Software Design Document

Purpose of the Software

The software component of the project ties the robot together. Without the software, the robot would be nothing more than a mechanoid.

The software should allow the robot to perform the following tasks:

- Receive instructions and information about the map through Wi-Fi.
- Calculate the robot's initial position and initial heading relative to the map.
- Navigate to any point on the field without veering off course.
- Avoid obstacles along its path.
- Track its position using internal measurements and calculation.
- Correct its orientation and position using external cues from the environment

To facilitate these needs, the following modules will be needed:

- An Odometer
- An Odometer Corrector
- A Filtered Ultrasonic Sensor
- A Filtered Color Sensor
- A Navigator
- A Pilot
- A Path Finder
- An Ultrasonic Localizer
- Color Sensor Localizer
- Ball Launcher
- Obstacle Avoider

We will now discuss the modules in depth below

Odometer

The odometer is an essential component of any autonomous machine. The machine should be able to estimate its position always. External environmental cues that can be used to estimate position and orientation are not always available.

The robot uses a differential drive system for locomotion. A differential drive uses two wheels connected to two motors to steer the robot. This system, while simple, facilitates a variety of movement paths. However, we have chosen to constrict the robot's movement to the first two types:

- Linear Motion – Forward and Backward movement.
- Rotational Motion – In place counter clockwise and clockwise rotation.
- Arc Motion – Combines linear and rotational motion to move along the arc of a circle.

This constriction simplifies our Odometry Correction module.

The below diagram demonstrates the two types of motion.

Linear Motions

Moving the robot forward is done by rotating both motors at the same angular velocity. To convert the angular velocity of the robot in linear velocity, we use the following formula.

$$v = \omega r$$

Rotational Motions

Rotating the robot in placed is achieved by rotating the wheels with the same magnitude and opposite directions to each other. Rotating the left wheel forward and the right wheel backwards will rotate the robot clockwise.

To calculate the angular velocity of the robot using the wheel base (b):

$$\omega_{chassis} = \frac{v}{b}$$

Calculations

The motors have a measurement device, called a tachometer, that tracks the rotation of the motors. We will use the tachometers present in the left and right motors to track the angular velocity of the wheels. Let ϕ_L and ϕ_R be the rotations of the left and right wheel in a small time Δt. Then the new orientation and position are given by

$$\theta_{t+\Delta t} = \theta_t + \sin\left(\frac{r(\phi_r - \phi_l)}{b}\right)$$

$$x_{t+\Delta t} = x_t + \frac{r(\phi_r + \phi_l)}{2}\sin(\theta_{t+\Delta t})$$

$$y_{t+\Delta t} = y_t + \frac{r(\phi_r + \phi_l)}{2}\cos(\theta_{t+\Delta t})$$

The calculations above combine both the linear and rotational motions of the robot. This form of motion is the last type. While our robot will never move in arcs, the calculations for arc movement generalize linear and rotational movements.

Implementation

The Odometer module must execute simultaneously while the robot is busy performing other tasks. To achieve this the Odometer must run as a thread. In each execution of the thread, the calculations cannot be interrupted, so the code needs to atomized. Java affords this capability as special code block called 'synchronized'. All code contained in this block cannot be interrupted while being executed.

The equations above have limitations in the practical aspect of the robot. The trigonometrical calculations can take an enormous amount of time, and will cause the odometry thread to delay other important modules. To avoid this, we use an approximation in the orientation calculation.

$$\theta_{t+\Delta t} \approx \theta_t + \left(\frac{r(\phi_r - \phi_l)}{b}\right)$$

This estimation is valid only for sufficiently small Δt. Hence the Odometry thread must be run frequently to avoid a rapid accumulation of angular error. This compromise is not a problem because of the extreme importance of odometry.

Localization

When the robot is initially placed on the field, there will be a human error present in the robot's position and orientation. To account for this, the robot will need to orient itself with respect to its surroundings. Localization is the estimation of initial position and heading using the environment.

Requirements

The robot will be placed in one of four corners on the field. The only information available about the robot's position is which corner it has been placed in and that it is along a tile's diagonal. The robot should obtain its actual position and heading using the corners of the wall and the gridlines on the floor.

