Computational Principles of Mobile Robotics

Fundamental problems

Some realities

- Robots are incredibly complicated pieces of machinery that must deal with a complex and difficult to model world.
- It can be beneficial to consider simplifications of the problem, if only to make solutions more tractable.
- A critical observation here though is that as the abstraction becomes less and less divorced from reality, the solutions developed can become very fragile to the subtleties associated with the original problem.

A point robot

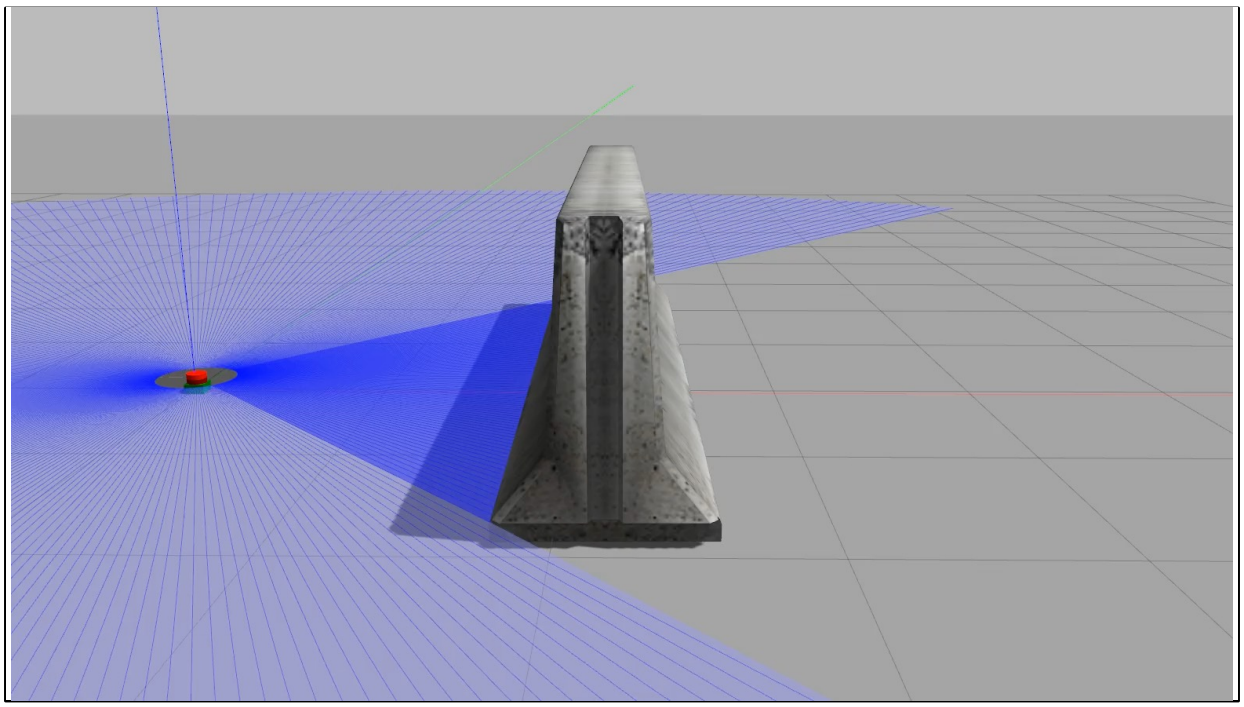
- A classic model of a robot is that the robot is a point, or perhaps a point with orientation.
  - No extension in space (literally a point).
  - No mass (and hence no problems in local motion modelling)
  - No difficulty in determining intersection with objects/
- So ignore orientation. The robot is full described as a point,
  - (x,y) on a Cartesian plane
  - This fully defines the robot's configuration.

Fundamental problems for a point robot

- *Path planning.* If we separate space into regions that are free C_{free} or occupied, is it possible to identify a continuous path from (a,b) to (c,d) that lies fully in C_{free} ?
- *Pose estimation.* Given local measurements of C_{free}, is it possible to determine where the robot is in C_{free}?
- *Robot perception.* How can the robot estimate if a location in space is in C_{free} ?
- *Robot mapping. Assuming that pose estimation is solved, how can a robot determine* C_{free}*?*
- *SLAM.* How can the robot determine its pose and C_{free} concurrently?

