```python
from controller import Supervisor
import math
import numpy as np
from heapq import heappush, heappop
from collections import defaultdict
from os.path import exists
import os

# Clamp a value into a closed interval
def clamp(v, lo, hi):
    return max(lo, min(hi, v))

# Wrap an angle into [-pi, pi] to keep heading errors stable
def wrap_pi(a):
    while a > math.pi:
        a -= 2 * math.pi
    while a < -math.pi:
        a += 2 * math.pi
    return a

```

This is simply the various imports needed for the python code to work along with 2 simple math functions that make sure a given value is in-between a given low and high value, and another function that makes sure that given angles remain between -pi and +pi.

```python
# Initialize the Webots supervisor and simulation timing
robot = Supervisor()
timestep = int(robot.getBasicTimeStep())

# Configure wheel motors for velocity control
motor_left = robot.getDevice('wheel_left_joint')
motor_right = robot.getDevice('wheel_right_joint')
motor_left.setPosition(float('inf'))
motor_right.setPosition(float('inf'))

# Enable camera and object recognition for jar detection
camera = robot.getDevice("camera")
camera.enable(timestep)
camera.recognitionEnable(timestep)

# Enable GPS and compass for localization and yaw estimation
compass = robot.getDevice('compass')
compass.enable(timestep)
gps = robot.getDevice('gps')
gps.enable(timestep)

# Load arm torso head gripper motors if present on this robot model
MOTOR_NAMES = [
    'torso_lift_joint',
    'arm_1_joint', 'arm_2_joint', 'arm_3_joint', 'arm_4_joint', 'arm_5_joint', 'arm_
    'head_1_joint', 'head_2_joint',
    'gripper_left_finger_joint', 'gripper_right_finger_joint',
]
motor_handles = {}
for name in MOTOR_NAMES:
    try:
        motor_handles[name] = robot.getDevice(name)
    except:
        pass
```

From there there is a lot of this type of code, which is essentially just the activation and naming of various parts of the robot, such as the lidar, gps, and arm joints that will later allow you to manipulate them in the code.

```python
# Robot wheel limits and global speed parameters
WHEEL_MAX = 10.1523

# Navigation gains and speed limits for waypoint following
NAV_P_FWD = 1.4
NAV_P_TURN = 3.0
NAV_MAX_SPEED = 5.5

# Reduce forward speed when heading error is large
NAV_ALPHA_SLOWDOWN_START = 0.35
NAV_ALPHA_SLOWDOWN_FULL = 1.00
NAV_MIN_FWD_SCALE = 0.08

# Path following tolerance and counter waypoint acceptance radius
PATH_POINT_REACHED = 0.22
GOAL_REACHED_COUNTER = 0.80

# Search spin cadence while scanning for jars
SEARCH_SPIN_SPEED = 0.45
SEARCH_PAUSE_EVERY = 18
SEARCH_PAUSE_STEPS = 6

# World frame standoff distance used before starting the arm grasp sequence
TARGET_DIST = 1.45
DIST_EPS = 0.06
SLOWDOWN_RADIUS = 0.75

# Back away behavior if the robot gets too close to the jar during approach
TOO_CLOSE_EPS = 0.10
BACKUP_GAIN = 2.8
BACKUP_MAX_SCALE = 0.65

# Camera visual servo configuration for horizontal centering
CAM_HORIZ_AXIS = 'y'
CAM_HORIZ_SIGN = -1.0

# Camera centering thresholds and stability holds
CAM_CENTER_EPS = 0.06
CAM_CENTER_HOLD_TICKS = 6
```

