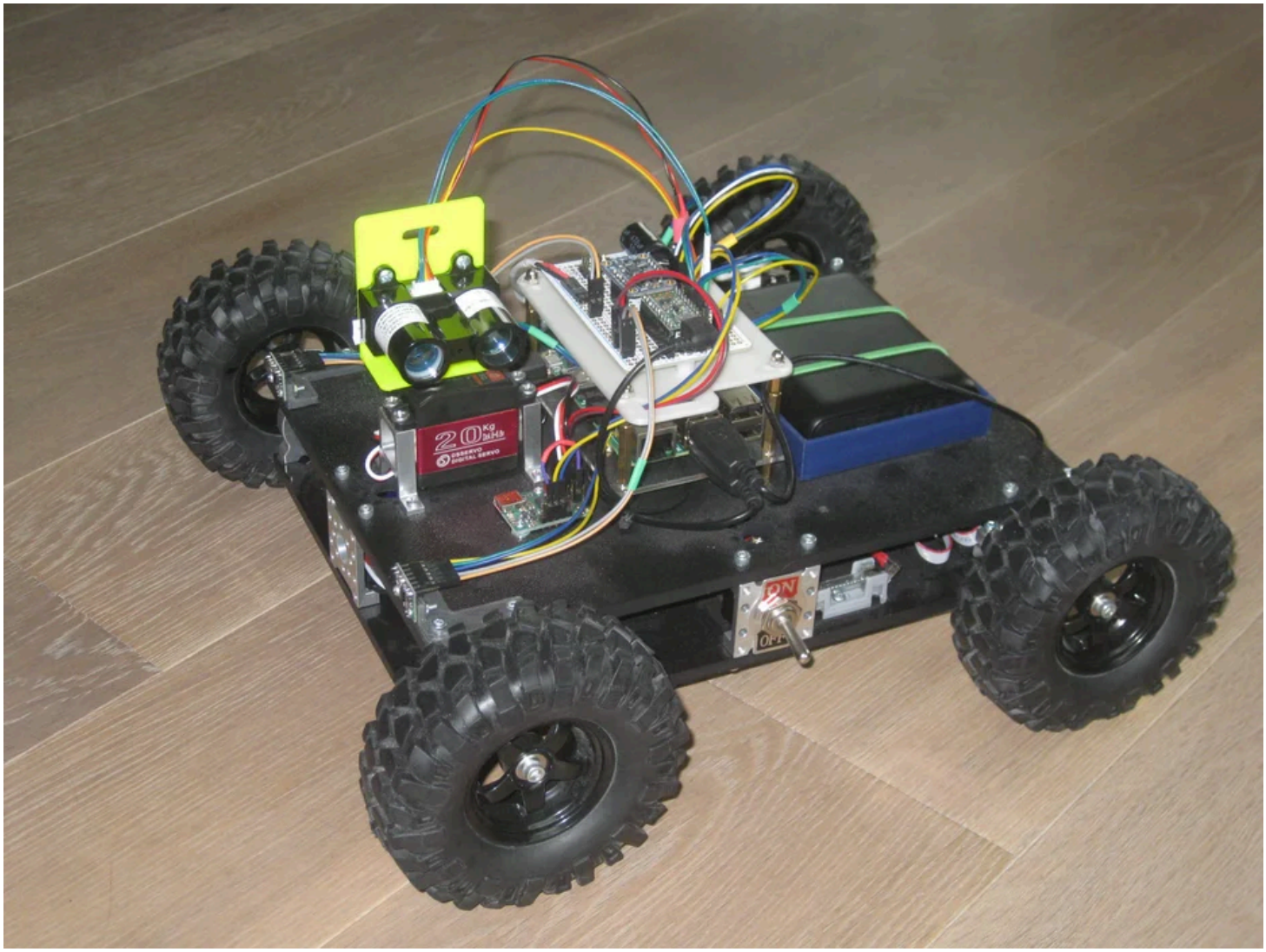
An Autonomous Rover

By GregF10 in CircuitsRobots

Introduction: An Autonomous Rover



The Autonomous Rover travels from one location in my house to another location in my house, with no help from a human. I designed it to experiment in robot autonomy.

To perform its task, the AR (short for Autonomous Rover) integrates three major capabilities:

- *Localization* — determines its current location in its environment
- *Navigation* — determines the optimal path from its current location to a target location
- *Locomotion* — moves along the path from the current location to the target location

Three things…. Sounds pretty simple, but the devil is in the details. With my fully custom design, using hobby-grade mechanical and electronic components, and my limited experience in autonomous technologies, it proved difficult, but both fun and educational.

In truth, the AR is still a work in progress. However, the AR can currently localize with modest accuracy, plan a path perfectly, and move along a rather complex path with remarkable accuracy. Thus, I think that the collection of hardware and software, and lessons learned, might help others with similar goals. At the same time, the AR should make an interesting entry in the current ***Robots*** contest; please consider voting for it.

This Instructable first describes in detail the implementation of the capabilities. The capabilities are covered in three sets of steps that follow:

- Locomotion: Describes how a 4-wheel drive platform equipped with an IMU is used to accomplish locomotion. See steps 1-4.
- Localization: Describes how a LIDAR subsystem (a servo-mounted LIDAR sensor) is used to accomplish localization. See steps 5-12.
- Navigation: Describes how software is used to accomplish navigation. See steps 13-19.

Locomotion and Localization include hardware components. For hardware, I discuss the components, criteria for selecting them, and relationships to other components. Worth noting is that the hardware components were chosen as long as three years ago. In some cases, the components I chose may no longer be the best choice, or in fact, may no longer be available. A key hardware component is a Raspberry Pi that provides the brains for the AR.

All three capabilities include software components. I chose Java as the programming language for the Raspberry Pi due to experience and performance advantages over Python. The choice also allowed me to develop code remotely using world class tools; see my Instructable on the subject. For software, I discuss overall architecture, algorithms and approaches, both empirical and heuristic, and on occasion mention alternatives considered or even implemented. I'll provide pseudocode to aid understanding. Since the AR is still a work in progress, at this time, I don't plan to make the 1000s of lines of code available, but might consider doing so in the future.

A final set of steps, 20-23, describe the integration of the three capabilities, weaving them together to produce the Autonomous Rover. The final steps include a video of the AR in action, and an analysis of an actual path traversal.

Step 1: Locomotion: Introduction

Locomotion is the way the Autonomous Robot moves. I describe locomotion first because it provides the platform for other hardware components, and it must be controlled and monitored by other components. It provides in fact, the *rover* part of the AR. The rover does provide "intelligent motion", but does not provide other aspects of autonomy.

Some important aspects of locomotion:

- Locomotion obviously requires a vehicle that moves. That means a chassis, motors, tracks or wheels, etc.
- Motors must be controlled. I did not want to spend time implementing PID algorithms for speed control, so I decided to use a motor controller to do that work.
- The motors require a battery capable of providing the proper current and voltage levels.
- When moving, it is very nice to know the direction the rover is moving, if the rover is rotating, experiencing unexpected acceleration, etc. That implies the rover needs an inertial management unit (IMU) to help sense such conditions.

- Something must provide overall command and control to provide "intelligent motion".

Step 2: Locomotion: Hardware

The first figure summarizes the overall high-level hardware design.

Command and control

The earliest decision I made regarding hardware is the "brain" for the Autonomous Rover, intended to provide overall coordination and control for locomotion, as well as provide full autonomy for the AR. I chose the Raspberry Pi 3 model B. I'd used earlier versions of the family in other projects, and I wanted to leverage that knowledge base. The Pi 3 was the most modern and powerful Pi available at the time. Even today, I'd probably still go with a Pi 3, as it is less power-hungry than a Pi 4, and has more computing power than say a Pi Zero or earlier models.

Chassis, motors, wheels

Having worked with tracked vehicles, I wanted to do something with wheeled vehicles. I briefly considered two wheeled vehicles (plus castors), but dismissed them as "ugly" and typically too small to accommodate the expected hardware components; later I came to regret dismissing such a vehicle. I briefly considered omni-wheeled vehicles, but feared that would require a lot of time to just get movement to work, reducing the time to investigate autonomy. I briefly considered steerable vehicles, but feared such could not handle the rather tight confines of my house. I eventually settled on a 4-wheel drive, "skid steered", vehicle. It is highly maneuverable, and in theory would allow me to leverage my experience with tracked vehicles.

With that decision made, I begin a search for candidates that:

- provided encoded motors, to support speed control and odometry
- provided enough space for mounting of lots of other components, since there will be controllers, sensors, batteries, wiring, etc
- supported relatively low speeds reliably, since my house is small
- looked cool, because I'm shallow

After a lot of investigation, and dismissal of many candidates due to concerns over quality, I chose a customized Scout Robot Chassis kit from ServoCity. The standard kit comes with unencoded 624 RPM motors, but they were happy to allow me to substitute encoded 116 RPM motors (encoded motors are extra cost, of course). I have to admit that my choice of the Scout over alternatives from other sources was highly influenced by extremely pleasant past experiences with ServoCity products and technical support.

Motor controller

I decided to get a "sophisticated" motor controller to do speed control. I looked at a few and chose the RoboClaw 2x15A from BasicMicro. The choice was driven by

- the controller's ability to drive two motors per channel (the 15A rating comes from driving two motors per channel, where both motors have a ~5A stall current)
- the controller's support for encoded motors
- the controller's excellent set of control capability (speed, distance, acceleration)
- the company's support for the Raspberry Pi
- the effective endorsement by ServoCity (the part is available in their online store)

The design connects the two left motors on the Scout to RoboClaw channel 1 and the two right motors on the Scout to RoboClaw channel 2. The design produces a "skid steered" vehicle, with characteristics somewhat like

a tracked vehicle.

The Raspberry Pi interacts with the RoboClaw via a USB serial connection.

Motor battery & connections

I decided to use a LiPo battery (3S) to power the motors. It offers the correct motor voltage and plenty of current capacity. I tend to over-engineer, and I bought a 6000 mAh battery so I could do many hours of testing between charges. After months of testing, the battery never came close to running out of charge before I ran out of time or energy to test. I suspect now that a capacity of 3000 mAh would be sufficient under the vast majority of circumstances.

The RoboClaw Data Sheet describes the connections between a battery and the RoboClaw. The circuit includes a switch, a fuse, a high current diode, and a resistor. The switch is desirable to act as a kill switch in case of an out-of-control robot. Conversations with BasicMicro indicated the resistor is only needed for battery voltages above 24V, so I eliminated it from my design. I decided to include a connector between the battery and the fuse to allow the battery to be easily detached and removed for balanced charging.

I worried about the current rating of the 28 AWG wire in the motor flat cable. Reliable sources suggest the maximum sustained current that gauge can handle is 1.4 A, but the motor could draw as much as ~5A. This source suggests, however, that in a ribbon cable, a wire can handle as much as 2x the current. I pondered over the ratings for the fuse (and switch). With four motors, the total current could reach up to 20A. To limit the current to what the ribbon cables can handle, one would need a fuse at about 5-6 A. I eventually settled on a 5A fast blow fuse to provide what is probably conservative protection. There have been no problems so far.

The RoboClaw Data Sheet also describes the connections between the motor and encoders and the RoboClaw. I won't repeat them.

Inertial Management Unit (IMU)

An IMU contains a magnetometer, an accelerometer, and a gyroscope. I expected the gyroscope to be important to rover motion control. From previous robots I built, I found that a magnetometer is useless in my environment, due to lots of rebar in the floor of my house (actually a condominium). For that reason, I did not attempt to use it it the AR. I also did not plan to use the accelerometer.

I initially chose an IMU product based on previous experience with it. However, after some testing I was disappointed in its gyroscope performance. After some searching, a comparison by Adafruit convinced me that the Adafruit Precision NXP 9-DOF Breakout Board was the best choice for my situation. Based on my experiences, the comparison was accurate. The gyroscope module on the NXP 9-DOF Breakout Board is the FXAS21002C. The Pi interacts with the FXAS21002C via I2C.

A bit of detail about using the FXAS21002C is significant. A gyroscope measures the angular rate of change. Like any semiconductor device, it is subject to various types of noise. An important one is zero offset. Fundamentally, the value output for a true zero rate of change can be non-zero, and can vary over time, primarily as a factor of temperature. This has two impacts; first, one must subtract the current zero offset from the value read from the gyro to get a valid value; second, one must find the current zero offset immediately prior to using the gyro to ensure the maximum accuracy.

Another important form of gyro noise seems to be more or less random; even after dealing with the zero offset, noise causes rate of change values that imply rotation, even when the rover is motionless. I found it necessary to create a “dead band” such that values within the band are considered noise and are ignored; my testing found that this dead band was relatively constant, and so it is hardcoded.

Since the Pi interacts directly with the motor controller, it seemed to make sense to connect the IMU directly to the Pi as well. The allows the Pi to detect unintentional rotation on a forward motion, and detect intentional

rotation on a rotate motion, and then make motor speed adjustments appropriately.