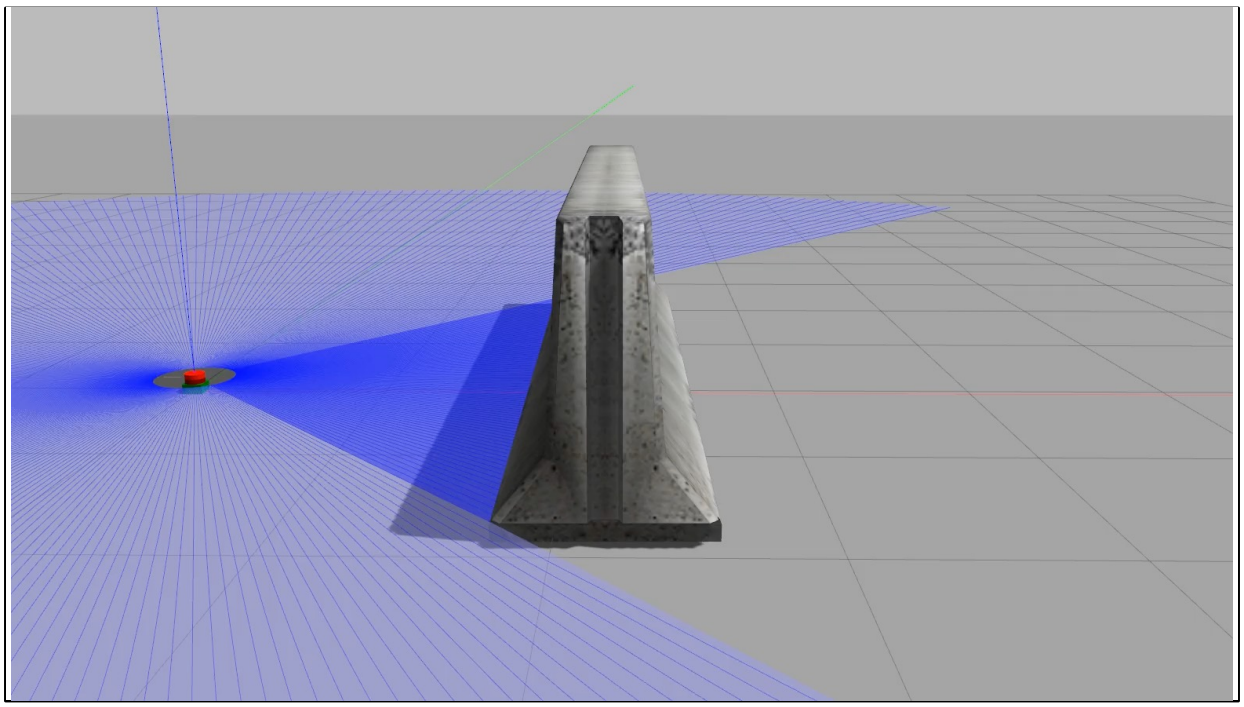
2.1 Defining a point robot

- A point robot on the plane is just that, a point (x,y).
- A point robot on the plane with orientation is (x,y,theta).
  - The 'block robot' of the text.
- We can imagine teleporting a robot from (a,b) to (c,d) assuming both (a,b) and (c,d) are in C_{free}



