Nolan Annis: noan9277@colorado.edu
Conor Fagan: faco1538@colorado.edu
Kate McCarthy: kamc8496@colorado.edu

Deliverables:

Area	Description
Mapping	Autonomous
Localization	Odometry
Computer Vision	Color Blob
Planning for Navigation	RRT w/ path smoothing
Manipulation	Autonomous w. waypoints
Yellow Blocks	Obtained: 5

NUMBER OF YELLOW OBJECTS GRABBED: 5

Mapping: Autonomous

Implemented By: Conor, Nolan

This implementation was pulled from our Lab 5 manual mapping, along with an aimless RRT algorithm to autonomously discover all the obstacles in the world, and adjusted to fit this world and coordinate system. The main issues we ran into with this was figuring out the differences in coordinates between the lab and the final project, and how this new coordinate setup correlated with the map we were drawing.

Process:

1. Define rad, pos_theta, wx, wy, rx, ry, and t.

```
n = compass.getValues()
 rad = -((math.atan2(n[0], n[1]))-1.5708)
 pose_theta = rad
 rx = -math.cos(alpha)*rho
 ry = math.sin(alpha)*rho
 t = pose_theta + np.pi/2
 wx =  math.cos(t)*rx + math.sin(t)*ry + pose_x
 wy =  math.sin(t)*rx - math.cos(t)*ry + pose_y
```

2. Initialize 2D array to be used as display map
3. Increment points on array correlating to the lidar readings

   map[map_j][map_i] += 0.003

Also includes checks to ensure that the values won't go out of bounds while attempting to take lidar readings and convert them to the 2D map

4. Make color change as Increment gets higher

```
g = map[map_j][map_i]
        if (g > 1):
          g = 1
        color = (g*256**2+g*256+g)*255
        display.setColor(int(color))
```

   As the lidar reads in the same spot multiple times, it begins changing the color, showing a degree of how likely there is an obstacle there

5. Using the RRT path planning algorithm with no goal, move Tiago through world to collect mapping data
6. Save map

   After the Tiago has gone through all of the map, pressing S, starts the saving process. Every point with a value over 0.6 in the 2D array(indicating there is likely an object there), is converted to a full white and the value of that spot set to 1, while every other is set to black on the map and the value in that spot set to 0. This 2D array now provides a visual map that can be interpreted easily, as well as a 2D array of zeros and ones that we can use for path planning.

Difficulties encountered:
Implementing the mapping is probably where we had the most difficulties. We started with our code from lab 5, so the first issue we encountered was trying to adjust the coordinate system to the final project world. We struggled getting this perfect, partly due to the next issue I'll discuss, but were able to get the correct orientation and general placement on the map. After this the readings we were getting just seemed to make zero sense, and we couldn't figure out why. We were able to figure it out outside of our main code, but when putting that code in with the rest of the project code we got an insane amount of noise and inconsistencies.

Future Work:
Well the current mapping is not very functional and could be cleaned up drastically. Along with this, after making the mapping fully function, we would have to make it work with the RRT algorithm we used. This is something we attempted to do, but weren't able to do any real testing with it considering we couldn't get a proper map and had some issues with the Tiago bot velocity problems.

RRT with Path Smoothing

Implemented By: Nolan
This algorithm is taken from a Homework 2 implementation and then updated to include some smoothing and to make it work with how our mapping was expected to work. Due to reasons discussed later in the odometry, while this algorithm implements a functional RRT with smoothing path, this couldn't be used in actual retrieval of blocks due to error and velocity problems pertaining to the Tiago robot.

Process:

1. Declare helper functions get_random_vertex, state_is_valid, get_nearest_vertex, steer, and check_path_valid
2. The algorithm initially checks if there is a goal point. These would be the points in front of the yellow blocks. In the case that there isn't Tiago moves on a random path, ideally taking him through the entire world to identify obstacles
3. When a Goal point is found the algorithm finds a random valid point and then checks if it is moving Tiago closer to the goal or not. If it is, it plots that point and continues the path from that one. All the points on the path are collected in an array of Tuples, giving Tiago a set of valid waypoints to travel to when the algorithm finishes

Difficulties Encountered:
The biggest difficulty we encountered here is that we were not able to test out the RRT algorithm, because we couldn't get a decent map, due to our mapping struggles. Along with this Tiago was experiencing some errors and velocity problems that after researching and going to office hours we couldn't figure out, and were told if we could fix would end up as extra credit for us. There weren't really any other difficulties we encountered here, as we were unable to actually test to see how well this worked.