Raspberry Pi and power

A critical consideration is powering the Pi. With a high capacity battery for the motors on board, I could have used a voltage regulator attached to that battery to power it. However, Raspbian OS does not like to be powered off without a shutdown, and I anticipated needing to turn off power to the motors on lots of occasions where I'd still want to be using the Pi. Thus I decided to use a power bank to power the Pi independently.

Physical construction

ServoCity published a video that shows the basic construction of the Scout chassis. An important feature of the Scout chassis is its two level design. That allowed me to mount all of the motor related hardware on the lower level, keeping the upper level available for other hardware components. The remaining figures show the lower level

- the overall construction
- the RoboClaw mounting and connection
- the switch/fuse between the battery and the RoboClaw
- the left motor connections
- the right motor connections

The motor battery is held stable with some angle brackets; cable ties provide some flexible tension. The motor cables are secured with some handmade supports; these could have been 3D printed.

Both the Pi and the IMU are mounted on the upper level. I mounted the Pi on a stacking platform to allow for additional components in close proximity. The location of the Pi itself is not critical, but is restricted for the purposes of localization. The location of the IMU, on the other hand, is critical. Since it is expected to measure rover rotation, to maximize accuracy, the IMU should be mounted at the centroid of the rover. Unfortunately, I did not achieve that goal exactly, which may account for some of the inaccuracies described later. Finally, I designed a holder for the power bank using a CAD program and had it 3D printed.

Step 3: Locomotion: Software

Overview

To achieve the overall goal of the Autonomous Rover, the locomotion software needs to deliver only two distinct motions:

- move forward a desired distance
- rotate a desired angle

Initial testing of forward motion showed the rover could not move straight over meaningful distances. This is really not surprising given that motors don't behave identically, the Scout tires are flexible and thus don't necessarily translate motor rotation accurately to linear movement, the system is noisy, etc. Similar limitations apply to rotation. Thus, performing either motion requires controlling the motors, above and beyond the lower level control provided by the RoboClaw. It also requires monitoring the gyroscope (or gyro) to inform motor control. As a result:

- Forward motion requires
    - driving the left motors and right motors forward at the same *nominal* speed
    - monitoring the gyroscope to determine the current heading error
    - adjusting the left and right nominal speeds to correct the heading error
    - monitoring the odometry (total motor encoder counts) to determine when to stop the motors
- Rotation motion requires
    - driving the left and right motors in opposite directions at the same speed
    - monitoring the gyroscope to acquire the current heading to determine when to stop the motors

The need for synchronized monitoring of the gyroscope and writing control commands to the RoboClaw suggests a threaded approach. Fortunately, Java provides a sophisticated threading subsystem. One thread, a class called **Rover** (since it does not implement the high level autonomous features), controls the motors via the class **RoboClaw** a representation of the RoboClaw. A second thread, a class called **Gyro**, interacts with the gyroscope device to produce an angle or heading; it uses class **FXAS21002C** that represents the gyro device. The two threads interact thru a synchronizing object, from a class called **Angle**. The gyro has a configurable sample rate, so it is perfect for pacing the entire system.

RoboClaw implementation

Class **RoboClaw** represents the RoboClaw motor controller and provides a method for all of the device commands listed in the RoboClaw User Guide. I modeled the implementation on the Python driver for the Raspberry Pi available from BasicMicro. While not necessarily productive, it was necessary to achieve the utmost performance and forced a deeper understanding of the device.

The Pi connects to the RoboClaw via a USB serial port. Initially I found three candidates for low level serial interaction using Java. In another bit of over-engineering, to allow different underlying implementations, I created an interface class called **RoboClawSerialPort**; **RoboClaw** uses it to communicate with the RoboClaw device. **RoboClawSerialPort** defines the methods:

- *write* — writes a byte array to the port
- *read* — reads a byte array from the port
- *flush* — ensures any buffered data gets read
- *drain* — ensures any buffered data gets written

Testing showed the best support for serial interaction over USB came from the jSerialComm library. Even though it was the best at the time, jSerialComm had some limitations. I had to use the default configuration of

the library because timeout values less than 100 milliseconds seemed to cause unpredictable results when using blocking writes and reads. Further, for blocking reads, the "available" method seemed to fail almost every time. Finally, all attempts to use semi-blocking reads also resulted in unpredictable results. All that said, at the time I implemented the RoboClaw driver, jSerialComm was at version 1.3.1.1; now it is at version 2.0.0.3, so the limitations may no longer exist.

A class called **ClawSerialImpl** actually implements the **RoboClawSerialPort** interface using jSerialComm v1.3.1.1. **ClawSerialImpl** also exposes a method called *getPortByDescription* that uses a (possibly) jSerialComm-specific capability that derives human readable identities from serial devices connected to the Pi. This allows a using class to look for the word “RoboClaw” among device identities of all USB attached devices to find the proper serial port for communication with the RoboClaw.

FXAS21002C Implementation

I created a singleton class called **FXAS21002C** to represent the gyroscope device. It supports the methods:

- *configure* — permits configuration of the full scale range, the low pass filter cutoff frequency, and the output data rate; all other device configuration parameters are fixed
- *readStatus* — read the status register that consolidates the status of the output data registers
- *readGyroZ* — reads the output data for the Z-axis (the one that is meaningful for the rover)
- *readGyro* — reads the output date for the X,Y,Z axes; the current design just reads the Z-axis, so this is no longer used

Adafruit has an Arduino library for the FXAS21002C. I used it as great guidance for the implementation of **FXAS21002C**.

FXAS21002C uses I2C to interact with the module. I found the Java library Pi4J that enables I2C communication via Java. It worked … mostly. However, it did not support the clock-stretching required by the device; thus, block reads, important for performance, could only start at address 0. I found that by dropping to the underlying wiringPi library, I could successfully read from any register; unfortunately wiringPi does not support block reads, so I had to use a mixed approach. I used Pi4J version 1.2-SNAPSHOT from late 2017; a released version became available in late 2019, and it may address the problem.

Angle implementation

Angle is quite simple. It implements a producer/consumer approach that has two methods:

- *get* — if data is current, the data is marked not current, waiting threads are awakened, and the data is returned to the consumer; if data is not current, the method waits, and is awakened when the producer calls *put*; at that point the data is returned, etc. (used by **Rover**)
- *put* — the data is stored and marked current, and waiting threads are awakened (used by **Gyro**)

Gyro implementation

Gyro interacts with **FXAS21002C** to provide what amounts to a current heading of the rover. In its constructor, **Gyro** configures the FXAS21002C for a full scale range of 250 degrees per second (the most sensitive), for a low pass filter cutoff of about 8 Hz (for best noise reduction), and for an output data rate of 50 Hz. **Gyro** has the following methods:

- *goActive* — starts the collection and integration of gyro readings to produce an angle and make the angle available to consumers (used by **Rover**)
- *goIdle* — stops the collection of gyro readings; this eliminates the CPU cycles that would otherwise be consumed (used by **Rover**)
- *setZeroOffset* — reads the gyro for some period of time, with the AR motionless, and averages the reading to produce the current zero offset
- *run* — for the threading subsystem
- *terminate* — terminates the thread in which the instance runs

Obviously *goActive* is the most interesting. The FXAS21002C device delivers an angular rate of change in degrees/second. Thus, to produce a heading, **Gyro** must integrate the change over time. As mentioned earlier, the gyro sets the pace for motion control. The FXAS21002C output data rate is configured to 50 Hz, but I felt that is too fast for doing motor control from the Pi since the RoboClaw does the low level control. I felt a better frequency for motor control at the level of the Pi would be 10 Hz. Thus, **Gyro** collects 5 samples to produce an average rate of change and integrates that, applies the zero offset, and the dead zone, to produce the current *relative* heading. Once the heading is calculated, it calls **Angle** to make it available to **Rover**.

In an effort to minimize CPU cycle consumption, gyro sample collection is a bit tricky. The sample period is 20 milliseconds. Rather than loop, checking the device status as fast a possible, **Gyro** sleeps for 18 milliseconds, and then checks device status as fast a possible. Once the status indicates the Z value is ready, **Gyro** reads it and does the averaging.

The pseudocode for the subset of the **Gyro** thread run() method when **Gyro** is active follows:

```
while active
        if initial sample
                read and discard 2 readings # for synchronization purposes
        else
                sleep for 18 ms
                wait for gyro reading available
                get the Z axis reading
                increment number of readings
                accumulate reading
                if have accumulated 5 readings
                        find the average
                        subtract zero offset
                        if average within noise band
                                average = 0
                        headingChange = average * device scale factor
                        angle = angle + headingChange
                        Angle.put(angle)
                        reset number accumulated and accumulation
```

Rover implementation

Rover implements the algorithms that produce the two types of motion required. **Rover** exposes the following methods:

- *execForward* — move forward a given distance (in cm)
- *execRotate* — rotate a given distance (in degrees)
- *awaitMotionCompletion* — wait until a motion completes
- *getMotionReport* — provides encoder counts and heading
- *run* — for the threading system
- *terminate* — terminates the thread in which the instance runs

To effect motion, instances of both the **Rover** and **Gyro** classes get created and their threads started using the Java Executor subsystem. Each thread goes idle, awaiting commands.

execForward

For forward travel, **Rover** monitors the current angle, or relative heading, and tries to ensure it remains at 0°. To do so, it uses a Proportional/Differential control approach to adjust the motor speeds to correct a heading error. No Integral component is used because it is more important to have the final relative heading as close to 0° as possible.

The following pseudocode describes the implementation of *execForward*:

```
Gyro.setZeroOffset
Gyro.goActive
calculate cruise distance # in encoder counts
RoboClaw.setSpeed(initial speed)
heading = Angle.get
phase 1: accelerate to high speed
phase 2: cruise at high speed until reach cruise distance
phase 3: decelerate to initial speed
RoboClaw.setSpeed(0)
heading = Angle.get
report distance traveled (in encoder counts) and final heading (in degrees)
Gyro.goIdle
```

All phases attempt to keep the relative heading close to 0°.

Phase 1 implements the following pseudocode:

```
do:
        increment nominal speed for left and right motors
        calculate speedCorrection using the heading error and PD constants,
                allowing for dead band # see below for information on dead band
        adjust nominal speeds with speedCorrection
        RoboClaw.setSpeed for left and right motors
        heading = Angle.get
until at maximum speed
```

Phase 2 implements the following pseudocode:

```
do:
        calculate speedCorrection using the heading error and PD constants,
                allowing for dead band
        adjust speeds with speedCorrection
        RoboClaw.setSpeed for left and right motors
        heading = Angle.get
        RoboClaw.getEncoderCounts
until reach cruise distance
```

Phase 3 implements the following pseudocode:

```
do:
        decrement nominal speed for left and right motors
        calculate speedCorrection using the heading error and PD constants,
                allowing for dead band
        adjust nominal speeds with speedCorrection
        RoboClaw.setSpeed for left and right motors
        heading = Angle.get
until at initial speed
```

Moving forward a specified distance proved a bit tricky:

- The Scout tires introduce a bit of a disconnect between motor rotation, which can be measured somewhat precisely using the odometry (encoder counts) capability offered by the RoboClaw, and actual travel on the floor.
- Deriving the so-called "PD constants" was a rather laborious exercise of testing and testing until "everything seems to work as best as possible". It was often difficult to tell good from bad given the seemingly random nature of some of the behaviors. I am not convinced I achieved "everything works as best as possible"; it may be more like "I worked on it until I can't tolerate working on it".
- Motor speeds below 160 PPS (pulses per seconds) produced jerky and unreliable forward motion. The "initial speed" mentioned above referenced in the pseudocode above is 160 PPS. Basically I asked the motors to get the that speed as fast as possible to avoid the inaccurate lower speeds.
- Since the speed control period is 100 ms, during the cruise phase (2), the AR travels about 2 centimeters for every iteration through the loop. The one could expect an error in the range ±2 cm. Nevertheless, the approach produces an average error of about ±1 cm, even for desired travel of 550 cm.
- The AR rarely travels truly straight. There are lots of sources of physical and electrical “noise”. As a result, the AR tends to drift to the left or the right as it travels; the direction of the drift and the magnitude of the drift seems to be random, and defies any attempt to eliminate it. That said, with extensive testing and PD control tuning, I’ve been able to limit the drift magnitude to less than 3 cm in the majority of cases for travel distances around 550 cm.
- The final heading reported by the gyro does not always match the heading measured manually (also prone to error) or determined visually (also prone to error). This could be due to improper mounting (not on the centroid) or noise or both.

One other physical phenomenon is worth mentioning. During testing, I noticed a periodic heading error with a period of about 1.5 seconds and a peak to peak magnitude of ±0.1°. It took me a while, but after looking at the encoder counts, the resolution of the motor encoders, and the theoretical diameter of the tires, I concluded that the error was related to tire rotation. I felt, therefore, that there was no way to correct it, and thus there was no reason to try. I implemented another “dead band” in the speed control approach; if the heading error was less than ±0.1°, I ignore it. So far, it seems to have worked well, with a final reported heading usually within ±0.2°.

