Autonomous Mobile Robots Template

Murat Aydemir

[[1]](#footnote-1)

***Abstract*—Write abstract at the end.**

***Index Terms*—Autonomous Mobile Robots, Obstacle Avoidance, Path Planing, TurtleBot**.

# I. INTRODUCTION

Discuss about Autonomous Mobile Robots and Obstacle Avoidance Path Planning problem. Explain the structure of the paper.

Autonomous Robots and robotic research are becoming more important as the robots are quickly becoming a valuable part of many industrial applications and becoming widespread in our everyday life such as hotels, restaurants, hospitals and even autonomous cars. There are many seemingly simple but rather difficult questions to answer in order to achieve autonomous functionality with the robots. For example, Path Planning approaches try to answer the question of how to get to the goal, and map representation techniques try to solve where have the robot been and can move to. In this paper, Obstacle Avoidance Path Planning problem will be examined and a bug-based algorithm with occupancy grid mapping will be demonstrated with kobuki robots.

# II. Related Works

Discuss about existing obstacle avoiding path planning approaches

One of the simplest but surprisingly powerful algorithm class is bug based path planning algorithms, where the robot knows the coordinates of the goal, but has no knowledge of the surroundings or the obstacles on the way. For example, an autonomous vacuum robot that is trying to reach to the charging dock station, but there is furniture in the way.

Bug-0 robots simply move towards the goal, or follow the obstacles until they can head to the goal again. Bug-0 algorithm is not a complete algorithm and the robots will be easily foiled by the complex obstacles. Bug-1 algorithm adds memory to bug-0, where it circumnavigates the obstacle when obstructed, and remembers the closest point to the goal around the perimeter. After that it navigates to the point of closes approach and heads towards the goal. Bug-1 algorithm is complete, where if there is a path exists, it will find a path in finite time, which makes it more predictable. Bug-2 algorithm is a greedy version of bug-1 algorithm, where it creates a virtual line between starting and the goal points, and follows the obstacle upon encountering, until it hits the line again. On average bug-2 will outperform bug-1, but bug-1 is more predictable. In this paper, Bug-2 algorithm is implemented on kobuki robots, and an 2D occupancy grid matrix is utilized for the mapping of environment and the obstacles.

# III. Proposed/Implemented Approach

## A. Algorithm

Discuss about the implemented or proposed algorithm. Sample Algorithm.

Left turning Bug 2 Algorithm is used for Path planning for a single Robot. Bug algorithms only have local environment knowledge, but no map information. The location of the goal is known, but the obstacles or possible paths are not clear at the beginning. Bug 2 algorithm is a wall following obstacle avoidance algorithm that creates an m-line between the start and goal, then follows the line towards the goal. If the Robot hits an obstacle it follows the obstacle until it hits the m-line again, then leave the obstacle and move toward the goal.

For the generation of movement with Kobuki Robots, two main robot modes are defined as GO\_TO\_GOAL\_MODE and WALL\_FOLLOWING\_MODE. State switches can be seen in the following state machine diagram:

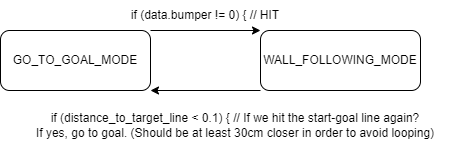


Figure 1: Main Robot movement modes.

The Robot starts with GO\_TO\_GOAL\_MODE. The current location and angle of the robot is calculated each cycle with odometry functions using the sensor readings of kobuki, and according to the error margins, the robot either goes straight to the goal, or adjusts the heading with the defined rotation speed. In the meantime, the mapping algorithm will mark the visited grids as empty on the occupancy grid matrix. The state machine can be seen as:

