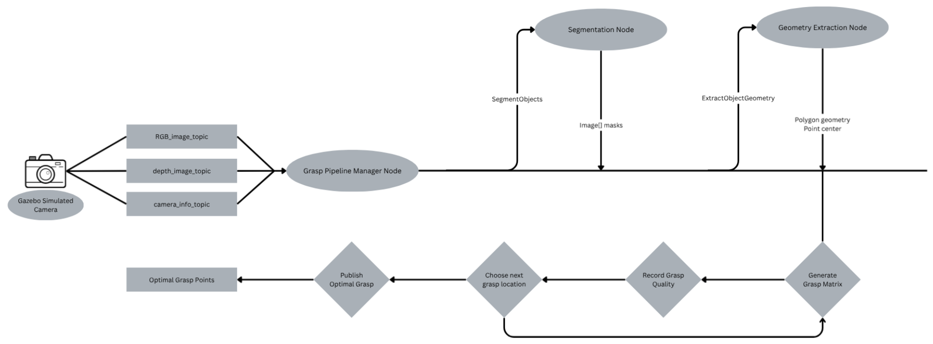
RBE 4540 — Group Assignment: Top Surface Grasping

Brent Weiffenbach, Isabella Rosenstein, Evan Carmody

October 10, 2025

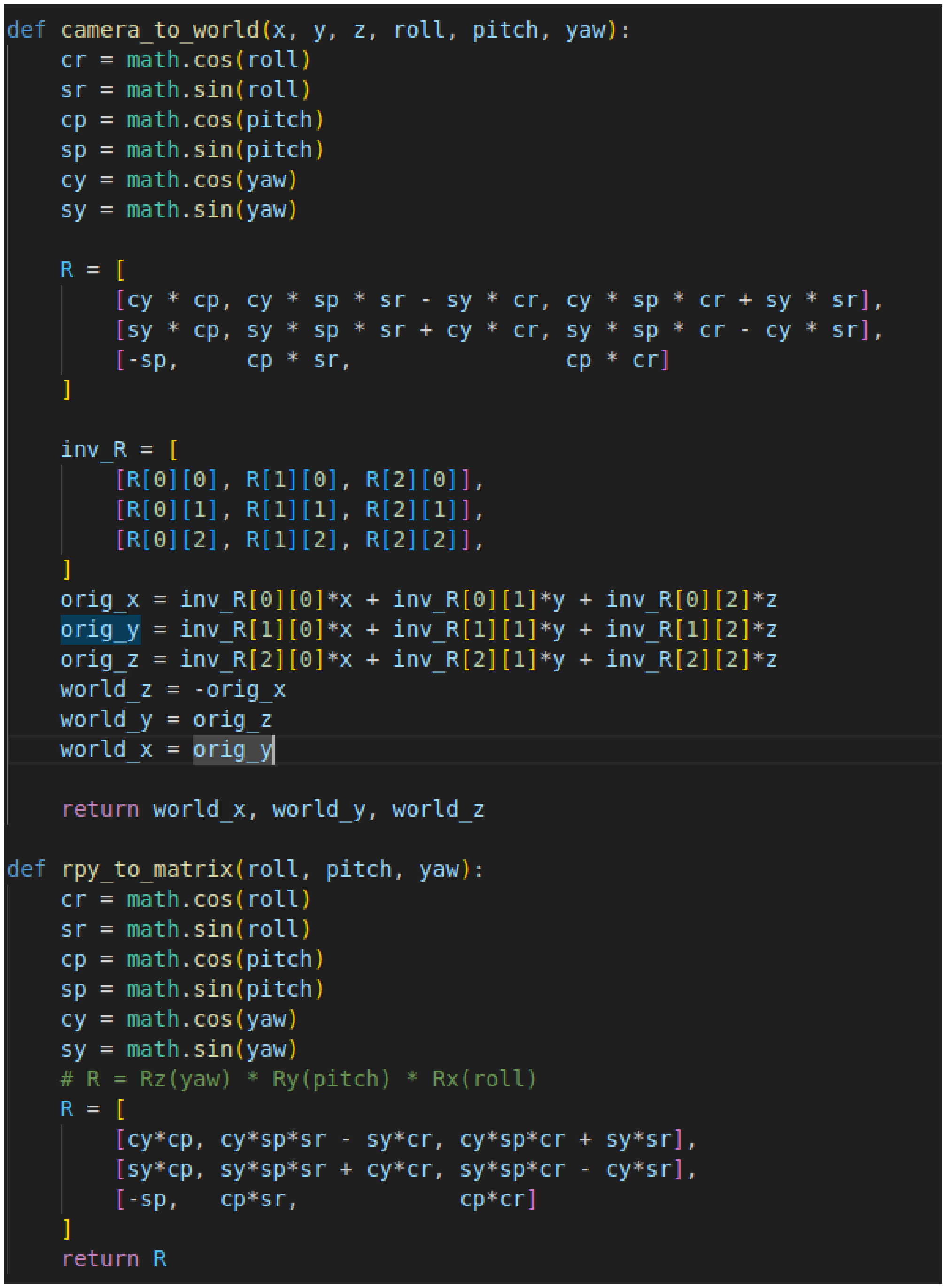
The goal of this project is to identfy the top surface of any number of objects, and estimate the shape of the surface as a 2D polygon. The polygon will be used to plan a grasp with the best quality metrics to pick up the object.