Using ultrasonic sensor in tandem with the walls allows us to measure with a high degree of accuracy what our position and orientation are. However, we cannot use the gridlines because the robot's initial position may place color sensor too far from any gridlines. Hence, to overcome this problem, we will initially use the ultrasonic sensor to estimate the robot's position and orientation so the robot can move to the nearest gridlines and use its light sensor to localize.

Ultrasonic Localization

Assuming the robot had been placed perfectly on a diagonal line, we can obtain information about our orientation in the following steps:

1. Rotate clockwise until a wall is visible, record the angle from the odometer as α
2. Rotate counter clockwise until a wall is visible, record the angle from the odometer as β
3. Stop

Using the angles α and β we can calculate our orientation with the following formula if the robot has not moved since we measured β

$$\theta(\alpha,\beta) = \begin{cases} \frac{5\pi}{4} + \frac{\beta-\alpha}{2}, & \beta < \alpha \\ \frac{\pi}{4} + \frac{\beta-\alpha}{2}, & \beta \geq \alpha \end{cases}$$

Color Sensor Localization

Once the robot has navigated to a suitable location, preferably one with gridlines in a known arrangement, we can use the geometry of the gridlines to calculate our position and angle.

The robot will rotate 2π radians and the light sensor will pick up four gridlines. Assuming the robot is in the 3rd quadrant with respect to the gridline cross point, we can calculate our distance from the cross point using the following

Let θ_x be the measured odometry angle between the horizontal lines, let θ_y be the measured odometry angle between the vertical lines and let ϵ be the offset of the light sensor from the center of rotation.

$$x = -\epsilon \cos\left(\frac{\theta_y}{2}\right)$$

$$y = -\epsilon \cos\left(\frac{\theta_x}{2}\right)$$

Implementation

The robot can be placed in any orientation along the diagonal, and this includes initially facing the wall. This is a problem because we are using falling edge localization, so we need the robot to initially facing away from the wall. To overcome this, we will rotate the robot until a wall can no longer be seen. The robot will continue to rotate for two seconds more using a delay function. This ensures that there is no wall in front of us.

To determine whether a wall is seen, we continuously poll the sensor and check if the distance falls below a constant called 'MIN_WALL_DISTANCE'. The detection of this falling edge is

subject to noise, however, experiments have shown that the filter used for the ultrasonic sensor negates the effect of noise.

Once we have determined our heading we can turn our robot towards the left wall and measure the robot's distance from it. This allows us to estimate our robot's position with sufficient accuracy to move close enough to a gridline cross point to use the light sensor.

After performing localization using the light sensor we perform ultrasonic localization again to correct for the error accumulated due to rotating in a circle.

Once we have determined our angle and position relative to the closest gridline cross. We can navigate towards it. Our robot believes itself to be at the point (0,0). We know the actual position of each of the gridline crosses, so we can simply set the odometer position to that.

The reason we allow the robot to believe that it is at (0,0) and in the first corner is to avoid code duplication. The robot's motions in each corner are identical

1. Calculate heading
2. Calculate position
3. Navigate towards (0,0)

The only differing aspect in regards to the starting corner is the actual position of the robot once the final step completes.

Filtered Sensors

The sensors provided with EV3 kits are not perfect. They are susceptible to noise, random error and systematic error. Using filters and experiments, we can minimize these sources of error.

Systematic error originates due to manufacturing defects or damage to the sensor. This kind of error is the simplest to account for. A few simple experiments with calibrated inputs allows us to calculate the systematic error. The error will be used to offset all future readings so they become accurate.

Noise and random error are unpredictable and can vary throughout the lifetime of the sensor. The only way to fix this is using a filter.

There are three basic filters that we can utilize for these sensors

- Mean Filtering
- Median Filtering
- Modal Filtering

Given a sample size of n, we can estimate the time complexity of each filter

- The mean filter will run in $O(n)$
- The median filter will run in $O(\log(n))$
- The modal filter will run in $O(n\log(n))$

Using the above calculation we can see that the modal filter is the worst type of filter, so we will restrict ourselves to the first two.

Ultrasonic Filtering

The ultrasonic sensor produces a series of sonic pulses and measures the time taken for each pulse to bounce off a surface and arrive back to the sensor. Using this information, the sensor can calculate the distance of a surface from itself.

