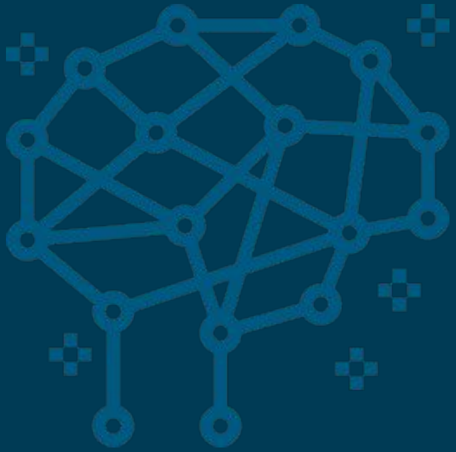
Turtlebot Autonomous Navigation in ROS Gazebo

EE4308 Part 1 Project

Tan Rong De Darius
Franky Laurentis

PRESENTATION OUTLINE

1. Problem Statement
2. Sensor Strategy
3. Navigation Strategy
4. Path Planning Strategy
5. Performance Analysis
6. Conclusion
7. Demonstration



PROBLEM STATEMENT

Achieve autonomous navigation in known & unknown worlds with a Turtlebot

Problem Specifications

Autonomous navigation:

- Path planning
- Obstacle avoidance

Robotics Platform

- Turtlebot
    - RGB-D Kinect sensor
    - IMU
    - Odometry
    - Translational & rotational motion

Simulation Platform

- Ubuntu Xenial 16.04
- ROS Kinetic Kame
- Gazebo
    - 9m x 9m grid world
    - Start: (0, 0)
    - Goal: (4, 4)

Sensor Strategy

robot_pose_ekf
pos_info

robot_pose_ekf

- Combines odometry (/odom) & IMU data (/mobile_base/sensors/imu_data) with Extended Kalman Filter
- Increased accuracy, reliability in pose estimation
- tf tree had to be edited in empty_world.launch

pos_info

- Subscribes to /robot_pose_ekf/odom_combined
- Publishes pose estimate for localisation on grid in /auto_ctrl/pos_info

Sensor Strategy

`depth_info`

Subscribes to raw RGB-D Kinect camera readings

- `/camera/rgb/image_raw`
- `/camera/depth/image_raw`

Publishes depth information directly in front of Turtlebot

RGB-D Kinect sensor (middle point)

- `/auto_ctrl/depth_info`

Acts as basic obstacle detection directly in path of Turtlebot

Sensor Strategy `scan_info`

Subscribes to laserscan_nodelet_manager for ‘fake’ laserscan values from depth image.

- `/scan`

Publishes depth information for left most, right most and middle readings in horizontal axis of sensor

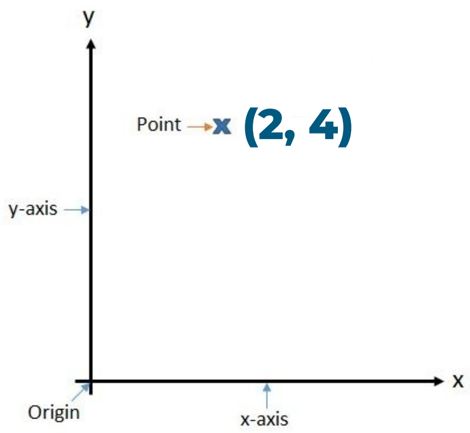
- `/auto_ctrl/scan_info`

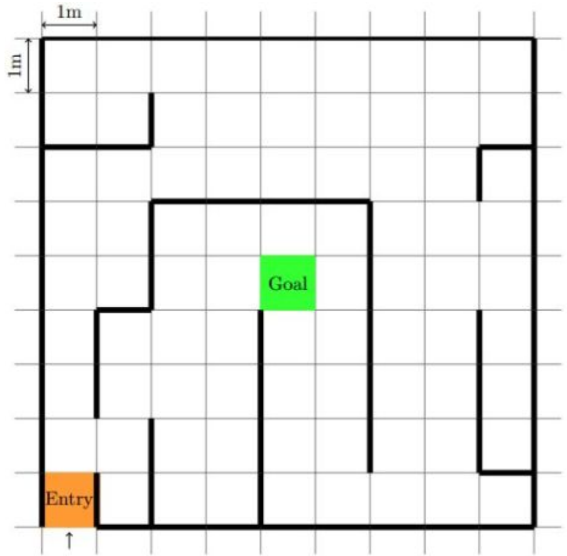
Acts as preemptive obstacle detection along Turtlebot current path