The figure shows the plot of the heading for a forward motion of 550 cm. You can see that it took about 26 seconds to travel 550 cm. The heading varies from -0.2° to 0.1°. In this trial, the finally heading reported -0.015°. If you look at the plot, you can see a ‘signal’ with a roughly 1.5 second period and a magnitude of roughly ±0.1° that I attribute to the "misshapen" Scout wheels.

execRotate

For rotation, **Rover** monitors the current angle, or relative heading, and tries to ensure it reaches the desired relative heading. To do so, it simply drives the motors and monitors the gyro until the rover achieves the heading. To support the expectations from Navigation, the AR needs to rotate either 90° clockwise or 90° counterclockwise.

During testing, I found that clockwise rotation was more accurate than counterclockwise rotation. I’ve still not discovered why; it could be due to hysteresis in the motor gearing or something else. So, even if the request is for a rotation of 90° counterclockwise, the rover will turn clockwise to achieve it, i.e., 270°.

At a high level, rotation occurs in two parts:

- rotate at high speed until the heading reaches a first pre-target angle, empirically determined to be around 12° less than the total physical rotation requested
- rotate at low speed until reach a second pre-target angle, empirically determined to be around 0.5° less than the total physical rotation requested

The following pseudocode describes the implementation of *execRotate*:

```
Gyro.setZeroOffset
Gyro.goActive
calculate physical angle from logical angle
calculate first and second pre-targets
RoboClaw.setSpeed (initial speed)
heading = Angle.get
Phase 1: accelerate to high speed
Phase 2: cruise at high speed until reach first pre-target heading
Phase 3: decelerate to low speed
Phase 4: cruise at low speed until reach second pre-target heading
RoboClaw.setSpeed (to zero)
heading = Angle.get
report distance traveled (in encoder counts) and final heading (in degrees)
Gyro.goIdle
```

Phase 1 implements the following pseudocode:

```
do:
increment/decrement nominal speed for left/right motors
RoboClaw.setSpeed for left and right motors
heading = Angle.get
until at high speed
```

Phase 2 implements the following pseudocode:

```
do:
RoboClaw.setSpeed for left and right motors (high)
heading = Angle.get
until reach first pre-target heading
```

Phase 3 implements the following pseudocode:

```
do:
decrement/increment nominal speed for left/right motors
RoboClaw.setSpeed for left and right motors
heading = Angle.get
until at initial speed
```

Phase 4 implements the following pseudocode:

```
do:
RoboClaw.setSpeed for left and right motors (low)
heading = Angle.get
until reach second pre-target heading
```

Rotation is perhaps more well-behaved than straight movement. All of the same noise sources exist, however, so rotation is not perfect. For example:

- The final heading is reported by the gyro to be within ±0.3° of the desired heading in the vast majority of cases. Visual estimation typically agrees with the reported heading.
- Rotation introduces translation in both axes. The magnitude has never measured more than ±2 cm on an axis. This seems to be another artifact of the "mushy" Scout tires, and there seems to be a random component to such behavior that could be due to gearing, motor differences, etcetera.

Step 4: Locomotion: Conclusion

I've described the rover implementation that provides Locomotion capability for the Autonomous Rover. It performs reasonably well. Forward motion is accurate to ±2 cm in the direction of travel. Unintentional drift (left or right) during forward travel is bounded to ±3 cm at the maximum forward distance of 550 cm; the drift for shorter distances is proportional to the distance. Rotations, constrained to ±90°, are accurate to ±0.3°. Unintentional drift during rotation is bounded to ±2 cm in both axes.

If I were to design and build the rover again, however, some of my choices would be different. The biggest change would be using a two wheeled vehicle, with stiff wheels/tires, to enable better accuracy during motion. Such a change should not be interpreted as denigrating the Scout. In my opinion the Scout, like other skid steered vehicles, was not designed my use case. So the real problem is mine, as I did not do enough research before choosing the Scout.

Step 5: Localization: Introduction

Localization is the way the Autonomous Rover determines its current location in its environment. Localization is about sensing the local environment and attempting to determine the *pose* within that environment, i.e., the position (in planar Cartesian coordinates) and the orientation (angle or heading) with respect to some reference frame. As any real system moves from a current pose to a target pose, there will be errors introduced, in both the position and orientation. This is certainly true for a system with the limitations or the rover described earlier.

What sensors help measure pose? The rover includes the equivalent of an odometer via the motor controller, which accumulates motor encoder pulses. The problem with odometry is that with the rover is that there is some disconnect between the rotation of the motors and linear motion of the rover, so the odometer has questionable accuracy. The rover also includes the equivalent of a compass via a gyroscope. The compass too appears to have some disconnect between the actual rover rotation and the heading reported, so the compass also has questionable accuracy. In any case, the odometer and compass can really only help estimate the *relative* pose error and don't really help understand the *absolute* pose error.

A hot topic in autonomous automobiles (and other mobile autonomous forms) is lidar. I decided to include a lidar subsystem to enhance localization. After testing the lidar subsystem I built, I found that it too had limitations in terms of accuracy for a number of reasons, such as fundamental accuracy limitations in the lidar device, differences in the characteristics of the surfaces in my house, and other "noise" soures.

Raw lidar data is nearly useless for localization. I had to find and implement a means of processing the raw lidar data into something more useful, like line segments. Unfortunately, since the raw data is inaccurate, the line segments are of suspect accuracy.

And finally, the lidar line segments alone generally are not sufficient to determine the pose. I made a map of my house and created a technique I call a virtual scan, which assumes an ideal pose and emulates a lidar sensor scanning from that ideal pose. Then I created a technique to match the results of the virtual scan to the results of the real lidar scan, enabling a way to estimate the pose error, and thus the actual AR pose.

Step 6: Localization: Hardware

Localization hardware design

A lidar subsystem fundamentally implies a lidar device that emits light and receives the reflection, and a means of calculating range from the time difference. It also implies some means of pointing the device in the proper direction to get a scan of the environment.

Most lidar subsystems do a full 360° scan, and in fact do so continuously. I decided to constrain the scan to 180°, as that seemed adequate for purposes of localization in my context. That allowed me to use a servo to point the lidar sensor.

I considered several options for controlling the lidar device, or sensor, and the servo. I wanted good synchronization between a lidar sample and servo movement. I concluded it would be a good idea to offload such “real time” tasks from the Raspberry Pi doing overall control and coordination. I initially used an Arduino to drive both the lidar sensor and a cheap servo I already possessed. Testing exposed obvious inaccuracies. I bought a much better servo and a real servo controller. These two changes improved lidar scan accuracy significantly.

The first figure summarizes the localization hardware design.

Lidar device

At the time I chose the lidar sensor (mid 2017), the Lidar Lite V3 (LLv3) from Garmin appeared to be the best option based on a combination of accuracy and cost. Lidar costs have decreased dramatically since that time, and other choices might now be better.

The LLv3 Operation Manual discusses the full details of the device. The most important features for this discussion are the range, accuracy, and interface. I used I2C to interact with the device. The maximum range is 40 meters, far more than I needed for my house. The accuracy for ranges 1 m to 5 m is ±2.5 cm, and for ranges above 5 m it is ±10 cm. The response is linear from 1 m to 40 m, but non-linear below 1 m. So, yet another source of error.

During testing I discovered that, unfortunately, like the gyroscope, the LLv3 must warm up before range values are reliable. Even more unfortunate, it can take minutes of warmup, though it can take much less, depending on circumstances.

The LLv3 physical design does not lend itself to mounting on a servo. I designed a mount using a 3D CAD program and had it 3D printed. I did not trust my CAD skills, or 3D printers, enough to deal with the servo spline, so attached the mount to a servo horn.

Servo

I chose a “good” servo because of some desirable characteristics:

- Its digital nature, high precision metal gears, and 3 microsecond dead band suggest better accuracy.
- Its high torque eliminates concern over moving the lidar device (and the mount).
- Its 270° rotation ensures that I can get a full 180° scan.
- Its operating voltage range included 5V, so I could power it with the existing power bank.

Servo controller

I chose the Pololu Micro Maestro servo controller based on its capabilities and previous experience. It allows serial control via USB or TTL and drives up to 6 servos with a 0.25 microsecond resolution. This really provides some nice accuracy, far superior to what an Arduino provides.

An important limitation must be discussed. I felt is was important to have the LLv3 positioned accurately at 0°, 90°, and 180°. I empirically determined the values for those three positions, using the Maestro Control Center, by measuring the angle of the LLv3, and adjusting the values until the angle was correct. The value to achieve any other angle can be determined mathematically using those three values. First, one can get a '0.25 microseconds/°' factor using the difference between the values for 180° and 0° divided by 180. Then, any other angle is the angle times the factor plus the value for 0°. But, in general, the factor is not integral, and the value sent to the Maestro must of course be an integer; thus, the actual angle achieved is only close to the angle desired, because of rounding.

Arduino

I chose the Pololu A-Star 32U4 Micro (a.k.a. A*) for the micro-controller based on its capabilities and previous experience. Since it is based on the 32U4, it supports a USB serial connection, which I used to interact with the Pi, and another TTL serial connection, which I used to interact with the Maestro. It of course supports I2C, which I used to interact with the LLv3. It is also small and relatively inexpensive.

Power

The LLv3, the servo, the Maestro, and the A* all need power (5V). All but the A* get power directly from the power bank that also powers the Pi. The Pi powers the A*.

Mounting the Lidar subsystem

The Lidar subsystem has to be mounted on the top level of the Scout chassis. I used a vertical servo mount from ServoCity to hold the servo. The LLv3 comes with a pre-made cable, so I had to make sure the spatial relationship between the Pi and the device allowed for enough slack so that servo rotation put no stress on the connection to the Pi.

I mounted the Maestro on the top level of the chassis next to the servo mount. The second picture shows the final result.

Step 7: Localization: Software

As indicated above, an Arduino drives both the servo (via the Maestro) and the LLv3. However, it is the Pi that processes the raw lidar scan to produce line segments and perform other higher level localization tasks. This split supports

- tight coordination between servo movement and LLv3 sampling, important for accuracy
- offloading of the low level functions from the Pi

I initially designed the localization capability around collecting lidar samples every 1° during a scan. After gaining some experience with the analysis of the raw scan, I eventually decided extra resolution would be a good thing and now collect samples every 0.5°. I think the change resulted in a significant improvement in the outcome from analysis.

A* software

Pololu provides a Maestro Arduino library supporting the Maestro Micro. It is useable as is.

Garmin provides an LLv3 Arduino library. Navigate to the src folder. I believe that LIDARLite.h and LIDARLite.cpp are the files you should use; however, those files are **not** the same as the library I downloaded in 2018. The others files in the folder are for other, newer devices. I can offer one warning. In the 2018 code I have and in the current library there are a couple of places that when errors occur, the library writes "> nack" to the Serial channel. That interferes with serial communication between the Pi and the A*. I made a copy of the library in my sketch and eliminated the behavior.

The A* and Pi communicate using USB serial. From the A* perspective, things are pretty simple. The *setup* function initializes the modules that drive the servo and the LLv3. The *loop* function waits for a single byte and calls the "command executor" which then calls the right function to perform the command. Each command waits for any additional bytes that provide parameters for the command, and then performs its function. For convenience, I separated the sketch into several files: a .cpp and .h for the LLv3 library, a code and .h file for the command executor, a code and .h for the lidar sensor, and a code and .h for the servo.

There are a few key commands:

- *lidar_scan* — initiates a scan producing 361 ranges (starts at 0° and ends at 180° with 0.5° increments); the samples are cached in the A*
- *lidar_scan_retrieve* — returns the 361 ranges from the scan; the Pi simply reads from the A*
- *get_servo_parameters* — returns the three values sent to the Maestro to position the servo at 0°, 90°, and 180°; this allows the Pi to calculate the actual angle at which each lidar range is taken; the Pi reads the three values from the A*
- *lidar_warmup* — a parameter indicates the length of the warmup, from 0 to 5; the A* takes 1000 to 21000 lidar samples, based on the parameter

The following is the pseudocode for *lidar_scan*:

```
set 'servo delay' to the parameter or to the default
send the servo to 0°
do a pre-scan warmup (relatively short)
for angle = 0 to 360
        get a range reading and store in array
        increment servo position by approximately 0.5°
        delay by 'servo delay' to allow servo to completely stop
```

There are some things worth elaboration.

- Note the need for a ‘servo delay’ between range samples. During testing I found that moving the servo and sampling too quickly afterwards generated a “noisy” result. I believe the servo was seeking its final position electronically, or perhaps mechanically vibrating. In any case, after additional testing I found that for my subsystem, a delay of 80 milliseconds eliminated the noise. I parameterized the command to accommodate testing, where utmost the accuracy was not necessary.
- Note the pre-scan warmup. During testing I found that even after very long warmups, periods of inactivity resulted in faulty ranges; a short warmup period seemed to eliminate the problem.
- Positioning the servo requires calculations described in the previous step. The servo parameters refer to the three values, in 0.25 microseconds, sent to the Maestro to position the servo at 0°, 90°, and 180°. These parameters were determined empirically, and are hardcoded, and are used to determine the value sent to the Maestro per the earlier discussion.