Future Work:
The future work for this section really pertains to the issues caused by adjacent parts. So, the first step would be to solve the mapping issues we were experiencing and then further solving the issues Tiago was having in the Webots world. Only after doing these things would we be able to make sure our RRT algorithm is working as intended, and improve upon it.

Color Blob Detection

Implemented By: Kate
There is fully implemented color blob detection in the robot. The robot contains a narrow enough yellow range that it only identifies the exact outline of the blocks detected, not the yellow candles or any other anomalies.

Process:

1. Define Yellow Pixel Range
    a. Defined As: yellow_range = [[210, 210, 0], [255, 255, 30]] for RGB colors.
2. Get Image from Robot and Establish Image Mask
    a. Uses Camera.imageGet[Color] for Red, Green and Blue values (see below)
    b. Automatically Detects if a Pixel is Yellow and demonstrates this in Mask
3. Detect Blobs (joint groups of pixels) in Image Mask
    a. Uses Expand_NR non-recursive function to establish blobs
    b. Taken directly from Homework 3
4. Get Blob Centroids
    a. Mean of x and y in the blob
5. Display Blob (optional)
    a. Displays white blobs (if blobs are large enough to be counted)

    b. Resets every 10 timesteps to prevent large streaks of blobs while the robot is moving.

Difficulties encountered:
In video demonstration and in testing trials, there was difficulty in the display functionality of the color detection. This introduced me to some specifics of image/camera extraction from the Tiago, and the matrix capabilities there within.
Perhaps it was the 2023a version of Webots that I have, but using standard Camera.image functionality wasn't working properly. As such, I had to resort to using Camera.imageGetRed, Camera.imageGetGreen and Camera.imageGetBlue and from there construct a separate pixel matrix. This added a large amount of computing power taken up, making color detection run much slower than I would've liked.
For the longest time, I had the for loop structured incorrectly so there was loss on the side of the image and every pixel in row i was shifted by i+1 pixels. This was caught and fixed, but threw off calculations for a large portion of the project.

Future Work:
Currently the maximum capacity of the color blob detection is the ability to detect where a blob is in a 2D image, not a 3D capacity. This results in the ability to provide a relative coordinate. At the moment the function contains the ability to identify whether a block is on the top or bottom shelf. In future implementation, I would incorporate the size of the block (bigger blobs are closer) and location in order to determine a 3D coordinate system.

Odometry

Implemented By: Kate
The odometry, in conjunction with the inverse kinematics, on the Tiago robot was perhaps the most time consuming challenge encountered in the project. This was due to inconsistencies in the wheel velocities of the Tiago robot. The solution will be discussed in a further section.

The solution to this difficulty was found in getting the *true* velocity of the wheels. Instead of checking what the inputted/current velocities of the wheels were, odometry and localization was based off of enabled position sensors in the wheels. Current linear velocity was calculated with:

```
curr_position = robot_parts["wheel_(L/R)_joint"].getPositionSensor().getValue()
change = curr_position-prev_position
lin_vel = ((change)/(timestep/1000.0))
prev_position = curr_position
```

Process:

1. Generating current linear velocities
    a. Above formula
2. Calculate distance change
    a. `dist = lin_velL/MAX_SPEED * MAX_SPEED_MS * timestep/1000.0`
3. Calculate new pose x and y.
    a. `pose_x += (distL+distR) / 2.0 * math.cos(pose_theta)`
    b. `pose_y += (distL+distR) / 2.0 * math.sin(pose_theta)`
4. Calculate pose theta

a. `pose_theta += (distR-distL)/AXLE_LENGTH`
b. `pose_theta = pose_theta%(2*np.pi)`

Odometry is established with (0,0) as the starting point of the robot, with roughly x = [-6, 6] and y = [-7, 14] for a range of positions in the global coordinates.

Difficulties Encountered:
The above velocity inconsistency was the primary difficulty encountered in this section. However, there were some other small sources of error. The largest one I found was the swaying of the Tiago's body. With the unstable body of the Tiago, it tends to sway back and forth, and this momentum marginally changes the speed/trajectory of the robot without impacting the wheel speed. Especially since my arm joint placements required I have the robot as tall as possible, the robot has a good amount of sway back and forth. While this was a small amount of error, as the robot went on longer and longer along paths, it would add more and more error to the odometry.

Autonomous IK with Waypoints

Implemented By: Kate
With a waypoint by each block, the robot effectively navigates between waypoints autonomously using bearing and distance errors. However, this was not without its difficulties.
I found that the Tiago is incredibly unresponsive to velocity changes in the wheels. This is especially prevalent when changing between direction states, such as turning and going straight. I found that while each wheel was being inputted with the same speed, but one would take nearly six times as long as the other to reach said speed in some cases. This was occurring even when both wheels were fully stopped between states, so even as both started from 0, one would take substantially longer to reach the end speed.

