Carnegie Mellon University

Robotics Capstone Project

Detailed Design

Friction Force Explorers:

Don Zheng
Neil Jassal
Yichu Jin
Rachel Holladay

supervised by
Dr. David Wettergreen

Version 1.0
December 5, 2016

Contents

List of Figures

1 Introduction

NJ: describes what detail design doc is for. also describes organization of doc - parts defined, then overall system operational modes and diagrams, then implementation, then fault-recovery

2 System Diagrams

NJ: Intro specifying that diagrams are for components/full system hardware all diagrams have short description

2.1 Full System Digrams

2.2 Component Diagrams

2.3 Wiring Diagrams

3 Operational Modes

This robot system can be defined by only a few operational modes. These modes represent various states that the system can take from the point of view of a user.

3.1 Mode: User Inputs

The first mode a user encounters is the user input mode. This mode involves the user creating and inputting an image to be drawn. This is all done via the user interface. In addition, this operational mode requires the user to setup the drawing surface, and place the robots in bounds.

3.2 Mode: Preprocessing and Planning

After the system is set up, the offboard processing system runs preprocessing to connect wirelessly to the robots, calibrate any localization, and determine the trajectories for the robot agents to follow. To do this, the planning system uses the user-generated drawing and parses it into a series of lines. It then runs these lines through the path planner to determine a series of trajectories for each robot. While these may be optimized to avoid collision, minute differences in performance and actual-robot speed require real-time collision detection as well.

3.3 Mode: Driving and Drawing

This is the main operational stage, in which the actual result of the system is generated. The offboard system will continually use updating localization and odometry to send commands to navigate the robot agents around the drawing space. In addition, the offboard processor will send commands to raise and lower the writing implement to create the drawing based on the robot position. Robot agents are responsible for parsing data to actuate motors, and sending odometry and other logging information back to the offboard processor. During this stage, the user observes the drawing process, pausing appropriately if they want to pause or stop execution.

3.4 Mode: Error Handling

The error handling operational mode does not occur regularly. This mode occurs whenever the offboard system receives information (from localization, or the robot agents) that an error has occurred. Depending on the issue, the system will shut down or pause. If the system pauses and it is possible to recover from the error, the user is responsible for correcting any issues and resuming operation. The user is unable to assist the system from recovering from fatal errors. Any errors and warnings are displayed on the UI for the user to see.

4 Hardware Fabrication Process

NJ: "explain the fabrication and implementation process"

4.1 Assembly Instructions

NJ: required according to instructions, this can fit all hw subsystems together

4.2 Writing Implement

4.3 Locomotion

4.3.1 Electronics

On each robot, the Raspberry Pi will be connected to an Adafruit DC Motor HAT, which is able to controll all 4 DC motors on the robot. Communication between the Raspberry Pi and the Motor HAT will be handled through the GPIO pins on the Pi.

4.4 Localization

The localization system is comprised of a ceiling-mounted webcam with full view of the workspace. AprilTags will be mounted on the corners of the workspace to allow the system to determine its boundaries. Tags will also be mounted on the robots for the system to determine their locations within the workspace. The webcam will be connected to the central laptop with a USB cable.

4.5 Image Processing

4.6 Communication

The built-in WiFi card on each robot worker's Raspberry Pi 3 will communicate with the offboard laptop computer. The WiFi card is 802.11n compliant, ensuring that packet loss will be minimized and high speed is achieved.

4.7 User Interface

The hardware for the user interface is the laptop on which the central processing is done. Users will be able to use the laptop's mouse and keyboard to specify parameters of operation, and the laptop's monitor to observe the workers' progress.

4.8 Power System

THe Raspberry Pi will draw its power from a 5V USB power bank. The battery pack can hold 3400 mAh of charge, which should allow more than 2 hours of operation. The battery packs are easily swappable and cheap, meaning that our budget allows for multiple packs. Additionally, battery packs can charge each other, making it uncessary to power down the Raspberry Pi when additional charge is needed.

