# CS 363 Robotics, Spring 2013

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

**Course Description**

Robotics addresses the problems of controlling and motivating mechanical devices to act intelligently in dynamic, unpredictable environments. Major topics will include: navigation and control, mapping and localization, robot perception using vision and sonar, kinematics and inverse kinematics, and robot simulation environments. To demonstrate these concepts we will be using medium sized mobile robots capable of functioning in human environments. Lab and assignments will focus on programming robots to execute tasks, explore, and interact with their environment.

**Prerequisites:** CS 251 or permission of instructor.

**Desired Course Outcomes**

A. Students understand and can implement fundamental algorithms for robot control and navigation.

B. Students understand and can make use of a variety of sensors to enable autonomous robot behaviors.

C. Students understand the concept of state estimation applied to sensing, localization, and mapping.

D. Students work in a group to design software for controlling a robot.

E. Students present methods, algorithms, results, and designs in an organized and competently written manner.

F. Students write, organize, and manage a large software project.

# 1 Robotics

What is a robot? Rod Brooks‘ definition:

- **Situated**: The robot has to live in a rich sensory environment and be capable of sensing the richness.
- **Enabled**: The robot has to be able to modify its environment or it is only an observer.
- **Motivated**: The robot has to be motivated to interact and modify its environment or it is a power tool.

## 1.1 A Brief History of Robotics

- 1206: Al-Jazari created the first programmable humanoid robot, a boat with four musicians that could be programmed to play different rhythms.
- 1868: Zadock Dederick designs and builds the first Steam Man, connected to a wagon.
- 1890: Nicola Tesla designs and builds radio-controlled robots, including an underwater robot.
- 1893: Georges Moore designs and builds a Steam Man, a biped which could walk at 5mph.
- 1940’s: Norbert Weiner designs the cybernetic AA gun, which uses radar to sense enemy planes.
- 1940’s: Scientists in Germany design the V1 and V2 rockets, which were arguably the first autonomous robots.
- 1950: W. Grey Walter designs and builds an autonomous turtle with sensors (eyes, ears, and feelers) and computational ability. The turtle could find its own recharging hutch and avoid obstacles.
- 1961: Unimate starts working on the GM assembly line.
- 1966-72: Development of SRI Shakey, the Stanford Cart, and the CMU Rover, autonomous robots with vision and sensing.
- 1979: Robotics Institute founded at CMU by Raj Reddy
- 1981: Direct Drive arm built by Takeo Kanade
- 1994: Dante successfully samples gasses from within a volcano
- 1995: Navlab 2 drives 98% of the way across America.
- 1997: Home-vacuum event at the AAAI Robot competition (not a great showing)
- 1997: Sojourner navigates around Mars for 83 days (expected to last 7)
- 2002: Roomba goes on sale and becomes the first commercial robot to sell over a million entities.
- 2004: Spirit and Opportunity land on Mars. Opportunity is still going in 2013
- 2005: 5 teams complete the off-road DARPA grand challenge, a 200 mile are across the desert
- 2007: DARPA Urban Challenge won by ”Boss” of Tartan Racing
- 2011: Robonaut 2 delivered to the International Space Station
- 2012: Over 3500 PackBots in use by the military

# 2 Robot Designs

Robots come in many configurations and designs. All robots can move (or they wouldn't be enabled), but there are many different mechanisms for movement. Before we can write algorithms for making robots move, we have to understand how they can move and how that movement is parameterized.

- Walking (legs)
- Rolling (wheel or wheels)
- Slithering (undulation)
- Hopping
- Flying
- Swimming

While this course will focus on wheeled robots, the algorithms and methods we are learning are applicable to any robotic system that needs to sense its environment and take action.

Robot configurations are different than natural configurations in most cases. Machines do not yet match nature in torque, response time, energy storage, or conversion efficiency at the same scales as biological systems.

Machines do have actively powered rotational joints, however, which biological systems do not (biological systems use passive joints with active muscles that contract or relax to cause the joint to move).

Key issues in locomotion:

- Stability:
  - Number and geometry of contact points
  - Center of gravity
  - Dynamic stability
  - Inclination of terrain
  - Potential external forces
- Characteristics of contact
  - Contact point/path size and shape
  - Angle of contact
  - Friction
- Type of environment
  - Structure
  - Medium

## 2.1 Legged Robots

Legged robots offer many advantages over wheel robots. Obstacles to rolling motion are not necessarily obstacles to legged motion.

- Ground clearance is generally higher
- Local gradient of the terrain is irrelevant if the feet can find purchase
- A robot with flexible legs can manipulate the environment (everything is an arm)

Legged robots are not as common, however, because they require more power than an equivalently sized wheeled robot, and generally have issues with weight and mechanical complexity. Newer motors and more compact batteries, however, have made legged robots more realistic within the past five years.

- Legs need more than one degree of freedom (at least 2, and 3 is nice)
- Legs must be strong enough to support part of the robot's weight
- Maneuverability requires many legs (or powered flexible joints) to produce forces in the direction required for a given maneuver (think spiders)

Legged robot configurations:

- Six legs plus: statically stable walking is possible. Hold three steady, move the other three.
- Four legs: statically stable walking is possible, and dynamically stable walking is possible with an active control system.
- Two legs: statically stable walking is possible, and dynamically stable walking is possible with an active control system.
- One leg: statically unstable configuration and requires a sophisticated control system.

If each leg can only lift, release, or hold, the number of possible events for a machine with k legs is:

$$N = (2k - 1)! \tag{1}$$

For example, a two legged robot can lift either leg (2 events), release either leg (2 events), lift both legs, or release both legs. A six-legged robot has over 39 million possible choices at any given time step. This is one reason why learning methods have been most successful at learning gaits for robots like the Sony Aibo, which has a multiple degrees of freedom in each leg and a range of velocities and angles it can move through.

## 2.2 Wheeled Robots

The advantage of wheeled robots is low power, low complexity, and higher speeds.

- Variety of configurations
- Efficient designs
- Good maneuverability
- Fast

The disadvantage is functionality in non-smooth terrain (non-zero local gradients)

**Wheels**

- Standard wheel: rotation around the wheel axle and around the contact point
- Castor wheel: rotation around the wheel axle and an offset steering joint
- Swedish wheel: rotation around the wheel axle, contact point, and rollers
- Ball or spherical wheel: rotation around the center of the sphere (difficult to implement)

Regardless of wheel selection, for anything but smooth ground you need a suspension system to keep the wheels in contact with the ground at all times. Sometimes the suspension system is as simple as an inflatable tire (effectively a tight spring), but sometimes it is much more complex.

Question: can you have only two wheels and not fall over?

Wheel geometries (Table 2.1 of Siegwart and Nourbakhsh)

- Bicycle/motorcycle
- Two-wheel differential drive w/COM below wheel axle - Cye robot
- Two-wheel differential drive with castor - Nomad/Magellan
- Two powered traction wheels, one steered front wheel - Piaggio minitrucks
- Two unpowered wheels, one steered, powered front wheel - Neptune (CMU)
- Three motorized Swedish wheels - Palm Pilot robot kit
- Three synchronously motorized & steered wheels ("Synchro drive") - B21r
- Ackerman wheel configuration, front and rear-wheel drive - automobiles
- Four fixed wheels using skid steering - ATRV
- Four steered and motorized wheels - Hyperion, construction vehicles
- Four Swedish wheels - Uranus (CMU)
- Two-wheel differential drive with two castors - Khepera
- Four motorized and steered castor wheels - Nomad XR4000

**Maneuverability v. Controllability:** the maneuverability, or flexibility of a robot is generally inversely related to its controllability. The most difficult robots to control are the swedish wheels and the steered castors, yet they also offer true holonomic ability (they can move in any direction instantaneously). The Ackerman steering system (think cars) is one of the least maneuverable, yet the control system for it is quite simple and the arrangement is very stable. Differential drive systems balance the two characteristics.