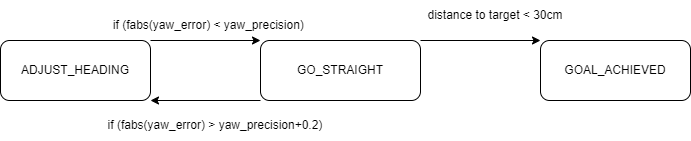
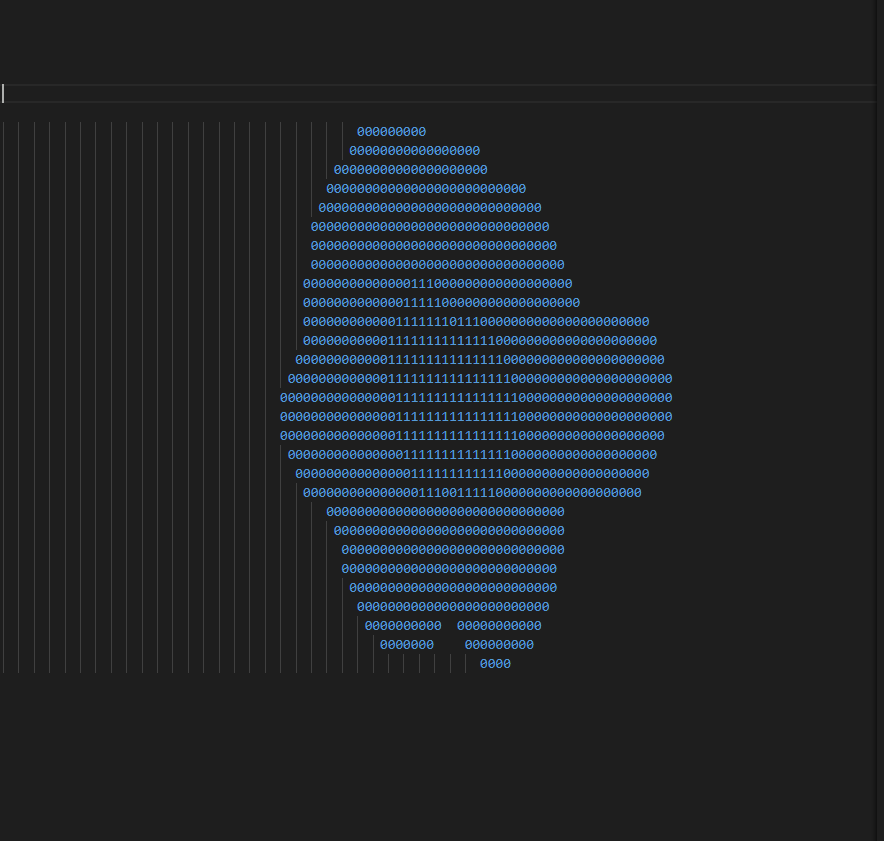


Figure 2: Moving modes state machine

When the robot hits an obstacle, it marks down the location of the bumper’s hit point as obstacle on the map, and also dilates the obstacle point by robot’s radius. The robot is represented as a point in this algorithm, therefore the dilation of obstacles allows the robot to avoid moving into those cells again. After the hit, the robot directly switches to WALL\_FOLLOWING\_MODE as stated in Figure 1.

If the Robot is in WALL\_FOLLOWING\_MODE, the Robot first rotates left until the bumper sensor reads zero; in other words, no more obstacles on the front facing side of the Robot. Then the robot checks the occupancy grid for any front, left or right obstacles on the way. If there are front obstacles, the robot keeps rotating left, until the front is facing an empty path. Then it starts moving by keeping its distance to the wall on the right by constantly checking occupancy grid matrix. If any other obstacles were hit, the Robot again starts turning left until the bumper reads zero again and start the wall following motions afterwards. When the robot reaches to the m-line, the mode is changed to GO\_TO\_GOAL\_MODE as in Figure 1. After the goal is reached the robot will move towards the new goal in a similar pattern, avoiding the saved obstacles on the occupancy grid, or exploring and saving the newly visited locations on the map for the future.

  
Figure 3: An example resulting occupancy grid after exploration where 0 is visited, 1 is obstacle cells.

For the future work, the robot will first check the occupancy grid for any possible shortest paths along the visited empty cells using A\* shortest path algorithm., then move toward the goal using the found path. If no path can be found, it will switch to bug-2 algorithm, and move toward the goal while exploring the obstacles on the map.

For multiple connected Robots, different variations of Bug 0 and Bug 2 algorithms can be used. E.g., one Robot with left turning variation of Bug 2, and one Robot with right turning Bug 2 algorithm can be exploring the map simultaneously, and after one of them reach to the goal, they will share the locations of the obstacles with each other, and then follow the shortest path on the way back without hitting to the obstacles again.