Step 1: Enviornment Setup

This project contains 5 ROS packages. A launch file at the root of the **TopSurface-Grasping** folder which is not part of a specific ROS2 package launches all the necessary nodes.

First we open a gazebo world with a table, 2 coke cans, banana, and mustard bottle in it. Then we spawn in the camera above the table looking at an angle towards the objects. To keep a global world frame we can use to identify what the 'top' of surfaces is, we use a static transform publisher to publish a transform from the camera frame to the world frame.

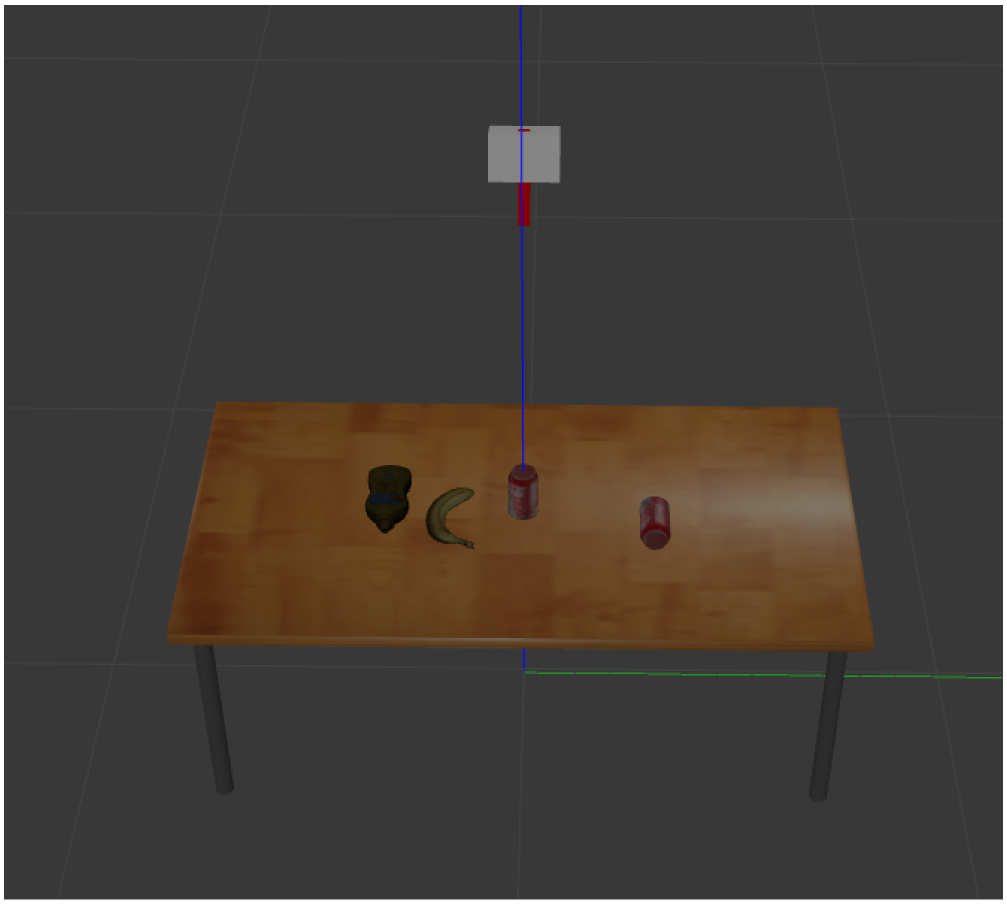


```python
def camera_to_world(x, y, z, roll, pitch, yaw):
    cr = math.cos(roll)
    sr = math.sin(roll)
    cp = math.cos(pitch)
    sp = math.sin(pitch)
    cy = math.cos(yaw)
    sy = math.sin(yaw)

    R = [
        [cy * cp, cy * sp * sr - sy * cr, cy * sp * cr + sy * sr],
        [sy * cp, sy * sp * sr + cy * cr, sy * sp * cr - cy * sr],
        [-sp,     cp * sr,                cp * cr]
    ]

    inv_R = [
        [R[0][0], R[1][0], R[2][0]],
        [R[0][1], R[1][1], R[2][1]],
        [R[0][2], R[1][2], R[2][2]],
    ]
    orig_x = inv_R[0][0]*x + inv_R[0][1]*y + inv_R[0][2]*z
    orig_y = inv_R[1][0]*x + inv_R[1][1]*y + inv_R[1][2]*z
    orig_z = inv_R[2][0]*x + inv_R[2][1]*y + inv_R[2][2]*z
    world_z = -orig_x
    world_y = orig_z
    world_x = orig_y

    return world_x, world_y, world_z

def rpy_to_matrix(roll, pitch, yaw):
    cr = math.cos(roll)
    sr = math.sin(roll)
    cp = math.cos(pitch)
    sp = math.sin(pitch)
    cy = math.cos(yaw)
    sy = math.sin(yaw)
    # R = Rz(yaw) * Ry(pitch) * Rx(roll)
    R = [
        [cy*cp, cy*sp*sr - sy*cr, cy*sp*cr + sy*sr],
        [sy*cp, sy*sp*sr + cy*cr, sy*sp*cr - cy*sr],
        [-sp,   cp*sr,            cp*cr]
    ]
    return R
```