A separate locomotive battery pack will provide power to the motors. This battery pack contains 8 rechargable AA batteries, giving 12V and containing 2000 mAh of charge. This allows around 30 minutes of continuous operation for the motors, requiring only a quick battery swap when charge is depleted.

4.9 Full System

4.9.1 Computation

A WiFi-compatible laptop computer will handle all offboard computation. This includes image processing, planning, scheduling, localization, and managing communication with the robot workers. Only communications, a computationally light task, must be managed in real time. This means that the computational speed of the laptop is largely unimportant to proper operation of the system.

Each robot worker will be equipped with a Raspberry Pi 3 to manage onboard computation, motor controller operation, and communication with the offboard system. The Raspberry Pi 3 is a fully functional Linux system with easily accessible GPIO pins, which can be used for the motor controller. The 1.2GHz quad-core processor will be more than sufficient for the required onboard computation. It

also has a built-in WiFi card, which simplifies communications and decreases the number of necessary parts.

5 Parts List

5.1 Part Name

Part Use: Foo
Supplier: Your Mom
Part Number: 8675309
Price: $0
Quantity: 20

6 Software Implementation Process

6.1 Design Hierarchy

NJ: "object oriented" or "structured design"'. probably best to split this into software subsections

6.2 Software Libraries and Packages

- C++14
- Python 2.7 [1]
- OpenCV 2.4 [2]
- Protocol Buffers 3 [3]
- AprilTags C++ Library [4]
- Python RPi.GPIO 0.6.3 [5]
- Boost.Python 1.60 [6]
- Python Tkinter [7]

6.3 Full System

Full system architecture is split into offboard and onboard, and handles the coordination between the various subsystem software. This section will describe the controller for the offboard and onboard systems separately.

6.3.1 Offboard Controller

The offboard controller's primary function is to continually send updated locomotion commands to the robot agents. This controller will operate based on the pseudocode listed below. Functions are classified by the subsystem they are a part of (Planner refers to the planning subsystem Sec. 6.8, Locomotion refers to the locomotion subsystem Sec. 6.5, etc.).

NJ: not sure how to format this in code typeset

It is important to note that while locomotion, writing implement, and error-related messages are all sent together, any action based on error reports executes with the highest priority on the onboard robot controllers.

The main loop will run continuously at a fixed rate until system operation is finished and the drawing is complete. In order to ensure a fixed-rate operation, large processing steps - such as updating paths, updating the UI, and localizing - will be done concurrently in separate threads. This will ensure the main controller is free to send any error messages immediately, or send an emergency kill command on user input.

Algorithm 1 Full System

```
Given: Input Image I
planner_input = ImageProcessing.processInputImage(I)
paths = Planner.planRobotTrajectories(planner_input)
Communication.connectToRobots
procedure !Planner.isFinishedDrawing
    robot_info = Communication.getTCPMessage(robots)
    localization = Localization.localizeRobots(camera_data, robot_info.odometry)
    paths = Planner.updatePaths(localization)
    locomotion_message = Locomotion.generateCommand(localization, paths)
    writing_message = WritingImplement.generateCommand(localization, paths)
    error_message = Communication.processErrors
    Communication.sendMessage(locomotion_message, writing_message, error_message)
    UI.displayErrors(error_message)
```

Algorithm 2 Offboard Processing

```
Communication.connectToOffboard()
procedure True
    message = Communication.getTCPMessage(offboard)
    locomotion = Communication.parseLocomotionMessage(message)
    writing = Communication.parseWritingMessage(message)
    errors = Communication.parseErrorMessage(message)
    if errors.emergencyOff then
        Locomotion.stopOperation
        Writing.stopOperation
    if errors.pause then
        Locomotion.pauseOperation
        Writing.pauseOperation
    Locomotion.parseErrors(errors)
    Writing.parseErrors(errors)
    Locomotion.actuateMotors
    Writing.actuateMotors
    if message.requestsLogging then
        Communication.sendMessage(Power.batteryLevel, Locomotion.encoders, Writing.encoders)
```

6.3.2 Onboard Robot Controller

The onboard robot controller is responsible for parsing incoming commands, and sending odometry, writing implement, and other logging information back to the offboard system.