Intel realsense depth camera will be used for improved obstacle detection, instead of relying on bumpers for detection. This will also eliminate the need of square motion at the wall following mode where the robot hits, turns 90 degrees, moves for certain distance and turns 90 degrees again only to hit the wall again. Instead, the robot can sustain certain distance to the wall, and keep moving forward until the edge of the obstacle is found. This will save considerable amount of time when the obstacle wall is great in length.

## B. Kobuki Robots

Discuss about Kubuki robots and Raspberry pi

Kobuki robotic turtle bot is used as the main platform in the project. It has 3 bumper crash sensors on the front, left and right. Kobuki c++ driver library is utilized to interface with the kobuki base controller. Raspberry pi 3B is connected to the kobuki robot with the USB cable where the sensor readout from kobuki, and commands to the kobuki robot is communicated.

## C. UWB

Discuss about Ultra-Wide band indoor localization.

Relying only on odometry for the navigation is error-prone since the robot’s wheels are not ideal for every type of environment or floor material, where it can slip or jump on some obstacles which creates some abnormal calculations of pose values. Ultra-wide band is a promising technology for indoor localization for determining the exact location of the mobile Robot in the environment.

D. General Architecture

Explain the general architecture of the implemented system. (using figures)

# IV. Evaluation

Discuss about the testbed environment and metrics (time, movement distance). Present figures about the distance and/or time. Present figures of testbed.

V. Conclusion

Write this at the end.

# Appendix

Provide source codes in formal format and colors.

#include <string>

#include <csignal>

#include <fstream>

#include <vector>

#include <sstream>

#include <chrono>

#include <ctime>

#include <ecl/geometry.hpp>

#include <ecl/time.hpp>

#include <ecl/sigslots.hpp>

#include <ecl/linear\_algebra.hpp>

#include <ecl/command\_line.hpp>

#include "kobuki\_core/kobuki.hpp"

#include "kobuki\_core/packets/core\_sensors.hpp"

#define FORWARD\_SPEED 0.25

#define ROTATION\_SPEED 1.0

#define FAST\_ROTATION\_SPEED 1.3

#define MAP\_SIZE 200 // n of cells

#define MAP\_ORIGIN 100 // origin point is at [100][100]

#define GRID\_SIZE 0.05 // m

#define ROBOT\_RADIUS 0.17 // m

#define LEFT\_BUMPER 4

#define RIGHT\_BUMPER 1

#define CENTER\_BUMPER 2

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Classes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class KobukiManager

{

public:

    KobukiManager(

        const std::string &device,

        const double &length,

        const bool &disable\_smoothing,

        const double target\_x,

        const double target\_y) : dx(0.0), dth(0.0),

                                 length(length),

                                 slot\_stream\_data(&KobukiManager::processStreamData, \*this)

    {

        this->temp\_target\_x = this->target\_x = target\_x;

        this->temp\_target\_y = this->target\_y = target\_y;

        kobuki::Parameters parameters;

        parameters.sigslots\_namespace = "/kobuki";

        parameters.device\_port = device;

        parameters.enable\_acceleration\_limiter = !disable\_smoothing;

        robot\_mode = GO\_TO\_GOAL\_MODE;

        moving\_state = ADJUST\_HEADING;

        for (auto &row : occupancy\_grid)

        {

            for (auto &column : row)

            {

                column = -1;

            }

        }

        kobuki.init(parameters);

        kobuki.enable();

        slot\_stream\_data.connect("/kobuki/stream\_data");

    }

    ~KobukiManager()

    {

        kobuki.setBaseControl(0, 0); // linear\_velocity, angular\_velocity in (m/s), (rad/s)

        kobuki.disable();

    }

    void processStreamData()

    {

        ecl::linear\_algebra::Vector3d pose\_update;

        ecl::linear\_algebra::Vector3d pose\_update\_rates;

        kobuki.updateOdometry(pose\_update, pose\_update\_rates);

        pose = ecl::concatenate\_poses(pose, pose\_update);

        current\_x = pose[0];

        current\_y = pose[1];

        pose[2] = current\_yaw = kobuki.getHeading(); // override broken odometry heading with the gyro data

        dx += pose\_update[0];  // dx

        dth += pose\_update[2]; // dheading

        data = kobuki.getCoreSensorData();

        processMotion();

        // testMotion(); / only for debugging

    }

    void dilateCell(int x, int y, int value) {

        int grid\_numbers\_in\_radius = (int)(ROBOT\_RADIUS/GRID\_SIZE)+1;

