Vision and Dialogue for an Autonomous Delivery Robot

Riley Ballachay

August 2024

1 Abstract

Automated robots have been playing an essential role in industrial settings for the last 20 years. As the technology that underpins these devices advances, the environments that researchers design them to navigate have grown increasingly complex. One such environment which holds great promise for automation is hospitals; the adoption of robots in hospitals and medical clinics has exploded in recent years. This growth has necessitated a similar explosion in technology that can respond to the unique physical and low cost constraints. Many turnkey solutions could be applied, however most are overly complicated and expensive to be universally useful. This work responds to that demand, implementing two of the fundamental tools that an autonomous hospital delivery robot requires: language and vision. The language model implemented in this work runs head-less-ly on the robot after startup, and allows the user to feed action commands to the robot in a hands-free manner. These actions are ported to the API developed for moving the Prometheus robot prototype and can successfully execute commands `move_forward` (with varying distance), `move_backward`, `turn_right` (with varying angle), `turn_left` (with varying angle) and `stop`. The model can run with real-time performance on a Raspberry Pi 5 with Jabra Speak 510 speakerphone. The vision model has two components: local path planning and ground segmentation; global localization and mapping. The first component has two proposed model: ray-marching and artificial potential fields. The first checks the distance in front of robot that may be traversed before running into an object. The

second suggests a heading vector based on objects surrounding the robot. These two values are accessible via a HTTP API, and can run real-time on a Raspberry Pi 5 with an Intel RealSense D415 camera. The global localization is run using ORB_SLAM3 visual SLAM method, and allows for map-free visual odometry and SLAM-based localization. Localization and mapping can be run in real-time on a Raspberry Pi 5 with an Intel RealSense D415 camera. The merits of the techniques introduced in this work are discussed and recommendations for future work are offered.

2 Introduction

Autonomous robots have been an ubiquitous part of industrial manufacturing since the turn of the 21st century [25]. As industrial robots have continued to improve in both mobility and decision-making capacity, the range of environments in which these robots may be deployed has expanded. The fields now extend to less structured, more dynamic environments including hospitals, public sidewalks/roads and retail stores. Robots in such environments must be able to safely adapt to highly dynamic obstacles and adapt to novel scenes. The medical industry is poised to take full advantage of the automation revolution; researchers have explored the possibilities as nurses assistants, porters and pharmacy assistants [13]. Autonomous navigation, particularly in dynamic environments like hospitals, is a challenging research question still being addressed in the literature [17]. Researchers are still exploring a variety of sensors (including LIDAR, RADAR and ultrasonic sensors) and local and global path planning algorithms (dynamic movement, reinforcement learning) to address the question.

A variety of turnkey robotic platforms exist, which come equipped with a sensor suite, powerful microprocessor, control board, motors, batteries and wheels. A variety of turnkey robot platforms, with TurtleBot being the most popular, having been used heavily in industry and research [3]. The primary challenge with autonomous navigation today is bridging the gap between environment perception and action, as environments can be complex and unknown, and often requires knowledge of pedestrian movement and space layout. Implementing a navigation algorithm from scratch in a

physical robot is a complicated task, meaning researchers often build upon ready-made solutions like TurtleBot. Platforms such as Turtlebot, however, rely upon higher-end graphical processing units (GPU's) and Light Detection and Ranging (LIDAR) for sensing, resulting in a product which is prohibitively expensive for many applications [2].

A handful of researchers have attempted to create a lower-cost, lower-tech alternative to TurtleBot. Kim et al. (2022) created their own mobile platform DPoom, equipped with a RGB-D camera, however the researchers used a Lattepanda Alpha 864 processor embedded on a TurtleBot chassis, which is not significantly lower-cost than the original [14]. The paper, however, demonstrates that an RGB-D camera can successfully be used to navigate complex indoor scenes. The authors use a plane-segmentation algorithm to identify traversable terrain in the local reference frame, and use this information to perform global navigation, with the aid of visual SLAM. The authors then use the combination of this information with a reinforcement learning paradigm to finalize the autonomous driving algorithm. Jo et al. (2022) created a much lower-cost SMARTmBOT, which uses a Raspberry Pi 4 processor and time-of-flight sensors for navigation [12]. While extremely low-cost, this robot lacks a voice module and capacity for global localization, minimizing its applicability in a healthcare setting.

This project aims build upon these works, moving towards a lower-cost autonomous vehicle that can assist in a hospital environment. To this end, it will have a dialogue system so that healthcare professionals and patients may make requests, local and global navigation systems based on a low-cost vision system, a low-cost processor and controller and a mobile chassis. The name of the complete platform is the Prometheus autonomous robot. The goal of the Prometheus platform is to create a complete product ready-to-deploy in hospital scenarios, however this project will focus on two of these aspects: the dialogue and vision systems. A chassis equipped with motors, controllers, batteries and a processor for receiving a limited number of commands including move_forward, move_backward, turn_left, turn_right and stop developed by a prior team. The dialogue system will need to augment the existing system by achieving the following objectives 1) Run headlessly on-board, activated by the keyphrase 'Hey Robot', 2) Dialogue naturally with the user, 3) Convert

commands uttered by the user into actions that can be run on the controller, 4) Call the existing API for robot actions. The vision system will be based on an Intel RealSense D415 RGB-D camera. The vision system must be capable of recognizing the local and global position of the robot, and identify traversable ground in a closed environment. This vision system will build the foundation for a complete autonomous navigation algorithm in the future.