The robot will parse commands as it recieves them, and use them to actuate motors for locomotion or raising and lowering the writing implement. It will also parse error information, for messages such as emergency stop, pause, or other noncritical errors. Error processing and action occurs with a higher priority than logging and motor actuation.

In order to send logging information in a way that is easiest for the offboard controller to process, the onboard system will only send logging information when requested by the offboard controller. Log information includes motor encoder data for odometry, battery state, and any debugging information.

The onboard robot controller will operate based on the following pseudocode. Functions are classified by the subsystem they refer to, similar to Sec. 6.3.1.

6.4 Writing Implement

Writing implement software is run both on the offboard system as well as on individual robot agents. This software will be a part of the main control system on the offboard system, written in Python [1]. Onboard, it will be parsed from the communication system Sec. 6.9 and used to command the writing implement.

The offboard system will combine localization and planning data to determine when the writing

implement should be raised or lowered. Once the controller decides the state of the implement, it uses logging information from the agent to determine how far up or down the implement must be moved. This delta is computed as a part of the proto3 message packets [3] and sent to the agent for command.

Onboard a robot agent, the robot controller is responsible for sending the current state of the writing implement back to the offboard system. This is sent using proto3 messages [3] as a part of the logging information. When the agent receives a message, it parses the writing implement command, and then commands the writing implement motor appropriately. This part of the onboard controller is written in Python [1], using the RPi GPIO Library for interfacing [5].

6.5 Locomotion

The software involved in locomotion exists both on the offboard processor as well as on individual robot agents. Robot agents must process commands sent from Sec. 6.9 and command the motors. The offboard processor must use localization from Sec. 6.6 and planning data from Sec. 6.8 to determine the current motor commands.

6.5.1 Libraries

Locomotion software makes use of Python 2.7 scripts [1] for commanding motors via Raspberry Pi [5] and generating proto3 commands offboard by combining localization and planning

6.5.2 Robot Agent

Robot agents read proto3 messages coming from the TCP connection, as specified in Sec. 6.9. These messages contain locomotion commands in the form of specific motor velocitoes; these motor values are relative to each other in value and together, will command the robot in the required direction. The robot parses this proto3 message, and commands each of the four motors via the Raspberry Pi GPIO pins [5]

6.5.3 Offboard Controller

The offboard controller is in charge of combining localization and planning information to create correct locomotion commands for each robot agent. The system must take into account the robot agent's current position and orientation, and then generate locomotion commands that will move the robot along the specified path. Locomotion commands will be generated using controlling python scripts [1].

Omnidirectional motion with mecanum wheels must have a specific controller, as directional movement is not as straightforward as with traditionally-wheeled robots. Controllers already exist, and reduce the problem to selecting a robot velocity, rotational velocity, and desired angle to be rotated to [8]. Localization provides information about the robot's current position and orientation. The path planning algorithm can be used in conjunction with localization to determine the next expected location and orientation of the robot. Given a current robot position and expected, the delta can be computed to represent the position and angle to be moved. Using speed constraints that align with **NJ: REFERENCE nfr:quality from req spec**, velocity commands for each motor can be computed. These are then added the the proto3 message, and sent to a robot for execution.

6.6 Localization

Localization uses data gathered from real-time overhead-mounted camera data to localize the drawing area bounds, and robot positions and orientations. Important locations are marked by AprilTags. AprilTags are the equivalent of 2D barcodes, which can be calibrated and detected at range to determine their orientation and position [4]. Localization software will use the AprilTags C++ library. In order to provide compatibility with the Python scripts used for locomotion and control Sec. 6.5, Boost Python will be used to wrap C++ functions into the Python layer [6].