        //cout << "d: " << x << " " << y << " " << grid\_numbers\_in\_radius << endl;

        for (int row = y - grid\_numbers\_in\_radius; row <= y + grid\_numbers\_in\_radius; row++) {

            if (row < 0 || row >= MAP\_SIZE) continue;

            //cout << "r " << row << " "<< endl;

            for (int column = x - grid\_numbers\_in\_radius; column <= x + grid\_numbers\_in\_radius; column++) {

                if (column < 0 || column >= MAP\_SIZE) continue;

                //cout << "c " << column << endl;

                if ((abs(row-y) + abs(column-x)) <= grid\_numbers\_in\_radius+1) {

                    if (occupancy\_grid[MAP\_SIZE-row][column] != 1) {

                        //cout << " marking " << " row " << row << " column " << column << " as " << value << endl;

                        occupancy\_grid[MAP\_SIZE-row][column] = value;

                    }

                }

            }

        }

    }

    // Technique used for mapping is Point-robot assumption. Meaning that the robot is represented as a point, and obstacles are Dilated by robot’s radius

    void updateMap()

    {

        // Mark the object cell from the hit point, and use Dilation function

        int x, y;

        //cout << "bumper: " << (int)data.bumper << endl;

        if (data.bumper == LEFT\_BUMPER) { // if left bumper is hit,

            x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw + ecl::pi \* 0.25)) / GRID\_SIZE) + MAP\_ORIGIN;

            y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw + ecl::pi \* 0.25)) / GRID\_SIZE) + MAP\_ORIGIN;

            cout << "left hit ["  << x << " " << y << endl;

            dilateCell(x, y, 1);

        } else if (data.bumper == RIGHT\_BUMPER) { // if right bumper is hit

            x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw - ecl::pi \* 0.25)) / GRID\_SIZE) + MAP\_ORIGIN;

            y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw - ecl::pi \* 0.25)) / GRID\_SIZE) + MAP\_ORIGIN;

            cout << "right hit ["  << x << " " << y << endl;

            dilateCell(x, y, 1);

        } else if ((int)data.bumper != 0) { // if any other bumper combination is hit, mark the front point

            x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw)) / GRID\_SIZE) + MAP\_ORIGIN;

            y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw)) / GRID\_SIZE) + MAP\_ORIGIN;

            cout << " hit ["  << x << " " << y << endl;

            dilateCell(x, y, 1);

        } else { // if no hit, mark the current cell as clean. Note that if the cell is already 1, it cannot be overwritten

            x = (int)round(current\_x / GRID\_SIZE) + MAP\_ORIGIN;

            y = (int)round(current\_y / GRID\_SIZE) + MAP\_ORIGIN;

            dilateCell(x, y, 0);

        }

        return;

    }

    void checkMap() // check obstables in front for movement controls

    {

        front\_obstacle = right\_front\_obstacle = right\_obstacle = left\_front\_obstacle = left\_obstacle = false;

        int x, y; // row, column

        // check front

        x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw)) / GRID\_SIZE) + MAP\_ORIGIN;

        y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw)) / GRID\_SIZE) + MAP\_ORIGIN;

        if (occupancy\_grid[MAP\_SIZE-y][x] == 1) front\_obstacle = true;

        // check right\_front

        x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw-ecl::pi/6)) / GRID\_SIZE) + MAP\_ORIGIN;

        y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw-ecl::pi/6)) / GRID\_SIZE) + MAP\_ORIGIN;

        if (occupancy\_grid[MAP\_SIZE-y][x] == 1) right\_front\_obstacle = true;

        // check right

        x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw-ecl::pi\*0.5)) / GRID\_SIZE) + MAP\_ORIGIN;

        y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw-ecl::pi\*0.5)) / GRID\_SIZE) + MAP\_ORIGIN;

        if (occupancy\_grid[MAP\_SIZE-y][x] == 1) right\_obstacle = true;

        // check left\_front

        x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw+ecl::pi/6)) / GRID\_SIZE) + MAP\_ORIGIN;

        y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw+ecl::pi/6)) / GRID\_SIZE) + MAP\_ORIGIN;

        if (occupancy\_grid[MAP\_SIZE-y][x] == 1) left\_front\_obstacle = true;

