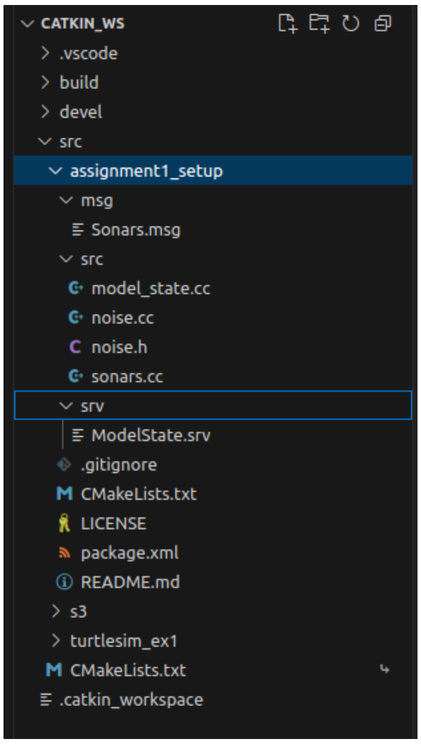
3806ICT ROBOTICS AGENTS AND REASONING ASSIGNMENT 1

Jack Millington

S5405915

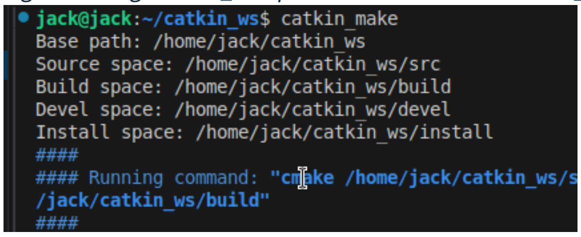
Task 1: Feedback Loop Control

This task was to setup the environment for the TurtleBot. Then use a PID feedback control algorithm, to drive the robot towards the box. Below is the setup process shown.



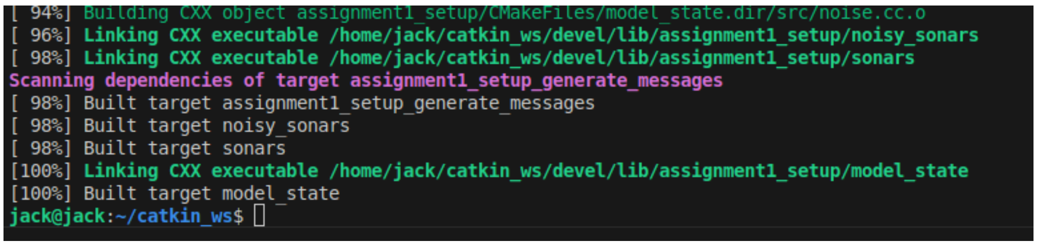
```
CATKIN_WS
  > .vscode
  > build
  > devel
  v src
    v assignment1_setup
      v msg
        Sonars.msg
      v src
        model_state.cc
        noise.cc
        noise.h
        sonars.cc
      v srv
        ModelState.srv
      .gitignore
      CMakeLists.txt
      LICENSE
      package.xml
      README.md
    > s3
    > turtlesim_ex1
    CMakeLists.txt
  .catkin_workspace
```

Figure 1 assignment1_setup.tar file extracted into catkin_ws/src folder



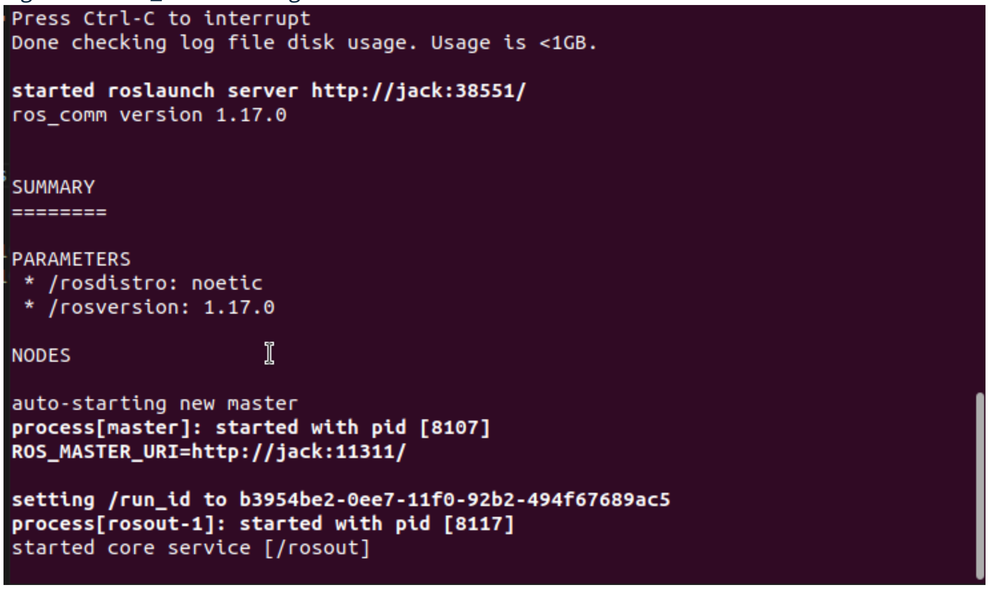
```
jack@jack:~/catkin_ws$ catkin_make
Base path: /home/jack/catkin_ws
Source space: /home/jack/catkin_ws/src
Build space: /home/jack/catkin_ws/build
Devel space: /home/jack/catkin_ws/devel
Install space: /home/jack/catkin_ws/install
####
#### Running command: "cmake /home/jack/catkin_ws/s
/jack/catkin_ws/build"
####
```

Figure 2 catkin_make ran img1



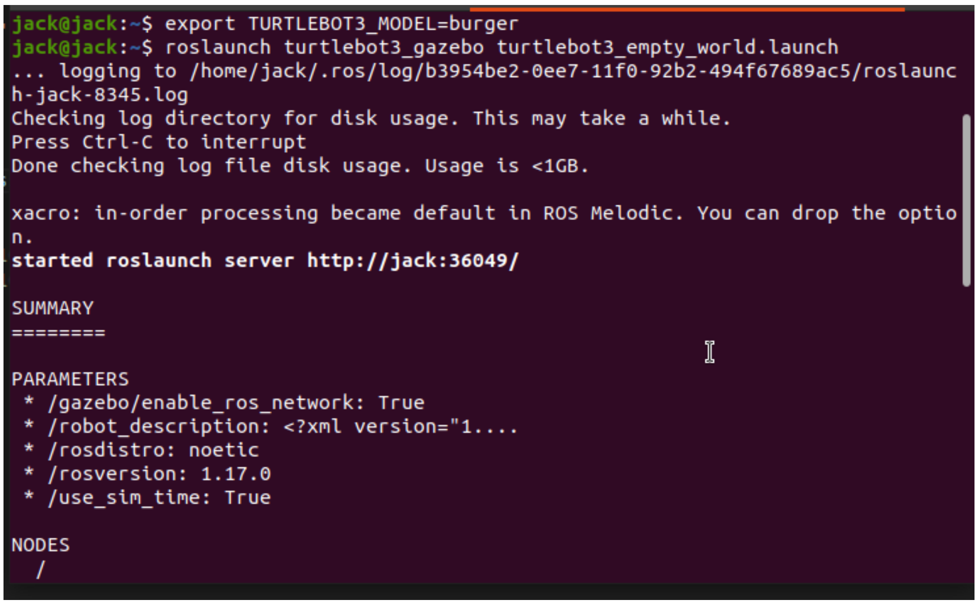
```
[ 94%] Building CXX object assignment1_setup/CMakeFiles/model_state.dir/src/noise.cc.o
[ 96%] Linking CXX executable /home/jack/catkin_ws/devel/lib/assignment1_setup/noisy_sonars
[ 98%] Linking CXX executable /home/jack/catkin_ws/devel/lib/assignment1_setup/sonars
Scanning dependencies of target assignment1_setup_generate_messages
[ 98%] Built target assignment1_setup_generate_messages
[ 98%] Built target noisy_sonars
[ 98%] Built target sonars
[100%] Linking CXX executable /home/jack/catkin_ws/devel/lib/assignment1_setup/model_state
[100%] Built target model_state
jack@jack:~/catkin_ws$
```

Figure 3 catkin_make ran img2



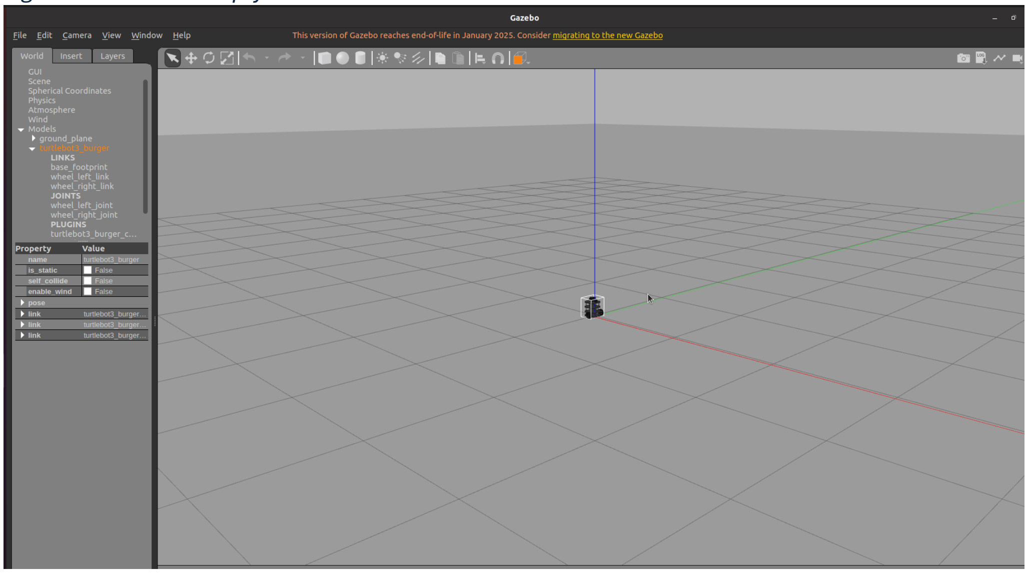
```
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://jack:38551/
ros_comm version 1.17.0


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.17.0

NODES

auto-starting new master
process[master]: started with pid [8107]
ROS_MASTER_URI=http://jack:11311/

setting /run_id to b3954be2-0ee7-11f0-92b2-494f67689ac5
process[rosout-1]: started with pid [8117]
started core service [/rosout]
```