Driving to a goal

- ROS relies on a controlled command velocity to change the position/orientation of a robot.
- So moving from (a,b) to (c,d) is not a process of interpolating along the line between the points
  - It is the process of developing a controller that converges to the goal state (c,d) given the kinematic and dynamic constraints of the device
  - This is much more realistic than just teleporting to intermediate points in the line from (a,b) to (c,d)



drive_robot.py

```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion
from nav_msgs.msg import Odometry

def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll, pitch, yaw
    quaternion = [x, y, z, w]
    """
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw


class MoveToGoal(Node):
    def __init__(self):
        super().__init__('move_robot_to_goal')
        self.get_logger().info(f'{self.get_name()} created')

        self._goal_x = 0.0
        self._goal_y = 0.0
        self._goal_t = 0.0

        self.declare_parameter('goal_x', value=self._goal_x)
        self.declare_parameter('goal_y', value=self._goal_y)
        self.declare_parameter('goal_t', value=self._goal_t)
        self.add_on_set_parameters_callback(self.parameter_callback)

        self._subscriber = self.create_subscription(Odometry, '/odom', self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist, '/cmd_vel', 1)


    def _listener_callback(self, msg, vel_gain=5.0, max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose

        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitch, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

        twist = Twist()
        if dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -max_vel)
            twist.linear.x = x * math.cos(cur_t) + y * math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y * math.cos(cur_t)
            self.get_logger().info(f"at ({cur_x},{cur_y},{cur_t}) goal ({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)

    def parameter_callback(self, params):
        self.get_logger().info(f'move_robot_to_goal parameter callback')
        for param in params:
            if param.name == 'goal_x' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_x = param.value
            elif param.name == 'goal_y' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_y = param.value
            elif param.name == 'goal_t' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_t = param.value
            else:
                self.get_logger().warn(f'Invalid parameter {param.name}')
                return SetParametersResult(successful=False)
        return SetParametersResult(successful=True)


def main(args=None):
    rclpy.init(args=args)
    node = MoveToGoal()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

```
class MoveToGoal(Node):
    def __init__(self):
        super().__init__('move_robot_to_goal')
        self.get_logger().info(f'{self.get_name()} created')

        self._goal_x = 0.0
        self._goal_y = 0.0
        self._goal_t = 0.0

        self.declare_parameter('goal_x', value=self._goal_x)
        self.declare_parameter('goal_y', value=self._goal_y)
        self.declare_parameter('goal_t', value=self._goal_t)
        self.add_on_set_parameters_callback(self.parameter_callback)

        self._subscriber = self.create_subscription(Odometry, "/odom", self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist, "/cmd_vel", 1)


    def _listener_callback(self, msg, vel_gain=5.0, max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose

        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitchc, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

        twist = Twist()
        if dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -max_vel)
            twist.linear.x = x * math.cos(cur_t) + y * math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y * math.cos(cur_t)
            self.get_logger().info(f"at ({cur_x},{cur_y},{cur_t}) goal ({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)

    def parameter_callback(self, params):
        self.get_logger().info(f'move_robot_to_goal parameter callback')
        for param in params:
            if param.name == 'goal_x' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_x = param.value
            elif param.name == 'goal_y' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_y = param.value
            elif param.name == 'goal_t' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_t = param.value
            else:
                self.get_logger().warn(f'Invalid parameter {param.name}')
                return SetParametersResult(successful=False)
        return SetParametersResult(successful=True)
```

```
class MoveToGoal(Node):
    def __init__(self):
        super().__init__('move_robot_to_goal')
        self.get_logger().info(f'{self.get_name()} created')

        self._goal_x = 0.0
        self._goal_y = 0.0
        self._goal_t = 0.0

        self.declare_parameter('goal_x', value=self._goal_x)
        self.declare_parameter('goal_y', value=self._goal_y)
        self.declare_parameter('goal_t', value=self._goal_t)
        self.add_on_set_parameters_callback(self.parameter_callback)

        self._subscriber = self.create_subscription(Odometry, "/odom", self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist, "/cmd_vel", 1)

    def _listener_callback(self, msg, vel_gain=5.0, max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose

        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitchc, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

        twist = Twist()
        if dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -max_vel)
            twist.linear.x = x * math.cos(cur_t) + y * math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y * math.cos(cur_t)
            self.get_logger().info(f"at ({cur_x},{cur_y},{cur_t}) goal ({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)

    def parameter_callback(self, params):
        self.get_logger().info(f'move_robot_to_goal parameter callback')
        for param in params:
            if param.name == 'goal_x' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_x = param.value
            elif param.name == 'goal_y' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_y = param.value
            elif param.name == 'goal_t' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_t = param.value
            else:
                self.get_logger().warn(f'Invalid parameter {param.name}')
                return SetParametersResult(successful=False)
        return SetParametersResult(successful=True)
```

Sets up node to receive updates (messages) and parameters updates. Also advertises that it will publish.