The drawing boundary, which is static throughout the course of a single drawing, will be represented by four AprilTags placed at the four corners. During the localization loop, these will be found and will remain the same at every iteration. By detecting all four corners, the maximum and minimum coordinates of tags in camera image space can be found. This will then define the bounds for computing locomotion commands. For the bounds, the orientation of the tags make no difference.

In contrast, tracking the robot agents requires both position and orientation tracking. Positions will be reported relative to the bounds, with the bottom left corner bounds marker being designated as (0,0). Similarly, orientations will be reported as an angle relative to the base of the detected bounds.

Algorithm 3 Planner.planRobotTrajectories

1: **Given:** Set of Line Coordinates L
2: $\{L_{R0}, L_{R1}\}$ = SCHEDULER.DISTRIBUTEWORK(L)
3: P_{R0} = PLANNER.GENERATEPLAN(L_{R0})
4: P_{R1} = PLANNER.GENERATEPLAN(L_{R1})
5: $\{T_{R0}, T_{R1}\}$ = PLANNER.GENERATETRAJECTORIES(P_{R0}, P_{R1})
6: **return** $\{T_{R0}, T_{R1}\}$

Algorithm 4 Scheduler.DistributeWork

1: **Given:** Set of Line Coordinates L
2: **RH: try geographic one.. leave open to others**
3: **return** $\{L_{R0}, L_{R1}\}$

Algorithm 5 Planner.generatePlan

1: **Given:** Set of Line Coordinates L_{Ri}
2: **RH: loop through lines and connecting pieces and call like generateControls**
3: **return** P_{Ri}

Algorithm 6 Planner.generateTrajectories

1: **Given:** Plan P_{R0}, P_{R1}
2: **RH: step through each time. if collision, insert pause on alternating robots**
3: **return** $\{T_{R0}, T_{R1}\}$

6.7 Image Processing

Image processing involves the user inputting a series of lines, and converting it into a format usable by the planner. The planner takes an ordered series of lines, and the image processing subsystem will perform this task. This system will create the ordered lines using Python [1] and OpenCV [2].

6.8 Work Scheduling, Distribution and Planning

RH: Need to add explanatory text

6.9 Communication

The communication software subsystem handles sending information to and from the robot agents and the offboard system. The offboard system will be tasked with mainly with sending locomotion and emergency commands to the robot agents. The robot agents will send logging information and odometry measured from motor actions back to the offboard system for processing. The offboard system can use received data to determine if a robot agent has fallen into any error states.

6.9.1 Libraries & Protocols

Communication will be handled via a TCP connection between a robot agent and the offboard system. Robot agents will not connect with each other. Given that the robots only receive and process commands relating to their own motion, there is no need for the robots to be able to directly communicate with each other.

Once a TCP connection is established, data will be sent using Protocol Buffers, a Google-designed standard for serializing structured data [3]. For this project, Protocol Buffers language version 3 (proto3) will be used due to its improved speed and features over the previous version, proto2.

Using proto3 will allow the offboard system to send commands in packets, called ḿessages. This way, the system can organize locomotion and emergency commands into separate messages or sections of a message, which can then easily be parsed by the robot agents.

6.9.2 Message Design

Message design will be addressed first with regard to messages sent to robot agents, then messages sent from robot agents to the offboard system.

All messages sent to the robot agents will involve emergency information commands or locomotion. Emergency commands can be separated into full system stop, and system pause. System stop is the emergency kill switch, which occurs by user command or when the system encounters a fatal error. This will end onboard robot operation. System pause occurs for errors that are recoverable; an example of this would be robot collision. The user has the option to assess and correct any issues, and resume operation. Both system stop and pause will halt all locomotion commands until the robot agent is commanded otherwise.