        // check left

        x = (int)round((current\_x + ROBOT\_RADIUS \* cos(current\_yaw+ecl::pi\*0.5)) / GRID\_SIZE) + MAP\_ORIGIN;

        y = (int)round((current\_y + ROBOT\_RADIUS \* sin(current\_yaw+ecl::pi\*0.5)) / GRID\_SIZE) + MAP\_ORIGIN;

        if (occupancy\_grid[MAP\_SIZE-y][x] == 1) left\_obstacle = true;

    }

    void testMotion()

    {

        kobuki.setBaseControl(0.0, -0.70);

        return;

    }

    void print(bool map)

    {

        auto now = std::chrono::system\_clock::now();

        auto now\_time = std::chrono::system\_clock::to\_time\_t(now);

        cout << endl << "Pose: [" << current\_x << ", " << current\_y << ", " << current\_yaw\*360.0/ecl::pi << "] " << std::put\_time(std::localtime(&now\_time), "%F %T") << std::endl;

        cout << "Obstacles: " << left\_obstacle << left\_front\_obstacle  << front\_obstacle << right\_front\_obstacle << right\_obstacle<< endl;

        // cout << "m0, target\_y: " << target\_y << " target\_x: " << target\_x << " desired\_yaw: " << desired\_yaw << " ";

        // cout <<  "yaw\_error: " << yaw\_error << " yaw\_precision: " << yaw\_precision << endl;

        if (!map) return;

        for (auto &row : occupancy\_grid)

        {

            for (auto &column : row)

            {

                if (column != -1) cout << column;

                else cout << " "; // print undiscovered cells as empty

            }

            cout << endl;

        }

    }

    double getYawError()

    {

        // Calculate the desired heading based on the current position

        // and the desired position

        double desired\_yaw = atan2(

            target\_y - current\_y,

            target\_x - current\_x);

        // How far off is the current heading in radians?

        double yaw\_error = desired\_yaw - current\_yaw;

        // cout << "m0, target\_y: " << target\_y << " target\_x: " << target\_x << " desired\_yaw: " << desired\_yaw << " ";

        // cout <<  "yaw\_error: " << yaw\_error << " yaw\_precision: " << yaw\_precision << endl;

        return ecl::wrap\_angle(yaw\_error);

    }

    // Generate motion

    void processMotion()

    {

        const double buffer = 0.05;

        double longitudinal\_velocity = 0.0;

        double rotational\_velocity = 0.0;

        // std::cout << "[" << ecl::TimeStamp() << "] Encoders [" << data.left\_encoder << "," << data.right\_encoder << "]" << std::endl;

        // std::cout << "Bumpers: " << (int)data.bumper << std::endl;

        checkMap();

        if (robot\_mode == GO\_TO\_GOAL\_MODE)

        { // "go to target mode"

            if (moving\_state == GO\_STRAIGHT && (data.bumper != 0 || right\_front\_obstacle || front\_obstacle || left\_front\_obstacle))

            {   // HIT

                robot\_mode = WALL\_FOLLOWING\_MODE; // wall following mode

                // save hit point coordinates:

                hit\_x = current\_x;

                hit\_y = current\_y;

                distance\_to\_goal\_from\_hit\_point = sqrt((

                        pow(target\_x - hit\_x, 2)) +

                        (pow(target\_y - hit\_y, 2)));

                cout << "m0, hit\_x: " << hit\_x << " hit\_y: " << hit\_y << " distance\_to\_goal\_from\_hit\_point: " << distance\_to\_goal\_from\_hit\_point << endl;

                std::cout << "robot\_mode: " << robot\_mode << " WALL\_FOLLOWING\_MODE, moving\_mode: " << moving\_state << endl;

                // stop and switch to wall mode

                kobuki.setBaseControl(longitudinal\_velocity, rotational\_velocity);

                return;

            }

            if (moving\_state == ADJUST\_HEADING)

            { // ADJUST HEADING

                double yaw\_error = getYawError();

                // cout << yaw\_error << endl;

                // Adjust heading if heading is not good enough

                if (fabs(yaw\_error) > yaw\_precision)

                {

                    if (yaw\_error > 0.5)

                    {

                        // Turn left (counterclockwise)

                        // cout << "turn L" << endl;

                        rotational\_velocity = FAST\_ROTATION\_SPEED;