```
class MoveToGoal(Node):
    def __init__(self):
        super().__init__('move_robot_to_goal')
        self.get_logger().info(f'{self.get_name()} created')

        self._goal_x = 0.0
        self._goal_y = 0.0
        self._goal_t = 0.0

        self.declare_parameter('goal_x', value=self._goal_x)
        self.declare_parameter('goal_y', value=self._goal_y)
        self.declare_parameter('goal_t', value=self._goal_t)
        self.add_on_set_parameters_callback(self.parameter_callback)

        self._subscriber = self.create_subscription(Odometry, "/odom", self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist, "/cmd_vel", 1)


    def _listener_callback(self, msg, vel_gain=5.0, max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose

        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitchc, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

        twist = Twist()
        if dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -max_vel)
            twist.linear.x = x * math.cos(cur_t) + y * math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y * math.cos(cur_t)
            self.get_logger().info(f"at ({cur_x},{cur_y},{cur_t}) goal ({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)
```

```
    def parameter_callback(self, params):
        self.get_logger().info(f'move_robot_to_goal parameter callback')
        for param in params:
            if param.name == 'goal_x' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_x = param.value
            elif param.name == 'goal_y' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_y = param.value
            elif param.name == 'goal_t' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_t = param.value
            else:
                self.get_logger().warn(f'Invalid parameter {param.name}')
                return SetParametersResult(successful=False)
        return SetParametersResult(successful=True)
```

We can be told to update our goal pose.

```
class MoveToGoal(Node):
    def __init__(self):
        super().__init__('move_robot_to_goal')
        self.get_logger().info(f'{self.get_name()} created')

        self._goal_x = 0.0
        self._goal_y = 0.0
        self._goal_t = 0.0

        self.declare_parameter('goal_x', value=self._goal_x)
        self.declare_parameter('goal_y', value=self._goal_y)
        self.declare_parameter('goal_t', value=self._goal_t)
        self.add_on_set_parameters_callback(self.parameter_callback)

        self._subscriber = self.create_subscription(Odometry, "/odom", self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist, "/cmd_vel", 1)

    def _listener_callback(self, msg, vel_gain=5.0, max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose

        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitchc, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

        twist = Twist()
        if dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -max_vel)
            twist.linear.x = x * math.cos(cur_t) + y * math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y * math.cos(cur_t)
            self.get_logger().info(f"at ({cur_x},{cur_y},{cur_t}) goal ({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)

