Jack Blackburn - jobl2177@colorado.edu
Kyle Paris - kyle.paris@colorado.edu
Ben Burkhalter - bebu6788@colorado.edu

Final Report

Components:

- **Mapping - Tier 1 - Manual**
    - **Led By Ben**
        - **Implementation:**
            - We used a lidar sensor and teleoperation to drive through the world
            - The lidar sensor determined where any obstacles are in the world view
            - Then this was translated into display coordinates
            - We mapped this to a .npy file that will be used later for path finding
        - **Issues:**
            - The lidar sensor beam had a ton of noise in it's mapping when it saw objects that weren't exactly in front of it
            - Getting all of the coordinate systems took awhile and was confusing but we handled it
            - Had to move Lidar up

- **Computer vision - Tier 3**
    - **Led By Kyle - Ben + Jack Assist**
        - **Steps:**
            - Training data was gathered by using the webots object recognition api, and manually driving around the world to gather over 5000 labeled images
            - Used the Ultralytics Hub to train a medium and small sized neural network on the training data
            - Downloaded the resulting neural network weights file and loaded it into webots.
            - Called the model.predict function to find the targets and found their angle from the robot

            - Located positions of objects by getting two photos of every object and triangulating the position of the object.
        - **Issues:**
            - Triangulating the objects in the map view to be able to deduce the exact location
            - Then moving the waypoints away from their exact location into a place that the robot can actually drive to
            - It was a struggle to move the code from a controller that only did 2d computer vision and implement it into our final controller flow with 3d target localization
            - We struggled getting the map coordinates rotated to be the correct way as we were coming down to the last minute

- **Manipulation - tier 2 - IK with teleoperation**
    - **Led by Jack- Kyle + Ben Assist**
        - **Implementation:**
            - Used the IKPY library implementation
            - Read the urdf file into a chain variable that maps out the joints, links, and their various constraints
            - Remove unnecessary links and joints
            - Ensure all links and joints are appropriately mapped as active and not-active
            - Initialize all the motors that map to joints
            - Use forward kinematics function to find the current position of the arm
            - Then use the inverse kinematics function to find desired joint angles
            - Map moving the arm in various directions to hot keys to be used for teleoperation to pick up objects
            - Store important positions, such as arm over basket and arm at top shelf, so we don't have to call inverse_kinematics each time to speed up computation
            - Once robot is within range of an object we use the keyboard to guide the arm to the object

        - Then we have a predefined key to hit for the robot to place the item in the bin
    - **Issues:**
        - It was a big struggle to get the urdf file to load in properly and make sure all the variables were set correctly
        - Then it was an issue to get the motors correctly integrated with the correct joints
        - Giving the correct initial position to the inverse_kinematics function and dealing with errors returned from the function. To solve this it took a lot of research into the library and going to piazza and getting information from there
        - We kept getting a “x0 is unfeasible” error that we resolved by doing a try and catch statement that returned the arm back to an acceptable position before the program crashed
        - Also difficult to get the code to run at a reasonable speed since the inverse kinematics is relatively slow for teleoperation

- **Navigation - tier 4 - RRT with Path Smoothing**
  - **Led by Ben - Jack Assist**
    - **Implementation:** .
        - Initialize the RRT with a starting state. This state represents the starting configuration of the robot or the object that needs to be moved.
        - Grow the RRT by iteratively adding new nodes to the tree. In each iteration, generate a random state in the configuration space and find the nearest node in the tree.
        - Then, attempt to connect the nearest node to the random state by extending the robot or object towards the random state while ensuring that the robot/object stays within the feasible region of the configuration space.

            - If a collision occurs, discard the new node and try again with a new random state. Otherwise, add the new node to the tree and update the nearest neighbors of the nodes in the tree.
            - Repeat step 2 until a goal state is reached or until a maximum number of iterations is reached. A goal state is a state that satisfies the task constraints or objective.
            - After RRT has generated our path, we call a path smoothing function that eliminates unnecessary points to optimize the path, we found a pseudocode algorithm that helped us write out code that is cited in the function definition
        - **Issues:**
            - The path smoothing algorithm gave us a lot of issues, we could not figure out the correct steps to handle this problem. It seemed simple in our heads. We knew what we wanted to do but had a lot of struggles getting python to do what we wanted. So, we went online, found some good pseudocode that's documented in the file at the function definition, and went from there and were able to get it implemented.
            - The robot would sometimes generate paths with collisions but it wouldn’t recognize it.

- **Localization - Tier 1**
    - **Led by Jack**
        - **Implementation:**
            - Used Webots library for GPS and Compass to determine the pose
            - Convert compass return value to radians that work for our world coordinate system
            - Store information
            - Print it to the console and compare against translation given in Webots tool bar
            - Use it for error heading driving code
        - **Issues:**

- This was an easy implementation, but we did take a minute to figure out that our radian calculation needed to be adjusted. But once we had a solid understanding of how the world coordinates related to the robot pose and our map coordinates, we resolved the issue.

Demonstration:

To display our **mapping**, we will teleoperate the robot and drive him throughout the world, and using the lidar sensor, we will map out our world. To demonstrate this mapping we will display the map generated by the lidar sensing.

To demonstrate the **computer vision** of our robot, we will use the display to show yellow lines of targets that the camera sees with the model. We trained a Yolo v8 model to learn what the targets look like (and all the other objects for that matter). We will do the object mapping simultaneously with our mapping, and to demonstrate the vision we will show a display showing the location of all of the objects on our map. We will then use these generated points as waypoints for our path planning.