3 Dialogue Model

3.1 Introduction

Hands-free audio systems are becoming increasingly common in autonomous robots, owing to their ease of use and intuitiveness. This ascent accelerated in the wake of the COVID-19 pandemic, when researchers scrambled to create sterile robots to assist and replace front-line workers. Grasse et al. (2021) outfitted a TurtleBot 2 with a dialogue system capable of taking specific delivery instructions [**?**]. Liu et al. (2024) developed an autonomous platform for assistive navigation capable of answering questions about the environment, making changes to ROS configuration and executing commands [16]. Zand et al. (2023) developed a more complicated graph-based voice system for a robot in assistive cooking applications [29]. These three works demonstrate a generalized architecture for a dialogue-based system for a robot, shown in Figure 1. The user speaks to the system until the end of speech is detected, at which point the audio is passed to a speech recognition module. This converts the speech to text, which is passed to a natural language understanding (NLU) model which parses the intent of the user from a limited set of intents that the model was trained to accept (for example 'go forward', 'go backward', 'stop') with slots ('5 metres', 'a couple yards'). These commands, or lack thereof, are passed to a dialogue policy, which communicates with the user if the command doesn't exist, or confirms the action. The dialogue policy then passes the intent and slots to the command type interpreter, which executes on the robot operating system (ROS). The response generation converts the text from the dialogue policy into audio that is spoken back to the user.

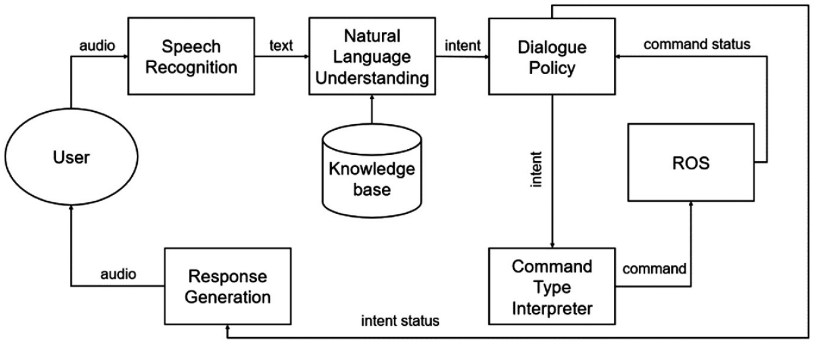


Figure 1: Generalized format for robot dialogue system. Adapted from [6]

Each of the three aforementioned projects utilize a variation of the architecture shown in 1. Grasse et al. (2021) have the simplest system for dialogue parsing, which requires that users speak certain phrases with a set structure and the commands are parsed directly using regex and other string operations - an example being "Robot, go forward". This is limiting in that all users must know the format of accepted commands and use the exact language. Liu et al. (2024) implemented a more advanced dialogue system, which uses Rasa, an open-source package for NLU and dialogue management. The authors leverage Rasa for intent parsing and slot filling, however the implement their own dialogue manager from scratch in python, instead of using the built-in dialogue manager from Rasa [4]. The authors use the Google TTS API for text-to-speech [1]. Zand et al. (2023) provide the most advanced dialogue system, which uses BERT for intention parsing and DistilGPT-2 for dialogue management. The authors do not include a dialogue response system however. This work combines multiple aspects from each of these papers into a faster, lighter-weight and more configurable dialogue system. FasterWhisper, a re-implementation of OpenAI's Whisper model using CTranslate2 is used for speech-to-text. Rasa NLU is used for intent parsing and slot filling, and Rasa dialogue engine is used for conversation management [4]. The lightweight NixTTS model is used for text-to-speech. The dialogue system from this work improves significantly upon multiple aspects introduced by Liu et al. (2024), providing an easily extendible model which can be quickly modified for new commands/intents and utilizes fast and small enough models to work in real-time on a Raspberry Pi 5. A voice activity detection module is implemented using the WebRTC VAD

engine, to determine when a user is done speaking [19].

Paper	STT	NLU	Dialogue	TTS
Grasse et al. (2021)	DeepSpeech	String Parsing	Code	GNUstep
Liu et al. (2024)	Speech Recognition	Rasa	Code	Google TTS
Zand et al. (2023)	Vosk	BERT	DistilGPT-2	N/A
This Work (2024)	FasterWhisper	Rasa	Rasa	NixTTS

Table 1: Comparison of components of complete robot dialogue systems for a variety of autonomous service robots. STT (speech to text) refers to speech recognition, NLU is natural language understanding, dialogue refers to the automated dialogue system and TTS (text to speech) is the system for converting text to audio.