Figure 4 Roscore started



```
jack@jack:~$ export TURTLEBOT3_MODEL=burger
jack@jack:~$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
... logging to /home/jack/.ros/log/b3954be2-0ee7-11f0-92b2-494f67689ac5/roslaunc
h-jack-8345.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

xacro: in-order processing became default in ROS Melodic. You can drop the optio
n.
started roslaunch server http://jack:36049/

SUMMARY
========

PARAMETERS
 * /gazebo/enable_ros_network: True
 * /robot_description: <?xml version="1....
 * /rosdistro: noetic
 * /rosversion: 1.17.0
 * /use_sim_time: True

NODES
  /
```

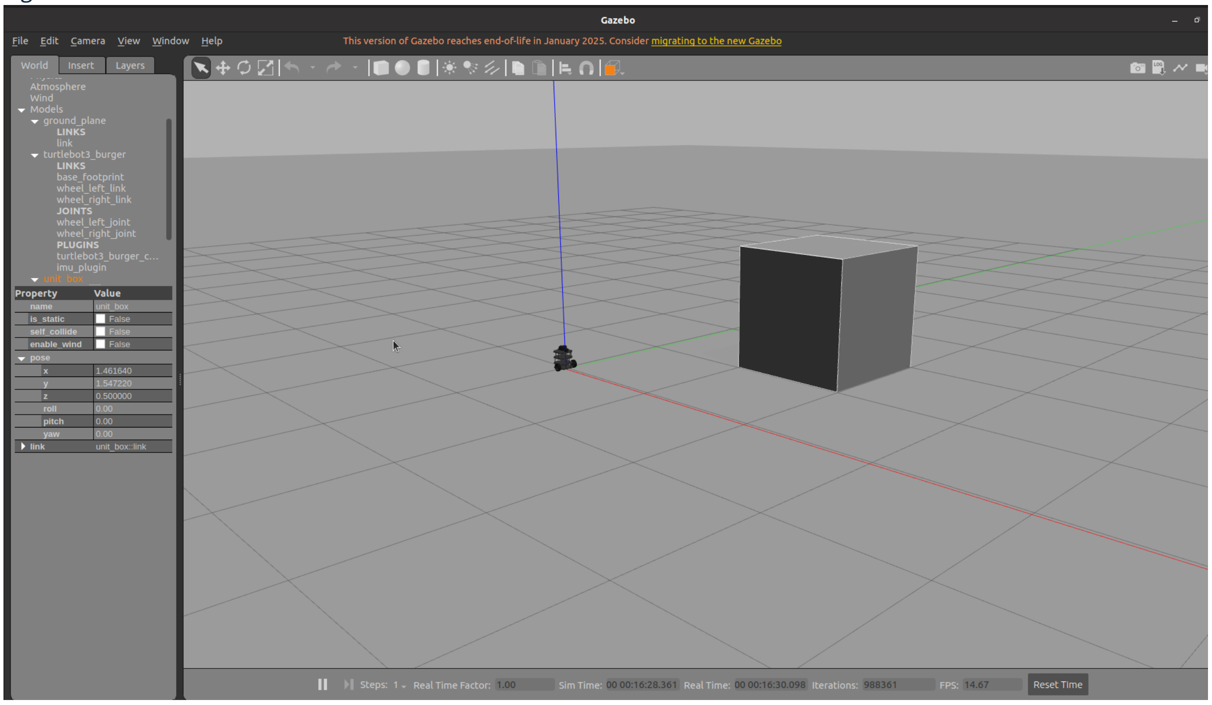
Figure 5 Turtlebot3 empty world launched



This version of Gazebo reaches end-of-life in January 2025. Consider migrating to the new Gazebo

Property	Value
name	turtlebot3_burger
is_static	☐ False
self_collide	☐ False
enable_wind	☐ False
pose	
link	turtlebot3_burger...
link	turtlebot3_burger...
link	turtlebot3_burger...

Figure 6 Gazebo Launched



This version of Gazebo reaches end-of-life in January 2025. Consider migrating to the new Gazebo

Property	Value
name	unit_box
is_static	☐ False
self_collide	☐ False
enable_wind	☐ False
pose	
x	1.461640
y	1.547220
z	0.500000
roll	0.00
pitch	0.00
yaw	0.00
link	unit_box::link

```
Steps: 1   Real Time Factor: 1.00   Sim Time: 00 00:16:28.361   Real Time: 00 00:16:30.098   Iterations: 988361   FPS: 14.67
```

Figure 7 Box Placed

I began encountering many errors when trying to run $ rosrun assignment1_setup sonars. After debugging and recoding certain sections of the sonars.cc file, and changing my cmaklists.txt and package.xml, I managed to get the sonar node working. However, it was still just the base setup. So the turtlebot only had 3 front facing 30 degree sensors. So this had to change. To implement this, I modified the sonars.cc file to include all 6 sensors, change them to 40 degree widths as specified in the assignment description, and modified the sonars.msg file. Below is the implementation of this:

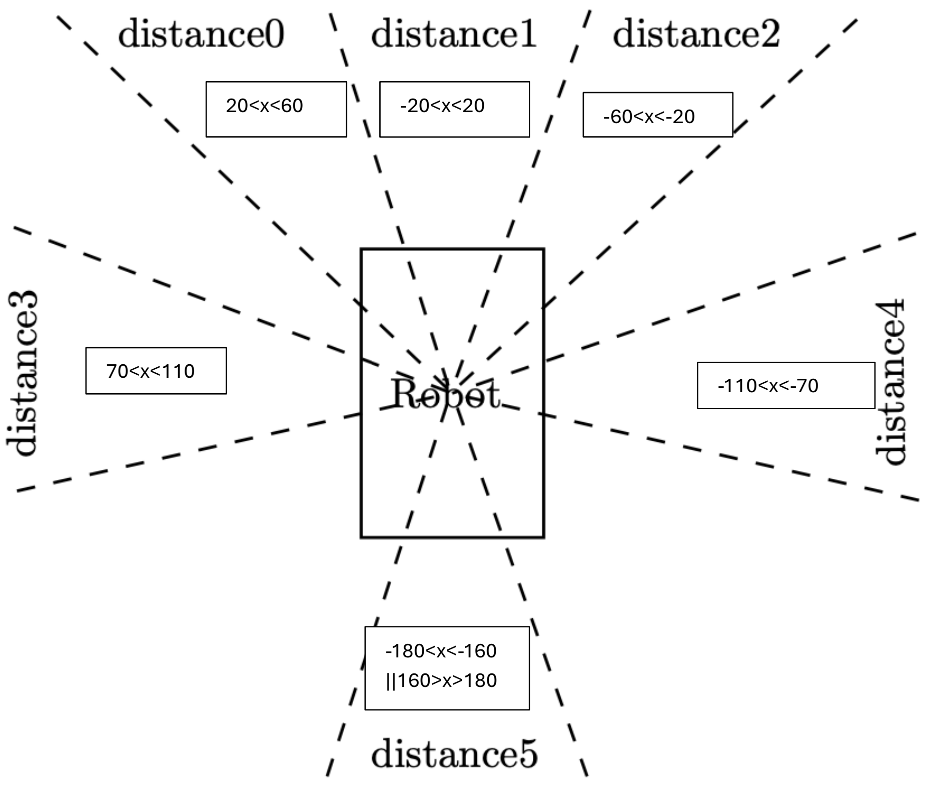
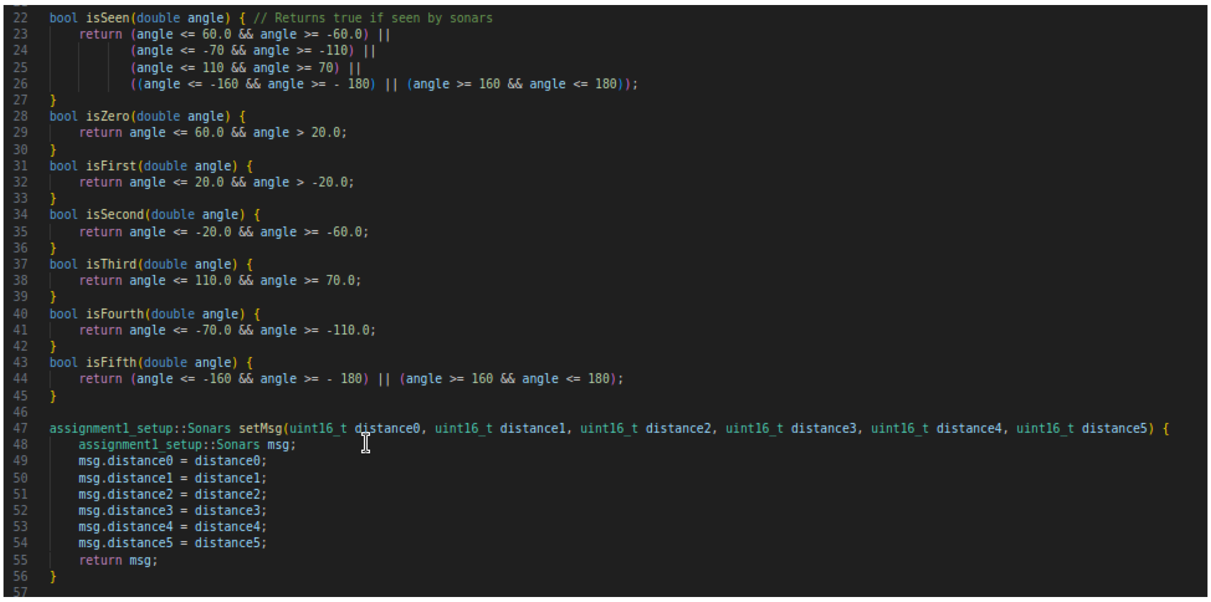


