif self._vertical_planar < self.height:
            # 调整xy参考面为车厢最大高度
            self._vertical_planar = self.height
            self._horizontal_planar = self.length
            if __is_planar_changed():
                flag.z == 0 # 放置失败并调整参考面
    # 否则 继续在当前层放置
```

```
# 否则 继续在当前层放置
else:
    for point in self._available_points:
        # 从可放置点列表中选出贴着yz参考面的放置点
        # 并且该放置点在xz面上
        # 放置不冲突且放置后货物满足容器垂直参考线的限制
        # 可以理解为当前层 第一行放满了 放第二行了
        if (
            point.x == self._horizontal_planar and
            point.y == 0 and
            self.is_encasable(point, cargo) and
            point.z + cargo.height <= self._vertical_planar
        ):
            # 标记为已放置已更新
            flag = point
            # 更新水平参考线
            self._horizontal_planar += cargo.length
            break
            # 放置了货物 不检测参考面改变
    # 可放置点列表中没有符合条件的放置点 那么货物未放置 更新yz参考面
    if not flag.is_valid:
        self._horizontal_planar = self.length
        # 更新yz水平参考线
        if __is_planar_changed():
            flag.z == 0 # 放置失败并调整参考面
```

```
# 货物已经成功放置
if flag.is_valid:
    # 将放置点赋值给货物放置点
    cargo.point = flag
    # 删除该放置点
    if flag in self._available_points:
        self._available_points.remove(flag)
    # 挪动货物 （函数中有判断是否需要挪动）
    self._adjust_setting_cargo(cargo)
    # 将货物添加到已放置列表
    self._setted_cargos.append(cargo)
    # 并将该货物放入后产生的三个新的放置点加入放置点列表
    self._available_points.extend([
        Point(cargo.x + cargo.length, cargo.y, cargo.z),
        Point(cargo.x, cargo.y + cargo.width, cargo.z),
        Point(cargo.x, cargo.y, cargo.z + cargo.height)
    ])
    # 对放置点进行排序
    self._sort_available_points()
return flag
```

4. 判断货物是否可放置。一个可以放置的货物需满足两个条件：其一，不和已经装入车厢的货物发生碰撞；其二，货物可以完全装入车厢内。首先检测是否

能装入车厢，再检测是否与其他货物碰撞。本实验通过碰撞检测的方法实现检测。将货物投影到坐标轴构成的平面上，检测长方形之间是否重叠，如果在三个投影面上，长方形均未发生重叠，则我们认为在空间上，货物之间也不会发生重叠。

```
# 货物之间是否冲突 三个投影面都没有冲突的时候 返回false 即无冲突
def _is_cargos_collide(cargo0: Cargo, cargo1: Cargo) -> bool:
    return (
        _is_rectangles_overlap(cargo0.get_shadow_of("xy"), cargo1.get_shadow_of("xy")) and
        _is_rectangles_overlap(cargo0.get_shadow_of("yz"), cargo1.get_shadow_of("yz")) and
        _is_rectangles_overlap(cargo0.get_shadow_of(
            "xz"), cargo1.get_shadow_of("xz"))
    )
```

```
# 判断长方形是否冲突 冲突返回1 不冲突返回0
def _is_rectangles_overlap(rec1:tuple, rec2:tuple) -> bool:
    return not (
        # 以下四项只要有一个是1（即不冲突） 那么返回就是0
        # 即长方体无冲突 右边的左上角比左边的右下角大 即无碰撞
        # xy面为例子：0和1是矩形左上角的坐标 2和3矩形右下角的坐标
        # 假设rec1是右边的
        rec1[0] >= rec2[2] or rec1[1] >= rec2[3] or
        # 假设rec2是右边
        rec2[0] >= rec1[2] or rec2[1] >= rec1[3]
    )
```

5. 挪动货物。为了避免挪动过程中对货物装载数据造成污染，需要复制原货物的装载数据，先在副本上进行挪动操作，挪动完成后，再将最终的放置点赋值到需要挪动的货物。