    def parameter_callback(self, params):
        self.get_logger().info(f'move_robot_to_goal parameter callback')
        for param in params:
            if param.name == 'goal_x' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_x = param.value
            elif param.name == 'goal_y' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_y = param.value
            elif param.name == 'goal_t' and param.type_ == Parameter.Type.DOUBLE:
                self._goal_t = param.value
            else:
                self.get_logger().warn(f'Invalid parameter {param.name}')
                return SetParametersResult(successful=False)
        return SetParametersResult(successful=True)
```

And we will respond to updates in the robot's pose

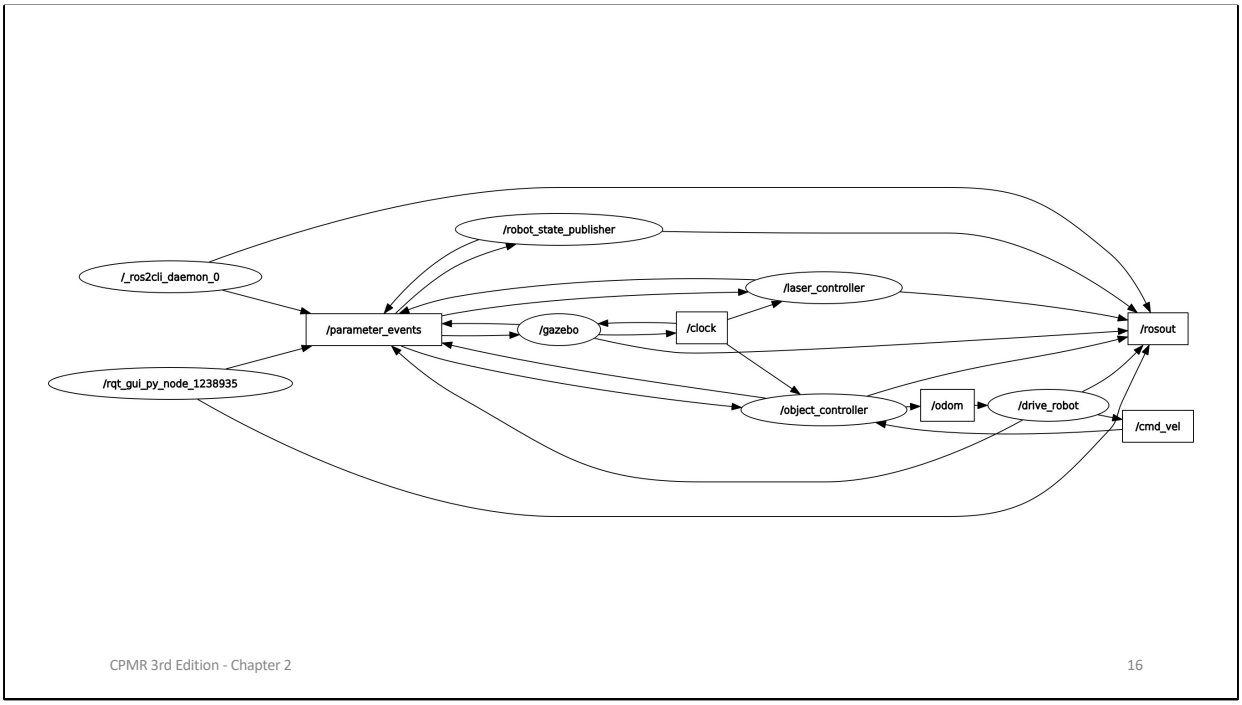
ROS parameters

- In ROS 2 parameters are associated with nodes and must be declared by them.