3.2 Implementation

The majority of dialogue system, except for audio recording, is implemented in Python. The entry-point for the dialogue system is language/daemon.py. After installing requirements, this can be run in headless mode with: `python3 daemon.py > /dev/null 2&1 &`. This will create a background process that will continuously run the voice model. See the complete schematic for the dialogue system in Figure 2

3.2.1 Porcupine Wake Detection

The first component of the dialogue system is the 'wake engine' which is intended to continuously listen on the robot and awake to a specific key-phrase, in this case, 'Hey Robot'. The lightweight framework Porcupine from Picovoice was chosen for the cross-platform ability and low memory consumption [18]. The models (for Mac and Raspberry Pi) for the Phrase 'Hey Robot' were created and downloaded from the Picovoice website to the folder language/models/Porcupine.

3.2.2 WebRTC Voice Activity Detection (VAD)

Once activated, the dialogue system launches the audio recording module language/src/stt/record_audio.c. The module starts recording when voice activity detection starts and stops after 1 complete second of silence has elapsed. The WebRTC VAD engine, isolated from the other components, is interfaced

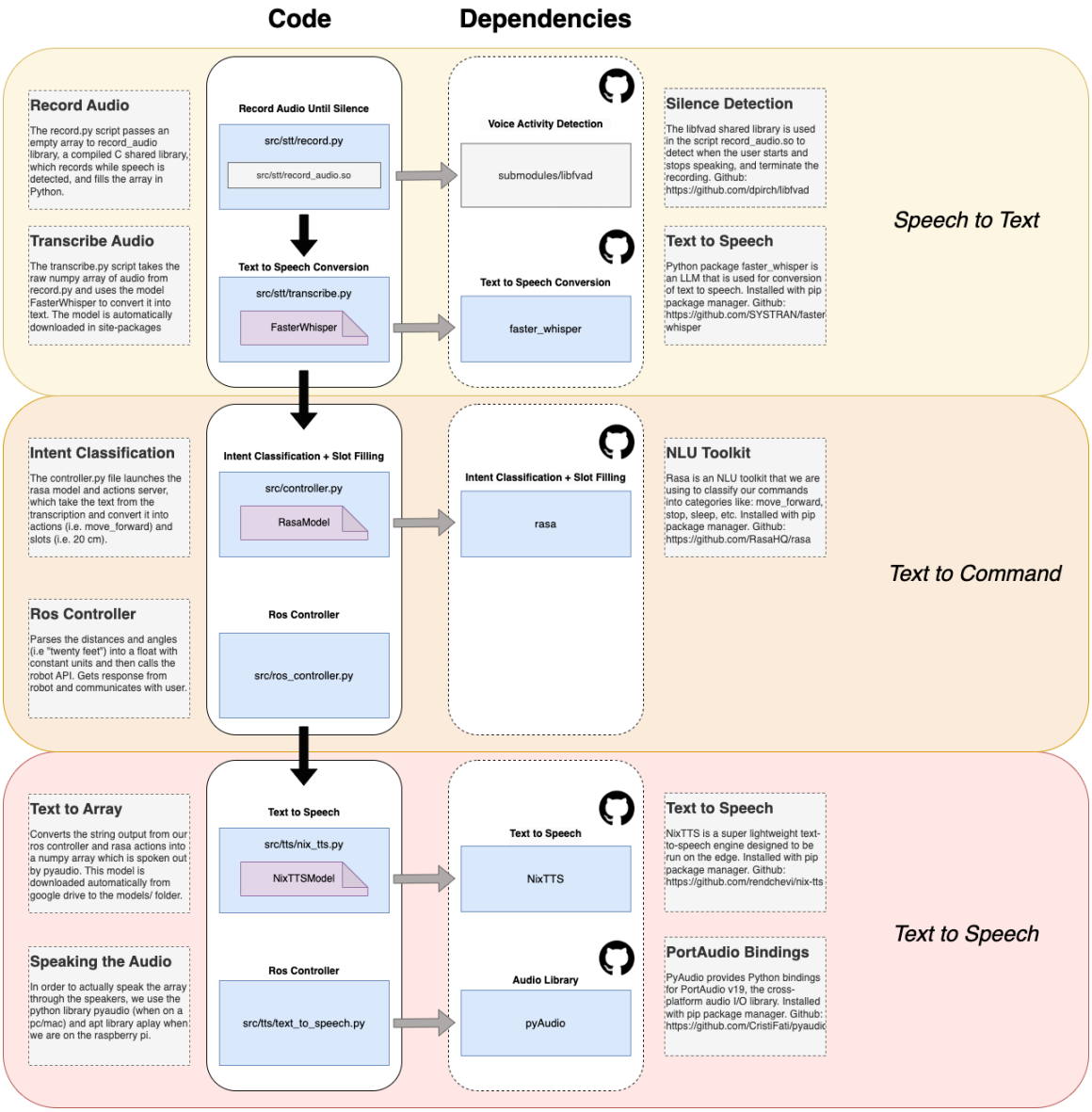


Figure 2: Complete schematic of dialogue system. Blue indicates Python, purple indicates ML models and grey indicates C dependencies. Available at language/README.md

using the libfvad library from GitHub [19].

3.2.3 FasterWhisper STT

The recorded audio is passed as an array to the next stage, the speech to text model, where it is converted into text. Whisper is an extremely popular open-source model for speech-to-text from OpenAI [20]. While robust, this model was not nearly fast enough to run on a CPU in real-time. Whisper-cpp, a higher-performance implementation of Whisper, was also investigated, but took several seconds to resolve responses [9]. FasterWhisper, an implementation of Whisper with CTranslate2 engine proved to be the fastest while retaining high enough accuracy [26]. The selected model, `tiny.en` is automatically loaded by the library and the inference is performed in language/src/stt/transcribe.py.