```python
def _adjust_setting_cargo(self, cargo: Cargo):
    site = cargo.point
    # 为避免污染 进行一个深复制操作
    temp = deepcopy(cargo)
    # 没有冲突 不需要挪动货物
    if not self.is_encasable(site, cargo):
        return None
    # 序列化坐标
    xyz = [site.x, site.y, site.z]
    # 序列化坐标以执行遍历递减操作, 减少冗余
    for i in range(3):  # 012 分别表示 xyz
        is_continue = True
        while xyz[i] > 1 and is_continue:
            # 依次在xyz三个方向上挪动货物 挪动的结果就是尽量使得货物靠边
            xyz[i] -= 1
            temp.point = Point(xyz[0], xyz[1], xyz[2])
            # 挪动后检测冲突 往边挪 不存在挪出容器的问题 所以 仅检测货物之间的冲突
            for setted_cargo in self._setted_cargos:
                if not _is_cargos_collide(setted_cargo, temp):
                    continue
                # 冲突了 挪回去 跳出循环
                xyz[i] += 1
                is_continue = False
                break
    # 在一系列挪动之后 更新货物的放置点
    cargo.point = Point(xyz[0], xyz[1], xyz[2])  # 反序列化
```

6. 输出货物装载结果。将货物的长宽高信息和坐标点信息写入 TXT 文件，并将结果进行三维可视化。

```python
# 保存已放置货物的放置点和放置状态信息
def save_encasement_as_file(self, filename):
    file = open(f"{filename}.csv", 'w', encoding='utf-8')
    file.write(f"index,x,y,z,length,width,height\n")
    i = 1
    for cargo in self._setted_cargos:
        file.write(f"{i},{cargo.x},{cargo.y},{cargo.z},")
        file.write(f"{cargo.length},{cargo.width},{cargo.height}\n")
        i += 1
    file.write(f"container,,,,{self}\n")
    file.close()
```

```python
def _plot_opaque_cube(x=10, y=20, z=30, dx=40, dy=50, dz=60):
    xx = np.linspace(x, x+dx, 2)
    yy = np.linspace(y, y+dy, 2)
    zz = np.linspace(z, z+dz, 2)
    xx2, yy2 = np.meshgrid(xx, yy)
    ax.plot_surface(xx2, yy2, np.full_like(xx2, z), cmap = 'winter')
    ax.plot_surface(xx2, yy2, np.full_like(xx2, z+dz), cmap = 'viridis')
    yy2, zz2 = np.meshgrid(yy, zz)
    ax.plot_surface(np.full_like(yy2, x), yy2, zz2, cmap = 'twilight')
    ax.plot_surface(np.full_like(yy2, x+dx), yy2, zz2, cmap = 'summer')
    xx2, zz2= np.meshgrid(xx, zz)
    ax.plot_surface(xx2, np.full_like(yy2, y), zz2, cmap = 'spring')
    ax.plot_surface(xx2, np.full_like(yy2, y+dy), zz2, cmap ='Greens')

def _plot_linear_cube(x, y, z, dx, dy, dz, color='red'):
    # ax = Axes3D(fig)
    xx = [x, x, x+dx, x+dx, x]
    yy = [y, y+dy, y+dy, y, y]
    kwargs = {'alpha': 1, 'color': color}
    ax.plot3D(xx, yy, [z]*5, **kwargs)
    ax.plot3D(xx, yy, [z+dz]*5, **kwargs)
    ax.plot3D([x, x], [y, y], [z, z+dz], **kwargs)
    ax.plot3D([x, x], [y+dy, y+dy], [z, z+dz], **kwargs)
    ax.plot3D([x+dx, x+dx], [y+dy, y+dy], [z, z+dz], **kwargs)
    ax.plot3D([x+dx, x+dx], [y, y], [z, z+dz], **kwargs)
```

5 总结

本实验完成了基础的和部分高级的要求：

1. 作业的基础部分要求所有的参数为整数，同时只需满足静态装箱即可，我们在实现的时候选择了动态装箱的算法，同时考虑了箱子的各个旋转情况，并且得出结果的时间小于 1 天，即满足了基础部分的要求。
2. 高级部分要求我们考虑箱子的到达顺序，先到达的先摆放，这和我们算法的过程一致。同时高级部分也要求我们考虑箱子的摆放顺序，即箱子是从内到外，从下向上的摆放顺序，我们也满足了这个要求。第三点高级部分也要求我们考虑物品不同的摆放状态，即考虑长、宽、高的排列组合顺序，我们在程序中也通过旋转方式满足了这一点，其次也考虑了小数点后两位。并且算法时间满足 2 秒内（每个货物）

我们的实验在大部分情况下都取得了 85%以上的空间利用率，具有较好的性能。但是由于本实验中使用的装箱方法，首先要遍历所有的货物进行装载，在遍历货物时 会遍历每个可放置点，每个可放置点中需要检测是否有碰撞，碰撞检测又会遍历已放置的点，导致时间复杂度略高。