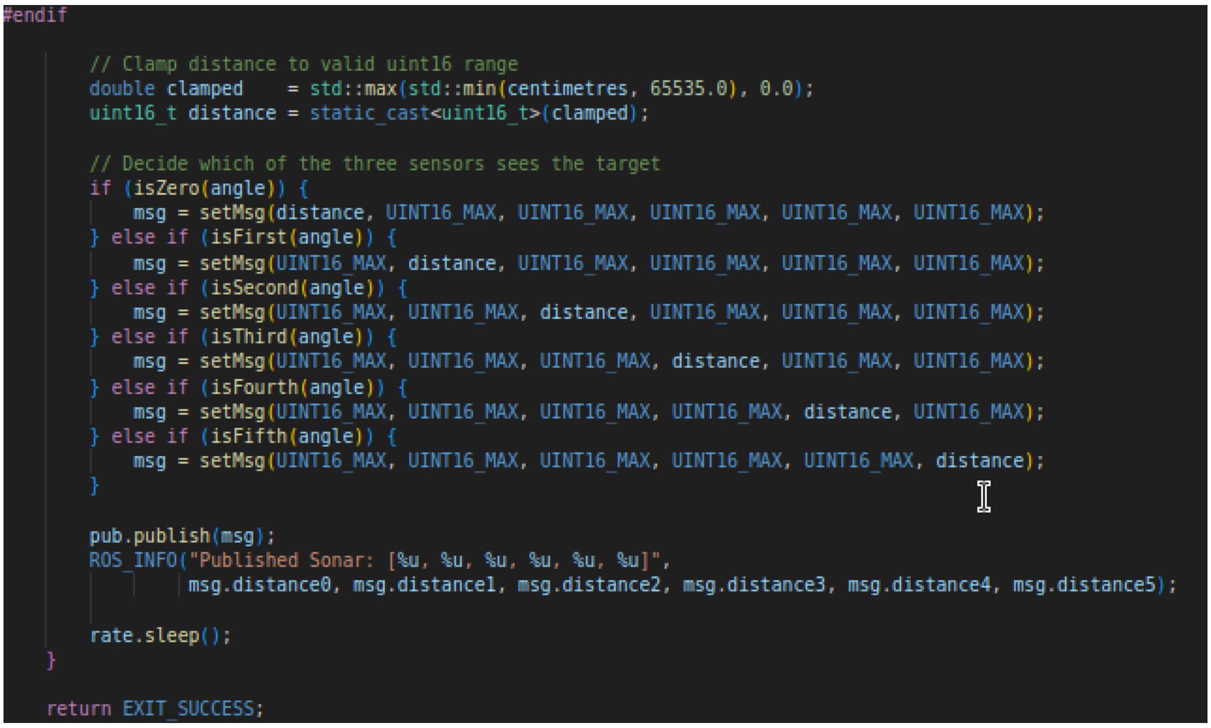
Figure 1: The Sonar Arrangement

Figure 8: The sonar arrangement



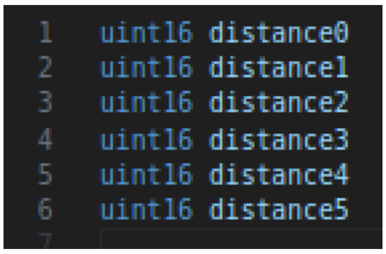
```
bool isSeen(double angle) { // Returns true if seen by sonars
    return (angle <= 60.0 && angle >= -60.0) ||
           (angle <= -70 && angle >= -110) ||
           (angle <= 110 && angle >= 70) ||
           ((angle <= -160 && angle >= - 180) || (angle >= 160 && angle <= 180));
}
bool isZero(double angle) {
    return angle <= 60.0 && angle > 20.0;
}
bool isFirst(double angle) {
    return angle <= 20.0 && angle > -20.0;
}
bool isSecond(double angle) {
    return angle <= -20.0 && angle >= -60.0;
}
bool isThird(double angle) {
    return angle <= 110.0 && angle >= 70.0;
}
bool isFourth(double angle) {
    return angle <= -70.0 && angle >= -110.0;
}
bool isFifth(double angle) {
    return (angle <= -160 && angle >= - 180) || (angle >= 160 && angle <= 180);
}

assignment1_setup::Sonars setMsg(uint16_t distance0, uint16_t distance1, uint16_t distance2, uint16_t distance3, uint16_t distance4, uint16_t distance5) {
    assignment1_setup::Sonars msg;
    msg.distance0 = distance0;
    msg.distance1 = distance1;
    msg.distance2 = distance2;
    msg.distance3 = distance3;
    msg.distance4 = distance4;
    msg.distance5 = distance5;
    return msg;
}
```

Figure 9: Sonars.cc edit 1



```
#endif

        // Clamp distance to valid uint16 range
        double clamped    = std::max(std::min(centimetres, 65535.0), 0.0);
        uint16_t distance = static_cast<uint16_t>(clamped);

        // Decide which of the three sensors sees the target
        if (isZero(angle)) {
            msg = setMsg(distance, UINT16_MAX, UINT16_MAX, UINT16_MAX, UINT16_MAX, UINT16_MAX);
        } else if (isFirst(angle)) {
            msg = setMsg(UINT16_MAX, distance, UINT16_MAX, UINT16_MAX, UINT16_MAX, UINT16_MAX);
        } else if (isSecond(angle)) {
            msg = setMsg(UINT16_MAX, UINT16_MAX, distance, UINT16_MAX, UINT16_MAX, UINT16_MAX);
        } else if (isThird(angle)) {
            msg = setMsg(UINT16_MAX, UINT16_MAX, UINT16_MAX, distance, UINT16_MAX, UINT16_MAX);
        } else if (isFourth(angle)) {
            msg = setMsg(UINT16_MAX, UINT16_MAX, UINT16_MAX, UINT16_MAX, distance, UINT16_MAX);
        } else if (isFifth(angle)) {
            msg = setMsg(UINT16_MAX, UINT16_MAX, UINT16_MAX, UINT16_MAX, UINT16_MAX, distance);
        }

        pub.publish(msg);
        ROS_INFO("Published Sonar: [%u, %u, %u, %u, %u, %u]",
                 msg.distance0, msg.distance1, msg.distance2, msg.distance3, msg.distance4, msg.distance5);

        rate.sleep();
    }

    return EXIT_SUCCESS;
```

Figure 10: Sonars.cc edit 2

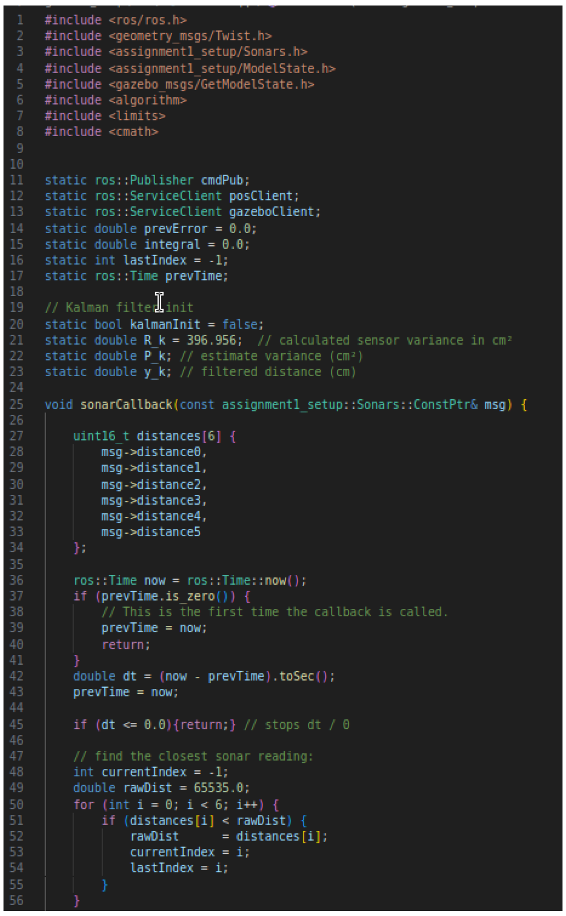


```
uint16 distance0
uint16 distance1
uint16 distance2
uint16 distance3
uint16 distance4
uint16 distance5
```

Figure 11: Sonars.msg

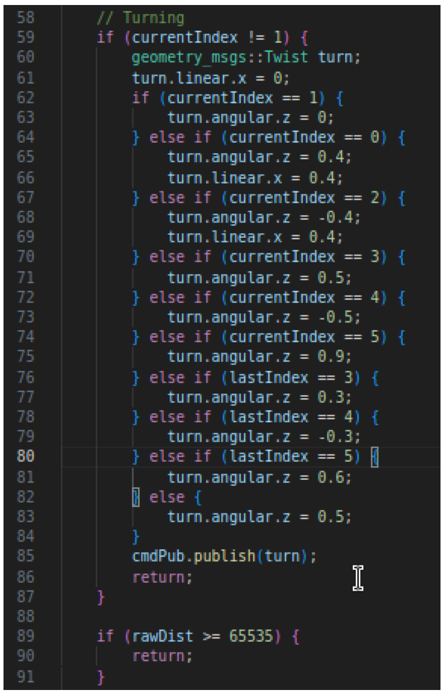
I then created my controller.cpp file which included the turning logic based on the sonars. The robot will spin left or right when there is no detection of the box, using the memory of its last sonar reading to calculate the shortest turn needed for the box to be sensed in the distance1 sonar. This is because sometimes none of the sonars will sense the box as it’s in a blind spot. This is because the sonars do not cover a 360-degree radius as seen in the diagram above. When the box is found, the bot turns to face the box, then uses a PID algorithm to accelerate towards the box, slowing down as it approaches the box for a smooth landing. The values for the PID algorithm were tweaked and refined for a smooth approach to the box, the discussion goes more into depth on this.

Below is the code implementing my controller.cpp.



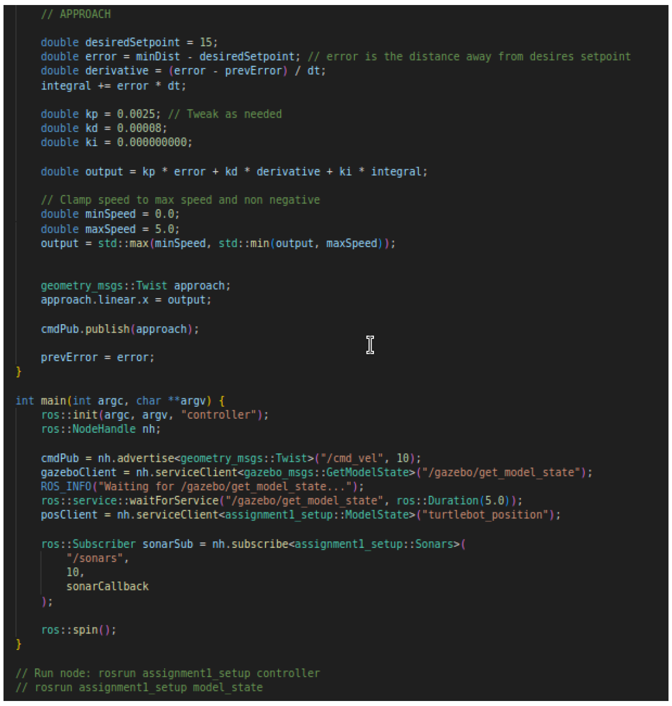
```cpp
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <assignment1_setup/Sonars.h>
#include <assignment1_setup/ModelState.h>
#include <gazebo_msgs/GetModelState.h>
#include <algorithm>
#include <limits>
#include <cmath>


static ros::Publisher cmdPub;
static ros::ServiceClient posClient;
static ros::ServiceClient gazeboClient;
static double prevError = 0.0;
static double integral = 0.0;
static int lastIndex = -1;
static ros::Time prevTime;

// Kalman filter init
static bool kalmanInit = false;
static double R_k = 396.956;  // calculated sensor variance in cm²
static double P_k; // estimate variance (cm²)
static double y_k; // filtered distance (cm)

void sonarCallback(const assignment1_setup::Sonars::ConstPtr& msg) {

    uint16_t distances[6] {
        msg->distance0,
        msg->distance1,
        msg->distance2,
        msg->distance3,
        msg->distance4,
        msg->distance5
    };

    ros::Time now = ros::Time::now();
    if (prevTime.is_zero()) {
        // This is the first time the callback is called.
        prevTime = now;
        return;
    }
    double dt = (now - prevTime).toSec();
    prevTime = now;

    if (dt <= 0.0){return;} // stops dt / 0

    // find the closest sonar reading:
    int currentIndex = -1;
    double rawDist = 65535.0;
    for (int i = 0; i < 6; i++) {
        if (distances[i] < rawDist) {
            rawDist      = distances[i];
            currentIndex = i;
            lastIndex = i;
        }
    }
```