We use these helper functions in the launch file to convert from camera coordinates to world coordinates, and keep the world frame aligned with the table surface.

With that we have a gazebo world with objects in it, and a camera looking at the objects, where the camera publishes RGB and depth information to ROS topics.

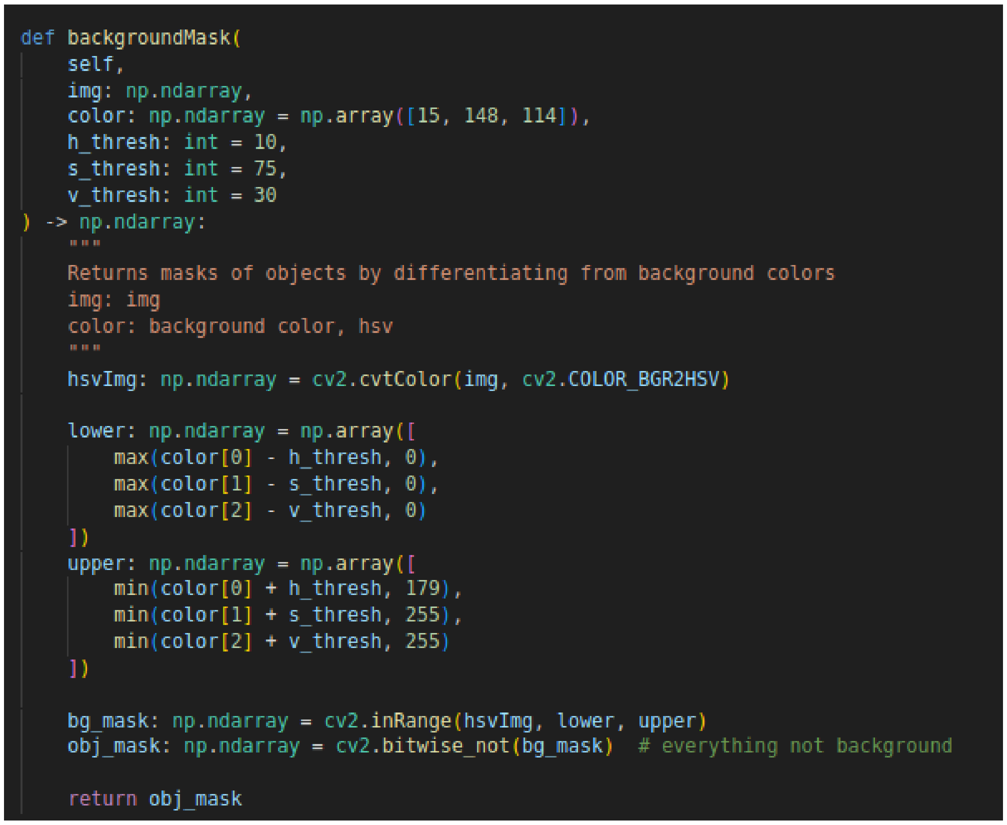
Finally, we have a grasp pipeline manager node which will manage the flow of data and publish debug information to topics we can view in rviz. The pipeline manager will call services on the segmentation and geometry extraction nodes.