# 3 Sensing

## 3.1 Proximity sensors

### Contact sensors

- Push-button: put a plate over a push-button switch, useful as a fail-safe to tell the robot to stop immediately. On the Roomba, they play a central role in obstacle avoidance and navigation.
- Contact surfaces: capacitive or optical screens that sense touch (e.g. iPhone). Handed robots often have contact sensors on the finger/gripper surfaces.
- Big red stop buttons: a critical safety feature, especially for big robots. They generally break power to the motors. These have been used on systems such as robotic combines and placed in strategic locations to they can be triggered by a person's head if the robot has pinned their arms against a wall (Carl Wellington, CMU Graduate)
- Feelers/Antennae: Put a wire on a contact switch. If the wire brushes something, the switch is closed. These can be very useful for avoiding feet or wall-following. It is also possible to put strain gages or flex sensors on an antenna, which give a continuous range of values depending upon the degree of

contact. These have been used to simulate a cockroach moving along a wall using only its feeler as a sensor (Lee et. al, "Templates and Anchors for Antenna-Based Wall Following in Cockroaches and Robots", IEEE Trans. on Robotics, 2008).

**IR sensors**

- The old versions of IR sensors used a focused IR LED to send out a beam of light and a simple IR photosensor to detect the intensity of the reflection. The output told you how much light was being reflected, but that was not always correlated with distance. Ambient light, or dark objects could cause biases in the range values returned.
- The new versions of IR sensors, the Sharp IR Rangers, use a linear strip of detectors and estimate range using triangulation rather that reflected intensity. They are available in a variety of distances from 0.3m to 5.5m, with different models have different minimum and maximum ranges. Some IR sensors provide a Boolean output, others a continuous output
- Both versions of the IR sensors have a fairly tight focus, especially compared to a sonar.
- The response of the IR sensors (raw voltages) are not linear with distance, and require a lookup-table. The noise profile is not bad (approximately Gaussian) once the data is linearized.

**Sonar sensors**

- Uses a piezoelectric material to send out a pulse of sound. The same material detects the reflection of the pulse and measures the time of flight. Sonars are useful out to 3-4m, but not as much use within 0.25m because there has to be sufficient time between sending the pulse and sensing its return for the vibrations to damp down.
- Senses within a fairly broad cone, and anything within the cone generates a signal.
- The major difficulty with sonars is reflection–at oblique angles the sound doesn't come back. Simple noise is not as much of a problem as noise due too reflections, which produce the equivalent of no obstacle. The noise profile is approximately Gaussian with no reflection, but reflection adds a shot noise component where some readings are extreme.
- Sonars are much more effective in underwater environments, and they are a primary method of mapping and localizing for marine robots.

**Laser sensors**

- Laser + camera - scan a laser across a scene and use a camera with a filter tuned to the laser's frequency to track where the laser reflects and use triangulation to estimate distance. The camera can be 2D or 3D, and must be calibrated with respect to the laser's position. Maintaining calibration can be difficult on moving vehicles, but this is generally the cheapest option for creating a simple laser scanner and can be implemented with a line laser and cheap camera.
- Laser range finder: 2D scan - time of flight to estimate distance using a pulsed signal. Time of flight is generally measured using the phase change of the signal.
- Laser range finder: 2D scan + PT mount - put a 2D scanner on a pan tilt mount and swing it around. Provides 3D information.
- Laser range finder: 3D scan - full 3D scan - time of flight using a 3D array. Enables 3D information with temporal coherence.

## 3.2 Proprioception

- Accelerometers - measure accelerations by using the phase response of lasers on a chip. Modern accelerometers are small and sufficiently accurate to permit double integration to estimate distance traveled.
- Gyroscopes - measure angular rotation, generally in three axes. Modern gyroscopes are laser-based rather than physical mechanisms.
- Compass - measures orientation relative to the local magnetic field. Electronic compasses have no moving parts and use the Hall effect to measure the local field.
- GPS - uses the Global Positioning Satellites to estimate 3D position and orientation. Modern GPS devices have accuracies of centimeters, but still may have issues with jumps from time to time.
- Computational load - robot knows when it is running into computational limits.
- Internal temperature sensors - used to manage power consumption or recognize dangerous situations.
- Battery life - robot knows when it needs to recharge itself.

## 3.3 Environmental Sensors

- Motion sensors - designed to respond to motion in the sensor field. Motion sensors can be IR sensors or temperature sensors looking for fluctuations in a small field of sensors.
- Temperature sensors - IR sensors that return a signal responsive to absolute temperature levels. These are used often in robotic USAR situations when looking for victims, who may be invisible to regular visual sensors and may not be moving.
- $CO^2$ sensors - measure the concentration of $CO^2$ gas, also commonly used in robotic USAR situations.
- Chemical sensors - there are a variety of special-purpose chemical sensors that are able to detect small concentrations of a particular chemical. Some researchers, inspired in part by ants, have looked into using them for laying down trails or assisting in searching an area by marking what locations have already been searched. Chemical sensors are a primary sensor for land mine detection robots.

## 3.4 Cameras

- Sensor technology
  - CCD - Charge-Coupled Device [CCD] cameras use a process that captures photons/electrons in silicon wells. The charge can be moved from well to well in order to read it off as a row. CCD cameras used to have lower noise and higher dynamic range because of their slightly larger well sizes.
  - CMOS - Complementary Metal Oxide Semiconductor [CMOS] cameras use a standard chip manufacturing process and generally integrate the electronics for each pixel next to the sensor well. The easy integration of electronics and sensor enables manufacturers to do some interesting on-chip processing.

- Greyscale - greyscale cameras have no filters over the sensor except possibly an infrared cutoff filter to increase the camera's sensitivity to visible light. Silicon is sensitive to IR wavelengths, so without the IR cutoff filter the camera tends to be saturated with the IR signal in natural light. It also possible, however, to remove the IR cutoff filter and block visible light, producing a mid-quality IR camera.
- Color - single CCD/CMOS color cameras use an array of red, green and blue [RGB] filters laid out in a grid called a Bayer pattern. Each set of four pixels has one red, one blue, and two green sensors. Human vision is most sensitive to wavelengths in the green channel, which is why they use a second green sensor to get a higher signal to noise ratio. Either the camera or software on the camera interpolates the local information in each color channel to produce an RGB value at each pixel. Two-thirds of the data in an image from a single sensor color camera, therefore, is a result of interpolation, not measurement. For this reason, greyscale cameras are often used where dynamic range, low noise, or low-light sensitivity is critical. A greyscale camera devotes four times as much pixel area to each measurement compared to a color camera, and integrates photons from across the visible spectrum. Note that in higher end cameras, the RAW format data holds only the actual sensor measurements, permitting customized software to implement the interpolation–or not–as desired.
- Multi-sensor - some cameras use three sensors instead of one, using a prism to split the incoming light into three parts and putting a different color filter over each sensor. A 3 sensor color camera generally has lower noise that a single CCD camera, even after splitting the incoming light, because each sensor is a greyscale sensor and has a much larger surface area than the pixels on a color camera.