3.2.4 Rasa NLU and Dialogue Manager

Once the recorded audio has been transcribed to text, a machine learning model is used to parse the text and convert it into an actionable command. The Rasa Dual Intent and Entity Transformer (DIET) model is used to parse the intents (move_forward, move_backward, turn_left, turn_right and stop) and associated entities/slots (distance, angle) [21]. The model can also accept combinations of intents, like move_forward and turn_angle, and performs the two in series. The full list of intents, actions and entities is found in the file language/data/rasa/v2_motor/domain.yml. To train the model, hundreds of samples of each of these actions, with the intent and entities indicated, are written to the file language/data/rasa/v2_motor/data/nlu.yml. The Rasa model has an additional dialogue manager, known as Rule and Transformer Embedding Dialogue (TED) policies, which manages conversation state. In this model, the behavior is configured using the file language/data/rasa/v2_motor/data/rules.yml, which dictates that certain actions are run after each intents are recognized. More complicated conversation paths can be written to the `stories.yml` in this folder, but this was not necessary for this case. Each of the actions in the `domain.yml` and `rules.yml` files correspond to a response entry, or an action. The actions are launched as a separate python process, and contain the logic for parsing the entities and returning the necessary information back to the dialogue controller. These actions are found in the file lan-

guage/data/rasa/v2_motor/actions/actions.py.

3.2.5 Command Interpreter

Once the actions generated inside of `actions.py` are sent back to the dialogue controller, they can be parsed and sent to the ROS. Each of the intents passed back to the dialogue controller correspond to a method of the RosControllerV2 class in language/src/ros_controller.py. For the move and turn actions, a distance or angle can be passed, and this is converted to a float prior to calling the TCP server.

3.2.6 Nix Text-to-Speech

The final stage in the dialogue pipeline is the speech model. This converts the text from the dialogue system into audio that is returned to the user. A variety of models were investigated, including FastSpeech 2 [23], StyleTTS 2 [15] and Espeak synthesizer [8]. NixTTS was the fastest option by a significant margin, with low memory consumption. The model is automatically downloaded inside the code from Google Drive, to the local folder `language/models/nix-ljspeech-deterministic-v0.1`.

4 Vision and Navigation

4.1 Introduction

A hospital is an extremely complex and unstructured environment. For a robot to safely navigate in a hospital, it must negotiate a variety of medical equipment, patients, medical professionals and visitors. Thus, in addition to a global path planning algorithm, an autonomous hospital-navigating robot must have a robust local path planning to avoid highly dynamic obstacles in a tight, cluttered environment [24]. Obstacles in a hospital environment can range from large dynamic obstacles (i.e. beds, tables) to smaller, dynamic objects with complex motion patterns (i.e. children). A fully autonomous robot would need to implement trajectory estimation planning for dynamic objects like pedestrians or other moving obstacles. This problem will need to be investigated in future versions

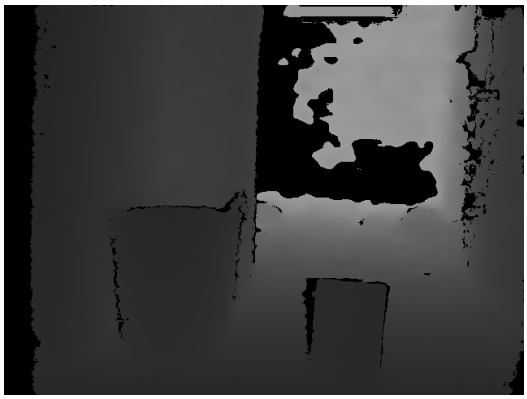
of the hospital robot, but has been omitted from the current work.

Autonomous navigation can be broadly divided into two components: global and local path planning. Global path planning consists of finding an optimal path between two points in a virtual/computational representation of the environment. Local path planning avoids dynamic and static obstacles (as well as determining if the ground plane is traversable) while remaining close to the globally optimal path. This work will work on developing a robust local path planning algorithm, capable of detecting local dynamic and static obstacles, while simultaneously identifying the regions of traversable ground.