Navigation Strategy `bot_control`

Coordinate System

- 9 x 9 grid representing world frame
- Cartesian coordinates

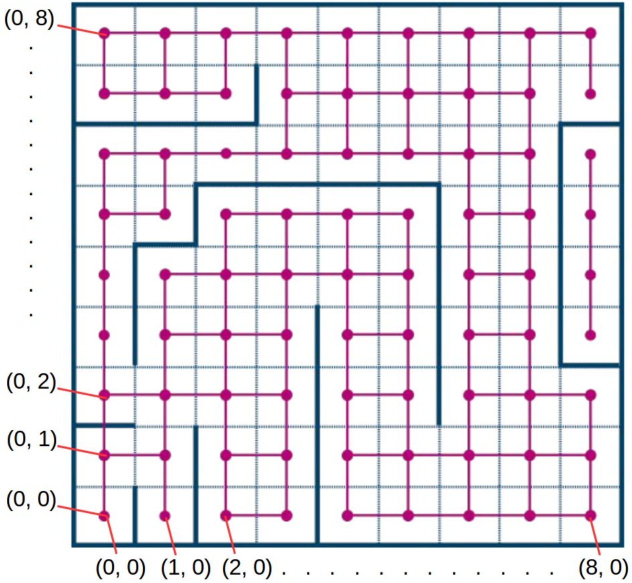




Navigation Strategy `bot_control`

Coarse Motion Control

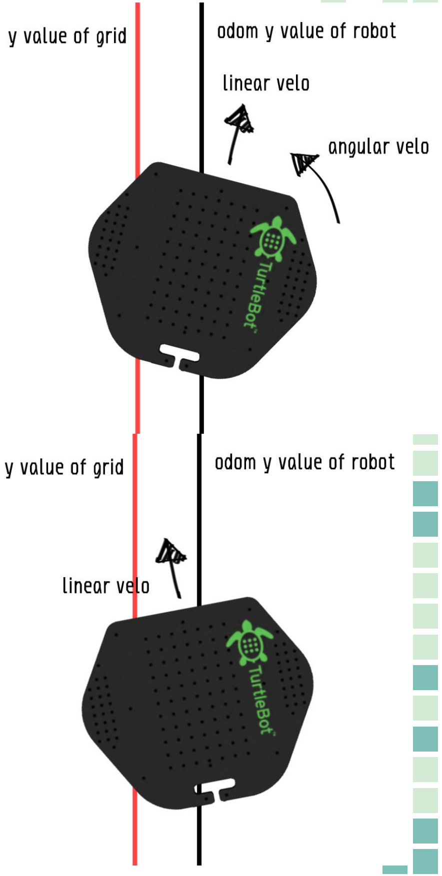
- Navigate between coordinates (middle of grid cells)
- 4 translational movement (up, down, left, right)
- 2 rotational movement (ACW, CW)
- Uses `pos_info` yaw data to check current direction Turtlebot is facing



Navigation Strategy `bot_control`

Fine Motion Control

- Compensates for movement errors and prevents bot from colliding into walls
- Continuous check for the 'middle' of each grid
- Adjusts pose by traversing towards said middle path based on distance / direction offset
- When Turtlebot drifts from middle line, it adds angular velocity to current linear motion to turn inwards
- If Turtlebot is already turned inwards, then no angular velocity is added



Obstacle Avoidance Strategy `bot_control`

Basic Obstacle Avoidance

- Detects obstacles directly in front of Turtlebot using `depth_info`
- Turtlebot has to turn to face the obstacle to detect it

Preemptive Obstacle Avoidance

- Detects obstacles along the path ahead of Turtlebot using `scan_info`
- Allows for continuous motion while scanning for obstacles ahead instead of stopping and turning
- Increases overall speed

Preemptive Obstacle Detection

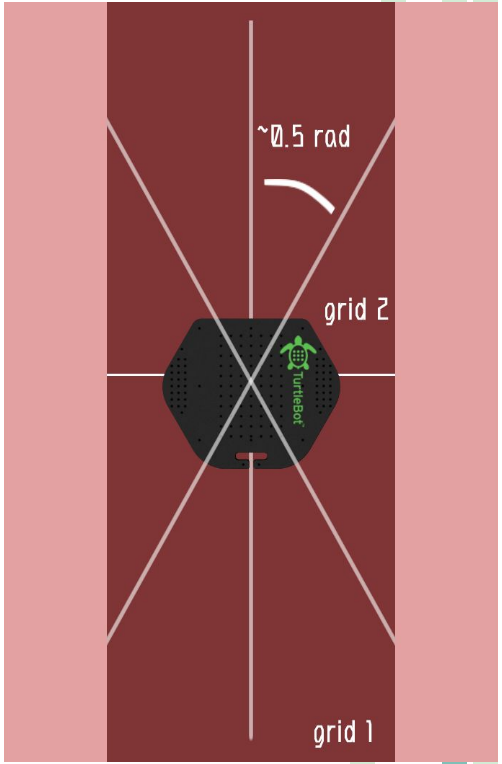
Works based on geometry of map grid and laser FOV

- Laserscan max right and left is ~0.5 rad (30 degrees) from the middle
- By keeping Turtlebot in middle of path with fine motion control, can detect presence of side walls effectively

Check is done when:

- Moving straight between coordinates
- Moved 0.5m from previous grid position

Updates internal map if walls are detected



Path Planning Strategy `algo.h`

Includes functions for creating / maintaining internal map grid as well as search algorithms

Internal map grid

- 9 x 9 2D array of cell structs
- Each cell contains:
  - Own coordinates
  - Neighbours coordinates
  - Previous cell coordinate in path
  - Presence of walls in 4 directions
  - Cost values for search algorithms

Search algorithms

- Flood Fill
- A*

A* Algorithm

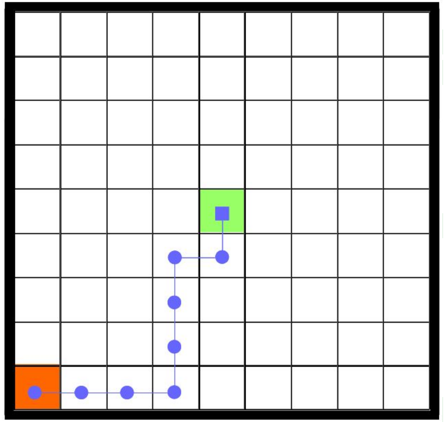
Slightly modified from typical A* implementation

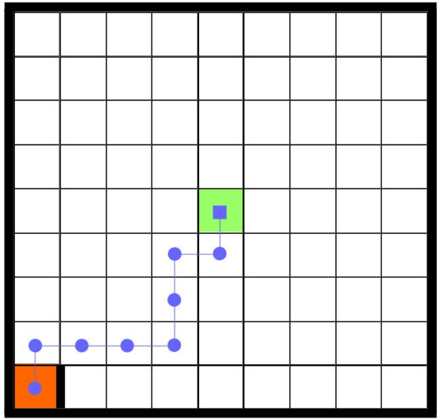
On initialisation:

- Algorithm will compute as if no obstacles in the maze except for maze boundaries

Upon moving:

- Continuous update of internal map grid when a wall is detected (either basic / preemptive)
- Algorithm will re compute to find new path, returning the next coordinate in path





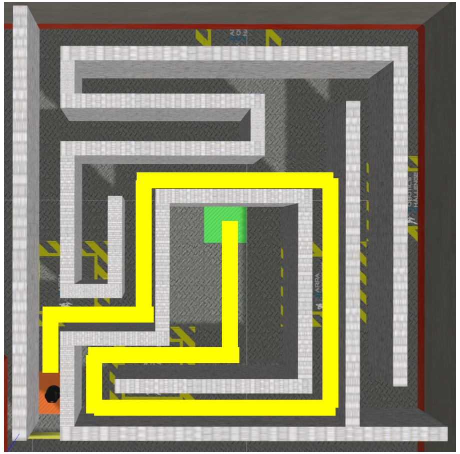
Performance Analysis

World 1

- Cleared with shortest possible route
- Average time to goal: ~50s
- Failure Rate: 0

World 2

- Cleared with shortest possible route
- Average time to goal: ~100s
- Failure Rate: 0



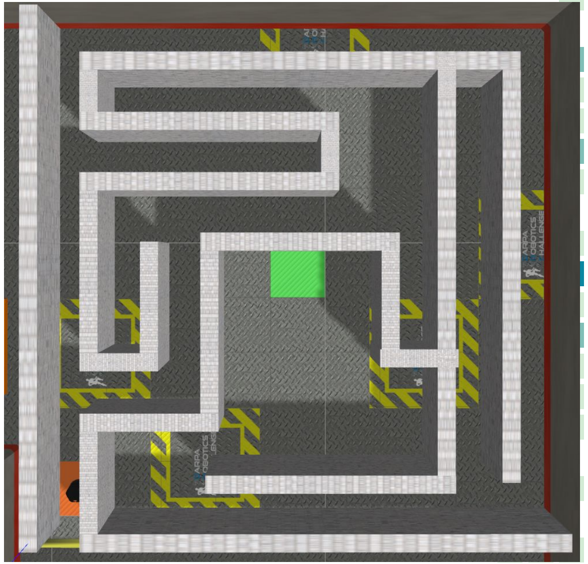
Performance Analysis

Custom World

- Worst case test of multiple dead ends
- Cleared after U-Turning at dead ends
- Average time to goal: ~2200s
- Failure Rate: 0

Most complicated world compared to Worlds 1 and 2

Proves algorithm implemented is able to recover from dead ends and still reach goal



Conclusion

Potential Improvements

- Smoothing turns to allow the robot to continue moving around a corner instead of stopping and rotating
- Allowing for diagonal grid motion

Knowledge Gained

- Understanding how to sync modules
  - Movement controller: Controls robot movement
  - Algorithm: Search for path and provide the next coordinate for Movement Controller
  - Sensors: Provide info for algorithm path planner

DEMONSTRATION

Known Environment (World 2)

DEMONSTRATION

Unknown Environment