- Image capture methods
  - Line scan - a line scan camera is a single linear array of sensors. Line scan cameras are often used in industrial processes on assembly lines for products like paper. They have also been used for high resolution aerial mapping. The benefit of a line scan camera is the data rate. Since it has only a single row of pixels, you can read out the pixels at a high frame rate and transfer the data easily.
  - Frame scan - a frame scan camera has a 2D pixel array with the odd rows assigned to frame 0 and the even rows assigned to frame 1. The camera reads out the frames in alternating order and packs them together in a single image. The scene, therefore, is imaged at 60Hz at half resolution or 30Hz at full resolution. One result of reading the frames at different times is that a moving object can appear in two places in the scene. Frame scan chips are rarely used now, especially with the advent of CMOS camera sensors. Some older stereo camera systems would take a half-height image from each camera, capture simultaneously, and put them together as the two frames of a single image.
  - Progressive scan - a progressive scan camera grabs the entire frame simultaneously. Most current camera systems are progressive scan. It is possible to get much higher frame rates from cameras, but most cameras have a frame rate of no more than 30Hz.
- Sensor geometries
  - Single camera - most camera systems are a single camera, possibly on a pan-tilt mount and with zoom capabilities.
  - Binary stereo - binary stereo requires two synchronized cameras in a fixed relative orientation. With proper calibration and software, binary stereo cameras can provide both reflectance and distance information. Depending on the scene, the distance information may be sparse or dense. Scenes with little texture or variation are difficult for stereo algorithms to analyze.
  - Multi-view stereo - binary stereo provides only two views of a scene. Adding more cameras provides more robustness to depth estimation, and there are simple algorithms for combining the information from many cameras. A common configuration for multiple cameras is an L.
  - Panoramic cameras - Many robot systems have used panoramic cameras, which combine a regular camera with a parabolic mirror. The camera points straight up at the parabolic mirror, which reflects information from $360^\circ$. A single image, therefore, contains an entire panoramic image. Objects closer to the robot are closer to the center of the image, while the horizon sits on a circle on the outer edge of the image. Given the geometry is known, it is possible to re-sample the circular image into a more traditional rectangular image.

# 4 Motion, Kinematics, and Control

Representation of robot state: where is the robot in the world (pick an origin)?

- Linear robot: $S = (x)$
- Planar mobile robot: $S = (x, y, \theta)$
- Mobile robot in natural terrain: $S = (x, y, z, \theta, \phi, \gamma)$
- Aerial/underwater robot: $S = (x, y, z, \theta, \phi, \gamma)$
- Jointed mobile robot (e.g. humanoid) on a plane: $S = (x, y, \theta, \vec{\Theta})$

Coordinate systems

- World - arbitrary definition of the origin, generally selected to be convenient (e.g. corner of a room).
- Robot - attached to the robot, generally defined as X forward, Y left, and Z up, specified relative to the world coordinate system.
- Camera - attached to the camera, specified relative to the robot.
- Laser - attached to the laser, specified relative to the robot
- Sonar - attached to each sonar sensor, specified relative to the robot
- Manipulator - attached to the end effector of the manipulator, transformations link each joint of the manipulator in a chain to the robot coordinate system

## 4.1 Transformations

If we describe a robot's pose as a vector, then we can describe both translations and rotations using matrices. Consider, for example, an oriented robot moving on a plane. The position of the robot is given by two parameters $(x, y)$ and the pose of the robot is given by $\theta$.

A translation by $(t_x, t_y)$ is defined by the matrix (2).

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

If we multiply the robot's pose by the translation matrix, the robot's pose changes by adding $t_x$ to the $x$ position and $t_y$ to $y$ position.

$$\begin{bmatrix} x_0 + t_x \\ y_0 + t_y \\ \theta \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ \theta \end{bmatrix} \tag{3}$$

A rotation by an angle $\theta$ can also be represented as a matrix as in (4).

$$R_Z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Rotation matrices are useful for relating the robot's local coordination system, which moves as the robot turns, to the global coordinate system, which is fixed. For a planar robot, the robot's local coordinate system is defined by its current orientation, $\theta$. Motion in the robot's local coordinate system can be described in the global coordinate system using a simple rotation matrix to transform the motion from a local velocity $(\dot{x}_l, \dot{y}_l, \dot{\theta})$ to a global velocity $(\dot{x}_g, \dot{y}_g, \dot{\theta})$. The robot's current orientation $\theta$ defines the transformation. To describe local coordinates in a global coordinate system, we need to rotate by the rotation matrix $R_Z(\theta)$.

$$\begin{aligned} \begin{bmatrix} \dot{x}_g \\ \dot{y}_g \\ \dot{\theta} \end{bmatrix} &= R_Z(\theta) \begin{bmatrix} \dot{x}_l \\ \dot{y}_l \\ \dot{\theta} \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_l \\ \dot{y}_l \\ \dot{\theta} \end{bmatrix} \end{aligned} \tag{5}$$

Using (5) we know how to update the robot's global position given its current velocity and heading. If the robot currently has a pose defined by $\theta$, then motion in the local coordinate system, specified by $(\dot{x}_l, \dot{y}_l)$ over a small time step $\Delta t$ will change the robot's global position $(x_g, y_g)$ by the amount $\Delta t(\dot{x}_g, \dot{y}_g)$.

Note that Siegwart and Nourbakhsh give a different definition of a rotation matrix that is the transpose of the one above. For rotation according to the right-hand rule, the proper matrix has the negative sine term in the first row, not the second. Consider, for example, rotating the vector $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^t$ by $\pi/2$, which should result in $\begin{bmatrix} -1 & 1 & 1 \end{bmatrix}^t$.

$$\begin{aligned} \begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix} &= R(\pi/2) \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\pi/2 & -\sin\pi/2 & 0 \\ \sin\pi/2 & \cos\pi/2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} \end{aligned} \tag{6}$$

Transposing the rotation matrix would result in the vector $\begin{bmatrix} 1 & -1 & 1 \end{bmatrix}^t$, which is rotation according to the left-hand rule.

## 4.2 Kinematics

Kinematics is the study of how physical systems can move. One way to view kinematics is, given a robot's pose, what are all the motions it can execute instantaneously. Another way to think about it is that the kinematics of an object describe all the possible poses the object can take given a starting location. Kinematics can be used to analyze both questions.

The movement, or current pose of any physical system can be described as a set of parameters. Each parameter describes either rotational motion around an axis or linear motion along an axis. By combining rotations and translations we can define all of the complex motions executable by a robot and estimate where a robot will be after the motion.

All robots have a kinematic description. In some cases, such as a planar robot with swedish wheels, the kinematics are simple: the robot is able to move in any direction instantaneously. In other cases, such as on a car, the kinematics tell us the possible arcs along which the car can travel. In a plane with no obstacles, any robot capable of moving in two independent directions can achieve any configuration, although different robots have to follow different paths. In a plane with obstacles, that is not necessarily the case.