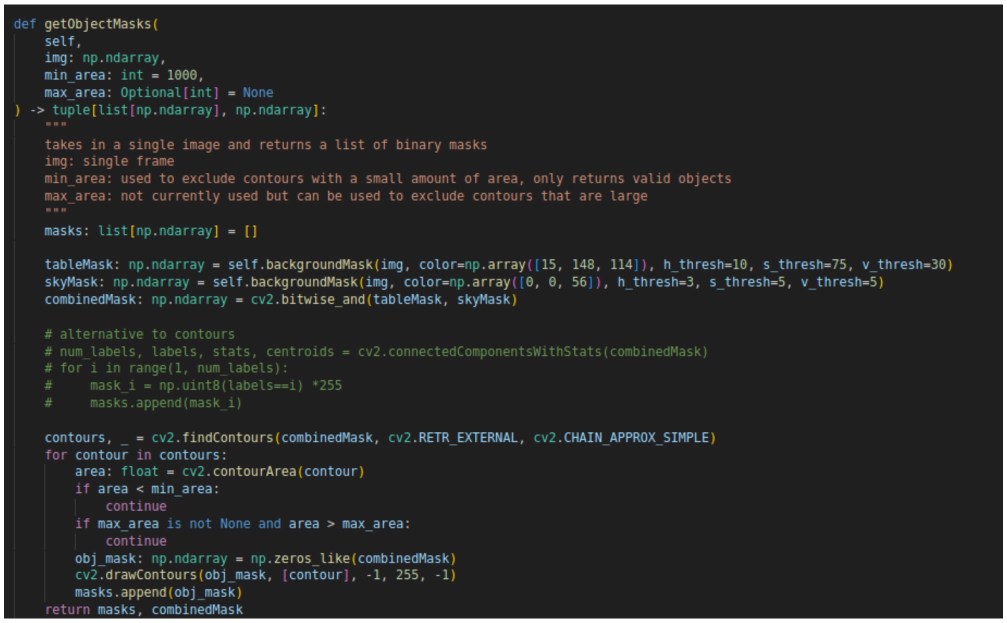
Step 2: Image Processing

Given an RGB image, we need to detect the objects in the image, and create a mask for each object. The `SegmentationNode` has a service `segment_objects` which receives an RGB image from the pipeline manager, and returns a list of masks for each detected object. It processes the image by masking out background regions like the table and sky to isolate the objects. Then contours are found in the masked image, and if the area is large enough, a mask is created for the object.

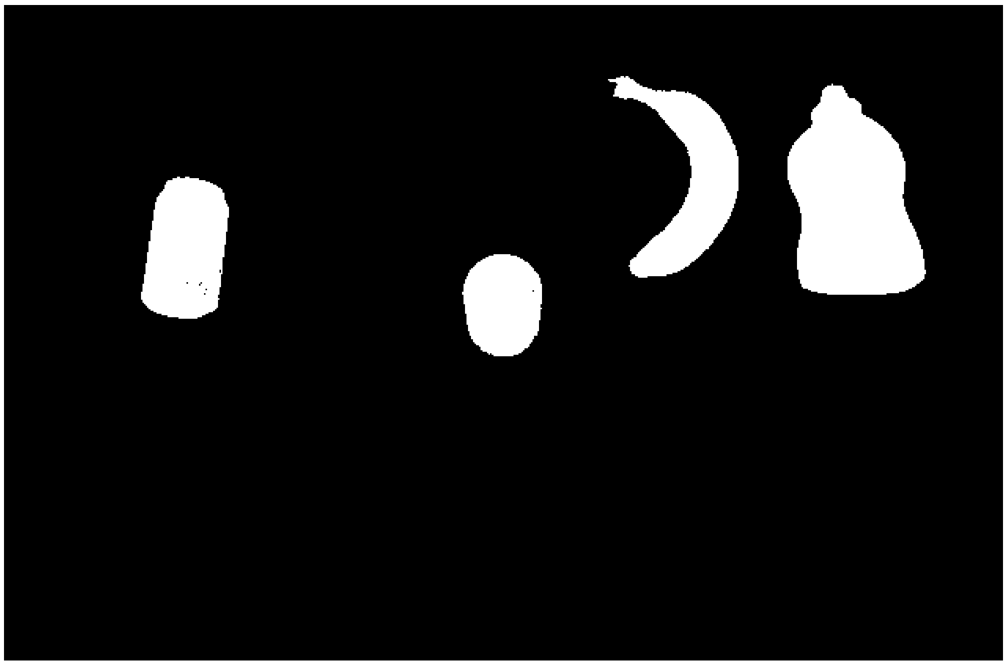
Code for getting the masks:



```python
def backgroundMask(
    self,
    img: np.ndarray,
    color: np.ndarray = np.array([15, 148, 114]),
    h_thresh: int = 10,
    s_thresh: int = 75,
    v_thresh: int = 30
) -> np.ndarray:
    """
    Returns masks of objects by differentiating from background colors
    img: img
    color: background color, hsv
    """
    hsvImg: np.ndarray = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    lower: np.ndarray = np.array([
        max(color[0] - h_thresh, 0),
        max(color[1] - s_thresh, 0),
        max(color[2] - v_thresh, 0)
    ])
    upper: np.ndarray = np.array([
        min(color[0] + h_thresh, 179),
        min(color[1] + s_thresh, 255),
        min(color[2] + v_thresh, 255)
    ])

    bg_mask: np.ndarray = cv2.inRange(hsvImg, lower, upper)
    obj_mask: np.ndarray = cv2.bitwise_not(bg_mask)  # everything not background

    return obj_mask
```