```
Ros2 param set /drive_robot goal_x 2.0
```

ROS message

- This node listens to Odometry messages (published on /odom)
- This node publishes Twist messages (published on /cmd_vel)
- Gazebo simulates the robot and publishes where it thinks the robot is (/odom).
- And updates that simulation based on Twist messages published on /cmd_vel.


/robot_state_publisher
/_ros2cli_daemon_0
/laser_controller
/parameter_events
/gazebo
/clock
/rosout
/rqt_gui_py_node_1238935
/object_controller
/odom
/drive_robot
/cmd_vel

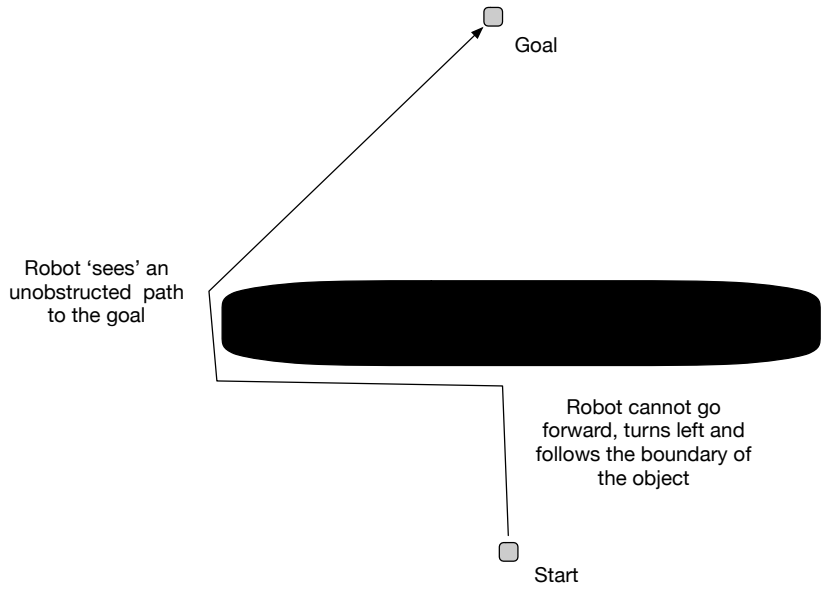
Summary

- Obtaining a continuous path from S1 to S2 for even a point robot can be a complex task.
- Problem becomes even more difficult when kinematic and dynamic constraints are considered and/or when paths must be chosen to avoid stationary or dynamic obstacles.
  - We will see many example algorithms here for this last problem late in the course.
- But let us move on to the (simple) problem of point robot path planning.

2.2 Path planning for a point robot

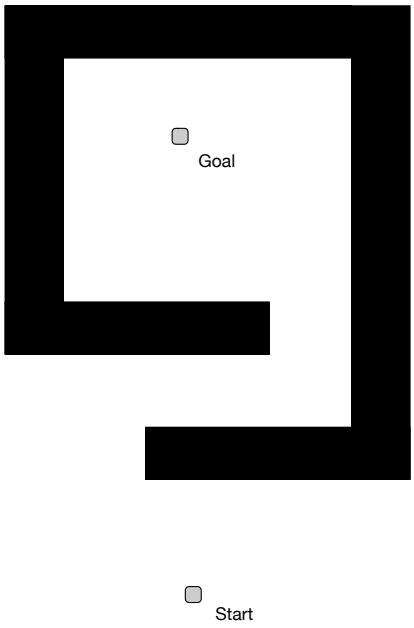
- Perhaps the simplest algorithms here are the Bug algorithms.
  - Loosely based on planning made by an insect (hence bug).
- Assumptions
  - Robot knows its position, goal, and can sense local obstacles.
- Bug 0 algorithm
  - Head towards goal
  - If robot encounters an obstacle, follow the edge of the obstacle until the robot can head towards the goal again
  - Repeat until success

Bug 0


Goal
Robot 'sees' an unobstructed path to the goal
Robot cannot go forward, turns left and follows the boundary of the object
Start

Bug 0

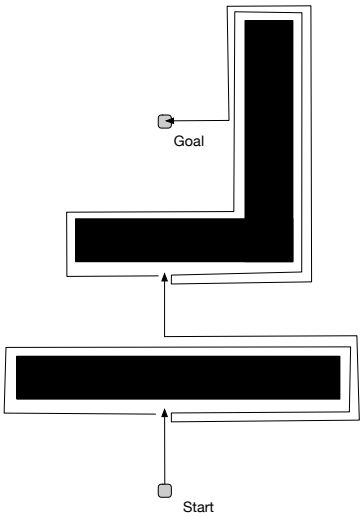
- Not guaranteed to find a path.
- Large number of assumptions
  - Many of which end up having to be encoded explicitly when you try to implement this.



Bug 1

- Add memory to the robot.
- When encountering and obstacle, circumnavigate and map it.
- Identify point of closest approach to the goal on the obstacle.
- Head there and depart for the goal.

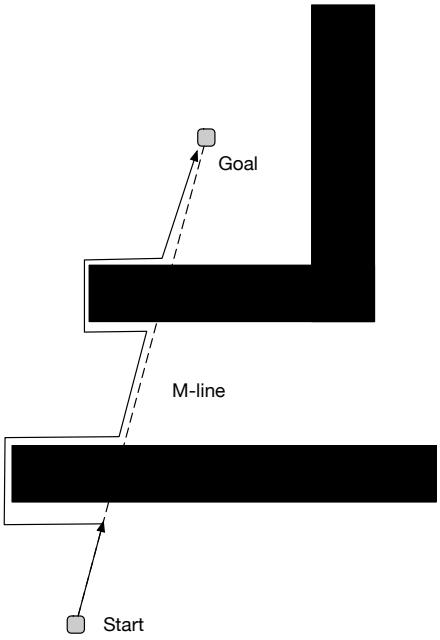
Bug 1



Guaranteed to work (in theory)
Worst case cost is easily computed as the robot must circumnavigate each object at most once.

Bug 2

- Draw a straight line from Start to Goal. Call this the m-line.
- Follow the m line to the goal.
- If an obstacle is encountered, follow it until you encounter the m-line again **closer** to the goal.
- Follow the obstacle and continue on the m-line until the goal is reached.


Goal
M-line
Start

2.3 Localization for a point robot

- If you look at the details of the Bug algorithms, the robot needs to be able to do a great number of (quite difficult) things for the algorithm to work.
- First, it needs to know where it is, all the time. This is the problem of localization.