Kinematics are especially important in path planning; the kinematics define the possible actions available to the planner.

### 4.2.1 Differential drive kinematics

Consider a differential drive robot with two standard wheels of radius $r$, each separately controllable. The center of the robot, or the origin of the robot is the point $P$. Let the wheels each be a distance $l$ from $P$. The angular velocity (radians/s) of the two wheels is given by $(\phi_1, \phi_2)$. When a wheel completes a full turn–$2\pi$ radians–then the wheel has traveled $2\pi r$ units. Therefore, it moves a distance $r$ per radian of rotation and the velocity of the wheel relative to the plane is given by $r\phi$ for a rotational velocity of $\phi$.

The equation of motion of the central point P due to turning wheel A can be expressed as a rotation about the location of wheel B. Likewise, motion due to turning wheel B can be expressed as a rotation about the location of wheel A. Motion by both wheels adds some degree of translation. If both wheels turn the same amount, the robot's orientation does not change. If the wheels turn equal, but opposite amounts, the robot's position does not change, but its orientation does.

One way to visualize the kinematics of a differential drive is to define the space of possible rotation centers. If both wheels are rotating in equal, but opposite directions the rotation center is the middle of the robot. If the two wheels are turning simultaneously forward, the center of rotation is at infinity along the y-axis. If they turn simultaneously backward, the center of rotation is at infinity along the negative y-axis. If the two wheels are moving differently, then the center of rotation lies somewhere on a line connecting the center of the robot and the points at $\pm\infty$ along the y-axis.

The kinematic description of a wheeled robot's position over time is complex to calculate because it depends upon the relative motion of the two wheels over time. At every instant the robot is rotating around some location along the axis of rotation of the two wheels, but the instantaneous center of rotation depends upon the relative speed of the two wheels and changes over time. Because the motion, and therefore the final position, of the robot depends upon the instantaneous center of rotation, the ordering in time of the wheel rotations is important.

A simple exercise demonstrates that the final position of the robot does depend upon the particular path

taken by the robot. Consider, for example, a sequence of operations where the robot first rotates the right wheel so the point of contact follows a path of length $d$, then rotates the left wheel by the same amount. The first rotation will move the origin of the robot along a circle of radius $l$ around the left wheel contact. The second rotation by the left wheel will rotate the robot around the new location of the right wheel. The end result will be the robot shifting to the left and forward of its starting location. By contrast, executing the rotations in the opposite order will shift the robot to the right and forward of its starting location. Executing the motion of the wheels simultaneously would move the robot forward by a distance d.

We can obtain the new location of the robot after a single rotation by using the translation and rotations matrices to rotate the origin about the left wheel contact. To rotate the plane about an arbitrary point, we use a a three-step method:

1. Translate the center of rotation to the origin using $T(-c_x, -c_y)$.
2. Rotate about the origin by $\theta$ using $R_Z(\theta)$.
3. Translate the center of rotation back to its original position using $T(c_x, c_y)$.

$$\begin{aligned} R_c(\theta) &= T(c_x, c_y) R_Z(\theta) T(-c_x, -c_y) \\ &= \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & c_x(1-\cos\theta) + c_y \sin\theta \\ \sin\theta & \cos\theta & c_y(1-\cos\theta) + c_x \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{7}$$

Therefore, if we multiply the origin $(0, 0)$ by $R_c(\theta)$ using a pivot point of $(c_x, c_y) = (0, l)$, then we get the location of the robot after rotating the left wheel.

$$\begin{bmatrix} l\sin\theta \\ l(1-\cos\theta) \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & l\sin\theta \\ \sin\theta & \cos\theta & l(1-\cos\theta) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{8}$$

Given the location of the robot‘s origin after the motion of the right wheel, we can use the same process to rotate about the right wheel as the left wheel rotates. However, the right wheel has moved from its original location, so the pivot point has changed. Therefore, the second rotation will move the robot in a circular arc centered on the new location of the right wheel. The end result will put the robot to the left and forward of its original location.

Because of the path dependence of the motion of a differential drive a planar mobile robot, it is actually easier to consider the relationship between wheel velocities and robot velocities than wheel position and robot position.

Instantaneous velocity by one wheel gives the robot's origin a velocity along the $x$-axis (which points forward). As the wheels are equidistant from the robot‘s origin, the origin moves by half the velocity of the wheel. A wheel of radius $r$ has a translational velocity along the ground that is equal to $r\phi_i$, and the contributions of two wheels add together. Therefore, the translation of the robot along the $x$-axis is given by (9).

$$\dot{x} = \frac{r\phi_1}{2} + \frac{r\phi_2}{2} \tag{9}$$

Instantaneous velocity in the $y$-axis is impossible because a standard wheel rotates about the $y$-axis.

$$\dot{y} = 0 \tag{10}$$

To obtain the rotational velocity of the robot $\dot{\theta}$, consider that rotation of one wheel will cause the robot's center to move in a circle of radius $l$, and the outer wheel in a circle of radius $2l$. The robot will complete a circle, which is a rotation of $2\pi$ radians, in the time $t$ it takes for the outer wheel to go a distance of $\pi 4l$. The distance traveled by the wheel given the wheel's rotational velocity $\phi_i$ is given by (11).

$$d = tr\phi_i \tag{11}$$

The time taken for the robot to traverse a complete circle is given by (12).

$$t = \frac{\pi 4l}{r\phi_i} \tag{12}$$

The angular velocity is given by $2\pi/t$, which gives us the angular velocity due to a single wheel.

$$\dot{\theta} = \frac{r\phi_i}{2l} \tag{13}$$

The contributions of each wheel add together, but cause opposite rotations. The total instantaneous angular velocity is given by (14).

$$\dot{\theta} = \frac{r\phi_1}{2l} - \frac{r\phi_2}{2l} \tag{14}$$

The above relationships exist in the robot's local coordinate system, so to transform them to the global coordinate system we need to multiply the local change by the inverse of the rotation matrix defined by the robot's current heading. The full model for a differential drive robot is given by (15).

$$\begin{bmatrix} \dot{x}_g \\ \dot{y}_g \\ \dot{\theta} \end{bmatrix} = R(\theta) \begin{bmatrix} \frac{r\phi_1}{2} + \frac{r\phi_2}{2} \\ 0 \\ \frac{r\phi_1}{2l} - \frac{r\phi_2}{2l} \end{bmatrix} \tag{15}$$

If we extract out the wheel velocities (joint velocities), then we can rewrite (15) as a relationship between the wheel velocities and the global translational and rotational velocities of the robot.

$$\begin{bmatrix} \dot{x}_g \\ \dot{y}_g \\ \dot{\theta} \end{bmatrix} = R(\theta) \begin{bmatrix} \frac{r}{2} & \frac{r}{2} & 0 \\ 0 & 0 & 1 \\ \frac{r}{2l} & -\frac{r}{2l} & 0 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ 0 \end{bmatrix} \tag{16}$$

Note that (16) explicitly gives what's called the Jacobian for the differential drive robot. The Jacobian defines the relationship between the joint angular velocities and the robot state velocities. Note that the Jacobian is dependent upon the current robot configuration ($\theta$) through the rotation matrix. Therefore, it must be computed dynamically at every time step. By separating the process into two parts, however, only the inverse of the rotation matrix is required, and the inverse of a rotation matrix is simply its transpose.