The solution I created to this was a little janky and incredibly slow, but proved to keep odometry and waypoint following working consistently. Instead of creating a set "stopped" state, I had the robot go straight from turning to driving. However, instead of sending the wheel the specified driving speed V, if one wheel was moving slower than the other, I sent the faster wheel the true speed of the slower wheel, so the two match. As such, this gave the slower wheel time to catch up to the speed of the faster wheel. However, while this does keep the robot from careening off to the side, it takes a while to perform properly.

So in code, with `TOLERANCE` being roughly 0.5:

```
if(vL==vR and vL!=0):
        if(lin_velL<lin_velR - TOLERANCE):
            vR = np.abs(lin_velL) + 0.1
        elif(lin_velR<lin_velL - TOLERANCE):
           vL = np.abs(lin_velR) + 0.1
```

The waypoints for 5 yellow objects are the following after extensive trial and error. Italicized waypoints are transition points, not actual cubes.

```
waypoints = [[0, 2.4], [1.16, 2.63], [6.724, 3.05], [7.342, 3.07],
[7.34, 1.8], [5.72,1.46], [2.6, 1.03],[2.52, 1.06], [0, 1.10]]

Note: The block at [2.6, 1.03] is close to a grab, but not quite.
```

Process:
1. Check odometry
2. Gather bearing and distance errors from current pose_x, pose_y to the waypoint
3. Check bearing to determine whether to turn left or right
4. Once bearing is less than a set tolerance, begin driving state
5. Use above velocity checks to ensure that both wheels are going at the same speed and one is not accelerating faster than the other.
6. The robot makes constant adjustments while driving due to constant calculation of the bearing error.
7. When robot reaches the waypoint, adjust to the next waypoint and check if there is a block to grab.

Difficulties Encountered:
On top of the above issues described, there were some more minor errors. First of all, waypoint detection was manual and trial and error. At times, the robot would take around 20 minutes to complete a full run on my CPU (a terrible CPU). As such, it took days at times to calculate correct waypoints.
Numerically, I was originally deriving a lot of error based on the np.arctan2() function, in regards to the concept that 0 and -2pi are the same number, but in my bearing formulas returned a distance of 2pi away from each other. As such, instead of a simple bearing = theta - pose_theta, I used the following function:

```
def angle_check(theta, pose_theta):
    pose_theta2 = 0
    if(pose_theta>0): #if positive
        pose_theta2 = pose_theta - 2*np.pi
    else: #if negative
        pose_theta2 = pose_theta + 2*np.pi
    if(np.abs(theta-pose_theta)<np.abs(theta-pose_theta2)):
        return theta-pose_theta
    return theta-pose_theta2
```
This therefore returned the proper

Future Work:
Given more time, I would probably try to get to the root of the Tiago's wheel inconsistencies and combat it at a machine level. My workaround is incredibly poor for performance/speed, and doesn't get to whatever the root of the issue is. I'm going to guess there is some problem with the momentum of the robot's wheel, whether this is arising from the robot itself or a machine issue on my end I do not know.

Robot Arm States
Implemented By: Kate

When landing at a waypoint with a cube, the robot goes through a series of states sequentially in order to properly grab and retrieve said object.

States:

- **Move:** Moving between waypoints, the transitory phase between block locations
- **Stop:** Stopping the robot when it hits a waypoint. Proceeds to next state when both true wheel velocities equal 0
- **Arm Out:** Sticks robot's arm straight in front. This is to allow for consistency in movement and prevents the robot from hitting a shelf when its arm is transitioning to the next state.
- **Arm Up or Down:** Depending on the location of the block, picks its setPositions from either an up or a down state. Hypothetically this puts the grippers around the block, if the waypoint is calculated correctly.
- **Grab:** Close grippers.
- **Arm to Basket:** Move arm and block over basket.
- **Drop:** Open grippers. From here, proceed to next waypoint with move and loop the cycle over again.

Video Link:

Slides:
https://docs.google.com/presentation/d/19du2Iwn3Hem5z-CLjSWxvBySgMaeAEKf5r88Nzwsr3A/edit?usp=sharing
YouTube:
https://youtu.be/uWPDbKVwc20

Github Link:

https://github.com/kimccarthy/CSCI-3302-Final
Caution with controller: Controller was made on Webots 2023a. Unknown if this causes errors with backdated versions of Webots.