After we have our list of waypoints of where all the objects are we will now use RRT to generate a valid path for our **navigation**, we enter a path planning state and display the path that our RRT path planning algorithm generates. We will first show the path generated by RRT then display the path after it has been smoothed. Then our robot will follow that path.

To demonstrate the **Localization** of our robot, we will print the world coordinates of the robot to the console as it moves through space. We will know this is working right if the robot goes to the locations of all the objects it is supposed to pick up. This is then used for our error heading code that guides the robot to its destination.

Once the robot has reached its waypoint our **Manipulation** will be apparent as the robot goes to grab objects. We are going for tier 2, implementing Inverse Kinematics with teleoperation. We will use the keys to grab the object and place it in the bin. Then we will go back into our path planning state in order to find a path to our next waypoint and repeat the manipulation until all objects have been collected. This will be demonstrated by watching the robot actually grab the objects and place them in the bin.

Post Completion Analysis

General Notes:

Project Video Demo: Robotics Final Project Demo.mp4

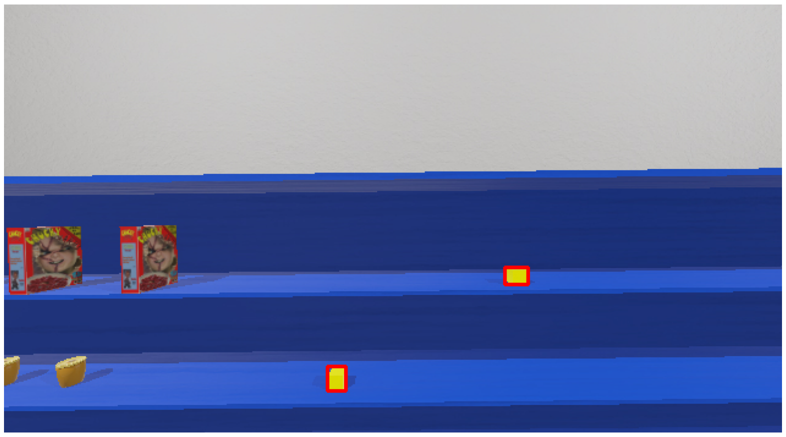
Several of the cubes fell straight through the basket after we dropped it in the basket, and per a piazza post we saw, it was said it can be ignored. We were able to collect all 10 items with this flow, plus the two bonus cubes. We had to do jump cuts due to the 2min time constraints. We also occasionally had to “free” the robot when it got stuck but this was fairly rare (also caused some cubes to fall out of the basket). Also make sure you back up the robot into the middle of the file before pressing “0” otherwise RRT might get called with the start configuration in an obstacle.

Code Notes and Structure:

We augmented the Tiago’s Lidar sensor to be at [0, 0, .8]. The Tiago Camera Slot has a width of 640 and height of 360. The display has a width of 483 and a height of 900. Overall, we used an onslaught of helper functions and a state machine within the main control loop to achieve our goal of picking up all the objects.

Control Flow:

1. Manually drive around the map. As we drive the lidar automatically fills out an obstacle map. During this mapping drive, we intermittently take photos with the camera. We take two photos of each cube, each from different directions so that we can triangulate the position of the cubes. The cubes are identified with a trained YOLOv8 model (“yoloWeights.pt” or “smallModel.pt”).
    a. Here is a demo image showing the bounding boxes of the targets that yolo sees. You can check out details on the model here: https://hub.ultralytics.com/models/bCebOBoiu4eDIZ5EJvTD



2. We then use these locations as goal waypoints for RRT to generate a path to them. Once RRT has returned a valid path, we smooth the path with another function.
3. Once we have a smoothed path, we use the Webots gps and compass localization to autonomously drive the robot to the waypoint. The robot then arrives at the waypoint and we use the IKPY library to teleoperate the robot arm to grab the object and place it in the bin. We also have to manually drive the robot a little bit when he’s reached the waypoint to allow for the correct positioning to grab the cube.
4. We then change our state to path planning again, to generate a path to our next cube. Then we drive there, grab the cube, place it in the bin, and repeat this process until all cubes have been grabbed.

Code Structure:
The structure of the code is as follows:

- Libraries included
- Global Variables
- Inverse Kinematics Setup
- Pre-included setup code for the robot and timesteps
- Helper Functions
- Main Control Loop
    - Manual Driving state
    - Mapping state
    - Path Planning state
    - Autonomous Driving State
    - Machine Learning Training State

Key Command Guide:

We used a lot of different keys for various commands in our controller and they are as follows. They are also commented in the same format in our controller file. Note: The robot is in its mapping mode until you hit ‘0’

ARROW KEYS:drive robot forward backward, left right
W, A, S, D: Control robot arm using IK in XY-axis
E, Q: Control robot arm using IK in Z axis
0: First iteration will display map, pixel inflated map,
on ALL iterations will call RRT and display original path and smoothed path
Upon X-ing out of the plot windows moves into autonomous path finding mode
N: Saves the map to map.npy file
V: Enables/Disables Computer Vision
O: Opens the robot gripper
C: Closes the robot gripper
L: Moves arm to intermediate state after grabbing object off of lower/middle shelf using IK
U: Moves arm to intermediate state after grabbing object off of upper shelf using IK
B: Moves arm to random point over basket for cube to be dropped in using IK

Concluding Note:

We all agreed that an equal amount of work/time was spent by each member in the completion of the project. Everyone pulled their weight and showed up to the meeting so we could get the project done.

Signed: Ben B [illegible]

[illegible]

[illegible]