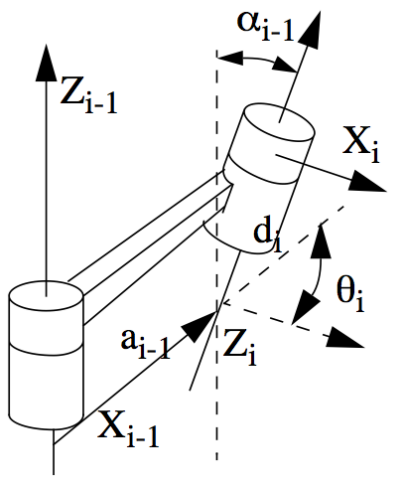



Figure 1: Two joints connected by a rigid linkage

### 4.2.2 Joint Kinematics

For linkages that can be represented as a rigid bar connecting two joints, we can generate a generic matrix transformation that applies equally well to linear and rotational joints. The transformation describes motion in one joint's coordinate system with respect to the other. Such a representation is important for robot manipulators, which are increasingly being used on mobile robot platforms for manipulation tasks.

Consider figure 1, which shows two joints connected by a rigid link.

- The motion of the joint is always defined to be around or along the local $z$-axis of the joint.
- The local $x$-axis of the joint always points towards the next joint.
- The angle between axis $Z_{i-1}$ and $Z_i$ is defined as $\alpha_{i-1}$.
- The distance along axis $X_{i-1}$ between axis $Z_{i-1}$ and $Z_i$ is defined as $a_{i-1}$.
- The angle around axis $Z_i$ between axis $X_{i-1}$ and $X_i$ is defined as $\theta_i$.
- The distance along axis $Z_i$ between $X_{i-1}$ and $X_i$ is defined as $d_i$.

The active motion of the $i$th joint is defined by either $\theta_i$ or $d_i$, while the linkage connecting joints $i-1$ and $i$ is defined by $\alpha_{i-1}$ and $a_{i-1}$. Using these definitions, we can define a transformation ${}^{i-1}_{i}T$ that transforms motion in frame of reference $i$ into motion in frame of reference $i-1$. The order or operation of transformation matrices is right to left given a column vector and right multiplication.

$$ {}^{i-1}_{i}T = R_X(\alpha_{i-1})T(a_{i-1},0,0)R_Z(\theta_i)T(0,0,d_i) \tag{17} $$

Recursive use of the kinematic relationship in (17) defines the position of the end effector at the last joint in the coordinate system of the first joint as a function of the joint parameters $(\theta_1,\ldots,\theta_N)$. The function $f(\theta_1,\ldots,\theta_N)$ is defined a the forward kinematic equation, and it tells us the position of the last linkage given the joint angles. However, it does not tell us how the end effector will move if, for example, we were to change a single joint angle $\theta_i$. Different joint angles will cause different types of motion.

**Example: Robonova arm**

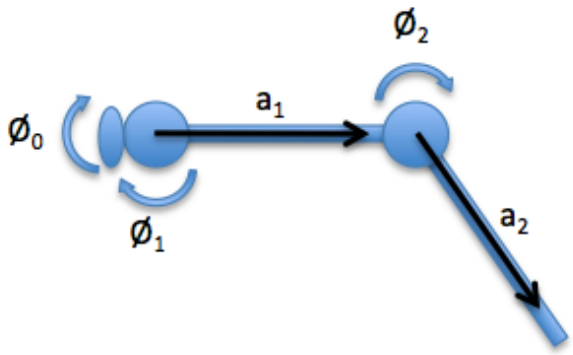



Figure 2: Diagram of Robonova arm joints

The Robonova arm has three joints, shown in figure 2: two at the shoulder and one at the elbow. We can organize the parameters of the arm, as in table 1, showing $\alpha_{i-1}, a_{i-1}, \theta_i$, and $d_i$ for each joint. Note that to get between the first and second joints, we have to insert a rigid joint to move the x-axis into the proper orientation. The last transformation gives the frame of reference for the end effector.

| Joint | $\alpha_{i-1}$ | $a_{i-1}$ | $\theta_i$ | $d_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | $\theta_0$ | 0 |
| - | $\pi/2$ | 0 | $\pi/2$ | 0 |
| 1 | $a_0$ | 0 | $\theta_1$ | 0 |
| 2 | 0 | $a_1$ | $\theta_2$ | 0 |
| 3 | 0 | $a_2$ | 0 | 0 |

Table 1: Joint parameters for the robonova arm.

A complete joint transformation between two frames is given by (18).

$$ {}^{i-1}_{i}T = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i \cos\alpha_{i-1} & \cos\theta_i \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -\sin\alpha_{i-1} d_i \\ \sin\theta_i \sin\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & -\cos\alpha_{i-1} & -\cos\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{18} $$

Plugging the values from the table into (18) gives us the each of the individual transformations from frame to frame. The multiplication of all of the frames gives us the complete transformation. The complete transformation matrix is an exact description of the end effector position given the joint angles. Note that most of the transformations are simple, as many of the parameters for each joint are zero.

### 4.2.3 Jacobian

In order to identify how the end effector will respond to a particular set of joint motions, or velocities, we need to calculate the Jacobian, which is a matrix of partial derivatives of the kinematic equation for each axis with respect to each joint angle. Consider, for example, a planar robot where the position of the end effector is defined by two functions $f_x(\theta_1, \ldots, \theta_N)$ and $f_y(\theta_1, \ldots, \theta_N)$, which define its $x$ and $y$ positions as a function of the joint angles.

The Jacobian of this system would be defined as in (19).

$$J(\Theta) = \begin{bmatrix} \frac{\partial f_x(\theta_1,\ldots,\theta_N)}{\partial \theta_1} & \cdots & \frac{\partial f_x(\theta_1,\ldots,\theta_N)}{\partial \theta_N} \\ \frac{\partial f_y(\theta_1,\ldots,\theta_N)}{\partial \theta_1} & \cdots & \frac{\partial f_y(\theta_1,\ldots,\theta_N)}{\partial \theta_N} \end{bmatrix} \tag{19}$$

The utility of the Jacobian is that if we have a set of joint velocities $(\dot{\theta_1}, \ldots, \dot{\theta_N})$, then we can calculate the instantaneous change, or the velocity of the end effector, which tells us how it moves in response to the joint angle velocities.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = J(\Theta) \begin{bmatrix} \dot{\theta_1} \\ \cdots \\ \dot{\theta_N} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_x(\theta_1,\ldots,\theta_N)}{\partial \theta_1} & \cdots & \frac{\partial f_x(\theta_1,\ldots,\theta_N)}{\partial \theta_N} \\ \frac{\partial f_y(\theta_1,\ldots,\theta_N)}{\partial \theta_1} & \cdots & \frac{\partial f_y(\theta_1,\ldots,\theta_N)}{\partial \theta_N} \end{bmatrix} \begin{bmatrix} \dot{\theta_1} \\ \cdots \\ \dot{\theta_N} \end{bmatrix} \tag{20}$$

Better yet, if we take the inverse of the Jacobian, $J^{-1}(\Theta)$, then we can identify the joint velocities necessary to achieve a certain motion of the end effector.

$$\begin{bmatrix} \dot{\theta_1} \\ \cdots \\ \dot{\theta_N} \end{bmatrix} = J^{-1}(\Theta) \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{21}$$

One of the uses for the Jacobian is identifying the sequence of joint angle velocities required to make an end effector follow a specific path. The technique is commonly used in robot manipulators. The difficulty with the method is that the forward kinematic equation is dependent upon the joint angles. Therefore, the Jacobian, which is based on the derivatives of the forward kinematic equation, changes as the joint angles change. In order to obtain the correct relationship between the joint angle and end-effector velocities, at each time step the current Jacobian matrix must be calculated and inverted. While this is no longer a computational issue given current processing power, the Jacobian is not guaranteed to be invertible, so care needs to be taken to handle special cases.