However, the ultrasonic sensor is extremely susceptible to noise and other sources of error. This includes variation in the density of air and the geometry of the surface. Performing experiments

on the outputs of the ultrasonic sensor provides important information about its behavior. If we collect a set of distance measurements.

If there are n samples from the sensor in the set and it should have been measuring a distance of x, we will see that the value x occurs 66.66% of the time. The second most common value is 255. This is the maximum distance that the sensor can measure in centimeters.

Using an averaging filter and if there are no values other than x and 255

$$\frac{x\left(\frac{2}{3}n\right) + (255)\left(\frac{1}{3}n\right)}{n} = \frac{2x}{3} + 85$$

This value is too far from the value of actual value of x. For example, if $x = 25$, the averaged value would be

$$\frac{2(25)}{3} + 85 = 101.66$$

Hence, we decided to use a median filter. The median filter would select the middle most value in any given set of measurements. This avoid the problems with average filter entirely. Tests have shown that this type of filter works extremely well. In fact, due to the filter's high accuracy, we can eliminate the noise margin entirely from the ultrasonic localization.

Implementation

There are two ways to implement this filter:

- Take continuous readings use a moving window to operate on a subset of values.
- Take a set of readings when prompted and operate on that set.

We have decided to implement it the second way because it would be the simplest to program. Furthermore, experiments have shown that it has a minimal effect on the operation the robot. So long, as the set size is sufficiently small.

In the robot, samples are fetched from the sensor in periodic intervals. The samples are placed into an array and sorted. By taking the set size and using integer division by two, we can obtain the middle value. Since the number of samples in the set is hard coded, we can ensure that it is always an odd number.

Let n be the number of samples and T be the sensor poll period, the polling frequency of the sensor is given by.

$$f = \frac{1}{nT}$$

Color Sensor Filtering

In contrast to the ultrasonic sensor, the measurements of the color sensor follow a Gaussian distribution. This means that we can apply a simple average filter to get a precise value

Once again, there are two ways we can implement this kind of filter:

- Take continuous readings use a moving window to operate on a subset of values.
- Take a set of readings when prompted and operate on that set.

We decided to use the second implementation.

In the color sensor's filter, we run a loop and sum each sample from the sensor. Once the for loop exits, we divide the sum by the number of values fetched.

Let n be the number of samples and T be the sensor poll period, the polling frequency of the sensor is given by.

$$f = \frac{1}{nT}$$

However, unlike the ultrasonic sensor, we have two color sensors running simultaneously. Hence we cannot use a large value for T or n. The delay would cause too much error in the odometer correction. Experiments show that the smaller values for n have a negligible effect on the filter.

Navigation

The purpose of this robot is to provide an interface for controlling the motors by specifying target coordinates or target headings. This facilitates a high level of control over the robot's movement and abstracts the motors' methods.

Navigator continuously reads information from the odometer to figure out its position relative to the target coordinate. Let (x, y) be the position of the robot and let (a, b) be the desired position of the robot.

The information relevant to controlling the motors, is the angle between (x, y) and (a, b) and the distance separating them. Let ψ be the angle between the two points and be given by

$$\psi = \tan^{-1}\left(\frac{b-y}{a-x}\right) = \cos^{-1}\left(\frac{b-y}{\sqrt{(b-y)^2+(a-x)^2}}\right)$$

Let ϵ represent the distance between the two points and be given by

$$\epsilon = \sqrt{(b-y)^2+(a-x)^2}$$

Implementation

Java provides a function called 'atan2' to convert a given coordinate into polar form and return the angle component. However, this function has a few limitations.

When the robot moves along the x-axis, the function atan2 returns either $-\frac{\pi}{2}$ or $\frac{\pi}{2}$, this causes the robot's navigator to experience a large bouncing effect as it attempts to correct motor drift.

The only to fix this would be to remove the angular correction as the robot moves. However, this causes an accumulation of a large error while the robot moves. However, if we add a delay before each angle correction we can significantly reduce the bouncing effect observed.

Pathfinder

To abstract away the complexity of navigating along each axis individually, we need a special class to convert target coordinates into per axis movement.