                    }

                    else if (yaw\_error > 0)

                    { // very close, rotate slowly

                        // Turn left (counterclockwise)

                        // cout << "turn L" << endl;

                        longitudinal\_velocity = FORWARD\_SPEED \* 0.5;

                        rotational\_velocity = ROTATION\_SPEED;

                    }

                    else if (yaw\_error < -0.5)

                    {

                        // cout << "turn R" << endl;

                        // Turn right (clockwise)

                        rotational\_velocity = -FAST\_ROTATION\_SPEED;

                    }

                    else

                    { // very close, rotate slowly

                        // cout << "turn R" << endl;

                        // Turn right (clockwise)

                        longitudinal\_velocity = FORWARD\_SPEED \* 0.5;

                        rotational\_velocity = -ROTATION\_SPEED;

                    }

                }

                else

                {

                    moving\_state = GO\_STRAIGHT;

                    std::cout << "robot\_mode: " << robot\_mode << ", moving\_mode: GO\_STRAIGHT " << moving\_state << endl;

                }

            }

            else if (moving\_state == GO\_STRAIGHT)

            { // GO STRAIGHT

                double position\_error = sqrt(

                    pow(target\_x - current\_x, 2) + pow(target\_y - current\_y, 2));

                if (position\_error > 0.2)

                {

                    longitudinal\_velocity = FORWARD\_SPEED;

                    // How far off is the current heading in radians?

                    double yaw\_error = getYawError();

                    // Adjust heading if heading is not good enough

                    if (fabs(yaw\_error) > yaw\_precision + 0.2)

                    {

                        moving\_state = ADJUST\_HEADING; // ADJUST HEADING

                        std::cout << "robot\_mode: " << robot\_mode << ", moving\_mode: " << moving\_state << endl;

                    }

                }

                else

                {   // If distance to target is smaller than 20cm

                    moving\_state = GOAL\_ACHIEVED; // finish successfully

                    std::cout << "robot\_mode: " << robot\_mode << ", moving\_mode: " << moving\_state << endl;

                    kobuki.setBaseControl(0.0, 0.0);

                    cout << "DONE!" << endl;

                    // raise( SIGINT);

                }

            }

            else if (moving\_state == GOAL\_ACHIEVED)

            { // GOAL\_ACHIEVED

                if (data.buttons & kobuki::CoreSensors::Flags::Button0)

                {

                    cout << "B0 pressed!!!" << endl;

                    if (fabs(target\_x) > 0.1)

                    {

                        target\_x = 0.0;

                    }

                    else

                    {

                        target\_x = temp\_target\_x;

                    }

                    target\_y = 0.0;

                    moving\_state = ADJUST\_HEADING;

                    std::cout << "robot\_mode: " << robot\_mode << ", moving\_mode: " << moving\_state << endl;

                }

            }

        }

        else if (robot\_mode == WALL\_FOLLOWING\_MODE)

        { // "wall following mode"

            // Distance to the line:

            double a = hit\_y - target\_y;

            double b = target\_x - hit\_x;

            double c = hit\_x \* target\_y - target\_x \* hit\_y;

            double distance\_to\_target\_line = abs(a \* current\_x + b \* current\_y + c) / sqrt(a \* a + b \* b);

            if (distance\_to\_target\_line < 0.10)

            { // If we hit the start-goal line again?

                //cout << "HIT the LINE! ";

                // Is the leave point closer to the goal than the hit point?

                // If yes, go to goal. (Should be at least 20cm closer in order to avoid looping)

                double distance\_to\_goal\_from\_crossing\_point = sqrt(

                    pow(target\_x - current\_x, 2) + pow(target\_y - current\_y, 2));

                if ((distance\_to\_goal\_from\_hit\_point - distance\_to\_goal\_from\_crossing\_point) > 0.2)

                {

                    cout << "HIT the GOAL LINE! ";

                    robot\_mode = GO\_TO\_GOAL\_MODE; // "go to goal mode"

                    moving\_state = ADJUST\_HEADING;

                    std::cout << "robot\_mode: " << robot\_mode << ", moving\_mode: " << moving\_state << endl;

                    kobuki.setBaseControl(longitudinal\_velocity, rotational\_velocity);

                    return;