## 4.3 Robot Control

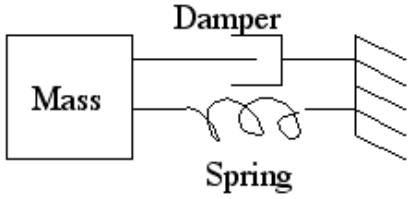
There are many approaches to robot control, many of which are based on sophisticated mathematics. There are also very simple approaches that use straightforward rules designed by programmers. In general, the more you can model your robot's response to inputs and use those models to define control strategies, the more sophisticated and robust your robot's behavior will be.

There are also many levels of robot control. At a very low level you want something on your robot to be looking out for obstacles and responding appropriately, which may include stopping as quickly as possible. At a high level, you may want your robot to follow walls, explore a maze, wander, or track through a series of waypoints.

## 4.4 Basic Control Theory

**Open loop control** is when you control something, like a robot, without any feedback from internal or external sensors to indicate how well you have achieved your goal. Its like walking down a corridor with your eyes close and hands behind your back: eventually you run into the wall. Using odometry to control a robot is insufficient over time, as errors in the position estimate grow without bound.

**Feedback control** uses internal or external sensing to determine the error between the actual and desired state of the device/robot. What to do at each instant of time is determined by the error and the control law. Most real systems are run as closed loop control systems, because otherwise small errors in motion eventually lead to big errors in position.




We can generally express a model for a physical system as a differential equation

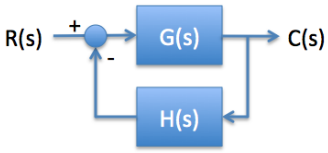
- Mass: 2nd order term, matched with acceleration
- Damper: 1st order term, matched with velocity
- Spring: 0th order term, matched with position

$$m\ddot{x} + b\dot{x} + kx = f(t) \tag{22}$$

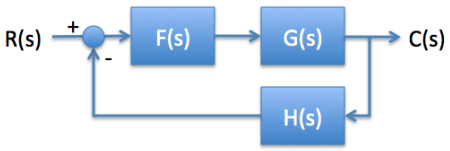
Motors are often modeled as masses with damping.

When designing a feedback loop, the feedback term is almost always negative. Positive feedback is generally bad because it drives the system to increase the control parameter. The transfer function of a closed loop system, which defines how the output is related to the input, is generally expressed as a fraction, where the terms describing the various parts of the system are given as their Laplace transform. The Laplace transform


(a) Open loop control




(b) Closed loop control




(c) Closed loop control

is a method of transforming differential equations from the time domain to the frequency domain, which often makes analysis simpler. A second order system, such as above, would be represented as in (23).

$$ms^2 + bs + k = F(s) \tag{23}$$

Given the Laplace representation of the parts of the control systems, we can write the open and closed loop feedback control as in (24) and (25), respectively.

$$C(s) = G(s)R(s) \tag{24}$$

$$C(s) = \frac{F(s)G(s)}{1 + F(s)G(s)H(s)}R(s) \tag{25}$$

(25) defines how the output $C(s)$ is related to the input signal $R(s)$.

- H(s) is the feedback function that is part of the control software.
- F(s) is a compensator function that is part of the control software.
- G(s) is the plant, or the robot's mechanical system and is described as a differential equation.

As an example, H might be the identity (1), and F might be a simple constant (P), in which case the transfer function looks like (26).

$$C(s) = \frac{PG(s)}{1 + PG(s)}R(s) \tag{26}$$

It is possible, given an accurate description of the plant and the control system, to demonstrate the expected behavior of the system as a whole given the feedback and compensator functions. An analysis of the feedback equations also shows us how different control parameters will affect the system. In general, for a given criterion, we can optimize the system performance by adjusting the control system parameters and form. Common criteria for optimization include: time to achieve the goal, and the degree of overshoot. Overshoot occurs when the control system goes past the desired goal and has to come back to it from the other side. A small amount of overshoot is often permitted because it speeds up the system's initial response to a control signal.

Without specific knowledge of the plant, it can be difficult to identify appropriate control systems. In general, though, if the plant is well-behaved, then there are some common approaches to control that provide acceptable performance given a small amount of experimentation.

### 4.4.1 Proportional control

The simplest control law is proportional control. Using proportional control, the control signal is proportional to the error between the desired and actual system outputs. A constant $k_p$, determines the speed and character of the system's response to a change in inputs.

In terms of the feedback loop diagram, Proportional control means setting $F = k_p$ and $H = 1$.

There are four regimes, or behaviors, in the response of the system to a proportional control law. The four regions are defined by the value of $k_p$ relative to the critical damping factor $k_c$ and the unstable damping factor $k_u$.

- $k_p < k_c$ **overdamped**: the response is a smooth exponential curve to the desired output
- $k_p = k_c$ **critically damped** the response is a smooth exponential curve to the desired output, and is as fast a response as possible without overshooting the desired output.
- $k_c < k_p < k_u$ **underdamped**: the response will overshoot the desired output but eventually settle to the desired output value.
- $k_c > k_u$ **unstable**: the response will never settle to the desired output and will eventually reach the physical limits of the system, possibly with very bad consequences.

The specific values for the constant are dependent upon the particular system and whether it is controlled by a continuous or discrete system.

- When you sample using a discrete system there is a delay between changing the control value and sensing the response of that change (think setting the shower temperature).
- The larger the delay, the smaller changes you need to make

### 4.4.2 Proportional-integral [PI] control

One solution to the problem of sampling or perturbations to the system is to put in a compensator Generally, this means that $F$ is a function of more than just the current value of the error.

- If you put an integrator in $F$, it will drive the error to zero, even with a disturbance, because small errors build up and force a stronger response.
- If there is a step-disturbance in the system this will handle it
- An integral may make the system respond in an underdamped fashion (with oscillations)

```
Error = Reference - Output
Integral = Integral + Error * T  // T = sampling rate
ControlSignal = Kp * Error + Ki * Integral
```

- Now there are two constants, $k_p$ and $k_i$, that are interdependent.
- The bigger they are, the faster the response, but it may introduce oscillations or instability.

### 4.4.3 Proportional-derivative [PD] control

A PI controller will drive long-term perturbations in a system to zero, but P and PI systems do not react to changes in the error, only their current value or a sum of their prior value. In a quickly changing, dynamic environment, we may want to make the control system responsive to the change in the error term, which provides a small amount of predictive performance enhancement. If the error is getting bigger, we can make the response of the control system larger. If the error is getting smaller, we can back it off. The result is proportional-derivative [PD] control.