Afterwards there is a long list of various constants used throughout different parts of the process all at the beginning for easy tweaking to fix or change various parts. All of them are pretty self explanatory except for ones ending in EPS which stands for Epsilon which is the accepted error range for that part of the process.

```python
# Arm poses for travel picking and post grasp transport
ARM_SAFE_POSE = {
    'arm_1_joint': 0.71,
    'arm_2_joint': 1.02,
    'arm_3_joint': -2.815,
    'arm_4_joint': 1.011,
    'arm_5_joint': 0.0,
    'arm_6_joint': 0.0,
    'arm_7_joint': 0.0,
    'head_1_joint': 0.0,
    'head_2_joint': 0.0,
}

ARM_PICK_POSE = {
    'arm_1_joint': 1.57,
    'arm_2_joint': 0.95,
    'arm_3_joint': 0.0,
    'arm_4_joint': 1.0,
    'arm_5_joint': 0.00,
    'arm_6_joint': 0.00,
    'arm_7_joint': -1.57,
}

ARM_POST_CLOSE_POSE = {
    'arm_1_joint': 1.57,
    'arm_2_joint': 0.95,
    'arm_3_joint': 0.0,
    'arm_4_joint': -0.32,
    'arm_5_joint': 0.00,
    'arm_6_joint': 0.00,
    'arm_7_joint': -1.57,
}
```

Next we have the 3 different arm poses that the robot uses during the process,the safe pose is used while moving whilst not holding a jar, the pick pose lowers the hand to pick up the jar, and the close jar lifts up the jar in order to avoid collision while moving it. The process to which I attained each of these values was simply trial and error until eventually I found a configuration which worked.

```python
# Stop the base immediately
def stop_base():
    motor_left.setVelocity(0.0)
    motor_right.setVelocity(0.0)

# Hold the base stopped for a fixed number of control steps
def force_stop_for_steps(n):
    for _ in range(n):
        stop_base()
        robot.step(timestep)
```

Here are 2 simple functions which are quite self explanatory, they are used to make sure the robot comes to a stop during certain steps to avoid issues in subsequent steps. For example after the robot centers itself on a jar it comes to a stop before then moving forward in order to avoid any issues caused by still being in motion when the next step starts.

```python
# Command a symmetric gripper opening
def set_gripper_open(width):
    width = clamp(width, 0.0, GRIPPER_MAX_OPEN)
    if 'gripper_left_finger_joint' in motor_handles:
        motor_handles['gripper_left_finger_joint'].setPosition(width)
    if 'gripper_right_finger_joint' in motor_handles:
        motor_handles['gripper_right_finger_joint'].setPosition(width)

# Open the gripper and hold briefly to settle motion
def open_gripper(width=GRIPPER_OPEN_GRASP, hold_steps=30):
    stop_base()
    set_gripper_open(width)
    for _ in range(hold_steps):
        stop_base()
        robot.step(timestep)

# Close the gripper and hold briefly to complete the grasp
def close_gripper(hold_steps=60):
    stop_base()
    if 'gripper_left_finger_joint' in motor_handles:
        motor_handles['gripper_left_finger_joint'].setPosition(GRIPPER_CLOSED)
    if 'gripper_right_finger_joint' in motor_handles:
        motor_handles['gripper_right_finger_joint'].setPosition(GRIPPER_CLOSED)
    for _ in range(hold_steps):
        stop_base()
        robot.step(timestep)
```

Some more helper functions, these work with the gripper opening and closing it, ensuting equal value for each finger and taking breaks in-between to ensure consistent behavior.

```python
# Read absolute finger force feedback values
def get_finger_forces():
    fL = 0.0
    fR = 0.0
    if 'gripper_left_finger_joint' in motor_handles:
        try:
            fL = abs(float(motor_handles['gripper_left_finger_joint'].getForceFeedba
        except:
            fL = 0.0
    if 'gripper_right_finger_joint' in motor_handles:
        try:
            fR = abs(float(motor_handles['gripper_right_finger_joint'].getForceFeedb
        except:
            fR = 0.0
    return fL, fR
```