Pi software

The lidar subsystem is represented by a class called **Lidar**. The class uses the jSerialComm library for communication with the A*. The same limitations for library usage noted in Part 2 apply.

Lidar is designed to run its own thread, which enables the warmup function to be run “off line”. The **Lidar** constructor gets the servo parameters to facilitate calculation of the real angle of each servo position in the scan.

Lidar exposes the following major methods:

- *getScanReport* — initiates a lidar scan by sending the proper command to the A*, retrieves the raw scan data from the A*, and calculates the Cartesian coordinates from the effective polar coordinates from the lidar subsystem; the calculation uses the servo servo parameters to determine the exact angle for each sample
- *startSensorWarmup* — starts a lidar warmup by sending the proper command to the A*
- *isWarmupRunning* — determines if warmup still running by attempting to read the return code from the A*
- *run* — for the threading subsystem
- *terminate* — terminates the thread in which the instance runs

The pseudocode for *getScanReport* follows

```
start the scan by sending the command and ‘servo delay’ to A*
wait for the scan to complete
retrieve the raw scan (ranges only) from the A*
calculate the Cartesian coordinates for scan [see below]
return the result
```

The calculation of the Cartesian coordinates is a bit interesting:

- As described in the previous step, the 0.25 microsecond resolution of the Maestro is not sufficient to produce exactly the angles desired. After servo calibration for precise angles at 0° and 180°, there are 28.88889 0.25 microsecond steps per 1°. The Maestro accepts only an integral number of steps, so, for example if the ideal angle is 5°, the actual angle is 4.985°.
- As mentioned earlier, the LLv3 is non-linear at ranges less than 1 m. Due to the tight confines of my house, such short ranges are a frequent occurrence. I found that to model the non-linearity, I had to use two linear equations, one for below 58 centimeters and second for 58 cm to 1 m.
- In addition, the LLv3 produces ranges of 0 or 5 for forms of invalid ranges (e.g., no signal return). Such values had to be mapped to obviously invalid coordinate (0, -1).

The following pseudocode shows the calculation:

```
for all 361 ranges
        calculate ideal angle for range (1/2 of loop index in degrees)
        calculate the real angle using the servo steps/degree
```

```
        if range <= 5
                mark as invalid (y=-1)
        else
                if range < 58
                        map range using equation 1
                else if range < 100
                        map range using equation 2
                else
                        range not mapped
                calculate Cartesian coordinate (x,y) from range and real angle
```

Putting it all together

The following pseudocode describes a simple user of **Lidar**:

```
create a Lidar instance
start the instance thread
start lidar warmup
wait for warmup done
get lidar report
```

At this point the report can be analyzed, stored, plotted, etc. I found it very convenient during testing to use a spreadsheet program (e.g., Excel or Numbers) to quickly plot the Cartesian coordinates. This allows a very quick look at what the LLv3 “sees”.

The figure is an example of scan of an area in my house with the LLv3 positioned orthogonal to the surfaces around it (as best as I could achieve that). This example demonstrates the challenge of making sense out of the “point cloud” produced by a lidar scan.The origin represents the LLv3. Each blue dot represents one of the 361 lidar range samples. The scan shows some “good” surfaces (smooth, flat, and non-reflective); examples include the more or less horizontal sets of samples at about y=380 (painted baseboards) and the more or less vertical sets of samples around x=-90 (painted door) and x=-80 (painted baseboard). The scan also shows some “bad” surfaces (not smooth, or not flat, or reflective); examples include the “squiggle” around x=-75 and y=10 (stainless steel) and the more or less vertical set of samples around x=-60 (wood with large wormholes). Finally the scan shows what is in effect “noise” due to things like chair legs and table legs seen as one to a few isolated samples, especially in the rectangle defined by the coordinates (0,150) and (200,300).

Step 8: Localization: Finding Surfaces in a LIDAR Scan

As seen in the previous step, the point cloud produced by a lidar scan can be quite a challenge to understand, even for a human. But localization requires exactly that — an understanding of the scan.

A huge amount of research has been done on the subject. In effect, surfaces get represented by lines or line segments, and in extreme cases, curves. I looked for an approach to finding line segments that (a) seemed to produce good results, and that (b) I thought I could implement. I found this paper that seemed to satisfy both conditions. The paper contains pseudocode for different aspects of the approach. In fact, at the bottom of page 6 of the paper, one can find a link to the code the authors used for testing; that is even better than pseudocode! Of course, I had to port the code to Java, which, for better or worse, forced me to really understand how the approach works.

It is worth noting that the approach actually does more than find lines. It finds line segments. This will be important in work described in a later step.

It is probably not necessary, but I'll emphasize that *all* of the work of finding line segments is implemented on the Raspberry Pi in Java.

LineFinder implementation

I created a class called **LineFinder** for the implementation. Page 6 of the paper includes table 2, which lists what I'll call the configuration parameters for the approach. I used the parameters, but the actual values depend on the physical system, so I had to play around with the parameters to produce what I felt were the best results given the LLv3 and my environment; I used the following:

- Minimum line segment length = 25 cm
- Minimum number of points in a segment = 8
- Number of points in a seed-segment = 4
- Distance threshold from a point to a line = 4 cm
- Distance threshold from a point to a point = 5 cm

LineFinder exposes two static methods:

- *findLines* —represents the implementation of the approach mentioned above that finds line segments
- *getLines* — returns the line segments; the description of the lines includes the line equation as well as the endpoints

I should mention that for a while, I experimented with finding other artifacts in a scan, in particular, corners. As with lines, research exists. I actually implemented one approach, but it proved to be far less reliable than **LineFinder** and I eventually stopped using it.

Using LineFinder

LineFinder works on the lidar scan report produced by **Lidar**, described above. Using **LineFinder** is quite simple.

```
configure LineFinder
[get a lidar scan report – see previous step]
LineFinder.findLines(using the scan report)
LineFinder.getLines
```

Now the lines/segments can be further processed. As with the raw scan, I found it convenient to plot the results. I could not figure out how to automate a spreadsheet to plot the lines along with the point cloud, so I created a

custom plotter program in JavaFX. The figure shows the raw scan and the derived lines. The point cloud is in green and the "real" lines found by **LineFinder** are in yellow. Note that the lines representing horizontal surfaces are pretty horizontal, and the lines representing vertical surfaces are pretty vertical. The biggest surprise is that the vertical surface around x=-60 (wood with large wormholes) actually resolved to a line.

For this particular situation, instead of a cloud of 361 points, we have just 9 line segments. Quite a bit of a "reduction of data"!

The collection of line segments provides a form of the actual pose. Without some ideal pose to compare against, however, there is no way to calculate the pose error.

Step 9: Localization: the Virtual Scan (continuous)

So, how does one find the ideal pose required to estimate the pose error? I struggled with this problem for quite a while, considering radio and infrared beacons, visual targets, etc. I looked at a number of research papers. I finally decided that a viable (and low cost) solution was to actually create a map of my house, or more accurately a map of the main space in my house in which the Autonomous Rover can roam. Given a map, I could do a *virtual scan* of the map, imitating a lidar scan, to produce a virtual equivalent of the real line segments. Comparing the real (actual) pose, represented by the real line segments, to the virtual (desired) pose, represented by the virtual line segments, results in the pose error.

The Map

Of course, nothing is as simple as it seems at first. The first figure shows the map I created, with a north orientation. At the bottom left, the intersection of the red arrows identifies the origin. The area is approximately 11.3 m high and 8.5 m wide.

The figure indicates the "vertical" surfaces in the map align with the compass direction north. While not exactly true, it is a useful simplification/fiction, because surface *orientation* is important for localization. Since there is no compass in the AR, the fact that "north" is not really true north doesn't matter.

The outside boundary in the figure represent fixed walls (really baseboards due to the height of the LLv3), doors (assumed shut), cabinets, shelves, appliances, and furniture. These surfaces don’t move.

Notice the two “islands” inside the exterior boundary. The left represents a true island with kitchen cabinets, pantry doors, and one piece of furniture; these surfaces don’t move. The right island represents a couch with an oddly shaped end table at each end. While the end tables are fixed, the couch is not.

But that is not all. There are additional furniture pieces, primarily tables and chairs, that are not represented. They move, or they are not orthogonal, or they are small enough to be seen only as “noise”, or some combination of the three. I’ll address the “noise” again later.

I described the map using a list of corners for each closed boundary. Thus, the map has three *sections*, one for the exterior boundary, a second for the left island, and a third for the right island. The corners are stored in a file for ease of manipulation, and use by multiple programs.

Reference Frames

In some research papers, a map of this sort is called the *physical ground truth*. Since it represents the entire world in which the AR exists, it establishes a *world reference frame*, or world coordinate system, abbreviated as *{W}*. I’ll use physical ground truth and {W} synonymously. The origin of {W} is the origin of the map, described above. The y axis of {W} is alway oriented "north".

Think about the lidar sensor (I use sensor because the same concepts apply to an ultrasonic device or similar range sensing devices) when doing a physical scan. In effect, the location and orientation of the sensor establishes another reference frame, called the *sensor reference frame*, or *sensor coordinate system*, abbreviated as *{S}*. The second figure shows a real scan with {S} identified. The y axis of {S} is alway oriented in the direction of forward travel of the rover, and always oriented with the long axis of the rover.

Clearly, to do a virtual scan that can be compared to a real scan, one must know the location of {S} in {W}, known as the *offset*.

Reference frame orientation

In the second figure, the sensor is oriented north like {W}. However, that is not always the case. The expectation is that AR will travel some combination of north, south, east, and west while following the path from the current pose to the target pose. Thus, it is also necessary to know the absolute orientation of the AR; the sensor inherits the same orientation.

The third figure helps explain. The top left part shows {S} oriented north, aligned with {W}. The top right part shows {S} oriented south, 180° from {W}. The bottom right part shows {S} oriented east, -90° from {W}. The bottom left part shows {S} oriented east, 90° from {W}. Note the {S} is used in the figure to disambiguate the orientation of {S}.

The figure highlights another important concept. The physical ground truth "seen" by the sensor depends not only on the location, but also the orientation.

Virtual scan terminology

To progress, I must introduce some fundamental terms and principles related to a virtual scan. The fourth figure shows a crude representation of the physical ground truth discussed earlier. To enable a virtual scan, the file containing corner coordinates for the surfaces gets processed to create an internal representation of {W}. The initial processing first produces a list of corners identified by section and position or index within that section. For example, the exterior boundary is section 0, with corners C0/0, C0/1, C0/2, C0/3.

Next the corners are processed to produce lines, or more accurately line segments, between the corners. Segments also have a section and index. For example, island 1 has lines L1/0, L1/1, L1/2, L1/3. Note that segments inherit their index from the first corner in the segment. Line segments are marked with an orientation and so can be treated as vectors. The vectors can be horizontal and point right (HR) or left (HL), or the vectors can be vertical and point up (VU) or down (VD).

As the vectors are defined, the corners are given a code that identifies the orientation of the two vectors that intersect at that corner; the corner code is derived from the orientation of the first or "incoming" vector and then the second or "outgoing" vector. For example, C0/1, C1/1, C2/1 all get the corner code HR_VU, and C0/2, C1/2, C2/2 all get the corner code VU_HL.

Virtual scan explained

Now consider that the lidar is positioned at the scan point indicated by SP in the fourth figure. The red lines indicate the 0° and 180° scan directions. The purple lines indicate some other critical scanning directions. We will do a visual scan, leveraging the ability to see the figure in totality.

Note that no corner or line below SP can be "seen" by the sensor. In this example, C0/0, C0/1, and L0/0 are invisible. Note also that no portion of a line below SP can be seen by the sensor. In this example, the bottom portion of L0/1 and L0/3 below the red line marking the SP are invisible. So far, all other corners and lines are potentially visible.

The virtual scan a 0° "sees" an intersection with L0/1, marked with a green star. When the scan continues counterclockwise between 0° and the first purple line, it sees L0/1. However, at the scan angle indicated by the purple line, the sensor encounters C2/1, blocking any further view of L0/1, at least for a while. There are important notions here. First, corner C2/1 is marked as a break, in that it impacts visibility of other artifacts. Second, the break creates an intersection (I1) with L0/1; now the portion of L0/1 between I0 and I1 is visible, but some or all of the rest of L0/1 is invisible.

As the scan continues counterclockwise it sees all of L2/0, C2/0, all of L2/3, and C2/3; all are marked visible. Note, however, that lines L2/0 and L2/3 occlude C0/2 and C2/2, making them invisible. In addition, that means

that L0/1 between I1 and C0/2 is invisible, and both L2/1 and L2/2 are invisible. C2/3 is another break; it causes an intersection with L0/2 (I2), and L0/2 between C0/2 and I2 is invisible.

As the scan continues from I2, it encounters C1/2, another break (marked visible); in this case, it causes an intersection with L0/2 (I3), and L0/2 between I2 and I3 is invisible, but some of L0/2 past I3 could be invisible.