Figure 12: controller.cpp snippet 1



```cpp
    // Turning
    if (currentIndex != 1) {
        geometry_msgs::Twist turn;
        turn.linear.x = 0;
        if (currentIndex == 1) {
            turn.angular.z = 0;
        } else if (currentIndex == 0) {
            turn.angular.z = 0.4;
            turn.linear.x = 0.4;
        } else if (currentIndex == 2) {
            turn.angular.z = -0.4;
            turn.linear.x = 0.4;
        } else if (currentIndex == 3) {
            turn.angular.z = 0.5;
        } else if (currentIndex == 4) {
            turn.angular.z = -0.5;
        } else if (currentIndex == 5) {
            turn.angular.z = 0.9;
        } else if (lastIndex == 3) {
            turn.angular.z = 0.3;
        } else if (lastIndex == 4) {
            turn.angular.z = -0.3;
        } else if (lastIndex == 5) {
            turn.angular.z = 0.6;
        } else {
            turn.angular.z = 0.5;
        }
        cmdPub.publish(turn);
        return;
    }

    if (rawDist >= 65535) {
        return;
    }
```

Figure 13: controller.cpp snippet 2



```cpp
    // APPROACH

    double desiredSetpoint = 15;
    double error = minDist - desiredSetpoint; // error is the distance away from desires setpoint
    double derivative = (error - prevError) / dt;
    integral += error * dt;

    double kp = 0.0025; // Tweak as needed
    double kd = 0.00008;
    double ki = 0.000000000;

    double output = kp * error + kd * derivative + ki * integral;

    // Clamp speed to max speed and non negative
    double minSpeed = 0.0;
    double maxSpeed = 5.0;
    output = std::max(minSpeed, std::min(output, maxSpeed));


    geometry_msgs::Twist approach;
    approach.linear.x = output;

    cmdPub.publish(approach);

    prevError = error;
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "controller");
    ros::NodeHandle nh;

    cmdPub = nh.advertise<geometry_msgs::Twist>("/cmd_vel", 10);
    gazeboClient = nh.serviceClient<gazebo_msgs::GetModelState>("/gazebo/get_model_state");
    ROS_INFO("Waiting for /gazebo/get_model_state...");
    ros::service::waitForService("/gazebo/get_model_state", ros::Duration(5.0));
    posClient = nh.serviceClient<assignment1_setup::ModelState>("turtlebot_position");

    ros::Subscriber sonarSub = nh.subscribe<assignment1_setup::Sonars>(
        "/sonars",
        10,
        sonarCallback
    );

    ros::spin();
}

// Run node: rosrun assignment1_setup controller
// rosrun assignment1_setup model_state
```

Figure 14: controller.cpp snippet 3

To run the node, first gazebo must be running, then assignment1_setup sonars must be called to activate the sonars. Then, assigment1_setup controller must be called.

As can be seen here, the robot successfully moves towards the box.

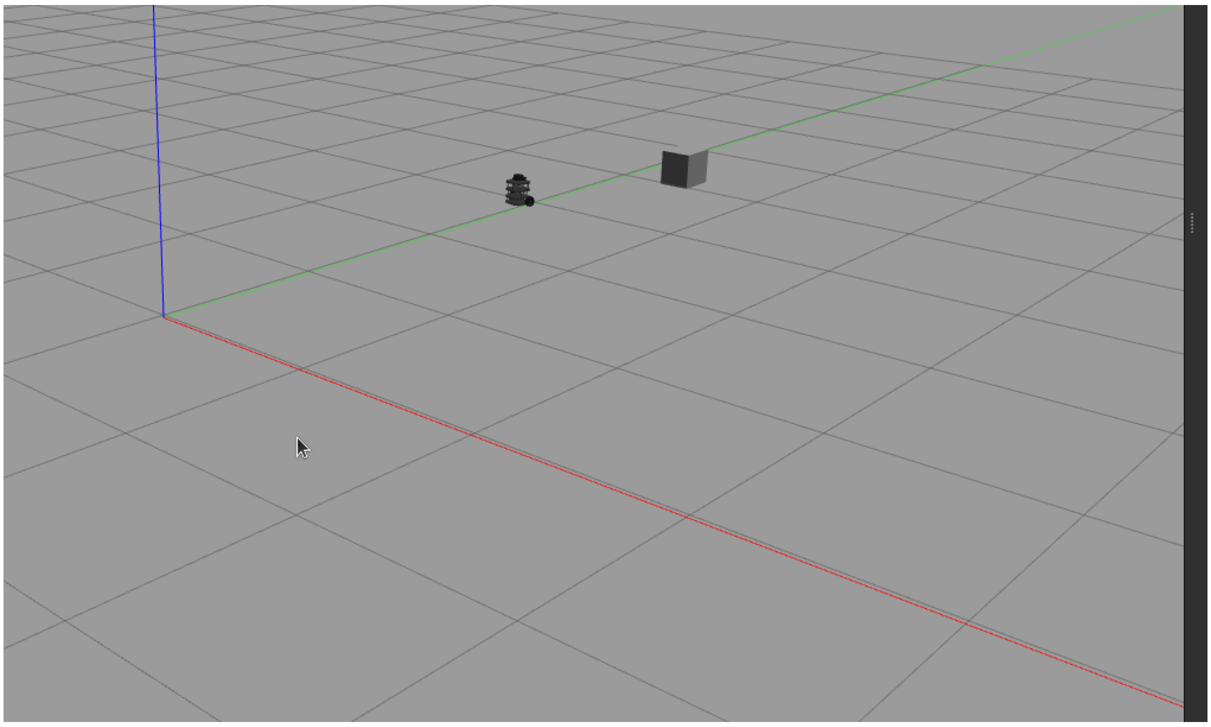


Figure 15: Robot moving towards box

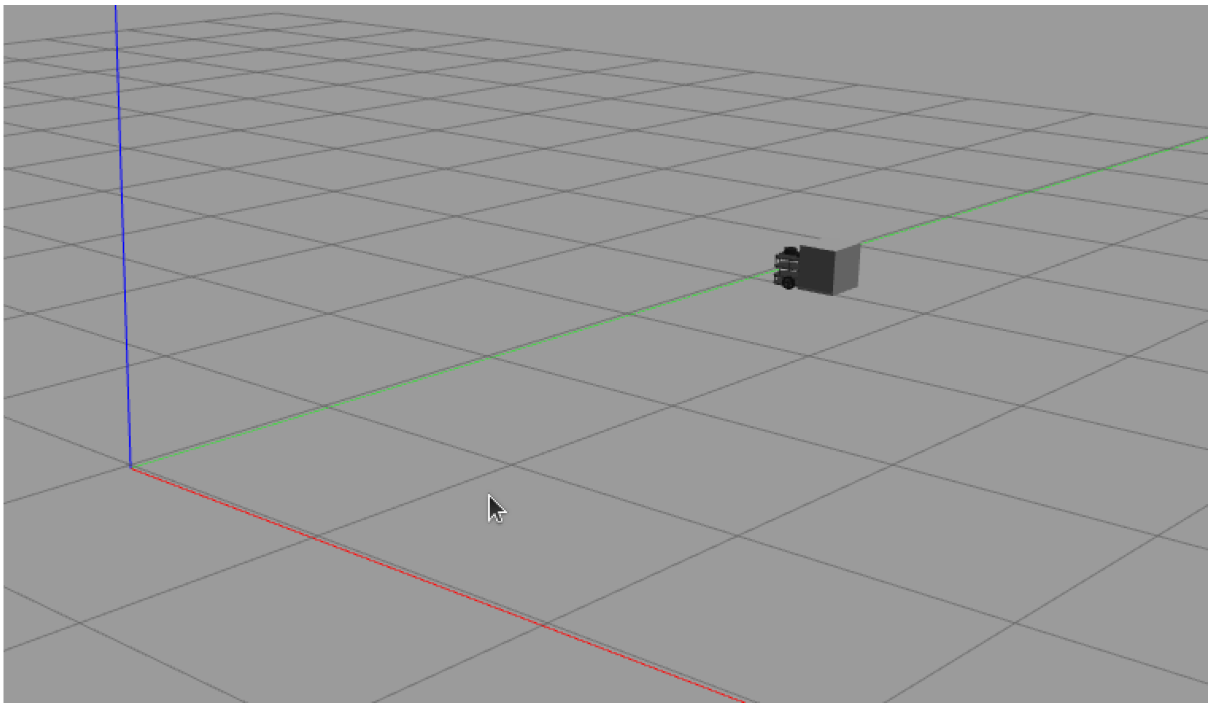


Figure 16: Robot moved to box

Task 1 Discussion:

Feedback loop vs open loop control:

In an open loop system, commands are issued without regard to the robot's actual state. An example of this would be for the robot to drive forward at a fixed speed for a set duration. The issue with this is that any variation in initial distance or changes in the environment can lead to large errors. By contrast, a feedback loop system will continuously measure the distance from the box via the sonars and adjust the speed and direction based on this, enabling the robot to self-correct itself on the fly, and adapt to a changing distance and environment.

Implementation and tuning:

The controller node subscribes to /sonars, computes the error, which is the distance from the box, and publishes a velocity command. To handle the blind spots, the last sonar that detected the box is tracked, to continue the angular movement of the bot as it moves to the front. Initially, when there is no sonar reading, the bot will drive forward until it finds the box in one of its 6 sonars. To tune the PID controller, ki and kd were set to zero, and kp was increased until the robot closed the gap in a reasonable time. Then a small amount of kd was added to dampen the overshoot and smooth the approach. Finally, a very small amount of ki was added to further smooth the approach as it is very close to the box.