The pathfinder class generates a list of coordinates that represents a path. The path is designed to be movement that is optimal for Odometry Correction. To simplify the design of the odometer we restrict movement to a per axis basis.

Implementation

In the constructor for the pathfinder we pass in the desired coordinate and the constructor breaks it down into several sub coordinates. Each sub coordinate is placed in a stack, so we must generate the path in the reverse direction of movement.

If (a, b) is the desired coordinate and (x, y) is the current coordinate of the robot we generate the coordinates as such

1. Push (a, b) onto the stack
2. Push (x, b) onto the stack

This is an extremely simplified implementation of the path finder. There are several ways to navigate towards (a, b) in the most optimal way for odometry correction. We can also break down the movement in each axis to movement on a per tile basis. This is the segmented pathfinder.

The segmented pathfinder uses several loops to break down each axis movement into small movements each of length $\delta = 30.48$ centimeters. Given a desired coordinate of (a, b) we can use the following formula to generate a path

$$T_x = \left\lfloor \frac{a - x_0}{\delta} \right\rfloor$$

$$T_y = \left\lfloor \frac{b - y_0}{\delta} \right\rfloor$$

The n[th] y-axis coordinates are given as $y_n = n\delta,\ 1 \leq n \leq T_y$

The n[th] x-axis coordinates are given as $x_n = n\delta,\ 1 \leq n \leq T_x$

1. Push (a, b)
2. Push $(x_n, T_y\delta)$ for $n = T_x, T_x - 1, T_x - 2, \ldots$
3. Push (x_0, y_n) for $n = T_y, T_y - 1, T_y - 2, \ldots$

Given (x_0, y_0) as the initial coordinate of the robot.

The segmented pathfinder was designed to be used alongside the 'one-shot navigator'. The simpler pathfinder was designed to be used with the 'continuously updated navigator'.

Pilot

The navigator class performs the low-level motor control and the pathfinder classes provides the low-level coordinate handling, we need a class to unify these two and provide a single control point for the robot's locomotion.

Implementation

The pilot class takes as input a pathfinder object. Using the coordinates on the stack and a navigator object the pathfinder calls the navigator using each coordinate from the pathfinder to move the robot.

This provides another advantage in terms of programming of the robot. The obstacle avoidance subroutine can be placed here. This allows us, in a high-level way, to interrupt the movement once an obstacle is visible and navigate around it.

Obstacle Avoidance

There are several ways to implement obstacle avoidance

- Using the A* algorithm to generate a path using a virtual map filled with information about the field collected by the ultrasonic sensor.
- Start a wall following routine and exit once the robot's direction has flipped 180°.
- Switch the order of the axis in the path generated by the pathfinder.

The first method is difficult to program and requires knowledge of the obstacle placement beforehand. In the second method, detecting whether the robot has flipped 180° is complicated and subject to a large error.

Hence we will use the third method.

Implementation
The pilot class will execute navigation object as a thread. This allows the pilot instance to poll the ultrasonic sensor for obstacles. If an obstacle is detected, the thread in interrupted and the motors are stopped to end their regulation threads.

The pilot class then generates a new pathfinder object with the same desired coordinates. However, the order that the axes are traversed is reversed. If the pilot was originally going to traverse the x-axis and then the y-axis, after detecting the object it would traverse the y-axis and then the x-axis.

This method is a brute force obstacle avoidance, however, for the purposes of this project the algorithm will be sufficient.

Ball Launcher
This class provides a high-level interface for controlling the firing and reloading motors.

Requirements
We need the robot to be able to store up balls and load them into the crossbow when requested. The robot should be able to fire, reload, and cock using only three motors.

There are two motors allotted to the ball launching mechanism.

- Motor M_1 reloads and fires the crossbow
- Motor M_2 cocks the crossbow

When M_1 rotates clockwise the firing pin is released and the crossbow launches the ball. If M_1 rotates counter clockwise, a ball is loaded into the crossbow.

Implementation
The basic operation to load a ball from its carrier into the crossbow and firing

1. Rotate M_2 to wind up the crossbow elastic and cock it
2. Reverse M_2 to unwind the string and allow the elastic to move freely
3. Rotate M_1 counter clockwise to push a ball into the crossbow
4. Rotate M_1 clockwise to release the firing pin
5. Go to step 1