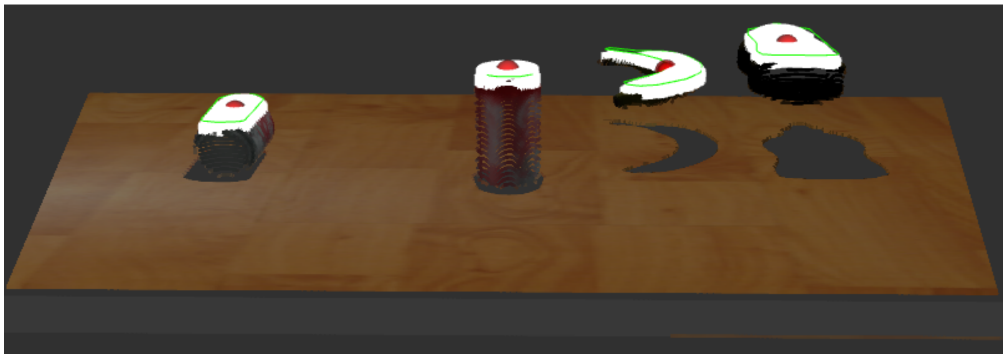


```python
def getObjectMasks(
    self,
    img: np.ndarray,
    min_area: int = 1000,
    max_area: Optional[int] = None
) -> tuple[list[np.ndarray], np.ndarray]:
    """
    takes in a single image and returns a list of binary masks
    img: single frame
    min_area: used to exclude contours with a small amount of area, only returns valid objects
    max_area: not currently used but can be used to exclude contours that are large
    """
    masks: list[np.ndarray] = []

    tableMask: np.ndarray = self.backgroundMask(img, color=np.array([15, 148, 114]), h_thresh=10, s_thresh=75, v_thresh=30)
    skyMask: np.ndarray = self.backgroundMask(img, color=np.array([0, 0, 56]), h_thresh=3, s_thresh=5, v_thresh=5)
    combinedMask: np.ndarray = cv2.bitwise_and(tableMask, skyMask)

    # alternative to contours
    # num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(combinedMask)
    # for i in range(1, num_labels):
    #     mask_i = np.uint8(labels==i) *255
    #     masks.append(mask_i)

    contours, _ = cv2.findContours(combinedMask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    for contour in contours:
        area: float = cv2.contourArea(contour)
        if area < min_area:
            continue
        if max_area is not None and area > max_area:
            continue
        obj_mask: np.ndarray = np.zeros_like(combinedMask)
        cv2.drawContours(obj_mask, [contour], -1, 255, -1)
        masks.append(obj_mask)
    return masks, combinedMask
```