This function also works with the gripper simply returning the absolute force each finger is experiencing at that time.

```python
# Move the arm to a named pose while the base is held stationary
def move_arm_pose(pose_dict, hold_steps=40):
    stop_base()
    for j, v in pose_dict.items():
        if j in motor_handles:
            motor_handles[j].setPosition(v)
    for _ in range(hold_steps):
        stop_base()
        robot.step(timestep)

# Set torso height within limits and hold until it settles
def set_torso_lift(pos, hold_steps=60):
    if 'torso_lift_joint' not in motor_handles:
        print("WARNING torso_lift_joint motor not found cannot adjust torso height")
        return
    pos = clamp(pos, TORSO_MIN, TORSO_MAX)
    motor_handles['torso_lift_joint'].setPosition(pos)
    for _ in range(hold_steps):
        stop_base()
        robot.step(timestep)
```

More helper functions were used to move the arm and torso of the robot, the set_torso_lift used to match the height of the arm to the jar of jam.

```python
# Convert a world pose to an occupancy grid pixel index
def world2map(xw, yw):
    world_x_min = -2.25
    world_x_max = 2.25
    world_y_min = -3.85
    world_y_max = 1.81
    px = int((xw - world_x_min) / (world_x_max - world_x_min) * 225)
    py = int((yw - world_y_min) / (world_y_max - world_y_min) * 284)
    py = 283 - py
    px = max(0, min(px, 224))
    py = max(0, min(py, 283))
    return [px, py]

# Convert a map pixel index back into a world coordinate
def map2world(px, py):
    world_x_min = -2.25
    world_x_max = 2.25
    world_y_min = -3.85
    world_y_max = 1.81
    xw = world_x_min + (px / 225) * (world_x_max - world_x_min)
    yw = world_y_min + ((283 - py) / 284) * (world_y_max - world_y_min)
    return (xw, yw)
```

These are helper functions used to convert between pixel coordinates and world coordinates and back again. This is used with the cspace map and A* navigation.

```python
# Compute A star heuristic cost in map pixel space
def heuristic(u, goal):
    return math.sqrt((goal[0] - u[0]) ** 2 + (goal[1] - u[1]) ** 2)

# Generate neighbors for A star traversal while respecting obstacles
def getNeighbors(u, grid):
    neighbors = []
    for i in range(-1, 2):
        for j in range(-1, 2):
            if i == 0 and j == 0:
                continue
            cand = (u[0] + i, u[1] + j)
            if (0 <= cand[0] < len(grid) and
                0 <= cand[1] < len(grid[0]) and
                not grid[cand[0], cand[1]]):
                neighbors.append((1.0, cand))
    return neighbors

# Plan a path in map space using A star and return a downsampled list of world coord
def pathfinding(start_map, goal_map, grid):
    queue = [(heuristic(start_map, goal_map), start_map)]
    distances = defaultdict(lambda: float("inf"))
    distances[start_map] = 0
    visited = set()
    parent = {}

    max_iterations = 15000
    it = 0
    while queue and it < max_iterations:
        it += 1
```

The next few functions all relate to A* navigation of the cspace map as implemented in previous assignments.

```python
# Compute yaw from compass and convert into the controller heading convention
def get_yaw():
    comp = compass.getValues()
    yaw = math.atan2(comp[1], comp[0])
    yaw -= math.pi / 2
    yaw = -yaw
    return yaw

# Drive toward a point using proportional control on distance and heading
def nav_drive_to_point(tx, ty, speed_scale=1.0):
    xw, yw, _ = gps.getValues()
    dist = math.sqrt((xw - tx) ** 2 + (yw - ty) ** 2)

    yaw = get_yaw()
    target_angle = math.atan2(ty - yw, tx - xw)
    alpha = wrap_pi(target_angle - yaw)

    a = abs(alpha)
    if a <= NAV_ALPHA_SLOWDOWN_START:
        fwd_scale = 1.0
    elif a >= NAV_ALPHA_SLOWDOWN_FULL:
        fwd_scale = NAV_MIN_FWD_SCALE
    else:
        t = (a - NAV_ALPHA_SLOWDOWN_START) / max(1e-6, (NAV_ALPHA_SLOWDOWN_FULL - NA
        fwd_scale = (1.0 - t) + t * NAV_MIN_FWD_SCALE

    v_fwd = dist * NAV_P_FWD * speed_scale * fwd_scale
    v_turn = alpha * NAV_P_TURN * speed_scale

    vl = clamp(v_fwd - v_turn, -NAV_MAX_SPEED, NAV_MAX_SPEED)
    vr = clamp(v_fwd + v_turn, -NAV_MAX_SPEED, NAV_MAX_SPEED)

    motor_left.setVelocity(clamp(vl, -WHEEL_MAX, WHEEL_MAX))
    motor_right.setVelocity(clamp(vr, -WHEEL_MAX, WHEEL_MAX))
    return dist, alpha