PID control algorithm:

The PID control algorithm goes as follows:

$$Y(t) = K_p e(t) + K_d \frac{de(t)}{dt} + K_i \int_0^t e(t')dt'$$

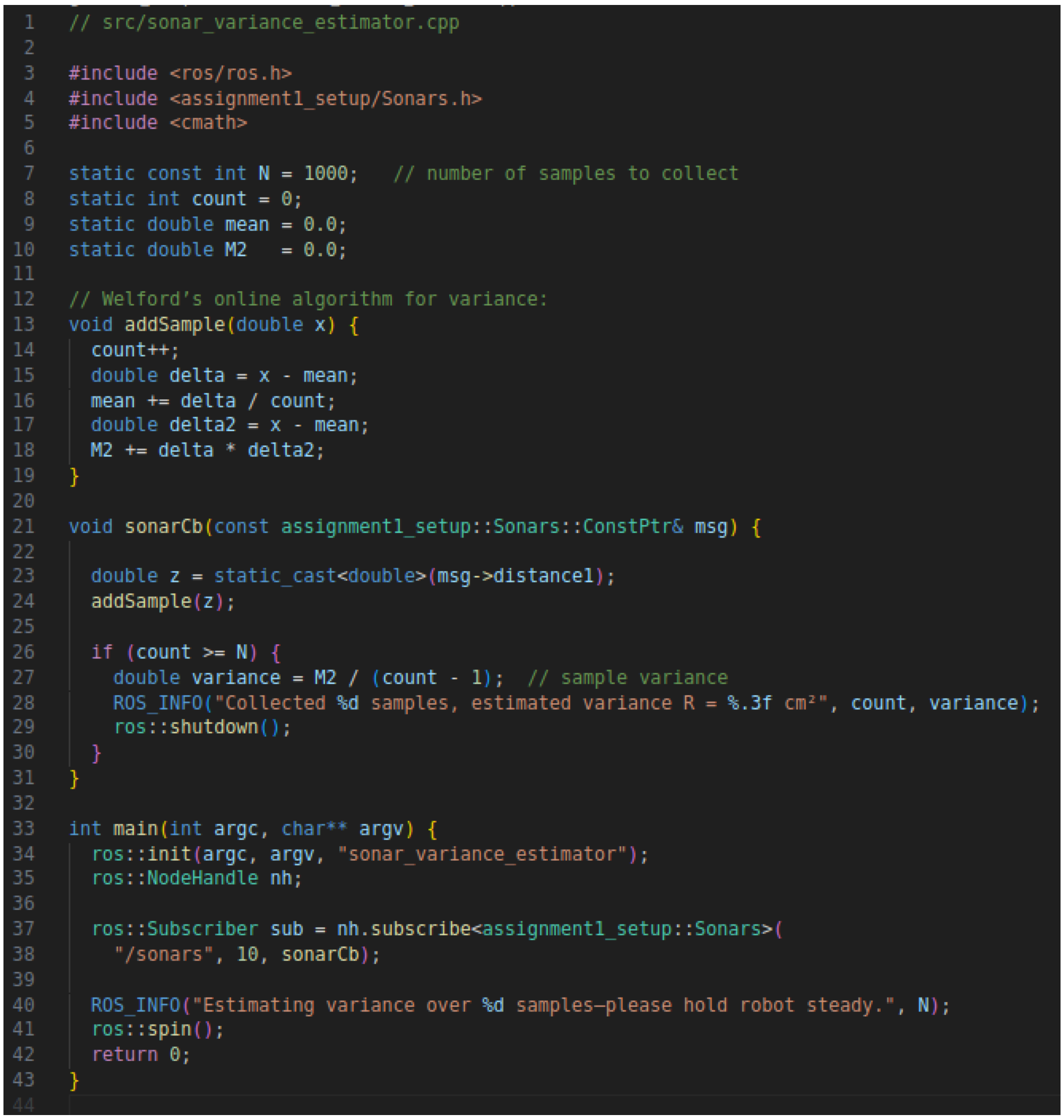
Proportional (P) acts on the current error, driving a velocity directly proportional to the distance error. Derivative (D) predicts future trends by reacting the errors rate of change, damping oscillations. Integral (I) accumulates past errors to eliminate persistent small offsets. Specifically in this task, kp speeds the robot towards the box, kd prevents overshoot, and ki corrects very small positional errors when close to the box.

Advantages and disadvantages of PID for this task:

Advantages of a PID control algorithm includes fast convergence to the box when well tuned, increase in accuracy by eliminating steady-state error with the I term. Automatically compensates for changes, simple to integrate in ROS, and very computationally lightweight. Disadvantages is that if not tuned well, overly large gains can cause instability, its sensitive to sensor noise, and requires manual tuning of three different parameters. Overall PID feedback control provides a flexible, robust, and computationally efficient way to approach and track a target with sonar sensors. The main challenge is noise sensitivity, which can be largely mitigated with the implementation of a Kalman filter.

Task 2: Using Noisy Sonar Sensors

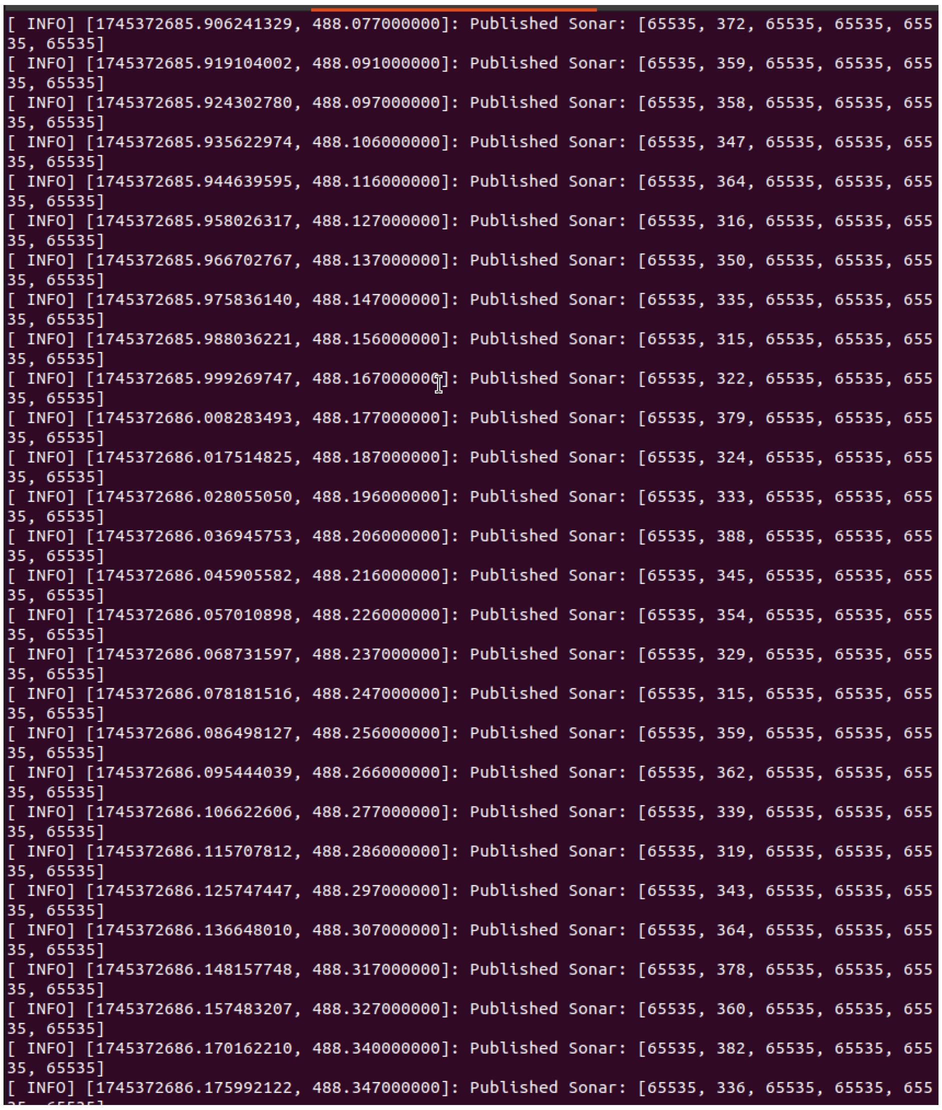
First, the estimated variance was calculated. This was implemented by placing the box Infront of the bot 350cm away. Then, Welford's online algorithm for variance was used to estimate the variance. To do this, a new node that calculated the variance was implemented, as seen here:



```cpp
// src/sonar_variance_estimator.cpp

#include <ros/ros.h>
#include <assignment1_setup/Sonars.h>
#include <cmath>

static const int N = 1000;   // number of samples to collect
static int count = 0;
static double mean = 0.0;
static double M2   = 0.0;

// Welford's online algorithm for variance:
void addSample(double x) {
  count++;
  double delta = x - mean;
  mean += delta / count;
  double delta2 = x - mean;
  M2 += delta * delta2;
}

void sonarCb(const assignment1_setup::Sonars::ConstPtr& msg) {

  double z = static_cast<double>(msg->distance1);
  addSample(z);

  if (count >= N) {
    double variance = M2 / (count - 1);  // sample variance
    ROS_INFO("Collected %d samples, estimated variance R = %.3f cm²", count, variance);
    ros::shutdown();
  }
}

int main(int argc, char** argv) {
  ros::init(argc, argv, "sonar_variance_estimator");
  ros::NodeHandle nh;

  ros::Subscriber sub = nh.subscribe<assignment1_setup::Sonars>(
    "/sonars", 10, sonarCb);

  ROS_INFO("Estimating variance over %d samples—please hold robot steady.", N);
  ros::spin();
  return 0;
}
```