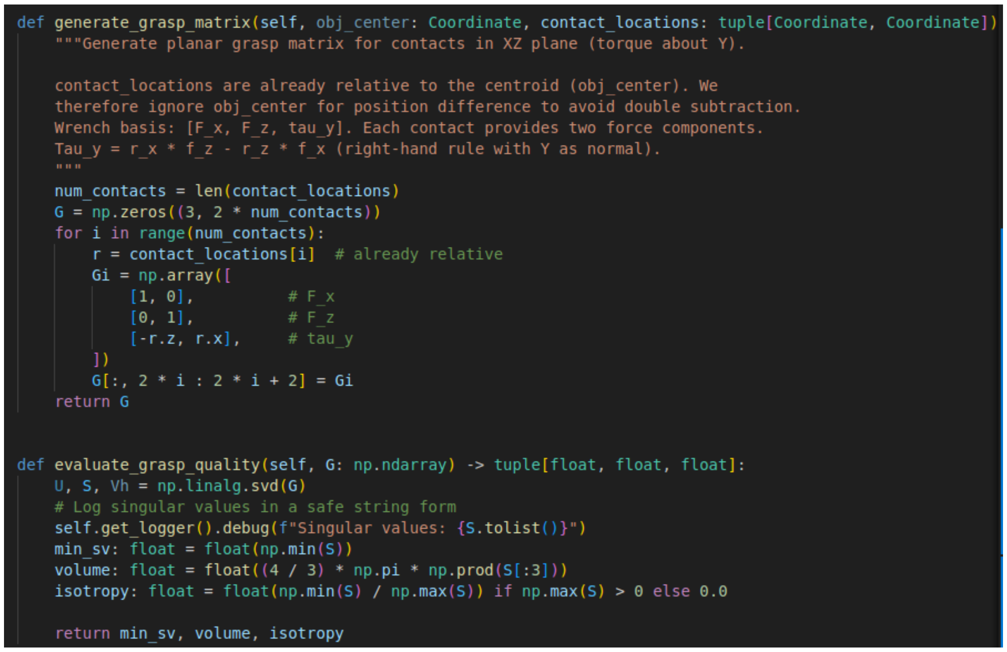
Step 3: Geometry Extraction

The `GeometryExtractionNode` has a service `extract_object_geometry` which receives a list of masks and a point cloud and returns a polygon representing the top surface and the center of mass projected onto the top surface. The node first masks the point cloud to isolate only relevant points of the object, then selects all the points with the minimum y-value (world coordinate system makes this the top surface), and projects these points into a 2D plane. Then using OpenCV contour detection, the top surface polygon is found. The center of mass is calculated by averaging the x and z coordinates of the top surface point cloud points, and using the minimum y-value for the y coordinate. The code callback for the service is too long to include in the report, but the code can be found in the submission files.



Step 4: Finding a Grasp

Once we have a polygon and center of mass, we can preform similar steps to previous homework assignments to find a grasp. We generate a list of grasp candiates on the object, and evaluate each candidate using grasp quality metrics. We find the minimum singular value, isotropy, and volume of ellipsoid metrics for each candidate, and select the candidate with the best metrics (if it has 2 metrics that are the best). Then these contact points can be visualized as markers in rviz.



```python
def generate_grasp_matrix(self, obj_center: Coordinate, contact_locations: tuple[Coordinate, Coordinate]):
    """Generate planar grasp matrix for contacts in XZ plane (torque about Y).

    contact_locations are already relative to the centroid (obj_center). We
    therefore ignore obj_center for position difference to avoid double subtraction.
    Wrench basis: [F_x, F_z, tau_y]. Each contact provides two force components.
    Tau_y = r_x * f_z - r_z * f_x (right-hand rule with Y as normal).
    """
    num_contacts = len(contact_locations)
    G = np.zeros((3, 2 * num_contacts))
    for i in range(num_contacts):
        r = contact_locations[i]  # already relative
        Gi = np.array([
            [1, 0],          # F_x
            [0, 1],          # F_z
            [-r.z, r.x],     # tau_y
        ])
        G[:, 2 * i : 2 * i + 2] = Gi
    return G


def evaluate_grasp_quality(self, G: np.ndarray) -> tuple[float, float, float]:
    U, S, Vh = np.linalg.svd(G)
    # Log singular values in a safe string form
    self.get_logger().debug(f"Singular values: {S.tolist()}")
    min_sv: float = float(np.min(S))
    volume: float = float((4 / 3) * np.pi * np.prod(S[:3]))
    isotropy: float = float(np.min(S) / np.max(S)) if np.max(S) > 0 else 0.0

    return min_sv, volume, isotropy
```

Code for grasp candidates:

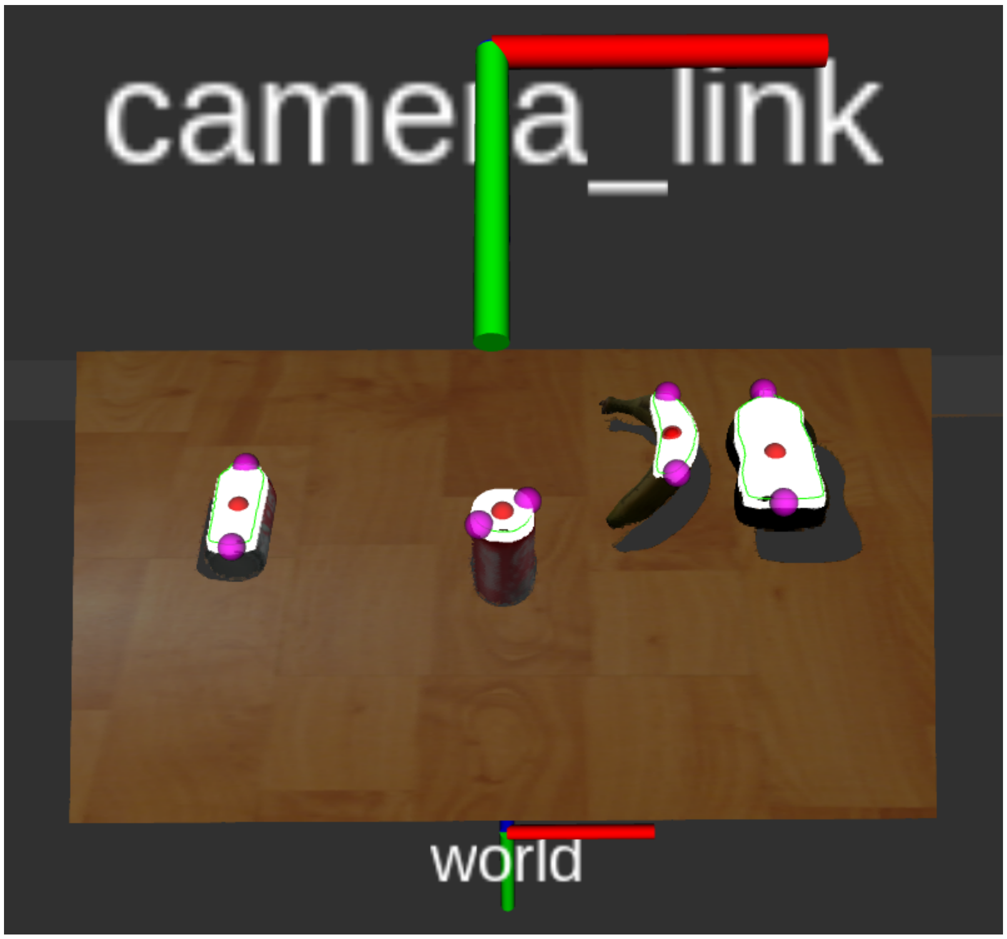
```python
def getNextPoint(point, centroid, geometry, thetaDelta):
    """
    point = current point
    centroid: center of polygon
    geometry:
    thetaDelta: increments of theta to change by in radians
    """
    points_list = list(geometry.points)

    ## Method 2
    if abs(point.x - centroid.x) < 0.00001: # vertical line
        if point.y > centroid.y:
            newTheta = math.pi/2 + thetaDelta
        else:
            newTheta = 3*math.pi/2 + thetaDelta
    else:
        m_old = (point.y - centroid.y)/(point.x - centroid.x)
        newTheta = math.atan(m_old) + thetaDelta

    m_new = math.tan(newTheta)
    b_new = centroid.y-m_new*centroid.x

    if m_new > 100000000000: # vertical line through centroid
        dummyPoint = Point32(centroid.x, centroid.y+10)
    else:
        x_dummy = centroid.x + 10
        y_dummy = m_new*x_dummy + b_new

        print(f'm: {m_new}\tb: {b_new}\tx_dummy: {x_dummy}\ty_dummy: {y_dummy}')

        dummyPoint = Point32(x_dummy, y_dummy)

    intersectingPoints = []
    for i in range(len(points_list)-1):
        curr = points_list[i]
        next = points_list[i+1]

        ip = getIntersectingPoint(rayPoints=(dummyPoint, centroid), sidePoints=(curr, next))
        if ip is not None:
            intersectingPoints.append(ip)

    min_diff = math.inf
    print(f'intersection points: {intersectingPoints}')
    if len(intersectingPoints) > 0:
        for intersection in intersectingPoints:
            # calculate the difference in theta and find the one that == delta theta
            angleInter = math.atan2(intersection.y - centroid.y, intersection.x - centroid.x)
            anglePt = math.atan2(point.y - centroid.y, point.x - centroid.x)
            # CCW angle difference
            diff = (angleInter - anglePt) % (2*math.pi)
            delta_error = abs(diff - thetaDelta)
            # print(f'angleInter: {angleInter}\tanglePt: {anglePt}\tdiff: {diff} \totherDiff: {(abs((abs(angleInter) - abs(anglePt))) - thetaDelta)}')
            if delta_error < min_diff:
                min_diff = delta_error
                closest_point = intersection

        return closest_point
    else:
        return None
```

```python
def getOpposite(currPt, centroid, geometry):
    """
    return the opposite point from the current point
    """

    points_list = list(geometry.points)
    points_list.append(Point32(0.0, 0.0))
    intersectingPoints = []
    for i in range(len(points_list)-1):
        curr = points_list[i]
        next = points_list[i+1]
        if currPt == curr or currPt == next:
            # print(f'skipped: {curr}, {next}')
            continue

        # print(f"check intersection: {((pt.x, pt.y), (centroid.x, centroid.y)), ((curr.x, curr.y), (next.x, next.y))}")

        point = getIntersectingPoint(rayPoints=(currPt, centroid), sidePoints=(curr, next))
        if point is not None:
            intersectingPoints.append(point)

    ## choose opposite point from intersections
    if len(intersectingPoints) > 0:
        dist_pt_to_centroid = math.hypot(centroid.x - currPt.x, centroid.y - currPt.y)
        farthest_intersection = None
        max_dist = 0.0

        for intersection in intersectingPoints:
            dist_pt_to_intersection = math.hypot(intersection.x - currPt.x, intersection.y - currPt.y)

            # Only consider intersections beyond the centroid
            if dist_pt_to_intersection > dist_pt_to_centroid:
                # Choose the farthest valid intersection (opposite side)
                if dist_pt_to_intersection > max_dist:
                    max_dist = dist_pt_to_intersection
                    farthest_intersection = intersection

        return farthest_intersection  # None if no valid intersection found
    else:
        return None
```

```
def getIntersectingPoint(rayPoints, sidePoints):
    (pt, centroid) = rayPoints
    (curr, next) = sidePoints