- But do we need to deal with this? How bad can it be?

Localization of a point robot

- Assume that every motion is corrupted by a small amount of noise
- That errors in x and y are independent
- That the variances are stationary
- That the covariances are independent

$$
\begin{aligned}
&\forall_i \mathrm{E}[\epsilon_i^x] = \mathrm{E}[\epsilon_i^y] = 0 \\
&\forall_i \mathrm{E}[(\epsilon_i^x - \mathrm{E}[\epsilon_i^x])(\epsilon_i^x - \mathrm{E}[\epsilon_i^x])] = \sigma^2 \\
&\forall_i \mathrm{E}[(\epsilon_i^y - \mathrm{E}[\epsilon_i^y])(\epsilon_i^y - \mathrm{E}[\epsilon_i^y])] = \sigma^2 \\
&\forall_{i \neq j} \mathrm{E}[(\epsilon_i^x - \mathrm{E}[\epsilon_i^x])(\epsilon_i^y - \mathrm{E}[\epsilon_i^y])] = 0.
\end{aligned}
$$

Localization of a point robot

- Then if we make N motions, where can we expect to get to?

$$
\begin{aligned}
\mathrm{E}[(x_N, y_N)] &= \mathrm{E}[(x_0, y_0) + \sum(\Delta x_i + \epsilon_i^x, \Delta y_i + \epsilon_i^y)] \\
&= (x_0, y_0) + (\mathrm{E}[\sum \Delta x_i + \epsilon_i^x], \mathrm{E}[\sum \Delta y_i + \epsilon_i^y]) \\
&= (x_0, y_0) + (\sum \Delta x_i + \sum \mathrm{E}[\epsilon_i^x], \sum \Delta y_i + \mathrm{E}[\epsilon_i^y]) \\
&= (x_0, y_0) + (\sum \Delta x_i, \sum \Delta y_i).
\end{aligned}
$$

Localization for a point robot

- And distribution around this expected value?

$$
\begin{aligned}
\sigma_{xx} &= \mathrm{E}[(x - \mathrm{E}[x])^2] \\
&= \mathrm{E}[(x_0 + \textstyle\sum(\Delta x_i + \epsilon_i^x) - \mathrm{E}[x_0 + \sum(\Delta x_i + \epsilon_i^x)])^2] \\
&= \mathrm{E}[(x_0 + \textstyle\sum \Delta x_i + \sum \epsilon_i^x - \mathrm{E}[x_0] - \sum \mathrm{E}[\Delta x_i] - \sum \mathrm{E}[\epsilon_i^x])^2] \\
&= \mathrm{E}[(\textstyle\sum \epsilon_i^x - \sum \mathrm{E}[\epsilon_i^x])^2] \\
&= \mathrm{E}[\textstyle\sum(\epsilon_i^x - \mathrm{E}[\epsilon_i^x])^2] \\
&= \textstyle\sum \mathrm{E}[(\epsilon_i^x - \mathrm{E}[\epsilon_i^x])^2] + 2\sum_{i<j} \sum \mathrm{E}[(\epsilon_i^x - \mathrm{E}[\epsilon_i^x])(\epsilon_j^x - \mathrm{E}[\epsilon_j^x])].
\end{aligned}
$$

$$
\Sigma = N \begin{bmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{bmatrix}
$$

Error grows with N

Localization for a point robot

- We need something, more than just odometry (counting the rolling of the wheels).
  - This traces its history back to Rome, Greece and China (at least) where the revolution of wheels on chariots were used to count distance travelled.
- Basic strategy to address this is related to navigation
  - Earliest examples of navigation away from land, using the stars can be seen as early as 3000BC in the Indio-Pacific (Taiwan)
  - Invention of the Marine Chronometer (clock) in the 1730's provided accurate navigation/localization on the sea.