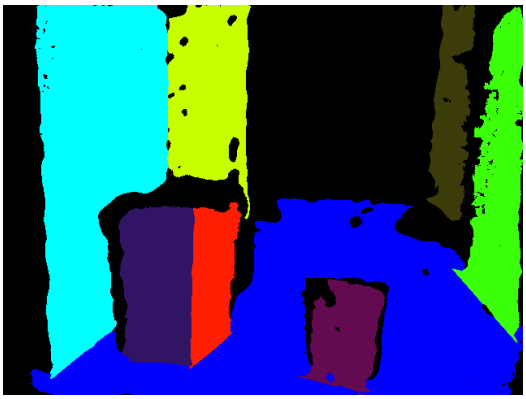
4.2 Ground Plane Segmentation

Identifying the 3D structure of the surrounding environment is an important pre-requisite for navigation in a dynamic environment. A robot must identify what parts of the surrounding consist of traversable surfaces (ground, stairs, ramp), environment (walls, ceiling, doors) and obstacles (chairs, pedestrians). The most important of these environmental structures is navigable ground surface. This is the area through which the robot may navigate unencumbered. A variety of methods have been proposed in the literature including CNN-based ground segmentation [28], path-marching algorithms [14] and point cloud-fitting plane segmentation [10]. The last of these, ground segmentation by plane fitting, is the most promising as it segments the ground and separate obstacle planes, allowing them to be identified and categorized. In this technique, a point cloud is generated from the RGB-D image and points are clustered to generate candidate planes and surface normals. The RGBDPlaneDetection Github repository is used for plane fitting [27], which is integrated directly into the morbius repository vision/src/rgbdSeg/include/plane. The segmented are then passed into a class that checks the normal of each plane, and picks the plane with the y normal closest to -1, which would be pointing directly upwards. This subroutine can be found in the class Surfaces in vision/src/rgbdSeg/artificialFields.hpp. See the results from the plane segmentation algorithm below in Figure 3.

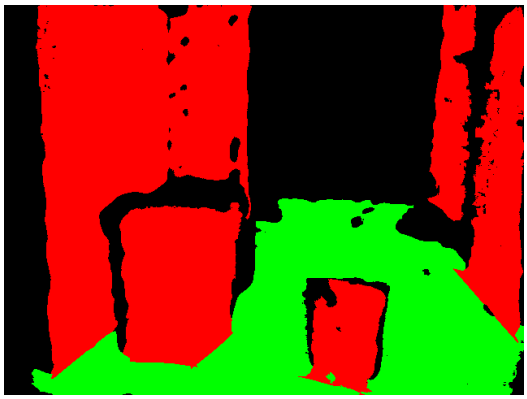
(a) RGB Image



(b) Depth Image



(c) Segmented planes



(d) Ground plane indicated in green

Figure 3: Pipeline for ground plane segmentation and identification.

4.3 Path Tracing Algorithms

The segmented ground plane indicates which parts of the environment belong to the ground surface, but it is not known whether these surfaces are traversable or not to the agent. Two separate methods were investigated in this work, and an API was provided that allows the developer to access them both, in addition to SLAM-based localization. This final API is called vision/visionServer.cpp. The API is launched after being built on `http://localhost:9080` and can be called using HTTP requests. These methods are 'ray-marching', based on the method developed by Kim et al (2022) and 'artificial potential fields' based on the method from Pugliese et al (2019).

4.3.1 Ray-Marching Method

The ray marching method helps to determine how far the robot can travel in a straight line on the ground surface before running into an obstacle plane. The algorithm, in Structure GroundVectors in vision/src/rgbdSeg/groundVectors.hpp, traces from the bottom of the segmented image upwards (the red and green image, starting on the green surface), and stops when there is less than AGENT_WIDTH (defaults to 25 cm) clearance to each side, meaning that the robot would not fit through the space. This assumes that the camera is placed on the front of the robot, and the robot is placed on the ground plane, so that the ground is the first surface detected in the bottom of the image. After launching the visionServer, the max distance traversable forward in mm can be accessed using a GET request, under the header max_distance. This is called in the language model when a move_forward request is made, see the handler in language/src/vision_handler.py. A visualization of the 'rays', or distance vectors, is generated when running the planeSegment script in vision/src/rgbdSeg. See the results below in Figure 4.

(a) Segmented ground and obstacle planes

(b) Traversable path for robot width 25cm

Figure 4: Example of ray-marching that calculates max traversable distance in front of the robot.

4.3.2 Artificial Potential Fields Method

The artificial potential fields method combines the solution from global path planning and combines it with knowledge of local obstacles to provide the 'best' local path. The algorithm starts with the

results from the ground-plane segmentation and follows the following steps:

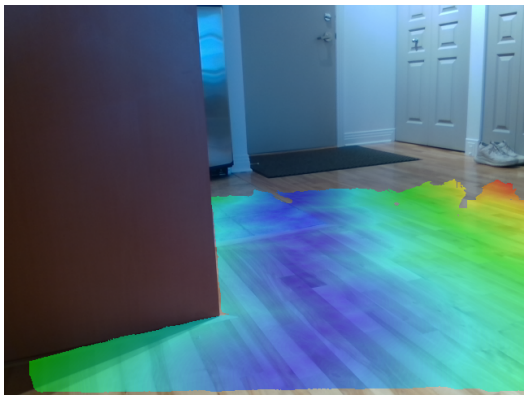
1. Calculate equation of ground plane using normal and center of plane
2. Convert the segmented RGB-D image to a point cloud with plane membership
3. Project 'obstacles' from plane detection onto ground plane
4. Refine points on ground to have one voxel in each 10 cubic mm
5. Calculate attractive force of destination and repulsive force of each obstacle point
6. Sum forces and calculate heading vector corresponding to 'best' path

The above algorithm provides a vector of length STEP_SIZE (defaults to 20mm) in the 'lowest energy' direction of the robot. The resultant vector is calculated in function resultantForces inside vision/src/rgbdSeg/artificialFields.hpp The visionServer uses the location calculated from SLAM, as well as this step, to calculate the best next position for the robot, which can be retrieved using a GET request to the endpoint `http://localhost:9080` under the label `local_destination`. A visualization of the best path to a given destination from the current position may be produced using the planeSegment script in vision/src/rgbdSeg. The path is generated by providing a destination, in this case 5 metres in front of the robot, and generating a path by iteratively moving the robot STEP_SIZE in the direction of the resultant vector, then virtually 'moving' the robot in the point cloud, calculating the resultant forces at the new position. See the results below in Figure 5.