Locomotion commands will be continually sent to robot agents. These commands will come in the form of relative motor powers to command each of the four mecanum wheels. These messages will be sent continually, to allow for stable corrections to robot motion to maintain smooth lines, which increases the overall quality of drawing.

Robot agents are responsible for sending odometry and logging information back to the offboard processor. Odometry information is sent for each of the four motors on the robot, by sending the encoder ticks since last requested. This odometry information can be incorporated into localization. Motor encoder movement for the writing implement motor will also be sent. Logging information can be used for debugging or system analysis. An example of logging information is the robot battery level.

6.10 User Interface

The user interface is a graphical interface that provides the user the option to enter settings, start, pause, and stop drawing execution, and display error messages. The interface will be written in Python using the Tkinter library [7]. This library makes for a straightforward interface to set user options, as well as display any warnings or errors during the drawing process. The interface will be run concurrently with the offboard control software.

6.11 Power System

Power system software is in charge solely of checking the battery power level of robot agents. As a part of the communication pipeline, battery information is sent as a part of logging information back to the offboard system Sec. 6.9. The offboard system will compare current battery level to the maximum level, and will pause or stop system operation when the battery falls below a threshold. This will prevent the system from entering a state of undefined behavior when not enough battery can be supplied. Power system control is handled by Python [1] in the main controller. This is done for simplicity, as proto3 messages are parsed in the same place Sec. 6.9.1.

7 Failure States and Recovery

NJ: probably best to organize failure states/recovery by subsystem section for describing fault recovery and "degraded modes" similar to failure modes from test plan

7.1 Degraded Mode: Driving but Not Drawing

This failure state occurs when a robot agent is able to drive properly using localization and planning, however the writing implement is not working properly. It can be caused by one of two factors: a broken actuation mechanism, or the writing material is out of ink. The writing implement running out of ink is significantly more likely, as it is a common occurrence and the system is designed around this limitation. For either degraded mode, the only solution is for the user to pause system operation, resolve the issue, and resume operation. The user can either replace the writing implement or replenish ink, or fix the actuation mechanism.

7.2 Degraded Mode: Erratic Driving or Drawing

Erratic driving or drawing occurs when the robot is moving, but not responding properly to the given commands. Alternatively, the commands being sent are incorrect. This failure state also occurs when the drawing actuation commands or actuation is incorrect. Invalid motion can be due to the system failing

to move omnidirectionally (likely due to issues with one or more mecanum wheels), or due to external factors such as slippage. Incorrect commands and undefined behavior can begin as a result of incorrect localization, loss of connection between offboard and onboard systems, battery problems, or the robot agent moving out of bounds. In these situations, the solution is to end system operation. Undefined behavior cannot be resolved without resetting the system.

7.3 Degraded Mode: Planning Failure

Planning failure is a degraded mode that occurs when the input image is unable to be processed into a path. When this happens, the planner is unable to generate a set of paths for the robot agents. The system cannot begin to draw the image without a valid path, so the user is required to enter a new image to begin system operation.

8 Installation Plan

NJ: "includes size, weight, power, other resource needs"

9 Traceability Matrix

NJ: "detailed design should make clear how system meets functional and non-functional requirements, can be done by tracing requirements to design attributes". Maybe another table linking requirements to sub/subsub sections in the doc

References

[1] "Python 2.7.0 release." [Online; accessed 3 December 2016].

[2] "Opencv open source computer vision." [Online; Accessed 3 December 2016].

[3] "Protocol buffers language version 3." [Online; accessed 3 December 2016].

[4] "Apriltags c++ library." [Online; accessed 3 December 2016].

[5] "Rpi.gpio 0.6.3." [Online; accessed 3 December 2016].

[6] "Boost c++ libraries." [Online; accessed 3 December 2016].

[7] "Tkinter - python interface to tcl/tk." [Online; accessed 3 December 2016].

[8] "Driving mecanum wheels omnidirectional robots." [Online; accessed 3 December 2016].