                }

            }

            // BUMPERS: 0, 1=R, 2=C, 4=L, 3=RC, 5=RL, 6=CL, 7=RCL

            // if hit

            if (data.bumper != 0)

            {

                // turn left

                rotational\_velocity = ROTATION\_SPEED;

                dx = 0.0;

                dth = 0.0;

                corners\_turned = 0;

            }

            else if (front\_obstacle || left\_front\_obstacle)

            {

                // turn left and move forward slowly

                //longitudinal\_velocity = FORWARD\_SPEED \* 0.5;

                rotational\_velocity = FAST\_ROTATION\_SPEED;

                dx = 0.0;

                dth = 0.0;

                corners\_turned = 0;

            }

            else if (right\_front\_obstacle)

            {

                // turn left and move forward slowly

                longitudinal\_velocity = FORWARD\_SPEED \* 0.5;

                rotational\_velocity = FAST\_ROTATION\_SPEED;

                dx = 0.0;

                dth = 0.0;

                corners\_turned = 0;

            }

            else if (right\_obstacle)

            {

                // move straight to follow the wall

                longitudinal\_velocity = FORWARD\_SPEED;

                //rotational\_velocity = FAST\_ROTATION\_SPEED;

                dx = 0.0;

                dth = 0.0;

                corners\_turned = 0;

            } else if (data.bumper == 0)

            {

                // turn right and move forward slowly

                longitudinal\_velocity = FORWARD\_SPEED;

                rotational\_velocity = -FAST\_ROTATION\_SPEED;

            }

            else if (data.bumper == 0)

            { // if no hit or no obstacles on the front and right -> move for <Lenght>cm, then turn 90 degrees and look for the wall

                if (dx >= (length) && dth <= -ecl::pi \* 0.5)

                {

                    // std::cout << "[Z] ";

                    dx = 0.0;

                    dth = 0.0;

                    corners\_turned++;

                }

                else if (dx >= (length + buffer))

                {

                    std::cout << "[R] " << dth << " " << dth\*360.0/ecl::pi;

                    rotational\_velocity = -ROTATION\_SPEED \* 2.0;

                }

                else

                {

                    // std::cout << "[L] ";

                    rotational\_velocity = -ROTATION\_SPEED;

                    longitudinal\_velocity = FORWARD\_SPEED;

                }

                // If completed the whole square but could not find the wall again,

                // recover and switch to go to goal mode. (avoid looping forever)

                if (corners\_turned > 3)

                {

                    corners\_turned = 0;

                    moving\_state = ADJUST\_HEADING;

                    robot\_mode = GO\_TO\_GOAL\_MODE; // "go to goal mode"

                }

            }

            else

            {

                dx = 0.0;

                dth = 0.0;

                corners\_turned = 0;

            }

        }

        kobuki.setBaseControl(longitudinal\_velocity, rotational\_velocity);

    }

    /\* const ecl::linear\_algebra::Vector3d& getPose() {

        return pose;

    } \*/

private:

    double dx, dth;

    int corners\_turned = 0;

    const double length;

    ecl::linear\_algebra::Vector3d pose;      // x, y, heading

    ecl::linear\_algebra::Vector3d test\_pose; // x, y, heading

    kobuki::Kobuki kobuki;

    ecl::Slot<> slot\_stream\_data;

    kobuki::CoreSensors::Data data;

    // Current position and orientation of the robot

    double current\_x;

    double current\_y;

    double current\_yaw;

    // Target coordinates from a file

    double target\_x;

    double target\_y;

    double temp\_target\_x;

    double temp\_target\_y;

    // Coordinates of the first hit point

    double hit\_x;

    double hit\_y;

    double distance\_to\_goal\_from\_hit\_point;

    // +/- 5.0 degrees of precision for the rotation angle

    double yaw\_precision = 5.0 \* (ecl::pi / 180);

    int occupancy\_grid[MAP\_SIZE][MAP\_SIZE];

    // variables for occupancy grid obstacles closer than 10cm

    bool left\_obstacle, left\_front\_obstacle, front\_obstacle, right\_front\_obstacle, right\_obstacle;

    /\*  ############# MAIN ROBOT MODES ###################

        "go to goal mode": Robot will head to an x,y coordinate

        "wall following mode": Robot will follow a wall \*/

    enum robot\_mode\_enum

    {

        GO\_TO\_GOAL\_MODE,

        WALL\_FOLLOWING\_MODE

    } robot\_mode;