4.4 Robot Localization and SLAM

Simultaneous localization and mapping (SLAM) is an integral part of modern autonomous robots. Through camera and sensor data, a SLAM system may be placed in any unknown environment and incrementally build a map of the surroundings while localizing itself within the map [7]. Visual odometry is an essential sub-component of a SLAM system, as it uses a series of frames to determine the position and orientation of the camera in the environment, and estimates how this is updating over time. One of the most popular packages for SLAM is ORB-SLAM3, which is capable of

(a) Segmented ground and obstacle planes



(b) Traversable path for robot width 25cm

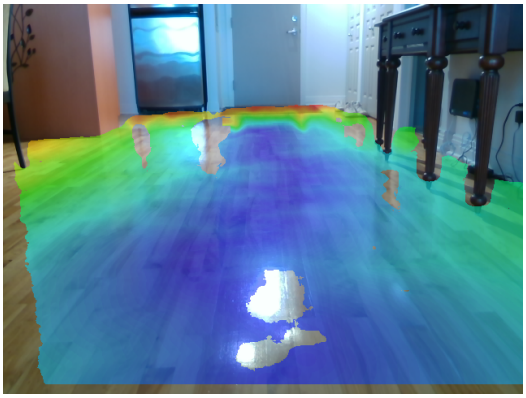
Figure 5: Example of artificial potential fields method for destination 5 metres in front of robot. Blue to red color scale indicates distance from best path, where blue is the best path.

performing real-time slam using RGB-D cameras like the Intel RealSense D415 [5]. The package was integrated into this project using the ORB_SLAM3 repository from the authors on Github. The SLAM system is automatically started when running the visionServer script, and a pre-made map can be provided as the fourth command line argument. The estimated location of the robot is updated inside of the API endpoint at `http://localhost:9080` under the title: `location`.

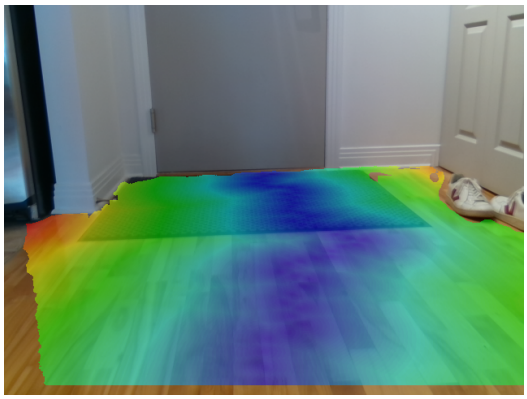
4.4.1 Running a Simulated Robot Session

The script vision/ros_sim.py was developed to test the localization capacity of the robot. The camera was placed upon a rolling chair, the visionServer service was started and the `ros_sim.py` script is used to track the position of the robot over two movement trajectories. The first is to move the camera forward a pre-determined distance, and the second is to move the camera forward a certain distance, turn to the left and move the camera another distance. See in Figure 6 two images from the first test, moving the camera forward 6 metres.

The robot location is recorded during the simulation, and the camera is moved very slowly to allow for the SLAM system to properly update key-frames and visual features. Figure 7 shows the real paths taken by the camera and the location estimated by the SLAM visual odometry system. A variety of tests were performed to assess the localization capacity of the ORB_SLAM3 system,



(a) First step, moving forward 6 metres. Destination is set to 1 metre in front of camera.



(b) Step at about 5 metres, moving forward 6 metres. Destination is set to 1 metre in front of camera.

Figure 6: Example of artificial potential field images taken when running a camera localization test.

and the system was shown to be robust in moving forward and directly to the side, but performance significantly declined once the camera was rotated. Fast rotation is a known problem in visual odometry systems, and rotation with this algorithm seems to cause significant problems [30].