```

Get yaw is quite self explanatory, and nav drive to point drives towards a given waypoint.

```
519 # Rotation and translation helpers for building camera to base transforms
520 def rotation_x(theta):
521     return np.array([[1, 0, 0, 0],
522                      [0, np.cos(theta), -np.sin(theta), 0],
523                      [0, np.sin(theta), np.cos(theta), 0],
524                      [0, 0, 0, 1]])
525
526 def rotation_y(theta):
527     return np.array([[np.cos(theta), 0, np.sin(theta), 0],
528                      [0, 1, 0, 0],
529                      [-np.sin(theta), 0, np.cos(theta), 0],
530                      [0, 0, 0, 1]])
531
532 def rotation_z(theta):
533     return np.array([[np.cos(theta), -np.sin(theta), 0, 0],
534                      [np.sin(theta), np.cos(theta), 0, 0],
535                      [0, 0, 1, 0],
536                      [0, 0, 0, 1]])
537
538 def translation(x, y, z):
539     return np.array([[1, 0, 0, x],
540                      [0, 1, 0, y],
541                      [0, 0, 1, z],
542                      [0, 0, 0, 1]])
```

The comment says it all

```python
# Compute camera frame to base frame transform using torso lift and head joints
def get_camera_to_base_transform():
    lift_value = encoders['torso_lift_joint'].getValue() if 'torso_lift_joint' in en
    head_pan = encoders['head_1_joint'].getValue() if 'head_1_joint' in encoders els
    head_tilt = encoders['head_2_joint'].getValue() if 'head_2_joint' in encoders el

    T0_1 = translation(0, 0, 0.6 + lift_value)
    Tt = translation(0.182, 0, 0)
    Trz = rotation_z(head_pan)
    T1_2 = Trz @ Tt
    T2_3 = rotation_y(head_tilt) @ rotation_x(-math.pi / 2)
    return T0_1 @ T1_2 @ T2_3

# Transform a point from camera coordinates into base coordinates
def transform_camera_to_base(p_cam):
    T = get_camera_to_base_transform()
    p = np.array([p_cam[0], p_cam[1], p_cam[2], 1.0])
    out = T @ p
    return out[:3] / out[3]

# Convert a camera observed object position into a world coordinate estimate
def compute_object_position_world(p_cam):
    p_base = transform_camera_to_base(p_cam)
    gx, gy, gz = gps.getValues()
    yaw = get_yaw()
    R = np.array([[np.cos(yaw), -np.sin(yaw), 0],
                  [np.sin(yaw),  np.cos(yaw), 0],
                  [0, 0, 1]])
    p_world_2d = R @ np.array([p_base[0], p_base[1], 0])
    return [gx + p_world_2d[0], gy + p_world_2d[1], gz + p_base[2]]