As the scan continues, all of L1/1, C1/1, and all of L1/0 are seen and marked visible. Lines L1/1 and L1/0 occlude C0/3 and C1/3, making them invisible. In addition, that means that L0/2 between I3 and C0/3 is invisible, as is at least some portion of L1/3 (since C0/3 is invisible), as are L1/2 and L1/3 (since C1/3 is invisible).

As the scan continues C1/0 is marked visible, but it also is identified as a break, producing an intersection with L0/3 (I4). This means that L0/3 between C0/3 and I4 is invisible, but some portion of that line below I4 is visible. As the scan continues to 180° more of line L0/3 is seen. At 180° the intersection with L0/3 (I5) marks the limit of visibility, so L0/3 is visible between I4 and I5.

The fifth figure shows the final visible line segments from the virtual scan in translucent orange lines. Note that the original corners can be visible or invisible. Visible corners can be breaks that impact visibility. The original line segments can be visible or invisible. Visible segments can be wholly visible or partially visible. Partially visible segments can be "clipped" at either end or at both ends. In theory, it is possible for the same original segment to produce multiple visible "snippets".

Virtual scan algorithm

Breaks are of critical importance, and the first step in a virtual scan is finding breaks. An algorithmic manner is to examine the relationship between the nature of the corner, indicated by its corner code, and the intersection of the scan vector. The sixth figure helps explain. The black vectors represent the direction of the line segments creating a corner. The red vectors represent the scan vector from the scan point. Any vector that "bisects" a corner is a non-break (8 cases) and any scan that is "tangent" to a corner is a break (8 cases). For example, the leftmost case in the top row depicts an HL-VD corner where the angle of the scan vector is less than 90° (l90). Since the vector is tangent, that corner is a break. To the right is an HL-VD corner where the angle is greater than 90° (g90). Since the vector bisects, that corner is not a break.

The algorithm confirms the findings in the example above. C1/0, C2/1, C2/3, and C1/2 are breakpoints.

The next step is finding the visible corners and intersections. A corner is examined to see if is above the scan point or obscured from the scan point. The former is discussed above and simply means the y coordinate of the corner is greater than the y coordinate of the scan point. The latter means that the scan point, the corner and an obscuring corner are collinear on x or y axis, and the obscuring corner is between the scan point and the corner. If either situation exists, the corner is invisible.

Then things get a bit more complex. For a corner that could still be visible, one must loop through all line segments looking for obscuring line segments. The seventh figure shows a few examples. Consider C2/0. The scan vector intersects L2/1 and L0/1. The distance from SP to C2/0 is less than the distance to the intersections with the two lines, thus C2/0 is visible. Consider C2/1. The scan vector through the corner intersects L0/1. The distance from SP to C2/1 is less than the distance to the intersection with the lines, thus C2/1 is visible; further, since C2/1 is a breakpoint, the intersection is persisted as I1. Consider C2/2. The scan vector intersects L2/3, L2/2, and L0/2. Since the distance to the intersection with L2/3 is less than the distance to C2/2, C2/2 is invisible. Consider C0/2. The scan vector intersects L2/3 and L2/1. Since the distance to the intersection with L2/3 is less than the distance to C0/2, C0/2 is invisible. Finally consider when the scan angle is 0°; this is treated as a special case; the intersection gets calculated and persisted; the same happens for 180°; these two intersections are added as corners to facilitate further processing. After finding all the visual corners, they are sorted according to the section and index within a section.

Now one can find visible line segments. Finding line segments starts with visible corners and includes persisted intersections. A visible corner is the start of a segment, the end of a segment, or both. It is important to realize that the corners that define a line segment must exist within the same segment.

Corners are examined in pairs. When a pair of corners are sequential in the same segment the two corners form a visible line segment (note there is a special case where the last corner in a section and the first corner in that section form a line, e.g., C2/3 and C2/0 in the above example). If that condition is not satisfied, the corner is considered isolated and then one must look for an intersection that can form a line segment with the corner. The intersected line containing the intersection must have the same section and index as the corner; if so, the corner and the intersection form a line segment.

After all the visible corners are processed, including those matched against intersections, not all interactions are necessarily consumed; the most prominent example is when a line gets clipped at both ends, as does L0/2 in the above example. In this case, the two intersections are combined to form a line. However, there are cases where there is an isolated intersection. In that case, one must look at the intersected line. If the first corner in the line is visible, the last corner gets replaced by the intersection; if the last corner is visible, the first corner gets replaced by the intersection. After complete processing, the visible lines get sorted according to section and index.

ContinuousTruth implementation

The class **ContinuousTruth** implements the virtual scan described in the above discussion. It has the following methods:

- *initializeTruthAll* — sets up the internal representation of the physical ground truth (corners and line segments) for all four orientations (N,S,E,W)
- *virtualScan* — performs a *continuous* virtual scan based on a scan point and orientation; in effect, it implements the algorithms described above

Step 10: Localization: the Virtual Scan (incremental)

Now is the time to share a dirty little secret. The virtual scan described in the previous step is “perfect”, or “analog”, or *continuous*. That means if the angle from the scan point (SP) to C2/1 in the example above is 14.0362°, that is used in all calculations. However, in reality, the lidar subsystem drives the servo in roughly 0.5° increments. Thus the lidar scan is actually *incremental* in nature. Fundamentally, the servo may not produce an angle of exactly 14°, much less 14.0362°.

Looking at the first figure for a simpler example of a physical ground truth, consider the vector on the right from the scan point to C0/2. Trigonometry shows the angle is 75.62°. In the best case, the servo can move to roughly 75.5° or 76°.

The second figure shows the impact of the actual scan angle around the corner. In that figure, the green dot shows corner C0/2 and the red dots shows the intersections on the surrounding lines for various angles. In particular, at 75.5° the intersection is below the corner and so the real endpoint of line segment L0/1 would be at that intersection = (300, 496.67), not (300, 500). The intersection for 76° is on a different line (L0/2), but still close to the corner. This shows that (a) the continuous scan produces incorrect endpoints, and (b) one has to be very careful about choosing the correct angle for the virtual scan.

Perhaps as interesting are the intersections of 75° and 74.5°, which are at y= 483.21 and y=470.59 respectively. The distance between these real sample points are large enough to terminate the line finding algorithm early, causing shortened lines. This phenomenon occurs because of the acute angle formed by the scan vector and the line, approaching being parallel. Note how the distance between samples becomes much shorter with the horizontal line, since the scan vector and the line are approaching perpendicular. This demonstrated the need for detection early termination of line segments, based on the **LineFinder** configuration parameter "Distance threshold from a point to a point". Unfortunately, I've not yet added that to the implementation.

Looking again at the first figure, consider the vector on the left from the scan point to the break at C0/5. Simple trigonometry shows the angle is 112.62°. In the best case, the servo can move to 112.5° or 113°. The third figure shows the difference between the continuous and incremental intersections is small for 112.5°, but for 113° the intersection is on a completely different line near the break! Again, this shows that care must be taken when choosing the angle for a virtual scan. When a break occurs, one must adjust both the intersection caused by the break, and the endpoint of the line containing the break.

I have to admit that I recognized the incremental nature of reality some time after the continuous virtual scan was implemented. Given that the continuous virtual scan worked well, I did not want to perturb it, so I decided apply the incremental reality to the results of the ideal continuous scan. There are two key capabilities necessary, *round down* and *round up*. Round down takes an ideal angle from an endpoint of a virtual line segment found during a continuous scan, calculates the nearest real angle achievable by the lidar servo that is lower than the ideal angle, and calculates the intersection with the virtual line to produce a real endpoint. Round down would be used for the endpoint C0/2 in the example above. Round up does just the opposite.

IncrementalTruth implementation

The class **IncrementalTruth** performs the *incremental* scan as described above, using the line segments found in the continuous scan. It has two methods:

- *incrementalScan* — performs the incremental scan, and persists the lines internally
- *getIncrementalLines* — returns the incremental lines

The *incrementalScan* method uses round down or round up on both endpoints of line segment based on the segment orientation, according the following pseudocode:

```
HL:
        first corner: round up
        last corner: round down
HR:
        first corner: round down
        last corner: round up
VD:
        if line to right of scan point
                if first corner angle not 0 or 180
                        first corner: round down
                if last corner angle not 0 or 180
                        last corner: round up
        else
                if first corner angle not 0 or 180
                        first corner: round up
                if last corner angle not 0 or 180
                        last corner: round down
VU:
        if line to right of scan point
                if first corner angle not 0 or 180
                        first corner: round up
                if last corner angle not 0 or 180
                        last corner: round down
        else
                if first corner angle not 0 or 180
                        first corner: round down
                if last corner angle not 0 or 180
                        last corner: round up
```

GroundTruth implementation

For convenience, I encapsulated the existence of **ContinuousTruth** and **IncrementalTruth** in the class **GroundTruth**. The class has the following methods:

- *initTruthFromFile* — reads a physical ground truth map from a file, and calls **ContinuousTruth**.*initializeTruthAll*
- *virtualScan* — calls **ContinuousTruth**.*virtualScan* and then calls **IncrementalTruth**.*incrementalScan* and it filters the resulting incremental virtual scan to eliminate lines that are
    - inherently invisible because the angle is so acute that the length of the line is in effect negative
    - less than the minimum length set for the algorithm used to find real lines (currently 25 cm)
- *getVirtualLineModels* — provides a representation of the virtual line segment in a form that is convenient for comparison with real line segments; the method processes the line segments to
    - translate the endpoints to {S} from {W}
    - calculate the intercept in {S}; the intercept is the midpoint of the segment
    - sort according to scan order, based on the angle of the first endpoint
    - recognize the possibility of merged lines (see below)

During analysis of real lines produced by the lidar subsystem, I discovered an interesting and perplexing phenomenon. Depending on the pose of the lidar unit with respect to surfaces that

- are consecutive in the sorted list
- have the same orientation
- have the last endpoint of one surface and first endpoint of a second surface that are “close” together
- have intercepts that are “almost the same" (4 cm or less different using the current configuration)

can be *merged* during processing of a real scan to find real line segments. So far, I’ve seen the phenomenon only when scanning three particular surfaces on the eastern boundary of the physical ground truth, shown highlighted in the fourth figure. In some poses, these three surfaces resolve to three segments. In some poses, they resolve to two segments, one for the bottom surface and one for the top two surfaces, or, one segment for the bottom two

surfaces and one for the top surface. In some poses, they resolve to one segment that includes all three surfaces. To address this problem I look for situations meeting the criteria above in the virtual line segments and then add additional segments, as appropriate, to the end of the list.

As an example of a virtual scan, the fifth figure shows the plot of the raw data (green dots), the real lines found during the analysis the the data (yellow lines), and the virtual lines found during a virtual scan (red lines). With the virtual lines, you might now be able to recognize that the area portrayed is the northwest corner of the physical ground truth. In theory, the lidar unit was placed perfectly to take the real scan. In theory, the real scan should match the virtual scan really well.

The actual result is both encouraging and discouraging. Encouraging is that there is clearly some affinity between the real and virtual scans. Discouraging is that the overall affinity is not as strong as one might hope. The maximum distance in X and Y is less than 5 m, so the lidar ranges should be accurate for ranges to ±2.5 cm. The differences in the intercepts of the lines are quite a bit larger in some cases. The real scan appears to be looking at an "expanded" physical ground truth, since in all cases the real lines corresponding to the virtual lines are “outside” the virtual lines. I have not yet discovered the cause of this phenomenon.

Step 11: Localization: Comparing the Real and Virtual Scans

At this point, the list of virtual line segments is ready to be compared to the list of real line segments produced by the lidar unit to estimate the pose error. As shown in the previous step, apparently, even with theoretically no error in pose, comparison of the real and virtual scan could show some error. That said, for completeness I will describe the technique used to calculate the pose error.

Classifying real lines

The first phase in the comparison is to filter and process the real lines to create a form more amenable to comparison with virtual lines. First lines are classified as either roughly horizontal (angle < 10° or > 170°), roughly vertical (angle = 80°-100°), or other. Horizontal and vertical lines have an intercept calculated based on the midpoint of the line segment. Those classified as 'other' are discarded since they would represent surfaces not modeled.

Calculating the pose angular error

The next phase calculates the *average angular error*. In theory this should be the angular component of the pose error. The calculation is simple, but with a twist. The reported angle of every real line is compared to either 90° (for vertical lines) or 0° or 180° (for horizontal lines) to get an error contribution. Based on observations during testing, a weight factor influences the error contribution. The weight factor has two components. The first is based on the length of the line; lines longer than 50 cm get a higher weight (they should be more accurately represented). The second is based on the intercept; lines with an intercept between 100 cm and 500 cm get a higher weight (the lidar sensor is most accurate within those ranges). The individual contributions are summed and divided by the total number of contributions to produce the average angular error.

Unfortunately, testing shows that the average angular error is not very accurate, and in fact, the heading produced by the gyroscope, despite itself being suspect, is likely a better estimate. Even so, the average angular error supports a sort of "early failure" mechanism. The theory is that if the pose angular error is too large, something is really wrong, and it is not really possible to do a reasonable calculation of the pose translation error due to some simplifying assumptions in that calculation, much less continue with any movement.