Figure 17: Variance Node

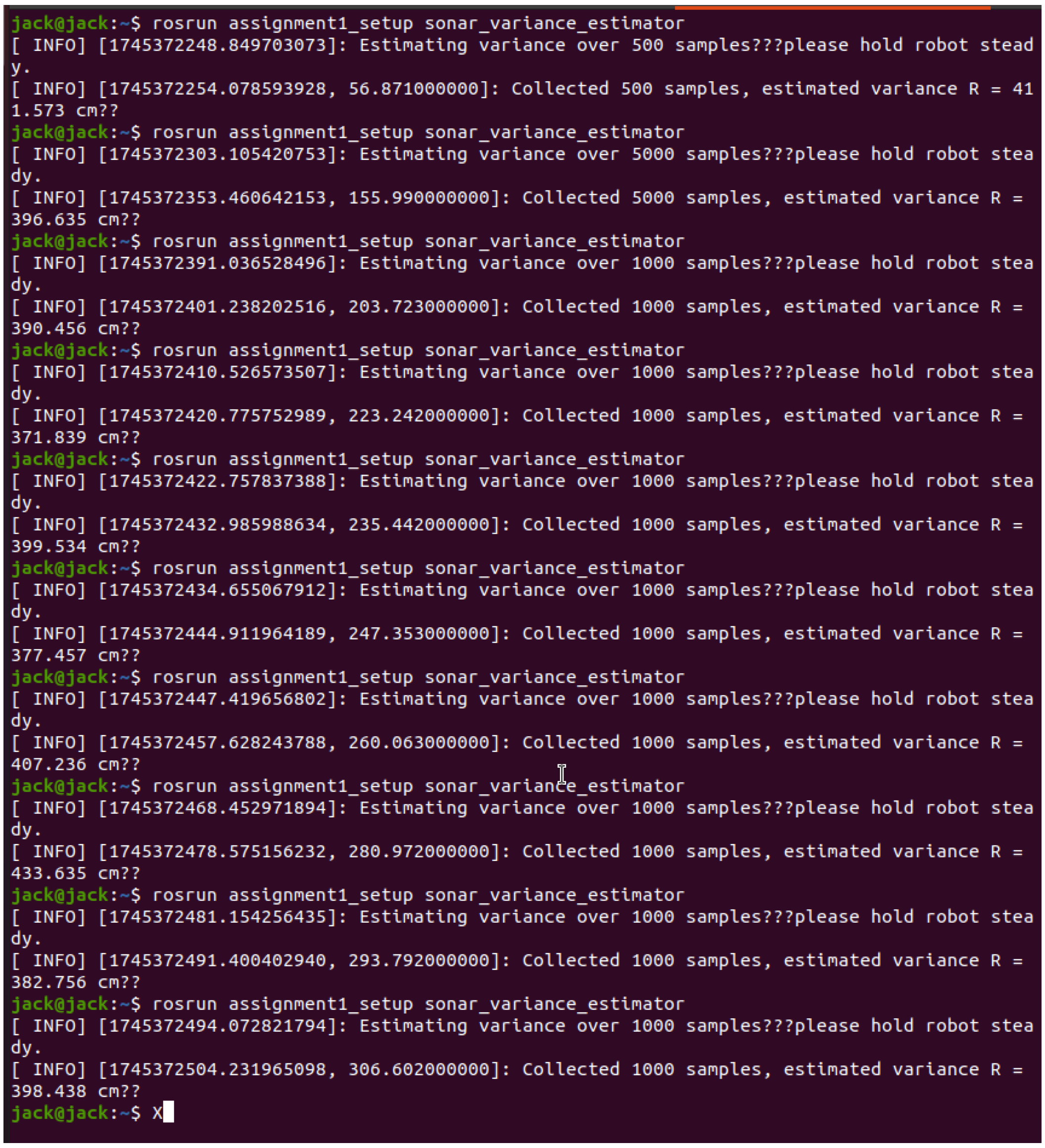
The noisy sonars node was then run with *assignment1_setup noisy_sonars*, the output can be seen here:



```
[ INFO] [1745372685.906241329, 488.077000000]: Published Sonar: [65535, 372, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.919104002, 488.091000000]: Published Sonar: [65535, 359, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.924302780, 488.097000000]: Published Sonar: [65535, 358, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.935622974, 488.106000000]: Published Sonar: [65535, 347, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.944639595, 488.116000000]: Published Sonar: [65535, 364, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.958026317, 488.127000000]: Published Sonar: [65535, 316, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.966702767, 488.137000000]: Published Sonar: [65535, 350, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.975836140, 488.147000000]: Published Sonar: [65535, 335, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.988036221, 488.156000000]: Published Sonar: [65535, 315, 65535, 65535, 655
35, 65535]
[ INFO] [1745372685.999269747, 488.167000000]: Published Sonar: [65535, 322, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.008283493, 488.177000000]: Published Sonar: [65535, 379, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.017514825, 488.187000000]: Published Sonar: [65535, 324, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.028055050, 488.196000000]: Published Sonar: [65535, 333, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.036945753, 488.206000000]: Published Sonar: [65535, 388, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.045905582, 488.216000000]: Published Sonar: [65535, 345, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.057010898, 488.226000000]: Published Sonar: [65535, 354, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.068731597, 488.237000000]: Published Sonar: [65535, 329, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.078181516, 488.247000000]: Published Sonar: [65535, 315, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.086498127, 488.256000000]: Published Sonar: [65535, 359, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.095444039, 488.266000000]: Published Sonar: [65535, 362, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.106622606, 488.277000000]: Published Sonar: [65535, 339, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.115707812, 488.286000000]: Published Sonar: [65535, 319, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.125747447, 488.297000000]: Published Sonar: [65535, 343, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.136648010, 488.307000000]: Published Sonar: [65535, 364, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.148157748, 488.317000000]: Published Sonar: [65535, 378, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.157483207, 488.327000000]: Published Sonar: [65535, 360, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.170162210, 488.340000000]: Published Sonar: [65535, 382, 65535, 65535, 655
35, 65535]
[ INFO] [1745372686.175992122, 488.347000000]: Published Sonar: [65535, 336, 65535, 65535, 655
```

Figure 18: Noisy sonars node output

Then, the variance estimator was run 10 times, 8 at 1000 samples, 1 at 500, and 1 at 5000 samples.



```
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372248.849703073]: Estimating variance over 500 samples???please hold robot stead
y.
[ INFO] [1745372254.078593928, 56.871000000]: Collected 500 samples, estimated variance R = 41
1.573 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372303.105420753]: Estimating variance over 5000 samples???please hold robot stea
dy.
[ INFO] [1745372353.460642153, 155.990000000]: Collected 5000 samples, estimated variance R =
396.635 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372391.036528496]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372401.238202516, 203.723000000]: Collected 1000 samples, estimated variance R =
390.456 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372410.526573507]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372420.775752989, 223.242000000]: Collected 1000 samples, estimated variance R =
371.839 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372422.757837388]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372432.985988634, 235.442000000]: Collected 1000 samples, estimated variance R =
399.534 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372434.655067912]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372444.911964189, 247.353000000]: Collected 1000 samples, estimated variance R =
377.457 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372447.419656802]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372457.628243788, 260.063000000]: Collected 1000 samples, estimated variance R =
407.236 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372468.452971894]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372478.575156232, 280.972000000]: Collected 1000 samples, estimated variance R =
433.635 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372481.154256435]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372491.400402940, 293.792000000]: Collected 1000 samples, estimated variance R =
382.756 cm??
jack@jack:~$ rosrun assignment1_setup sonar_variance_estimator
[ INFO] [1745372494.072821794]: Estimating variance over 1000 samples???please hold robot stea
dy.
[ INFO] [1745372504.231965098, 306.602000000]: Collected 1000 samples, estimated variance R =
398.438 cm??
jack@jack:~$ X
```

Figure 19: Variance Estimator

To calculate the overall estimated variance, the average of all these readings was taken. This led to an average estimated variance of 396.956. This coincides with the data from figure 14, as sqrt (396.956) is 19.92, which looks to be the standard deviation that the readings vary by.

A Kalman filter was then implemented into my controller.cpp. Following this formula:

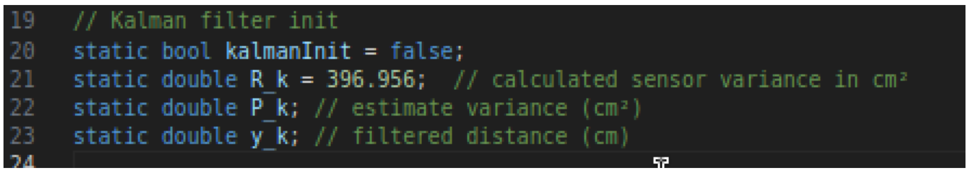
$$K_i = \frac{P_i^-}{P_i^- + R_i}$$

$$y_i = y_i^- + K_i(z_i - y_i^-)$$

$$P_i = (1 - K_i)P_i^-$$

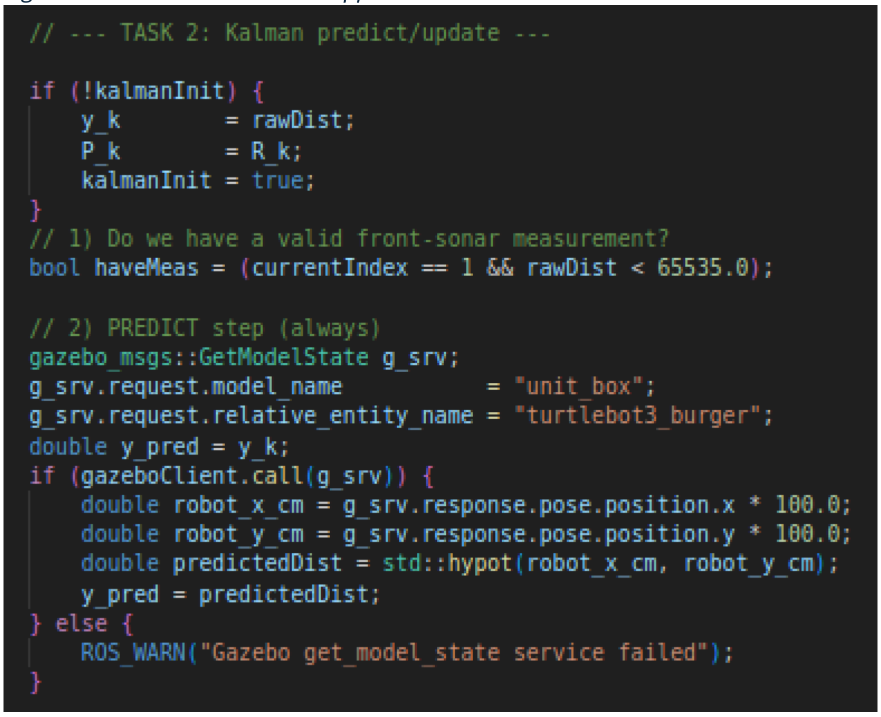
Figure 20: Kalman Filter Formula

This can be seen in my code here where the gazebo/get_model_state service is used to find the position of the robot, and the noisy_sonar_sensors for the position of the box:



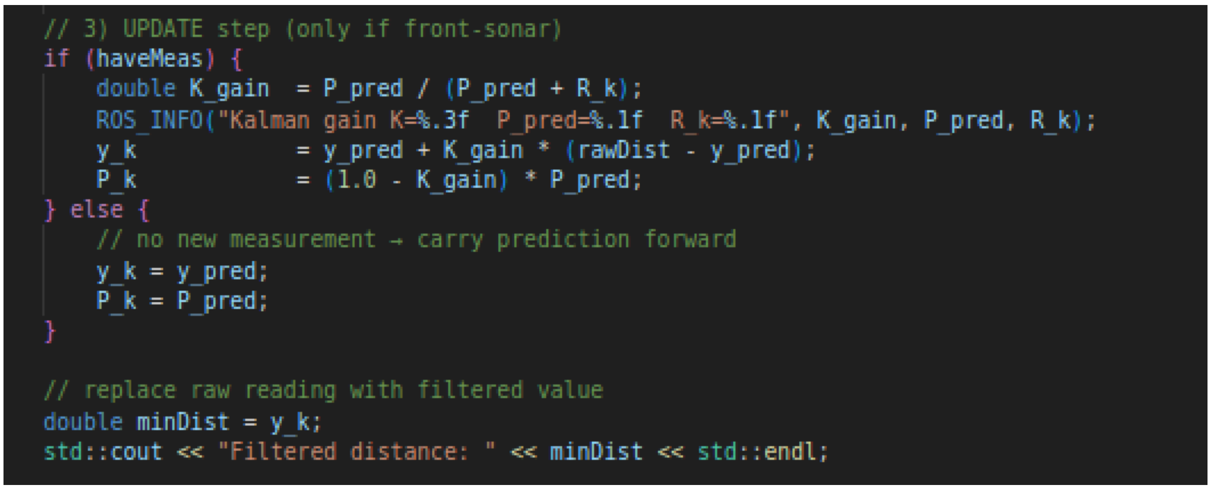
```
// Kalman filter init
static bool kalmanInit = false;
static double R_k = 396.956;  // calculated sensor variance in cm²
static double P_k; // estimate variance (cm²)
static double y_k; // filtered distance (cm)
```

Figure 21: Kalman Filter Code Snippet 1



```
// --- TASK 2: Kalman predict/update ---

if (!kalmanInit) {
    y_k        = rawDist;
    P_k        = R_k;
    kalmanInit = true;
}
// 1) Do we have a valid front-sonar measurement?
bool haveMeas = (currentIndex == 1 && rawDist < 65535.0);

// 2) PREDICT step (always)
gazebo_msgs::GetModelState g_srv;
g_srv.request.model_name           = "unit_box";
g_srv.request.relative_entity_name = "turtlebot3_burger";
double y_pred = y_k;
if (gazeboClient.call(g_srv)) {
    double robot_x_cm = g_srv.response.pose.position.x * 100.0;
    double robot_y_cm = g_srv.response.pose.position.y * 100.0;
    double predictedDist = std::hypot(robot_x_cm, robot_y_cm);
    y_pred = predictedDist;
} else {
    ROS_WARN("Gazebo get_model_state service failed");
}
```

Figure 22: Kalman Filter code snippet 2

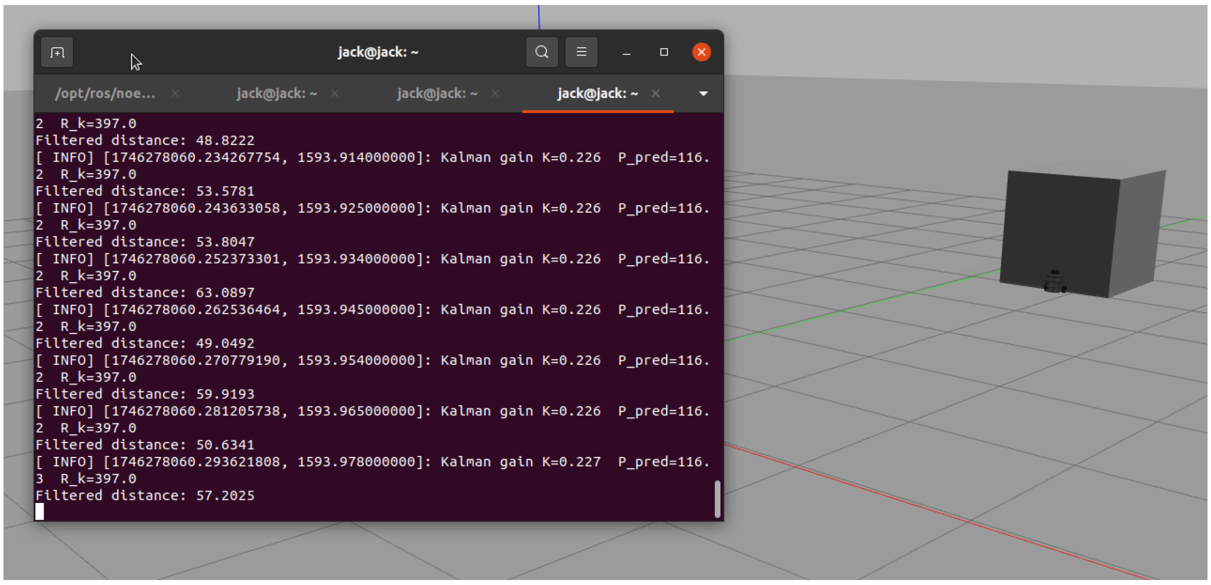


```
// 3) UPDATE step (only if front-sonar)
if (haveMeas) {
    double K_gain   = P_pred / (P_pred + R_k);
    ROS_INFO("Kalman gain K=%.3f  P_pred=%.1f  R_k=%.1f", K_gain, P_pred, R_k);
    y_k             = y_pred + K_gain * (rawDist - y_pred);
    P_k             = (1.0 - K_gain) * P_pred;
} else {
    // no new measurement → carry prediction forward
    y_k = y_pred;
    P_k = P_pred;
}

// replace raw reading with filtered value
double minDist = y_k;
std::cout << "Filtered distance: " << minDist << std::endl;
```

Figure 23: Kalman Filter Code Snippet 3

This filter worked successfully, as can be seen here, where the noisy sonars are in operation, and the robot moved to the box relatively smoothly. The Kalman gain is calculated to be 0.226, which shows that 22.6% of the new estimate comes from the sonar reading, and 77.4% of the new estimate comes from the model based prediction.

This low Kalman gain value is respectable for the nature of this environment, as the box does not move towards or away from the robot as it is static, and there are no other obstacles in the environment.



```
2  R_k=397.0
Filtered distance: 48.8222
[ INFO] [1746278060.234267754, 1593.914000000]: Kalman gain K=0.226  P_pred=116.
2  R_k=397.0
Filtered distance: 53.5781
[ INFO] [1746278060.243633058, 1593.925000000]: Kalman gain K=0.226  P_pred=116.
2  R_k=397.0
Filtered distance: 53.8047
[ INFO] [1746278060.252373301, 1593.934000000]: Kalman gain K=0.226  P_pred=116.
2  R_k=397.0
Filtered distance: 63.0897
[ INFO] [1746278060.262536464, 1593.945000000]: Kalman gain K=0.226  P_pred=116.
2  R_k=397.0
Filtered distance: 49.0492
[ INFO] [1746278060.270779190, 1593.954000000]: Kalman gain K=0.226  P_pred=116.
2  R_k=397.0
Filtered distance: 59.9193
[ INFO] [1746278060.281205738, 1593.965000000]: Kalman gain K=0.226  P_pred=116.
2  R_k=397.0
Filtered distance: 50.6341
[ INFO] [1746278060.293621808, 1593.978000000]: Kalman gain K=0.227  P_pred=116.
3  R_k=397.0
Filtered distance: 57.2025
```

Figure 24: Robot using Kalman Filter

Task 2 Discussion:

Theory of the Kalman Filter:

The Kalman filter is a recursive estimator that fuses the predictions of the model with the noisy measurements to produce an optimal estimate of the state of the box relative to the robot. At each time step i, it performs two main operations; predict, and update.

Predict step:

$$y_i^- = f(y_{i-1}), \qquad P_i^- = P_{i-1} + Q$$

Where,

- y_i^- is the *a priori* state estimate before seeing the new measurement
- P_i^- is its associated uncertainty (covariance)
- Q is the process noise covariance (capturing model uncertainty)
- $f(\cdot)$ is the state transition (in this case y_{i-1})

Update step:

$$K_i = \frac{P_i^-}{P_i^- + R}, \qquad y_i = y_i^- + K_i\,(z_i - y_i^-), \qquad P_i = (1 - K_i)\,P_i^-.$$

Where,

- z_i is the noisy measurement
- R is the measurement noise variance
- K_i (the Kalman gain) balances trust between prediction and measurement
- y_i and P_i are the *a posteri* estimate and its covariance.

Using this task as an example:

The Kalman filter was used in this task to smooth the noisy /sonars readings so the PID controller sees a stable distance. As stated earlier, the estimate measurement variance R was found to be 396.956 cm^2. The filter is bootstrapped on the first reading to $y_0 = z_0$, $P_0 = R$.

Then on each iteration, the predict step takes place where the process noise is assumed to be $Q = 0$:

$$y_i^- = y_{i-1}, \qquad P_i^- = P_{i-1}$$

Then the update step takes place:

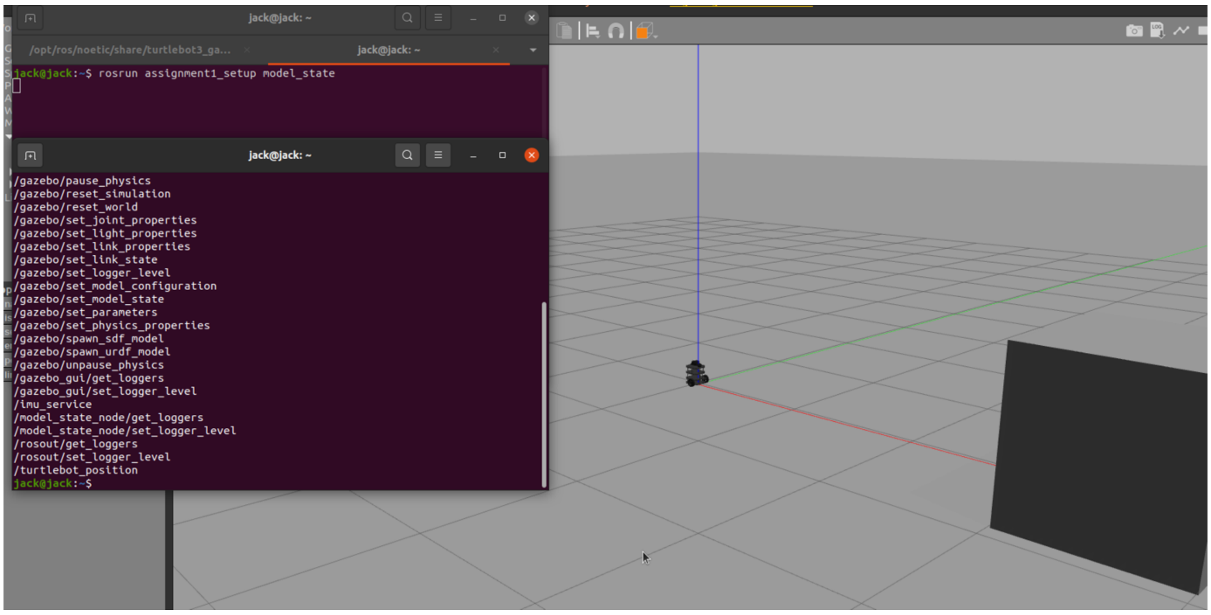
$$K_i = \frac{P_i^-}{P_i^- + R}, \qquad y_i = y_i^- + K_i\,(z_i - y_i^-)$$

As R reflects how jittery the sonar is, the Kalman gain naturally downweighs erratic spikes which generates a smooth, trustworthy distance for the PID controller to use.