```

Next is a series of helper functions that help convert what the camera sees to world position data, this is used to hone in on the jars, and determine the robots distance from them for smooth consistent pickups.

```python
# Model string matching helper for jar recognition objects
def _jar_match(model_str: str) -> bool:
    m = (model_str or "").lower()
    return ("jar" in m) or ("jam" in m)

# Safe object id extraction for recognition objects
def _safe_get_id(obj):
    try:
        return int(obj.getId())
    except:
        return None

# Extract image features used for centering and tracking continuity
def _get_obj_image_features(obj):
    z = float(obj.getPosition()[2])

    cam_w = camera.getWidth()
    cam_h = camera.getHeight()

    # Prefer image based features when available
    if hasattr(obj, "getPositionOnImage") and hasattr(obj, "getSizeOnImage"):
        cx, cy = obj.getPositionOnImage()
        bw, bh = obj.getSizeOnImage()

        nx = (float(cx) - (cam_w / 2.0)) / (cam_w / 2.0)
        ny = (float(cy) - (cam_h / 2.0)) / (cam_h / 2.0)

        ex = nx if (CAM_HORIZ_AXIS == 'x') else ny
        ex *= CAM_HORIZ_SIGN

        area = float(bw) * float(bh)
        return ex, float(cx), float(cy), area, z
```

What follows next is another list of helper functions that helps detect the jar, if it is detecting multiple jars picking one of them, and then being able to center the robot on the jar making sure it remains straight ahead.

```python
# Resample the jar world pose multiple times and return a robust median estimate
def acquire_and_refine_locked_target(samples=10, prefer_id=None, prefer_model=None):
    stop_base()
    xs, ys, zs = [], [], []
    last_cx = None
    last_cy = None
    last_z = None

    for _ in range(samples):
        robot.step(timestep)
        stop_base()
        obj, model, oid, ex, cx, cy, z = get_locked_or_best_jar(
            prefer_id=prefer_id, prefer_model=prefer_model,
            last_cx=last_cx, last_cy=last_cy, last_z=last_z
        )
        if obj is None:
            continue
        last_cx, last_cy, last_z = cx, cy, z
        p_cam = list(obj.getPosition())
        world = compute_object_position_world(p_cam)
        xs.append(float(world[0]))
        ys.append(float(world[1]))
        zs.append(float(world[2]))

    if not xs:
        return None
    xs.sort()
    ys.sort()
    zs.sort()
    return (xs[len(xs)//2], ys[len(ys)//2], zs[len(zs)//2])
```

This helper function gets multiple readings of the jar's world position, takes the median then uses that info to determine the torso height for the robot. It ultimately barely affects the torso height of the robot, and has little if any effect on the actual run, but it can help account for possible extenuating circumstances.

```python
# Backup primitive that drives backward a fixed distance while maintaining heading
def backup_distance_straight(distance_m, reverse_speed=BACKUP_AFTER_CLOSE_SPEED):
    stop_base()
    force_stop_for_steps(5)

    x0, y0, _ = gps.getValues()
    yaw_ref = get_yaw()

    while robot.step(timestep) != -1:
        xw, yw, _ = gps.getValues()
        d = math.sqrt((xw - x0) ** 2 + (yw - y0) ** 2)
        if d >= distance_m:
            break

        yaw = get_yaw()
        heading_err = wrap_pi(yaw_ref - yaw)
        turn = clamp(BACKUP_AFTER_CLOSE_YAW_KP * heading_err,
                     -BACKUP_AFTER_CLOSE_TURN_MAX, BACKUP_AFTER_CLOSE_TURN_MAX)

        vl = -reverse_speed - turn
        vr = -reverse_speed + turn
        motor_left.setVelocity(clamp(vl, -WHEEL_MAX, WHEEL_MAX))
        motor_right.setVelocity(clamp(vr, -WHEEL_MAX, WHEEL_MAX))

    stop_base()
    force_stop_for_steps(10)

# Nudge forward primitive used during placement
def nudge_forward_distance(distance_m, forward_speed=PLACE_NUDGE_FWD_SPEED):
    stop_base()
    force_stop_for_steps(5)

    x0, y0, _ = gps.getValues()
    yaw_ref = get_yaw()

    while robot.step(timestep) != -1:
        xw, yw, _ = gps.getValues()
        d = math.sqrt((xw - x0) ** 2 + (yw - y0) ** 2)
        if d >= distance_m:
            break
```