Calculating the pose translation error

The last phase is finding the *translation* component of the pose error. The process is driven by real lines, as testing shows there are generally fewer real lines than virtual lines. Every virtual line gets compared to a real line of the same orientation (horizontal or vertical) to find the best match. After ensuring the candidate lines overlap (if not, it is very unlikely the two relate to the same surface), the following metrics for the real and the virtual candidate get compared:

- angle of first endpoint
- angle of last endpoint
- angle of midpoint
- intercept
- length

The differences in these metrics between the real and virtual candidates are calculated, persisted, and checked for being a new minimum. All the metric differences for the virtual candidates are *scored*.

A metric score is based on the magnitude of the metric difference. Different magnitudes are given different weight factors. As an example, for the intercept metric, if a virtual line difference from the real line is the minimum for all virtual lines, the score=0; if it is within 4 cm of the minimum, the score=1; if it is within 10 cm of the minimum, the score=2; if it is within 30 cm of the minimum, the score=3; otherwise the score=100.

Different metrics have different magnitudes and different weights. All the individual metric scores for a virtual line get added to create a combined score.

At this point, one can look thru all the combined scores to find the best match. In theory this means the lowest score. However, there are some additional conditions that must be investigated, based on experimental results:

- The last figure of the last step contains a real line on the left that starts at y=~160 and ends at y=~200. There is no corresponding virtual line; likely because it after the incremental scan it ended up less than 25 cm.
- It is possible for virtual lines to have the same combined score. I've been experimenting with surface types to break such ties. For example if the surface type of one virtual line is "good" and the other is "bad", the "good" surface wins.

I must admit that the matching approach is totally empirical, based on some observations during testing. It requires additional thought and tuning on both what metrics get used and the scoring mechanism. While the current approach seems to work reasonably well, it sometimes fails to match accurately.

Once the real and virtual line segments get matched, the intercept error for every matched pair gets calculated. The sum of the intercept errors for horizontal lines and the sum of the intercept errors for vertical lines get calculated and then divided by the number of horizontal pairs and vertical pairs respectively, to produce the average translation error in the direction of travel (Y) and the drift perpendicular to the direction of travel (X).

When calculating the average translation error, there is no weighting applied (more opportunity for tuning). Note that, in general, there may be no horizontal matches or no vertical matches to compare (never both, as far as I have seen), so it may not be possible to calculate one of the error components.

Estimator implementation

The class **Estimator** implements the heuristic approach described above to find the components of the pose error. It exposes the following methods:

- *findAvgAngularError* — examines the angles of the nearly vertical and nearly horizontal "real" lines derived from the lidar scan to estimate the angular error of the lidar sensor during the scan
- *matchRealToVirtual* — performs a virtual scan and then matches virtual lines to the nearly vertical and nearly horizontal "real" lines; it compares the intercepts of matched pairs to estimate the translation error of the lidar sensor during the scan

An example

Now we will look at an example. The figure shows the point cloud from the lidar scan (green dots), the real lines (in yellow), and the virtual lines (in red) for a pose after the AR traveled 50 cm north from the position in the previous example. It is easy to recognize features common to both scans. Due to the movement, the new rover pose for this scan is will not be perfect. It is not likely the rover moved exactly 50 cm or ended up oriented exactly north.

The matched lines are marked in the figure. The match is actually pretty good. That said, due lack of foresight, the plotter program set the bounds of the plot based on the real scan; as a result, some virtual lines are clipped, and some are clipped completely. Some more observations:

- The diagonal real line with intercept around x=425 represents a large chair that blocked the lidar from seeing the surfaces represented by the virtual lines, so no harm done.
- The nearly horizontal line with intercept around y=210 represents a surface that I'm actually shocked resolved to a line. There is certainly no model for the surface in the physical ground truth, so again, no harm done.
- The virtual line with intercept around x=300 is definitely obscured by trash or "noise" (chairs and a table) and so it is no surprise that roughly the have is not detected by the lidar.
- The almost vertical real line with intercept around x=75 is an artifact of "noise" (chairs and a table). Thus there is no virtual line which could match it.
- The three nearly vertical lines match their counterpart virtual lines. That, of course, is as hoped.
- The horizontal virtual line with intercept around y=90 does not have a matching real line. While I have not investigated to confirm, I'm sure the potential real line resolved to be less than 25 cm long, and thus was not generated.

The north-most horizontal virtual line is worth additional consideration. The spotty rendition of the surface represented by the long virtual line is due to the trash (chairs and a table) obscuring the surface. As a consequence the real scan resolves two shortened lines, one with a midpoint around x=75 and another with a midpoint around x=-20. Fortunately, both get matched to the correct virtual line.

In this example, using the matched pairs, the calculated average angular error is 0.79°. The translation error in the direction of travel (Y) is 6.32 cm and the drift (X) is -1.81 cm. Thus, the pose error

$[e_X, e_Y, e_\theta] = [-1.81, 6.32, 0.79°]$

This seems consistent with a visual inspection of the figure. That said, in reality, given that the rover was in theory placed perfect prior to motion, and in fact, the move of 50 cm produced a translation error for $x < 1$ cm, a translation error for $y < 2$ cm, and a heading error (according the the gyro) $\approx -0.1°$, the calculations are clearly wrong. However, they are certainly good enough to determine that the AR is not wildly out of place, and that path following can continue. More tuning!

Step 12: Localization: Conclusion

The Localization capability described clearly works. That said

- The lidar scan does not appear as accurate as one should expect, given the device specifications, despite months of effort to make that happen. I'm not sure how to improve it, or even how to begin investigating.
- Finding "real" line segments works quite nicely. However, given that the input scan is suspect, the real lines segments are suspect.
- The virtual scan appears to work perfectly. That said, there are additional ideas that could improve it, e.g., identifying where virtual line segments terminate early due to acute scan angles.
- The matching algorithm is firmly based in heuristic and empirical approaches that require additional investigation, tuning, and testing.

So, Localization works, but not as well as intended. It actually appears to be less accurate than the odometry and gyroscope heading from Locomotion, even with the known limitations in that capability. In its current state, Localization can only be used to detect significant errors that should terminate continuing on the path. That, however, is not a bad thing!

Step 13: Navigation: Introduction

Navigation is the way the Autonomous Rover determines the optimal path from its current position to a target position. I did a fair amount of research on navigation, or path planning, approaches that could leverage and build upon the work already completed for localization. I chose a well-known approach called *wavefront expansion*, or simply *wavefront*. Many research papers, books, and web pages address the approach. For example, see this, this, this, this, and this.

Wavefront is based on partitioning a *map*, or *configuration space*, into cells (generally fixed in size). Cells can be marked a *free* or *unfree*. An unfree cell can be considered an obstacle to be avoided. This pre-defined approach to describing the environment seemed attractive in that it allows me to deal more easily with objects in the environment that are not easily described using the physical ground truth map using in localization.

Step 14: Navigation: Melding Wavefront and Localization

For Localization, remember that to define the environment in which the AR can travel, I used a map to describe the physical ground truth. It describes all the fixed orthogonal surfaces in the environment, and a few (mostly) orthogonal surfaces that are not fixed. I also mentioned that my house contains “trash” or "junk" that is not fixed, like tables, chairs, lamps, etc., the map cannot adequately describe. The first figure shows the approximate location of such junk using cloud symbols. In more technical terms, the junk represents *obstacles*. One of the great aspects of wavefront is that obstacles can be designated by unfree cells.

As stated earlier, wavefront divides a map or configuration space into a grid of cells. In most discussions of wavefront, the configuration space is a rectangular grid. Given this, I had to address two key issues:

- what is the extent of the configuration space (the rectangular grid)
- what is the size of a grid cell

Configuration space extent

An obvious answer to the extent issue is “the whole thing”, meaning the entire physical ground truth, {W}. I recognized, however, if the grid overlaid the entire physical ground truth, nearly the entire north quarter of grid would be unfree. That seemed wasteful. I realized I could define the extent of the grid based on the simpler goal of demonstrating autonomy rather than truly allowing the AR to travel anywhere in the environment.

In effect, the configuration space defines a *logical ground truth*. It is clearly related to and derived from the physical ground truth, but not necessarily equal to it. The physical ground truth is used for localization and locomotion, while the logical ground truth is used for navigation.

The second figure, using a red rectangle, shows an approximation of the logical ground truth with respect to the physical ground truth. Note that the entire north end of physical ground truth does not exist as far as the AR is concerned, since it is not defined in the logical ground truth.

The logical ground truth establishes its own reference frame, called *{T}*. There obviously must be a geometric relationship between {W} and {T}, in this case a simple translation. Assume the origin O_T of {T} in {W} is $O_{TW} = (\boldsymbol{x}_W,\boldsymbol{y}_W)$ which is a constant. Thus, if a point in {T} is $P_T=(x_T,y_T)$, P_T in {W} is

$$P_{TW} = (x_T + \boldsymbol{x}_W, y_T + \boldsymbol{y}_W)$$

Grid cell size

Apparently, in general, the wavefront grid cell size is somewhat arbitrary. This book chapter (see page 240) mentions an implementation with a cell size of 2 cm, but there are very few autonomous robots of that size. Another paper states “the wavefront propagation algorithm assumes that the robot is less than a single grid cell wide”. I decided to go with the latter assertion. That means an examination of the physical structure of the AR.

The salient aspects of that physical structure derive from the Scout robot kit. The third figure shows that the chassis body (in blue) is 19 cm wide and 27 cm long (in the direction of forward travel). However, accounting for the wheels and tires (in black) mounted on the motor shafts (in gray) the overall width is 30 cm and overall length is 33.5 cm.

However, there are other important considerations. The Scout based rover can go forward, or can rotate around its centroid (the red X). The large red circle shows the extent of the rover assuming perfect rotation; the diameter of that circle is about 45 cm. The description of Locomotion mentioned that rotation is not perfect; there is an

associated translation along both axes. Thus it is prudent to make the cell size a bit bigger than 45 cm. That led the the choice of a cell size = 50 cm x 50 cm, shown by the green square.

The final logical ground truth

The combination of logical ground truth extent independence from the physical ground truth and the ability to define obstacles wherever needed led to the actual {T} shown in the fourth figure as a red grid. Note that it is rectangular, not square, and is 11 cells wide by 12 cells high. Obstacles, or unfree cells, are shown in yellow.

On the whole {T} seems reasonable. It is possible to travel from any free cell in {T} to any other free cell in {T}. On the other hand, there are several "corridors" that are only one cell wide. This results in a couple of concerns. First, there is not much room for errors in locomotion; that said, testing so far has shown this not to be as problematic as feared. Second, there are a large number of cells in which the distance between the lidar sensor and the scanned surfaces will be less than 100 cm, the minimum range for linear behavior. This could lead to significant problems with localization; fortunately, while this has been a bit problematic during testing, the impact has been slight.

More on reference frames

One can view {T} as a reference frame with an 11 x 12 grid of 50x50 cm cells. A cell has an origin in {T} $O_C = (c_X, c_Y)$. The location of O_C in {T} is

$$O_{CT} = (c_X * 50, c_Y * 50) \text{ with } c_X \in [0,10] \ \& \ c_Y \in [0,11]$$

By substitution from above, the location of the cell origin in {W} is

$$O_{CW} = ((c_X * 50) + \boldsymbol{x}_W, (c_Y * 50) + \boldsymbol{y}_W)$$

Any point in a cell can be represented by

$$P_C = (x_C, y_C) \text{ with } x_C, y_C \in [0,49]$$

Thus any point in a cell can be represented in {W} as

$$P_{CW} = ([x_C + (c_X * 50) + \boldsymbol{x}_W], [y_C + (_{cY} * 50) + \boldsymbol{y}_W]) \text{ with } c_X \in [0,10],\ c_Y \in [0,11], \text{ and } x_C, y_C \in [0,49]$$

In theory, with {T} established, the AR always resides in one of the free cells in {T}. In fact, the centroid of the rover is located at the centroid of the cell. The rover centroid is important because it is the axis of rotation; thus the rover has its own *rover reference frame {R}* with the origin O_R at the rover centroid. The origin O_{RC} of {R} in a cell is at the centroid of the cell = (25,25). Thus substituting in the previous equation, the origin of {R} in {W}

$$O_{RW} = ([25 + (c_X * 50) + \boldsymbol{x}_W], [25 + (c_Y * 50) + \boldsymbol{y}_W]) \text{ with } c_X \in [0,10],\ c_Y \in [0,11]$$

Any point in {R}, $P_R = (x_R, y_R)$, can now be represented in {W} by

$$P_{RW} = ([x_R + 25 + (c_X * 50) + \boldsymbol{x}_W], [y_R + 25 + (c_Y * 50) + \boldsymbol{y}_W]) \text{ with } c_X \in [0,10],\ c_Y \in [0,11]$$

The origin of {R} is important for locomotion and for navigation, but in theory, not so much for localization, which only cares about the sensor reference frame {S}. As a result, we must now again look at the Scout geometry to determine the relationship between {S} and {R}. The fifth figure shows a slightly simpler version of the previous figure of the Scout. As expected, {R} is at the centroid of the rover. Based on statements from tech support folks about the structure of the lidar sensor, and my measurements of the lidar sensor mounting on the

rover, the {S} origin is at (0, 9.5) in {R}. With that knowledge, any point in {S}, $P_S = (x_S, y_S)$, can now be represented in {W} by

$$P_{SW} = ([x_R + 25 + (c_X * 50) + \boldsymbol{x}_W], [y_R + 34.5 + (c_Y * 50) + \boldsymbol{y}_W]) \text{ with } c_X \in [0,10], c_Y \in [0,11]$$

Sadly, all of the above equations apply only when the rover, and thus the lidar unit, are oriented north. As described earlier, the rover, and thus the lidar unit, can also be oriented south, east, or west, as shown in the sixth figure. Suffice it to say that this is very important for localization and navigation. The exercise of deriving the equations for the other orientations is left as an exercise for the reader.