    # print(f"pt: {pt}")
    print(f'intersection between {((pt.x, pt.y), (centroid.x, centroid.y)), ((curr.x, curr.y), (next.x, next.y))}')

    if pt == curr: # or pt == next:
        print(f'point == curr')
        return None

    a_ray = centroid.y - pt.y
    b_ray = pt.x - centroid.x
    c_ray = a_ray*pt.x + b_ray*pt.y

    a_side = next.y - curr.y
    b_side = curr.x - next.x
    c_side = a_side*curr.x + b_side*curr.y


    ## find intersection
    d = a_ray*b_side - a_side*b_ray
    if abs(d) < 1e-9:
        print(f'lines are parallel')
        return None
    else:
        x = (b_side*c_ray - b_ray*c_side)/d
        y = (a_ray*c_side - a_side*c_ray)/d
        # print(f"{(x,y)}")

    if x >= min(curr.x, next.x) and x <= max(curr.x, next.x) and y >= min(curr.y, next.y) and y <= max(curr.y, next.y):

        intersection = Point32(x=x, y=y)
        return intersection
    else:
        print(f'does not fit in intersection')
        return None
```

```python
def get_all_possible_grasps(geometry, centroid, thetaDelta):
    """calculate all possible grasp points"""

    graspPoints = []
    theta = 0

    # start somewhere, draw a line through the center to the other side to get second point
    points_list = list(geometry.points)
    print(f"points list: {points_list}")
    maxDist = 0
    farthest = None
    for p in points_list:
        dist = math.dist((centroid.x, centroid.y), (p.x, p.y))
        if dist > maxDist:
            maxDist = dist
            farthest = p

    currPoint = farthest

    while theta < 2*math.pi:
        print(f'\ncurrPoint: {currPoint}\ttheta: {theta}')

        # get opposite point
        oppPoint = getOpposite(currPoint, centroid, geometry)
        print(f"opposite: {oppPoint}\n")
        if oppPoint is None:
            print(f"opposite is None")
            return
        graspPoints.append((currPoint, oppPoint))

        theta += thetaDelta
        currPoint = getNextPoint(currPoint, centroid, geometry, thetaDelta)
        if currPoint == None:
            print("currentPoint is None")
            return

    # get rid of duplicates
    unique_grasps = []

    for p1, p2 in graspPoints:
        is_duplicate = False

        for o1, o2 in unique_grasps:
            d1 = math.dist((o1.x, o1.y), (p1.x, p1.y))
            d2 = math.dist((o2.x, o2.y), (p2.x, p2.y))

            # Check if both points match (or reversed match)
            if (d1 < 0.001 and d2 < 0.001) or \
            (math.dist((o1.x, o1.y), (p2.x, p2.y)) < 0.001 and \
                math.dist((o2.x, o2.y), (p1.x, p1.y)) < 0.001):
                is_duplicate = True
                break

        if not is_duplicate:
            unique_grasps.append((p1, p2))

    ## Debugging
    print(f'calculated {len(unique_grasps)} total grasp positions')

    for (p1, p2) in unique_grasps:
        print(p1, p2)

    return unique_grasps
```

Discussion



We had resulting grasps with the following metrics:

- **Object 1 (Standing Coke Can):**
  Best grasp with $\min\_sv = 0.0428$, $volume = 0.3588$, $isotropy = 0.0303$
- **Object 2 (Lying Down Coke Can):**
  Best grasp with $\min\_sv = 0.0870$, $volume = 0.7286$, $isotropy = 0.0615$
- **Object 3 (Mustard Bottle):**
  Best grasp with $\min\_sv = 0.1209$, $volume = 1.0130$, $isotropy = 0.0855$
- **Object 4 (Banana):**
  Best grasp with $\min\_sv = 0.0892$, $volume = 0.7476$, $isotropy = 0.0631$

As seen in the images, the grasps are reasonably placed and could sufficently pick up the objects. Our best grasp was on the mustard bottle, which is the object with the largest top surface area. The grasps on the two coke cans are also good, with the grasp on the lying down can being better than the standing can. The grasp on the banana is the worst, which makes sense as the top surface is very small and curved. We also are modeling the grasps as hard contacts, which could result in these grasps not being the exact ideal grasps a human would use.