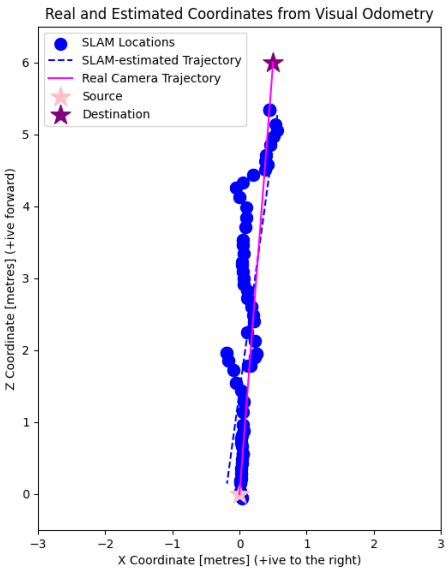
5 Materials

5.1 Camera

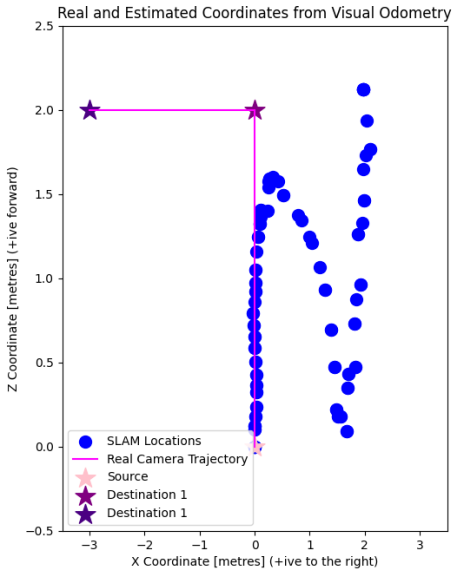
An Intel RealSense D415 stereo depth camera is to be used on-board the robot for localization, motion detection and path planning [11]. The camera uses two grey-scale images in stereo to estimate depth, and a single RGB camera to capture color. The camera is able to capture depth in a range from 30 cm up to 4 metres.

5.2 Speakerphone

A Jabra Speak 510 speakerphone was used for audio input and output. The speaker may be attached to the Raspberry Pi via USB and is easy to access via Python and C.



(a) Moving forward 6 metres + 0.5 metres to the right.



(b) Moving 2 metres forward + 3 metres to the left, with rotation in the middle.

Figure 7: Visual odometry tests using ORB_SLAM3 localization under two conditions.

5.3 Processor

A Raspberry Pi 5, Soc BCM2712 with 8 GB of RAM was selected as the best processor capable of running the SLAM, visual and language models consecutively [22]. The system is connected directly to an outlet through the USB-C, and to the speaker and camera via USB-3 ports. The Pi is accessed and code is run via SSH. It is important that the Pi is plugged into a direct power source instead of a laptop, as it will not provide sufficient voltage.

5.3.1 Language Model

The full language model is described in the Dialogue Model section, but the model consists of a 'daemon' that should be constantly running on the Raspberry Pi, awaiting the keyphrase "Hey Robot". Once that is uttered, the full language model is run in a loop until the user asks it to sleep. The full pipeline is profiled on the Raspberry Pi and the results are shown below in Table 2. This

includes the average response time for each step, memory consumption and CPU usage shown using the `top` command. Generally, the language model showed to be very responsive while maintaining low memory consumption on the Pi.

Function	Time (s)	Memory (MB)	CPU (%)
"Hey Robot" Daemon	N/A	256	2
Wake-up	5	512	5
Full Language Model	5	650	105
Shut-Down	5	512	105

Table 2: Performance metrics for components of the language model. Time refers to the average max time for a response from the system, which ranged from 1-5 seconds for each stage.

5.3.2 Vision Model

The vision model is run in two different scripts: `visionServer` and `planeSegment`. The server creates a http server and uses ORB_SLAM3 to estimate the location position relative to the start position, while also providing the best heading vector from the artificial potential fields and the max forward distance traversable from the ray marching method. The planeSegment script produces the images that visualize these two methods, and displays them to the user. Both of these scripts run with a good speed and relatively low memory usage, see the results in Figure 3

Script	Time (s)	Memory (MB)	CPU (%)
visionServer	2	640	150
planeSegment	1.5	240	110

Table 3: Performance metrics for vision scripts. Note that time refers to a single update, which corresponds to a total of 10 frames captured from the camera.

6 Conclusion and Discussion

Two essential components of an automated delivery robot - language and vision - were conceptualized and implemented in Python, C and C++. The entirety of the code, including the setup scripts for the Raspberry Pi, and certain setup scripts for Mac, were provided in the Morbius Repository.

6.1 Language

The language model was developed almost entirely in Python, and was demonstrated to work seamlessly on a Raspberry Pi 5. This portion of the work is the least error prone. It has been tested in a variety of environments and is robust and low-latency. If future students want to expand the model, or change the intents, one must simply change the .yml files in language/data/rasa and update the corresponding actions in the ros_controller.py. In addition, students may want to increase the ability to dialogue with the robot, which can be modified by expanding the file language/data/rasa/v2_motor/data/stories.yml. For more information about stories and dialogue with rasa models, see this resource.

6.2 Vision

The vision model was developed entirely in C++, and certain parts were demonstrated to work with varying efficacy on a Raspberry Pi 5. The two local positioning algorithms worked fairly well, however are very sensitive to noise in the point cloud received from the camera. The first, the ray-marching method, works okay, however is limited by the fact that the camera can only detect the ground 1-2 metres ahead of its location. This is largely due to reflections on the ground, or other lighting artifacts. It is unlikely that the robot would need to move more than this distance at a time, however. The artificial potential fields method works a little less well, as the point cloud is extremely noisy, and the heading vector can sometimes point in the wrong direction. With that said, the method used to segment the ground plane is extremely effective, so the objects detected using this method can be used directly in another path planning algorithm. The global positioning algorithm, ORB_SLAM3, created a SLAM map successfully with accurate dimensions, however this takes a considerable amount of time. When using without a pre-generated map, the localization is accurate until the robot is turned. An important test that was not completed in this work is to create a SLAM map and then try to localize the robot in a second session; this will likely be a lot more accurate. Visual odometry using SLAM seems to be a very promising method for localization.