Step 15: Navigation: Pathfinding With Wavefront

Given the abundant references that describe wavefront, I won't describe how it works. I will talk about how I initially implemented it, adapted it for the AR, and improved it (I think).

I must mention that even with all the references, I attempted to find an implementation to use as a starting point. I found one on YouTube, included in a wonderful set of videos that cover a specific variant of wavefront called A* (not the Arduino used in Locomotion). The first in the set does a great job of explaining the approach. The second and third in the set show the implementation. The rest deal with optimizations that were not compelling, so my implementation stopped with the third video.

My initial implementation was simply a port of the code in the videos to Java. There are three classes for the initial implementation of A*:

- **Grid** represents the grid of nodes or cells in the A* algorithm
- **Node** is an element of the grid, with fields indicating the location in the grid, free or unfree, and the costs
- **PathFinder** finds a path from a current node (cell in {T}) to a target node (cell in {T}) using the A* algorithm

Grid Implementation

For the AR, the **Grid** class represents an implementation of the A* algorithm. The implementation is adapted from the videos mentioned above. In addition to the private methods supporting A*, the class has the following methods:

initializeGrid — creates a grid of the proper size, populates the grid with **Node** instances, and marks the instances free or unfree based on information from the class **LogicalGroundTruth**

LogicalGroundTruth implementation

LogicalGroundTruth represents the logical ground truth ({T}), a logical version of the physical ground truth ({W}), to support the wavefront path finding approach. The class holds information about the size of {T}, the cell size, the origin of {T} in {W}, and finally a list of cells that are considered obstacles; this list corresponds to the yellow shaded cells in {T} shown in a figure in the previous step. It exposes the following methods:

- *findIdealScanPoint* — calculates the origin of the lidar sensor reference frame ({S}) in {W}, assuming the rover centroid is at the centroid of a cell in {T}
- *findTCellInW* — calculates the {W} coordinates for the origin corner of a cell or the upper right corner of a cell

PathFinder implementation

PathFinder has the following methods:

- *initialize* — initializes **Grid** by calling **Grid**.*initializeGrid*
- *findPath* — finds the path from one cell in {T} to another cell in {T} using the A* algorithm, just like the video; it calls A* derived methods defined in **Grid** to do so

Initial testing using the configuration space used in the video produced the same results. It was exciting to see it worked.

Step 16: Navigation: Adapting and Improving PathFinder

If you pay close attention to the videos mentioned, you will notice the implementation assumes diagonal moves (in increments of 45°) are possible. The Locomotion component constrains the rover to ±90° rotations only. So I had to modify the algorithm to allow only orthogonal moves.

With diagonal moves, the algorithm must examine all 8 neighbor nodes or cells. The cost of moving orthogonally is (1 x 10 = 10), and the cost of moving diagonally is (√2 x 10 ≈14), as shown on the left of the first figure. I basically limited examination to the orthogonal neighbors, so the cost of a move is always (1 x 10 = 10) as shown on the right of the figure. This change impacted the algorithm methods *getNeighbors* and *getDistance*.

With this adaptation in place, and with the current cell [0, 10] and the target cell [8, 1] in {T}, **PathFinder**.*findPath* produces the following path:

[0, 10] ➔ [0, 11] ➔ [1, 11] ➔ [2, 11] ➔ [3, 11] ➔ [4, 11] ➔ [4, 10] ➔

[4, 9] ➔ [4, 8] ➔ [4, 7] ➔ [4, 6] ➔ [4, 5] ➔ [4, 4] ➔ [4, 3] ➔ [4, 2] ➔

[4, 1] ➔ [5, 1] ➔ [6, 1] ➔ [7, 1] ➔ [8, 1]

Compressing the path

Note that the path above, found by the basic A* adapted for orthogonal travel, consists of a series of one-cell-long sub-paths. At least in the case of the Autonomous Rover, that is inefficient. I decided to compress the path by finding the maximum sub-paths that go in a single direction. The approach is simple. The pseudocode for the private *compressPath* method, starting with the first cell (always the current cell) is

```
start a new sub-path with the first (current) cell and second cell
find the direction of travel between the two
for rest of path
        find the direction of travel to the next cell
        if direction changes
                end sub-path
                start new subpath with new direction
```

Using compression, the path between [0, 10] and [8, 1] in {T} becomes:

[0, 10] ➔ [0, 11] ➔ [4, 11] ➔ [4, 1] ➔ [8, 1]

Clearly, compression works nicely to produce a more efficient path.

Optimizing the path

During testing, I found that my implementation could produce a sub-optimal path, even after compression. While not completely certain, I think the problem is related to the removal of diagonal moves; however, it could be due to an obscure bug in my implementation of the basic algorithm. But, it could also be due to a different understanding of "optimal", as A* appears to be designed to "hug" any obstacles.

I have not encountered a sub-optimal path in the actual {T}, so I'll have to demonstrate the problem with a simpler example. Consider the {T} in the second figure. The compressed path produced for the current cell = [4, 4] and the target cell = [7, 1] is shown in orange. For the AR, the path results in a total of 4 forward moves and 3 rotations; that is 7 total movements. It is always better to minimize movements, especially turns. An alternative path with fewer movements is shown in green; it produces a total of 3 forward moves and 2 rotations; that is 5 total moments — much better!

The “magic” for detecting an optimization is finding three moves (between 4 cells) where the moves have the pattern orientation 1 — orientation 2 — orientation 1. If the pattern is found, the proper correction (cell replacement) depends on the orientation of the third move, and whether the path cell to be replaced is free or not. The pseudocode for an internal *optimizePath* method follows:

```
for all cells in path, starting at 4th cell
        orientation 1 = orientation of move from cell(n-3) to cell(n-2)
        orientation 2 = orientation of move from cell(n-2) to cell(n-1)
        orientation 3 = orientation of move from cell(n-1) to cell(n)
        if ((orientation 1 equals orientation 3) and (orientation 2 not equals
                orientation 3))
                replaceIndex = 0
                if (orientation 3 equals N)
                        cellX = cell(n-1)
                        cellY = cell(n-3)
                        replaceIndex = n-2
                else if (orientation 3 equals S)
                        cellX = cell(n-2)
                        cellY = cell(n)
                        replaceIndex = n-1
                else if (orientation 3 equals W)
                        cellX = cell(n)
                        cellY = cell(n-2)
                        replaceIndex = n-1
                else # orientation E
                        cellX = cell(n-3)
                        cellY = cell(n-1)
                        replaceIndex = n-2

                if replaceIndex not equals 0
                        replacement cell location = (x = cellX.x, y = cellY.y)
                        if sub-paths to and from replacement cell are open/free
                                set the replacement cell location at the replaceIndex in path
                        advance loop pointer by 3
```

It turns out that in some “corner case” situations, there can be more than one spot in an original path that can be optimized. In some cases, by shear luck, one pass through the optimizer catches multiple spots, generally if they are widely separated. Maximum optimization, however, would require repeated invocations of *optimizePath* until no more optimizations occurred. I have not yet implemented multiple passes. One more point. After optimization, it is sometimes necessary to again compress the resulting optimized path using *compressPath*.

PathFinder implementation revisited

Given the above improvements, the pseudocode for the *findPath* method becomes, in effect:

```
initialPath = find initial path (orthogonal version)
compressedPath = compressPath(initialPath)
optimizedPath = optimizePath(compressedPath)
finalPath = compressPath(optimizedPath)
```

Step 17: Navigation: Producing Motions From a Path

PathFinder produces a path that is just a sequence of cell locations in {T}. The optimized and compressed path found for the example in the previous step is

[0, 10] ➔ [0, 11] ➔ [4, 11] ➔ [4, 1] ➔ [8, 1]

The sequence must be transformed into a series of motions compatible with Locomotion's rover described earlier. To move from the current cell to the target cell, the rover can only perform two forms of motion:

- forward a distance of N cm in one of four compass headings (N, S, E, W)
- rotate 90° CW or CCW to orient the rover in the proper heading for the next forward motion

The distance and the proper heading for forward motion from path cell n to path cell n+1 derives from the coordinates of the two consecutive cells. The rotation angle necessary to reorient from one heading to the next derives from the coordinates of cells n-1, n, n+1. Note that path compression assures the orientation of n-1 to n and n to n+1 are different.

It is worth noting that the series of motions will always start with a forward motion followed by a rotation motion. The very last motion is always a forward motion.

Motion implementation

Class **Motion** describes the distance and type of a motion. Two subclasses exist. **Forward** adds an orientation field; **Rotate** adds nothing.

MotionFinder implementation

Class **MotionFinder** has the following methods:

- *determineMoveOrientation* — determines the orientation (N,S,E,W) of a forward move from one cell to another; this is trivial, and is based on the cell coordinates; for example, assume the first cell is [J, K] and the second cell is [J, L]; the move is along the N-S axis; if K*getMotionsFromPath* — produces the motions required to allow the rover to follow the path identified by **PathFinder**

The pseudocode for *getMotionsFromPath*:

```
for all cells in path, starting at 3rd cell
        orientation 1 = orientation of move from cell(n-2) to cell(n-1)
        orientation 2 = orientation of move from cell(n-1) to cell(n)
        add to motions: forward with orientation 1 and distance from cell(n-2) to cell (n-1)

        if (orientation 1 equals N)
                if (orientation 2 equals W)
                        rotate distance = 90
                else # must be E
                        rotate distance = -90
        else if (orientation 1 equals S)
                if (orientation 2 equals W)
                        rotate distance = -90
                else # must be E
                        rotate distance = 90
        else if (orientation 1 equals W)
                if (orientation 2 equals N)
                        rotate distance = -90
                else # must be S
                        rotate distance = 90
        else # orientation E
```

```
            if (orientation 2 equals N)
                    rotate distance = 90
            else # must be S
                    rotate distance = -90
    add to motions: rotate with rotate distance from above
```

From the path above, **MotionFinder** produces the following series of motions:

1. FORWARD: 50 cm going North
2. ROTATE: -90
3. FORWARD: 200 cm going East
4. ROTATE: -90
5. FORWARD: 500 cm going South
6. ROTATE: 90
7. FORWARD: 200 cm going East

Step 18: Navigation: Checking the Safety of a Motion

It is great to have an optimal path, and the motions required to move along that path. One must remember, however, that the path is derived from a logical, not physical perspective. Thus, it is conceivable that the physical environment contains obstacles not described in the logical ground truth. Such unexpected obstacles could be a human, a pet, or maybe a misplaced package. Thus, as a safety measure, there should be a check for anything in the path of a forward motion prior to executing that motion. Rotations are not checked, as they theoretically don't leave the boundary of a cell already occupied by the rover.

How can one check for unexpected obstacles? The expectation is that after every motion, either forward or rotate, AR performs localization, based on a lidar scan. That scan can be used to detect unexpected obstacles.

A forward motion in {T} in theory occurs in a rectangle one cell wide and one or more cells long. A reasonable assumption is that any obstacle in that rectangle would show up in the lidar scan. Consider the first figure. It shows the AR about to move on a path one cell (50 cm) long. In an ideal world, one would check for any lidar samples within the boundaries of that single cell defined by the lower left corner S and the upper left corner E. Any samples within that rectangle indicate unexpected obstacles.

The world is not ideal. Testing showed that when the LLv3 is close (less than ~30 cm) to a surface to the left or right, noise in the range samples can cause false obstacle detection. I had to apply a noise factor (nf) to S and E to avoid false positives. So the boundary within which samples indicate obstacles is defined by S' and E'. Notice that the distance between the rover tires and cell boundary is 10 cm. I currently use nf=5.

Per the discussion at the end of step 1, it is necessary to recognize that because {W} and {T} are always north oriented, but the rover can orient N,S,E,W, translation must be applied S' and E' to properly check the lidar scan (in {S}) for obstacles.

PathChecker implementation
I created a class **PathChecker** with a single method *isPathClear* to check the path. The method has parameters: the starting cell in {T}, the ending cell in {T}, the scan point in {W}, and of course, the results of the lidar scan in {S}.

S derives from the lower left corner of the starting cell and E derives from the upper right corner of the ending cell. Both S and E are in {W}. If the motion orientation is not north, the S and E have to adjusted to a "physical" S and E to align with the lidar scan in {S}; these are termed SP and EP. The second figure shows the relationships. In effect, the coordinates of SP and EP have to be derived from proper translation from S and E.