These helper functions simply help move the robot forwards and backwards for various parts of the picking up and dropping process, such as the robot moving forward before dropping, moving backwards after dropping, and moving backwards before lifting the jar after gripping it.

```python
# Tight docking controller to hit an exact table point without driving into the tabl
def dock_to_exact_point(goalx, goaly):
    xw, yw, _ = gps.getValues()
    dist = math.sqrt((xw - goalx) ** 2 + (yw - goaly) ** 2)

    yaw = get_yaw()
    target_angle = math.atan2(goaly - yw, goalx - xw)
    alpha = wrap_pi(target_angle - yaw)

    # If close enough stop and hand off to yaw facing stage
    if dist <= TABLE_DOCK_EPS:
        stop_base()
        return True, dist, alpha

    # Rotate first when significantly misaligned
    if abs(alpha) > DOCK_ALIGN_ONLY_ALPHA:
        v = 0.0
        w = clamp(P_DOCK_TURN * alpha, -WHEEL_MAX * DOCK_MAX_SPEED_SCALE, WHEEL_MAX
    else:
        v = clamp(P_DOCK_FWD * dist, 0.10, WHEEL_MAX * DOCK_MAX_SPEED_SCALE)
        w = clamp(P_DOCK_TURN * alpha, -WHEEL_MAX * DOCK_MAX_SPEED_SCALE, WHEEL_MAX

    motor_left.setVelocity(clamp(v - w, -WHEEL_MAX, WHEEL_MAX))
    motor_right.setVelocity(clamp(v + w, -WHEEL_MAX, WHEEL_MAX))
    return False, dist, alpha

# Rotate in place until yaw matches the desired table placement yaw
def face_yaw(target_yaw):
    yaw = get_yaw()
    err = wrap_pi(target_yaw - yaw)
    turn = clamp(TABLE_YAW_KP * err, -WHEEL_MAX, WHEEL_MAX)
    motor_left.setVelocity(-turn)
    motor_right.setVelocity(turn)
    return err
```

These two functions perform the duties of moving the robot to a specific waypoint and facing a specific way.

```python
# Place sequence that nudges opens and wiggles to reduce drop jamming
def place_release_sequence():
    stop_base()
    force_stop_for_steps(10)

    # Lower torso before release so the jar is closer to the surface
    if 'torso_lift_joint' in motor_handles:
        set_torso_lift(PLACE_TORSO_DOWN, hold_steps=80)

    # Small controlled forward nudge to place closer to the intended point
    nudge_forward_distance(PLACE_NUDGE_FWD_DIST, forward_speed=PLACE_NUDGE_FWD_SPEED

    # Open gripper fully and hold to ensure release
    set_gripper_open(GRIPPER_MAX_OPEN)
    for _ in range(10):
        stop_base()
        robot.step(timestep)
    set_gripper_open(GRIPPER_MAX_OPEN)

    # Wiggle wrist joint to reduce sticking
    if 'arm_7_joint' in motor_handles:
        base = float(ARM_PICK_POSE.get('arm_7_joint', -1.57))
        for _ in range(PLACE_WIGGLE_CYCLES):
            motor_handles['arm_7_joint'].setPosition(base + PLACE_WIGGLE_DELTA)
            for _ in range(PLACE_WIGGLE_HOLD_STEPS):
                stop_base()
                robot.step(timestep)
            motor_handles['arm_7_joint'].setPosition(base - PLACE_WIGGLE_DELTA)
            for _ in range(PLACE_WIGGLE_HOLD_STEPS):
                stop_base()
                robot.step(timestep)
        motor_handles['arm_7_joint'].setPosition(base)

    # Keep commanding open during the hold window
    for _ in range(PLACE_OPEN_HOLD_STEPS):
        set_gripper_open(GRIPPER_MAX_OPEN)
        stop_base()
        robot.step(timestep)
```