- If the error is decreasing $E_{i+1} < E_i$ the derivative will slow down the response.
- If the error is increasing $E_{i+1} > E_i$ the derivative will increase the response.

```
Initialization:
LastError = 0

Run loop:
Error = Reference - Output
dError = (Error - lastError) / T  // T = sampling rate
ControlSignal = Kp * Error + Kd * dError
LastError = Error
```

- Just as in PI control there are two constants, $k_p$ and $k_d$, that are interdependent.
- The derivative term can contain significant noise, so it is important to find a $k_d$ that balances response with noise sensitivity.

### 4.4.4 Proportional-integral-derivative [PID] control

Trying to predict where the circuit is going also helps to improve the response of the system. If you add a derivative function to $F$ it helps the system look ahead to see what's coming.

- The combination of the integral and derivative terms lets the control system response to long-term disturbances and short term changes in the error.

```
Error = Reference - Output
Integral = Integral + Error * T
Derivative = (E - E1) / T
E1 = E
ControlSignal = Kp * Error + Ki * Integral + Kd * Derivative
```

- There are many ways to calculate integrals and derivatives that may be more accurate.
- Using more samples gives you a smoother result with less error, but is less responsive.

## 4.5 Reactive Navigation

So far we haven't looked at using sensor information except for odometry. In order to navigate in the real world, however, which is not an infinite plane with no obstacles, we need to be able to react to external sensor information. Reactive navigation is an approach that does not maintain or use significant state information, such as a map. A reactive approach uses rules that connect sensor readings to a behavior. Achieving a location using a control law and odometry is one example of such a reactive rule. Stopping if something is in the way of the robot is another.

Reactive behaviors are extremely useful in robotics because they permit the robot to respond quickly and appropriately to changes in its environment. Almost all robots have reactive behaviors built in to their software at the layer closest to the robot hardware. They are like reflexes that can override instructions from a higher layer in the architecture.

### 4.5.1 Free space following

One straightforward task is **free space following**, where the robot tries to keep moving as quickly as possible without running into obstacles. The idea is to keep the robot moving quickly and as far away as possible from obstacles. So long as the robot has a clear path in front of it, it should just go.

For free space following we need to use the robot's sonars in order to discover where there is free space for the robot to traverse. One problem that arises with sonars is that sometimes a single sonar misses an obstacle because of specular reflection of the sound away from the robot. The problem is usually an issue for surfaces at an oblique angle relative to the sensor. For a robot equipped with a ring of sonars and/or infrared sensors we can reduce errors by grouping the sonars into virtual sensors, where the closest reading of any sonar in the group is the output of the group. If the robot also has IR sensors, we can merge each sonar/IR pair into a single reading and then form the groups.

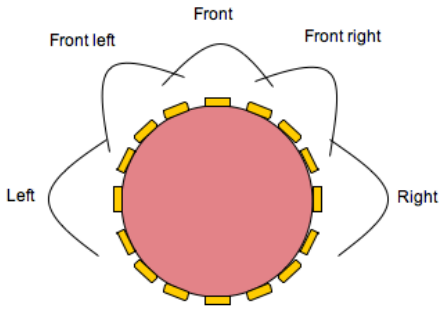



Figure 3: Useful sensor groups for free space following and obstacle avoidance.

Consider a robot with 16 sonars spread out evening in a ring. For free space following, it is useful to make three to five groups out of the front and side sensors as in figure 3. Note that each of the groups overlaps its neighbor by one sonar.

Given the sensor inputs $\{S_{FL}, S_F, S_{FR}\}$, we can formulate a simple free space following algorithm in terms of their inputs. Velocity should be proportional to the amount of free space in front of the robot, while the angular velocity should be proportional to the difference between the left and right groups.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} k_v & 0 & 0 \\ 0 & k_\omega & -k_\omega \end{bmatrix} \begin{bmatrix} S_F \\ S_F L \\ S_F R \end{bmatrix} \tag{27}$$

The following conditions are required for safety, since the robot may run into problems in a tight space.

- The translational and angular velocities must be limited to the maximum safe values.
- If the front sonar reading is below a threshold, the robot should stop and turn in an arbitrary direction.

The method of turning ends up being important. If the robot always turns one direction, it can easily get stuck in loops. Picking a random direction is more challenging, because at each time step the robot is asking the question, "what do I do now". If the robot picks a random direction at each time step, it will sit and oscillate randomly. Therefore, turning in a random direction requires the robot to hold state information. It must remember the direction in which it started turning and continue in that direction until it is able to move forward again. As the robot may move forward a small amount and then need to turn again, the direction picked should stay picked for some amount of time and forward motion.

### 4.6 Wall Following

Wall following is actually a non-trivial problem, especially when one considers doors and corners. For now, we will restrict ourselves to the problem of traversing a perfectly straight infinite wall. The method here is from Bemporad, De Marco, and Tesi, 1998.

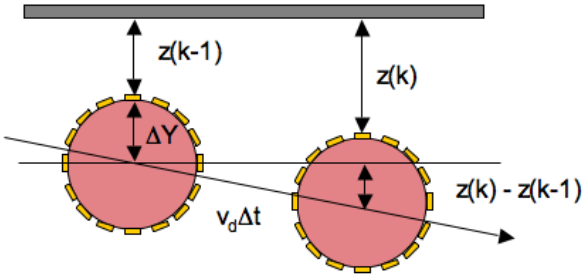



Figure 4: Measuring the orientation of the robot with respect to the wall based on sensor data.

A straight wall can be described by (28). $(x_m, y_m)$ is a representative point on the wall, and $\gamma$ is the orientation of the wall.

$$(x - x_m)\sin\gamma - (y - y_m)\cos\gamma = 0 \tag{28}$$

A simple control law within such an ideal framework is given by (29), where $k$ and $\beta$ are positive scalars, $d$ is the distance of the robot from the wall, and $v_d$ is the desired velocity of the robot. This law works for right-handed wall following. Left-handed wall following would negate the rotational velocity terms.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} v_d \\ -\frac{k(d-d_0)}{v_d \cos(\theta-\gamma)} - \beta \tan(\theta - \gamma) \end{bmatrix} \tag{29}$$

Unfortunately, the ideal control law will not work perfectly due to limitations of the odometry and sonars. We also do not have direct information about the orientation of the wall, $\gamma$. Therefore, we need to use range measurements to the wall within the control loop.

As shown in figure 4, it is straightforward to calculate $\sin(\theta - \gamma)$ based on sequential measurements of the distance between the robot and the wall. If we assume that the robot is approximately following the wall to start with, then we can use the approximation $(\theta - \gamma) = \sin(\theta - \gamma)$. If we let the orientation error $\sigma = \theta - \gamma$, then the angular error term $\sigma$ can be approximated as in (30), where $z(k)$ is the measured distance from the wall. $T_c$ is the time between measurements, and $v_d$ is the forward velocity of the robot.

$$\sigma(k) = \frac{z(k) - z(k-1)}{T_c v_d} \tag{30}$$

Substituting $z(k)$ and $\sigma(k)$ into the previous control law (29), we get (31), where $\Delta_y$ is the distance between the origin of the robot and the sonar sensor.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} v_d \\ -\frac{k(z(k)+\Delta_y - d_0)}{v_d \cos(\sigma(k))} - \beta \tan(\sigma(k)) \end{bmatrix} \tag{31}$$

As noted by Bemporad *et al*, the above control system has four potential problems.