- Basic idea is to use triangulation or trilateration to figure out where you are given some measurements.

Many solutions here

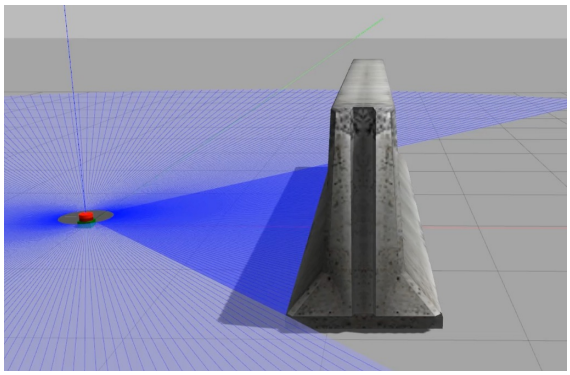
- If your sensor is very powerful (e.g., can measure range with no error) then three markers are sufficient on the plane.
  - Can generalize this to GPS
- But even simple solutions can work. Lorenz beam from the 1940's used two radio transmitters (one sending dashes the other dots) so that when you were equidistant from the two you received a constant signal. This was sufficient to allow aircraft to find runways and land even in very bad conditions.
- We will explore a number of solutions here in later chapters.

2.4 Sensing for a point robot

- In a perfect 2D world, with well behaved sensor noise (independent additive Gaussian, for example) we can exploit standard data merging tools (e.g., least squares) to fit data to an underlying model.
  - Sensor noise is never like this.
- Some examples here
  - LIDAR (laser) dislikes specular surfaces (sees the reflection).
  - IR dislikes non-reflective surfaces (absorbs the IR signal) and dislikes reflective structures.
  - SONAR dislikes sound absorbing surfaces (ceiling tiles)
- Almost all active sensor technology dislikes other active emitters.

Sensors in ROS/Gazebo

- Gazebo has a number of simulated sensors (e.g., planar LIDAR).
  - Provides very simple (perfect) error models for the sensor
- Expect the real world to be more complex/difficult even if the format of the data is similar.



2.5 Mapping for a point robot

- If we assume perfect odometry, then a range of simple solutions exist.
  - Build a 2D map of points.
  - Whenever a laser gives a valid reading, make that a point in a 2D map
  - Aggregate over motion
- Under the assumption of perfect odometry, even if the LIDAR measurements are corrupted with independent noise then we can use standard statistical tools to aggregate the data
  - Often the tools are designed to exactly deal with this sort of noise.

2.6 SLAM for a point robot

- Suppose you can identify a unique something (a corner) using a sensor like LIDAR.
- Then given the standard noise assumptions (even in robot motion) the corner becomes a robust feature in space. Its location can be estimated relative to other such markers.
  - A corner-based map.
- LIDAR contour can do the same, for a more dense map representation
  - LIDAR scan matching (Lu and Milios, as an example)

2.7 Looking forward

- Later chapters will generalize the simple robot model here to more realistic robot hardware.
- Code provided with the text is structured so that there is minimal relationship between the code from one chapter to another.
  - Results in some code duplication but allows modification to code in one chapter without breaking code in others.