This helper function works to drop the jar when specified with various checks and movements used to ensure the jar is actually dropped and does not remain in the hand of the robot.

```
844 # Load the occupancy grid map and inflate to create a navigation cspace
845 print(f"Map file {os.path.join(os.getcwd(), MAP_FILE)}")
846 if exists(MAP_FILE):
847     print("Map exists loading")
848     cspace = np.load(MAP_FILE)
849     print("Loaded map")
850 
851     if cspace.dtype != np.bool_:
852         cspace = (cspace > 0.5)
853 
854     print("Inflating cspace for safer navigation")
855     cspace_nav = inflate_cspace(cspace, radius_cells=12)
856     print("Inflation complete")
857 else:
858     print("ERROR cspace npy not found this controller expects an existing map")
859     while robot.step(timestep) != -1:
860         stop_base()
861 
862 # Move the arm to a safe travel posture before starting navigation and manipulation
863 print("Stopping base before moving arm to safe pose")
864 stop_base()
865 force_stop_for_steps(8)
866 
867 print("Moving arm to safe pose base stopped")
868 move_arm_pose(ARM_SAFE_POSE, hold_steps=80)
869 print("Arm safe pose command complete")
```

These initialize the map and position of the hand of the robot right before the main control loop finally starts.

```
871 # State machine labels for navigation search grasp transport docking and placement
872 STATE_PLAN_TO_COUNTER = "PLAN_COUNTER"
873 STATE_FOLLOW_TO_COUNTER = "FOLLOW_COUNTER"
874 STATE_SEARCH = "SEARCH_FOR_JAR"
875 STATE_ORIENT = "ORIENT_TO_JAR_VISUAL"
876 STATE_APPROACH = "APPROACH_TO_DIST_VISUAL"
877 STATE_ARM_POSE = "MOVE_ARM_PICK_POSE"
878 STATE_ARM_LEVEL = "ALIGN_ARM_LEVEL"
879 STATE_OPEN_AND_CREEP = "OPEN_AND_CREEP"
880 STATE_CLOSE = "CLOSE_GRIPPER"
881 
882 STATE_PLAN_TO_TABLE = "PLAN_TABLE"
883 STATE_FOLLOW_TO_TABLE = "FOLLOW_TABLE"
884 STATE_DOCK_TABLE = "DOCK_TABLE_EXACT"
885 STATE_FACE_PI = "FACE_TABLE_YAW"
886 STATE_PLACE_AT_TABLE = "PLACE_AT_TABLE"
887 STATE_DONE = "DONE"
```

Finally the main code loop is a state machine that goes through each of the states shown above. The way this works is PLAN_COUNTER and FOLLOW_COUNTER use the previous helper functions to compute an A* path and then follow the made path to the counter waypoint. SEARCH_FOR_JAR makes the robot spin in place with pauses until it finds a jar. ORIEN_TO_JAR_VISUAL makes the robot turn until the robot is centered on a jar within an acceptable error range. APPROACH_TO_DIST_VISUAL the robot will reverse if too close or move forward if too far, all the while making sure it is still centered on the jar.

MOVE_ARM_PICK_POSE moves the arm to the pick pose, and then ALIGN_ARM_LEVEL adjusts the torso to the right height. OPEN_AND_CREEP then slowly moves forwards the robot forwards keeping the robot centered through visual and force detected on the fingers, until both fingers at once detect a force at which point it shifts to CLOSE_GRIPPER, which does exactly as it says, backs up the robot, then lifts the jar up. PLAN_TABLE and FOLLOW_TABLE do what you would assume and DOCK_TABLE_EXACT makes sure the robot is very close to the table waypoint before continuing to reduce error. FACE_TABLE_YAW makes the robot face the table. PLACE_AT_TABLE then moves the robot forwards, releases the grip, returns arm to safe position, and then backs up. Finally the robot checks whether or not it has done all 3 at which point it will go to the DONE state where it simply stops, and if it isn't it repeats from the original state.