References

[1] Google cloud text-to-speech. `https://cloud.google.com/text-to-speech?hl=en`. Accessed: 2024-07-30.

[2] Ehab I. Al Khatib, Mohammad Abdel Kareem Jaradat, and Mamoun F. Abdel-Hafez. Low-cost reduced navigation system for mobile robot in indoor/outdoor environments. *IEEE Access*, 8:25014–25026, 2020.

[3] Robin Amsters and Peter Slaets. *Turtlebot 3 as a Robotics Education Platform*, pages 170–181. 01 2020.

[4] Tom Bocklisch, Joey Faulkner, Nick Pawlowski, and Alan Nichol. Rasa: Open source language understanding and dialogue management, 2017.

[5] Carlos Campos, Richard Elvira, Juan J. Gomez Rodriguez, Jose M. M. Montiel, and Juan D. Tardos. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, December 2021.

[6] Evgenii Chepin, Alexander Gridnev, and Margarita Erlou. Developing a voice control system for a wheeled robot. In *Biologically Inspired Cognitive Architectures Meeting*, pages 208–215. Springer, 2023.

[7] Hugh Durrant-whyte and Tim Bailey. Simultaneous localization and mapping: Part i. *Robotics Automation Magazine, IEEE*, 13:99 – 110, 07 2006.

[8] eSpeak NG contributors. espeak ng: Text-to-speech software. `https://github.com/espeak-ng/espeak-ng`, 2024. Accessed: 2024-08-05.

[9] Georgi Gerganov. whisper.cpp, 2023. Accessed: 2024-07-30.

[10] Dirk Holz, Stefan Holzer, Radu Rusu, and Sven Behnke. Real-time plane segmentation using rgb-d cameras. volume 7416, 01 2012.

[11] Intel. Intel realsense depth camera d415, 2024. Accessed: 2024-08-13.

[12] Wonse Jo, Jaeeun Kim, Ruiqi Wang, Jeremy Pan, Revanth Krishna Senthilkumaran, and Byung-Cheol Min. Smartmbot: A ros2-based low-cost and open-source mobile robot platform, 2022.

[13] Mari Kangasniemi, Suyen Karki, Noriyo Colley, and Ari Voutilainen. The use of robots and other automated devices in nurses' work: An integrative review. *International Journal of Nursing Practice*, 25, 05 2019.

[14] Taekyung Kim, Seunghyun Lim, Gwanjun Shin, Geonhee Sim, and Dongwon Yun. An open-source low-cost mobile robot system with an rgb-d camera and efficient real-time navigation algorithm. *IEEE Access*, 10:127871–127881, 2022.

[15] Yinghao Aaron Li, Cong Han, Vinay S. Raghavan, Gavin Mischler, and Nima Mesgarani. Styletts 2: Towards human-level text-to-speech through style diffusion and adversarial training with large speech language models, 2023.

[16] Shuijing Liu, Aamir Hasan, Kaiwen Hong, Ruxuan Wang, Peixin Chang, Zachary Mizarchi, Justin Lin, D. Livingston McPherson, Wendy A. Rogers, and Katherine Driggs-Campbell. DRAGON: A dialogue-based robot for assistive navigation with visual language grounding. *IEEE Robotics and Automation Letters*, 9(4):3712–3719, 2024.

[17] Saeid Nahavandi, Roohallah Alizadehsani, Darius Nahavandi, Shady Mohamed, Navid Mohajer, Mohammad Rokonuzzaman, and Ibrahim Hossain. A comprehensive review on autonomous navigation, 2022.

[18] Picovoice. Porcupine wake word engine, 2024. Accessed: 2024-07-30.

[19] David Pircher. libfvad, 2023. Accessed: 2024-07-30.

[20] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022.

[21] Rasa. Introducing dual intent and entity transformer (diet): State-of-the-art performance on a lightweight architecture, 2020. Accessed: 2024-07-30.

[22] Raspberry Pi Foundation. Raspberry pi 5 documentation, 2024. Accessed: 2024-08-15.

[23] Yi Ren, Chenxu Hu, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao, and Tie-Yan Liu. Fastspeech 2: Fast and high-quality end-to-end text to speech, 2022.

[24] C. Rondoni, F. Scotto di Luzio, C. Tamantini, et al. Navigation benchmarking for autonomous mobile robots in hospital environment. *Scientific Reports*, 14:18334, 2024.

[25] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, Cambridge, MA, 2nd edition, 2011.

[26] SYSTRAN. faster-whisper, 2023. Accessed: 2024-07-30.

[27] Chao Wang. Rgbdplanedetection, 2024. GitHub repository.

[28] Kailun Yang, Luis Bergasa, Eduardo Romera, and Kaiwei Wang. Robustifying semantic cognition of traversability across wearable rgb-depth cameras. *Applied Optics*, 58:3141, 04 2019.

[29] Manizheh Zand, Krishna Kodur, and Maria Kyrarini. Automatic generation of robot actions for collaborative tasks from speech. In *2023 9th International Conference on Automation, Robotics and Applications (ICARA)*, pages 155–159. IEEE, 2023.

[30] Ji Zhang and Sanjiv Singh. Visual-lidar odometry and mapping: Low-drift, robust, and fast. volume 2015, 05 2015.