The Kalman filter is used as it recursively updates in real time, and can trust measurements more when uncertainly is low, and trust predictions more when measurements are noisy.

Task 3:

The model state was run and turtlebot_position was a new rosnode as seen below.

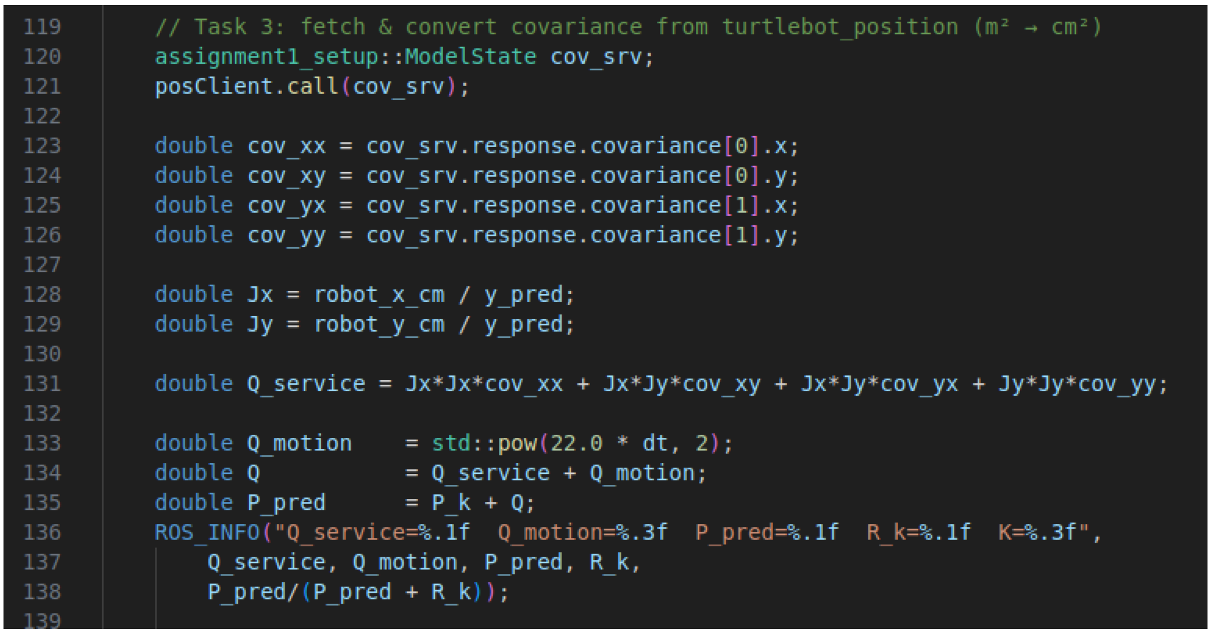


```
jack@jack:~$ rosrun assignment1_setup model_state
```

```
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_light_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physics_properties
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
/gazebo_gui/get_loggers
/gazebo_gui/set_logger_level
/imu_service
/model_state_node/get_loggers
/model_state_node/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
/turtlebot_position
jack@jack:~$
```

Figure 25: model_state startup, turtlebot_position service shown

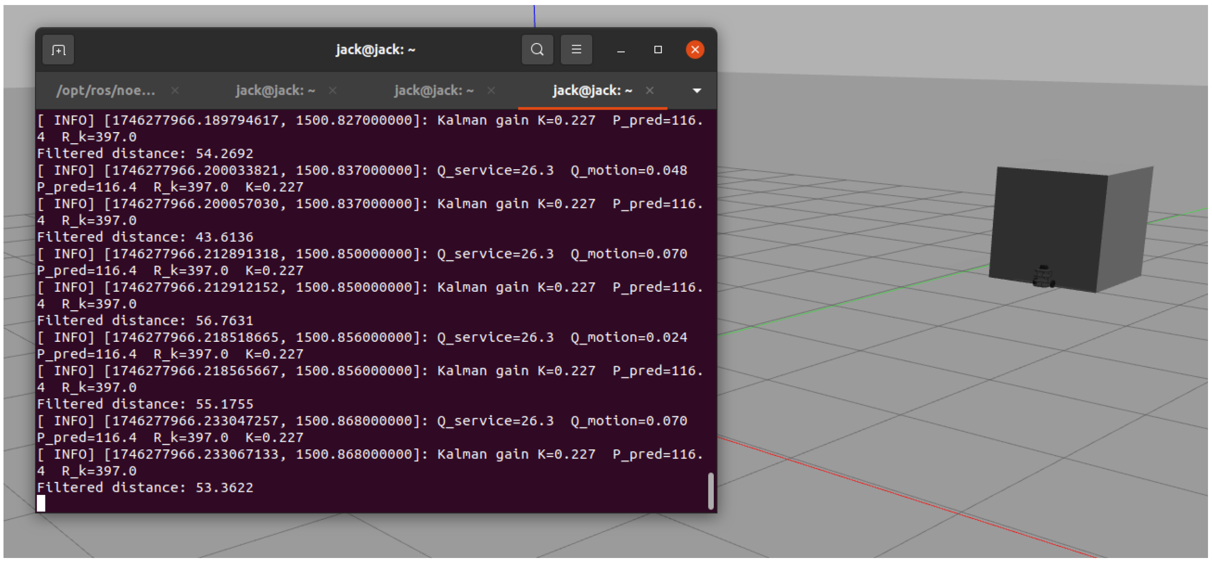
Then the covariance matrix was added into the code here.



```cpp
    // Task 3: fetch & convert covariance from turtlebot_position (m² → cm²)
    assignment1_setup::ModelState cov_srv;
    posClient.call(cov_srv);

    double cov_xx = cov_srv.response.covariance[0].x;
    double cov_xy = cov_srv.response.covariance[0].y;
    double cov_yx = cov_srv.response.covariance[1].x;
    double cov_yy = cov_srv.response.covariance[1].y;

    double Jx = robot_x_cm / y_pred;
    double Jy = robot_y_cm / y_pred;

    double Q_service = Jx*Jx*cov_xx + Jx*Jy*cov_xy + Jx*Jy*cov_yx + Jy*Jy*cov_yy;

    double Q_motion    = std::pow(22.0 * dt, 2);
    double Q           = Q_service + Q_motion;
    double P_pred      = P_k + Q;
    ROS_INFO("Q_service=%.1f  Q_motion=%.3f  P_pred=%.1f  R_k=%.1f  K=%.3f",
        Q_service, Q_motion, P_pred, R_k,
        P_pred/(P_pred + R_k));

```

Figure 26: Covariance Matrix Addition

As can be seen, the bot successfully moves toward the box.



```
[ INFO] [1746277966.189794617, 1500.827000000]: Kalman gain K=0.227  P_pred=116.
4  R_k=397.0
Filtered distance: 54.2692
[ INFO] [1746277966.200033821, 1500.837000000]: Q_service=26.3  Q_motion=0.048
P_pred=116.4  R_k=397.0  K=0.227
[ INFO] [1746277966.200057030, 1500.837000000]: Kalman gain K=0.227  P_pred=116.
4  R_k=397.0
Filtered distance: 43.6136
[ INFO] [1746277966.212891318, 1500.850000000]: Q_service=26.3  Q_motion=0.070
P_pred=116.4  R_k=397.0  K=0.227
[ INFO] [1746277966.212912152, 1500.850000000]: Kalman gain K=0.227  P_pred=116.
4  R_k=397.0
Filtered distance: 56.7631
[ INFO] [1746277966.218518665, 1500.856000000]: Q_service=26.3  Q_motion=0.024
P_pred=116.4  R_k=397.0  K=0.227
[ INFO] [1746277966.218565667, 1500.856000000]: Kalman gain K=0.227  P_pred=116.
4  R_k=397.0
Filtered distance: 55.1755
[ INFO] [1746277966.233047257, 1500.868000000]: Q_service=26.3  Q_motion=0.070
P_pred=116.4  R_k=397.0  K=0.227
[ INFO] [1746277966.233067133, 1500.868000000]: Kalman gain K=0.227  P_pred=116.
4  R_k=397.0
Filtered distance: 53.3622
```

Figure 27: Robot movement after covariance matrix

Task 3 Discussion:

Why a Covariance Matrix matters:

When extending a Kalman filter from 1D to a 2D position (x, y), a single variance no longer suffices. Real world sensors exhibit different noise level along each axis and cross axis correlations. Slippage often pushes the robot diagonally linking error in both x and y. This can be captured in a 2x2 covariance matrix.

$$\Sigma = \begin{pmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 \\ \sigma_{yx}^2 & \sigma_{yy}^2 \end{pmatrix}$$

Where

- $\sigma_{xx}^2, \sigma_{yy}^2$ are the variances in x and y.
- $\sigma_{xy}^2 \ = \ \sigma_{yx}^2$ measures how deviations along one axis accompany deviations along the other.

Challenges in real measurements:

Data collection requires many ground truth samples under varied conditions. Noise stats shift as environments change or hardware gets older. Turns on uneven ground can create complex errors, and its computationally expensive as propagating a full matrix is heavier than a scalar variance.

Role in the Kalman Filter:

When implementing into the Kalman filter, the predict step becomes $P_i^- \ = P_{i-1} + Q$ where $Q \ = \ \Sigma$. In the update step, the filter calculates a gain based on how uncertain the prediction is vs the measurement noise and uses that to blend the predicted position with the new x, y reading. It then shrinks the overall uncertainty most in the directions where the sensors are more precise, keeping any error correlations between x and y. By modelling both individual variances and their covariance, the 2D Kalman filter fuses the noisy internal pose estimates with the sonar readings, leading to smoother and more accurate robot positionings for the PID controller.