    /\*  ############ GO TO TARGET MODES ##################

        adjust heading: Rotate towards the target

        go straight: Go straight towards the target

        goal achieved: Reached the goal x, y coordinates \*/

    enum moving\_state\_enum

    {

        ADJUST\_HEADING,

        GO\_STRAIGHT,

        GOAL\_ACHIEVED

    } moving\_state;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Signal Handler

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool shutdown\_req = false;

void signalHandler(int /\* signum \*/)

{

    shutdown\_req = true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Main

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*\*argv)

{

    fstream fs;

    fs.open("target.txt", ios::in);

    vector<vector<float>> floatVec;

    string strFloat;

    float targetX;

    float targetY;

    int counter = 0;

    getline(fs, strFloat);

    cout << fixed;

    cout.precision(3);

    std::stringstream linestream(strFloat);

    linestream >> targetX;

    linestream >> targetY;

    std::cout << "x: " << targetX << " y: " << targetY << std::endl;

    ecl::CmdLine cmd\_line("Uses a simple control loop to move Kobuki with bug 2 algorithm with sides of length 0.5m", ' ', "0.2");

    ecl::ValueArg<std::string> device\_port(

        "p", "port",

        "Path to device file of serial port to open",

        false,

        "/dev/kobuki",

        "string");

    ecl::ValueArg<double> length(

        "l", "length",

        "traverse square with sides of this size in length (m)",

        false,

        0.15,

        "double");

    ecl::SwitchArg disable\_smoothing(

        "d", "disable\_smoothing",

        "Disable the acceleration limiter (smoothens velocity)",

        false);

    cmd\_line.add(device\_port);

    cmd\_line.add(length);

    cmd\_line.add(disable\_smoothing);

    cmd\_line.parse(argc, argv);

    signal(SIGINT, signalHandler);

    std::cout << "Demo : Example of simple control loop. l: " << length.getValue() << std::endl;

    KobukiManager kobuki\_manager(

        device\_port.getValue(),

        length.getValue(),

        disable\_smoothing.getValue(),

        targetX,

        targetY);

    ecl::MilliSleep sleep(1);

    ecl::linear\_algebra::Vector3d pose; // x, y, heading

    try

    {

        while (!shutdown\_req)

        {

            sleep(200);

            kobuki\_manager.print(false);

            kobuki\_manager.updateMap();

            // pose = kobuki\_manager.getPose();

            // std::cout << "current pose: [" << pose[0] << ", " << pose[1] << ", " << pose[2] << "]" << std::endl;

        }

    }

    catch (ecl::StandardException &e)

    {

        std::cout << e.what();

    }

    sleep(300);

    kobuki\_manager.print(true);

    return 0;

}

References

1. J. U. Duncombe, “Infrared navigation—Part I: An assessment of feasibility,” *IEEE Trans. Electron Devices*, vol. ED-11, no. 1, pp. 34–39, Jan. 1959, doi: 10.1109/TED.2016.2628402.
2. E. P. Wigner, “Theory of traveling-wave optical laser,” *Phys. Rev*., vol. 134, pp. A635–A646, Dec. 1965.
3. P. Kopyt *et al., “*Electric properties of graphene-based conductive layers from DC up to terahertz range,” *IEEE THz Sci. Technol.,* to be published, doi: 10.1109/TTHZ.2016.2544142. *(Note: If a paper is still to be published, but is available in early access, please follow ref [5]).)*
4. R. Fardel, M. Nagel, F. Nuesch, T. Lippert, and A. Wokaun, “Fabrication of organic light emitting diode pixels by laser-assisted forward transfer,” *Appl. Phys. Lett.*, vol. 91, no. 6, Aug. 2007, Art. no. 061103.
5. D. Comite and N. Pierdicca, "Decorrelation of the near-specular land scattering in bistatic radar systems," *IEEE Trans. Geosci. Remote Sens.*, early access, doi: 10.1109/TGRS.2021.3072864. (*Note: This format is used for articles in early access. The doi must be included.)*
6. H. V. Habi and H. Messer, "Recurrent neural network for rain estimation using commercial microwave links," *IEEE Trans. Geosci. Remote Sens.*, vol. 59, no. 5, pp. 3672-3681, May 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9153027

1. [↑](#footnote-ref-1)