The pseudocode for *isPathClear*:

```
calculate orientation of motion using starting cell and ending cell
find S and E in {W}
if (orientation equals N)
        transform S and E to {S} using scan point to produce SP and EP
else if (orientation equals S)
        transform S, E and scan point to a south orientation
        transform S and E to {S} and using scan point and coordinate swapping to
                produce SP and EP
else if (orientation equals W)
        transform S, E and scan point to a west orientation
        transform S and E to {S} and using scan point and coordinate swapping to
                produce SP and EP
else # orientation E
        transform S, E and scan point to an east orientation
        transform S and E to {S} and using scan point and coordinate swapping to
                produce SP and EP
```

```
adjust SP and EP to account for the noise factor to produce S’ and E’
for all samples in lidar scan
        if (sample is within the rectangle defined by S’ and E’)
                indicate obstacle found
return true if obstacle found
```

Step 19: Navigation: Conclusion

The Navigation capability described works quite well.

- It finds an optimal path from a starting location to a target location. This works perfectly.
- It translates that path into a set of executable motions. This works perfectly.
- It checks for obstacles in the subpath of an anticipated forward motion. This is currently not perfect, in that it depends on "noisy" data from a lidar scan.

.

Step 20: An Autonomous Rover: Summary of Capabilities

This step summarizes the major components that provide the capabilities described in detail in earlier steps, and describes how an integration, like the Autonomous Rover, uses them. Recognize that some of the components are simply initialized by the AR and are then used "under the covers" by other components; thus some of the key methods for implementing the capabilities are not used directly by the AR and don't appear below.

Localization

class **Lidar**

Represents the physical LIDAR (or lidar) subsystem, with a lidar sensor and a servo. It exposes the following methods:

- *getScanReport* — initiates a lidar scan, waits for completion, and processes raw data into a consumable report
- *startSensorWarmup* — initiates calibration of the lidar sensor to support accurate measurements
- *isWarmupRunning* — checks if sensor warmup has completed
- *terminate* — terminates the thread in which the class instance runs

class **ArtifactFinder**

Analyzes a lidar report to find line segments representing surfaces in the environment. It exposes the following methods:

- *initialize* — initializes the class to use the right parameters for the specific lidar device and environment characteristics
- *findArtifacts* — analyzes the lidar report and finds "real" line segments representing surfaces in the environment

class **GroundTruth**

Represents the physical ground truth, {W}, describing the environment, to provide support for virtual scans. It exposes the following methods:

- *initTruthFromFile* — reads the description of {W} from a file and produces an internal representation of {W} to support virtual scans from any orientation

class **Estimator**

Analyzes "real" lines derived from the lidar scan to estimate the angular and translation errors in the actual pose of the rover at the time of a lidar scan. It exposes the following methods:

- *findAvgAngularError* — examines the angles of the nearly vertical and nearly horizontal "real" lines derived from the lidar scan to estimate the angular error of the lidar sensor during the scan
- *matchRealToVirtual* — performs a virtual scan and then matches virtual lines to the nearly vertical and nearly horizontal "real" lines; it compares the intercepts of matched pairs to estimate the translation error of the lidar sensor during the scan

Navigation

class **LogicalGroundTruth**

Represents the logical ground truth, {T}, a logical subset of {W}, to support the wavefront path finding approach. {T} can be considered a grid of cells. It exposes the following methods:

- *findIdealScanPoint* — calculates the position of the lidar sensor reference frame, {S} in {W}, assuming the rover centroid is at the centroid of a cell in {T}

class **PathFinder**

Finds an optimal path from one cell in {T} to another cell in {T} using a wavefront approach. It exposes the following methods:

- *initialize* — initializes the class with knowledge of {T} and the structures needed to run the the A* algorithm
- *findPath* — finds the optimal path from one cell in {T} to another cell in {T}

class **MotionFinder**

Translates a path, a series of {T} cell locations, to a series of motions executable by class **Rover** (see below). It exposes the following methods:

- *getMotionsFromPath* — processes the path into a set of forward motions in the correct orientation, alternating with a set of rotations to establish the correct orientation for a forward motion

class **PathCheck**

Examines the proper portion of a lidar report to determine if unexpected obstacles exist in the path of an upcoming forward motion. It exposes the following methods:

- *isPathClear* — examines a rectangular area in a scan report to determine if any lidar samples, representing unexpected obstacles, exist within that rectangle

Locomotion

class **Rover**

Represents the ability to execute motions. It exposes the following methods:

- *execForward* — move forward a given distance (in cm)
- *execRotate* — rotate a given distance (in degrees)
- *awaitMotionCompletion* — wait until a motion completes
- *getMotionReport* — provides encoder counts and heading
- *terminate* — terminates the thread in which the class instance runs

class **Gyro**

Represents the physical gyroscope. It exposes the following methods:

- *setZeroOffset* — in effect, calibrates the gyroscope to support accurate measurements
- *terminate* — terminates the thread in which the class instance runs

Step 21: An Autonomous Rover: Implementation

The form of the Automated Rover described here is the minimal form that could be constructed using the capabilities. That said, the implementation does use all the capabilities and demonstrates they function properly, if not always to the level of accuracy desired.

The pseudocode below describes just the use of the components implementing the capabilities to achieve its task. In the real implementation, I include points where it waits for me to indicate it is OK proceed, for example, before any motion. The real implementation also has debug code that either writes to a file or to the terminal. I will not include either of these aspects of the actual implementation in the pseudocode.

There are certain initial conditions that are expected by the implementation:

- a current cell and target cell are hard coded, and both are free of known obstacles
- the rover is position in the ideal location in the current cell; that means the centroid of the rover is on the centroid of the cell and orientation is correct for the initial forward movement

The pseudocode for the current form of the Autonomous Robot:

```
start the threads for: Gyro, Lidar, Rover
ArtifactFinder.initialize
PathFinder.initialize
GroundTruth.initTruthFromFile

Lidar.startSensorWarmup
path = PathFinder.findPath from current cell to target cell
motions = MotionFinder.getMotionsFromPath(path)
while Lidar.isWarmupRunning
        do nothing
realReport = Lidar.getScanReport

for motion in motions
        if motion is Forward
                get fromCell and toCell for motion
                get orientation for motion
                scanPoint = LogicalGroundTruth.findIdealScanPoint(fromCell,
                        orientation)
                if PathChecker.isPathClear(fromCell, toCell, scanPoint, realReport)
                        continue
                else
                        abort
                Gyro.setZeroOffset
                Rover.execForward(motion.distance)
        else # Rotate
                Gyro.setZeroOffset
                Rover.execRotate(motion.distance)
                orientation = calculated value after rotation

        Rover.awaitMotionCompletion
        motionStatus = Rover.getMotionReport

        scanPoint = LogicalGroundTruth.findIdealScanPoint(toCell, orientation)
        realReport = Lidar.getScanReport
        ArtifactFinder.findArtifacts(realReport)
        angularError = Estimator.findAvgAngularError # uses real lines
        if angularError OK
                continue
        else
                abort
        translationError = Estimator.matchRealToVirtual(scanPoint, orientation)
        if translationError OK
```

```
        continue
else
        abort
```

Step 22: An Autonomous Rover: Test Results

This step is an analysis of an actual test of the Autonomous Rover as it moves from a start location to a target location. It includes plots of what the lidar sensor "sees" during a lidar scan, as well as the results of matching the lidar scan and the corresponding virtual scan.

The first figure shows the physical ground truth, {W}. The logical ground truth, (T}, with obstacles, overlays {W}. **S** in the figure indicates the start cell [0,10] for the AR, and **T** indicates the target cell [8,1].

The optimized path from **S** to **T** calculated by the Navigation capability is

[0, 10] ➔ [0, 11] ➔ [4, 11] ➔ [4, 1] ➔ [8, 1]

The rover must be placed in cell [0,10] with the rover centroid over the cell centroid and facing north. Running the AR described in the previous step results in individual motions as the AR moves along the sub-paths in the path. The motions derived from the path are:

```
FORWARD: 50 cm going North
ROTATE: -90
FORWARD: 200 cm going East
ROTATE: -90
FORWARD: 500 cm going South
ROTATE: 90
FORWARD: 200 cm going East
```

The attached video (second figure) shows the AR moving along the path. I apologize for the snippet approach, but because of the nature of the path around obstacles, it was impossible to position myself to get decent videos without changing positions.

The AR starts with a lidar scan. The results are shown in plot 610 (third figure). The individual lidar ranges are shown with green dots. The "real" lines derived from analysis of the lidar scan are shown in yellow. The virtual lines derived from a virtual scan are shown in red. Since the rover should be positioned "perfectly", one would expect the real and virtual lines to align better. **Estimator** calculates the pose error

$[e_X, e_Y, e_\theta] = [-0.8, 7.2, 0.95°]$

Again, since there has been no movement, there should be no error. The error is concerning, but not large enough to abort.

Next is the forward motion of 50 cm north. Plot 611 shows the results. Again, the alignment between real and virtual lines is disappointing. The pose error

$[e_X, e_Y, e_\theta] = [-1.8, 5.5, -0.33°]$

In this case there was movement, but the translation error should be < ±2 cm in the direction of travel (Y) and < ±1 cm in X, and the angular error should be < ±0.1° (the gyroscope reported -0.05°). Nevertheless, the error is not large enough to abort.

Next is the rotate motion to east. Plot 612 shows the results. The alignment between real and virtual lines is better. The pose error

$[e_X, e_Y, e_\theta] = [-8.4, 1.5, -2.2°]$

Again there was movement, but the translation error should

angular error should be < ±0.3° (the gyroscope reported 0.3°) and the . The error is concerning, but is not large enough to abort.

Next is the forward motion of 200 cm east. Plot 613 shows the results. The alignment between real and virtual lines is again decent. The pose error

$[e_X, e_Y, e_\theta] = [-3.7,-0.9, -1.7°]$

Again there was movement, but the translation error should be < ±2 cm in Y (and it is!) and < ±2 cm in X, and the angular error should be < ±0.2° (the gyroscope reported 0.03°) . The error is not large enough to abort.

Next is the rotate motion to south. Plot 614 shows the results. The alignment between real and virtual lines is again decent. The pose error

$[e_X, e_Y, e_\theta] = [0.4, 2.6, -0.6°]$

That is pretty good. Again there was movement, but the translation error should be < ±2 cm in both axes (and it is for X!) and the angular error should be < ±0.3° (the gyroscope reported 0.2°). The error is not large enough to abort.

Next is the forward motion of 500 cm south. Plot 615 shows the results. The alignment between real and virtual lines is again somewhat disappointing. The pose error

$[e_X, e_Y, e_\theta] = [5.5, -0.4, 0.5°]$

Again there was movement, but the translation error should be < ±2 cm in Y (and it is!) and < ±3 cm in X, and the angular error should be < ±0.2° (the gyroscope reported 0.07°). The error is not large enough to abort.

Next is the rotate motion to east. Plot 616 shows the results. The alignment between real and virtual lines is again disappointing. The pose error

$[e_X, e_Y, e_\theta] = [-0.4, -2.8, -0.6°]$

Again there was movement, but the translation error should be < ±2 cm in both axes (and it is for X!), and the angular error should be < ±0.3° (the gyroscope reported -0.01°). The error is actually quite good, and certainly not large enough to abort.

Next, and last, is the forward motion of 200 cm east. Plot 617 shows the results. The alignment between real and virtual lines is decent. The pose error

$[e_X, e_Y, e_\theta] = [-2.1,-0.5, -1.2°]$

Again there was movement, but the translation error should be < ±2 cm in Y (and it is!) and < ±2 cm (really close!) in X, and the angular error should be < ±0.2° (the gyroscope reported -0.05°). The error is actually quite good, and certainly not large enough to abort.

After all the movements, the effective overall pose error

$[e_X, e_Y, e_\theta] = [-2.1, -0.5, -1.2°]$

That is quite remarkable for 7 different movements over a linear distance of 950 cm and a rotation of 450°. That said, the measured translation error was (1,-6), still quite good. The visual angular error was much less than 1°. Altogether, rather successful.

Step 23: An Autonomous Rover: Conclusion

The results described in the pervious step lead to the following assessment: The *Localization*, *Navigation*, and *Locomotion* capabilities, while not perfect, are quite adequate to enable the *Autonomous Rover* to accomplish its task of moving from one location to another without help.

The AR described herein just moves from location A to location B and it is done. However, that is only the simplest way the capabilities could be combined. A somewhat more complex task would be to move from A to B, from B to C, C to …. Another task might be A to B, from B to C, and then return to A. And so on.

Of course, it would be possible to add or improve capabilities for more sophistication. For example, with improved localization, it might be possible to start an an unknown location and orientation, figure out the start location and orientation, and them move to a target location. My wife wants the AR to bring her coffee. Maybe someday ….

I hope you enjoyed reading this Instructable, and learned something useful while doing so. Please consider voting for it in the ***Robots*** contest.