- The odometric and sonar information are completely independent. We could, for example, try to use the sonar information to refine our estimate of the orientation of the robot relative to the wall.
- The estimate of orientation requires differentiation of a noisy quantity, amplifying errors.
- The robot may violate the orientation constraint $\sigma \in [-15°, 15°]$.
- The equation does not take into account velocity constraints.

To overcome these problems, Bemporad *et al* suggest a modified version of the control law (32).

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \alpha v_d \\ -\alpha \left[ \frac{k(z(k)+\Delta_y - d_0)}{v_d} - (\beta_0 + \beta_1 |z(k) + \Delta_y - d_0|) \tan(\hat{\theta} - \gamma) \right] \end{bmatrix} \tag{32}$$

$\beta_0$ and $\beta_1$ are control constants, $\alpha$ is a modifier that keeps $v$ and $w$ within safe ranges, and $\hat{\theta}$ is an estimate of orientation that takes into account both odometry and sonar information over time. In particular, it is the output of a Kalman filter used to estimate orientation relative to the wall. Bemporard et al. also suggest the relationship $\beta_0 = c\beta_1$ where $c \in [0.05, 0.1]$.

In a real robot it is important to remember to limit the maximum speed of the wheels. This can be done by separate limits on the translational and angular velocities, or by a single limit on the rotational velocity of the wheels. In either case, to maintain the integrity of the control law, there must be a single $\alpha$ modifier on the translational and rotational velocities in order to maintain the proper relationship between forward velocity and turning. Given a differential drive robot with radius $e$, wheel radius $r$ and max wheel velocity $\Omega_{max}$, we can write either of the following expressions for $\alpha$.

$$\alpha = \min\left\{1, \frac{v_{max}}{v}, \frac{w_{max}}{w}\right\} \quad \text{or} \quad \alpha = \min\left\{1, \left|\frac{r\Omega_{max}}{v \pm ew}\right|\right\} \tag{33}$$

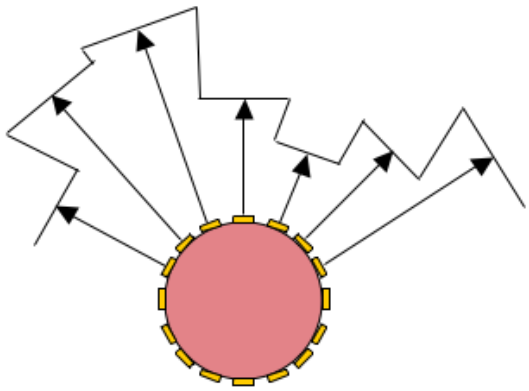


Figure 5: Visualization of representing sensor readings as a set of line segments.

### 4.7 Velocity Space Navigation

Obstacle avoidance is a key robot ability. We don't usually want the robot to stop when it reaches an obstacle. Instead, it should follow some algorithm to go around the obstacle and get itself to the goal. There are two methods of integrating obstacle avoidance into goal achievement: global and local. Global approaches require knowledge of the environment–i.e., a map–and generate a motion plan that avoids obstacles while achieving the goal, if the goal is achievable.

Local approaches examine only the obstacles in their immediate area–obstacles that can be sensed–and try to achieve the goal by heading towards it, turning aside only as required to avoid obstacles. The problem with local approaches is that they may become stuck in places where there is no good local decision for achieving the goal, such as inside a U-shaped obstacle. The benefit of local approaches is that they do not require a complete map and can work effectively in dynamic environments. Global approaches, on the other hand, require replanning when obstacles move.

Using simple control laws to manage reacting to obstacles while also trying to achieve a goal is one approach to the problem of low-level goal achievement. However, such control laws generally do not explicitly take into account robot dynamics. Nor do they maximize the robot‘s ability to move given current knowledge of obstacles. Velocity space planning, also called dynamic window planning, explicitly plans paths that the robot is capable of traversing that move the robot as quickly as possible towards the goal while still maintaining an explicit safety margin (Fox, Burgard and Thrun 1997).

Velocity space computation begins with the assumption that the robot can move locally only along a set of circular trajectories, which limits the search space to two dimensions, defined by $v$ and $\omega$.

The second required component is a map of the local environment generated by the robot‘s sensors. The map could be represented as an **occupancy grid** or as a set of line segments representing the sensor readings, as shown in figure 5. In either case, the maximum admissible velocity along any circular arc is the maximum velocity that permits the robot to stop before colliding with an obstacle while also staying within the safety/power limits of the motors. Note that the size of the robot must be taken into account in these calculations. If all calculations are made assuming the robot is a point at the origin, stopping the center of the robot at the wall means the front of the robot has already collided.

The third required component of the velocity space computation is knowledge of the robot‘s ability to accelerate and deccelerate, $\dot{v}$ and $\dot{\omega}$. Given the robot‘s current velocity $(v, \omega)$, only those velocities within

the range $v - \dot{v}\Delta t < v < v + \dot{v}\Delta t$ and $\omega - \dot{\omega}\Delta t < \omega < \omega + \dot{\omega}\Delta t$ are achievable within the next time step $\Delta t$. Therefore, the total range of the search space is the intersection of the set of admissible velocities and the set of achievable velocities.

The optimal trajectory within the search space is the one that has the highest likelihood of achieving the goal; all trajectories within the search space are safe. One way to define the likelihood, proposed by (Fox, Burgard and Thrun, 1997), is to use the objective function defined in (34).

$$G(v, \omega) = \sigma(\alpha * H(v, \omega) + \beta * D(v, \omega) + \gamma * V(v, \omega)) \tag{34}$$

- $D(v, \omega)$ is the distance to the closest obstacle on the arc defined by $(v, \omega)$. $D$ is maximized when there are no obstacles detected along the arc.
- $H(v, \omega)$ is a function of the difference between the robot‘s alignment and the target direction given that the robot travels along the specified arc for time $\Delta t$. The function is given by $\pi - \theta$, where $\theta$ is the angle of the target point relative to the robot's heading. The maximum value of $H$ occurs when $\theta = 0$ and the robot will end up heading towards the target.
- $V(v, \omega)$ is the maximum velocity possible for the specified arc. $V$ is maximized when the robot can safely move at its maximum velocity.
- $\alpha$ is a constant weight, with a suggested value of 2.0.
- $\beta$ is a constant weight, with a suggested value of 0.2.
- $\gamma$ is a constant weight, with a suggested value of 0.2.
- $\sigma$ is a smoothing function, such as a Gaussian, so that the score of any given point in velocity space is a weighted sum of the scores within a local area.

A simple method of implementing velocity space is to discretize the set of $(v, \omega)$ velocities into a 2D grid representing a range of translational and angular velocities, up to the maximum safe velocities for the robot, where each grid square represents a specific velocity pair $(v_i, \omega_j)$. At each time step, for each grid square within the current dynamic window $\{(v - \dot{v}\Delta t, \omega - \dot{\omega}\Delta t), (v + \dot{v}\Delta t, \omega + \dot{\omega}\Delta t)\}$, calculate the intersection of the arc determined by $(v_i, \omega_j)$ with the local obstacle map and calculate the value of the functions $H$, $D$, and $V$. Finally, execute the smoothing function on the updated area of the velocity space grid and identify the maximum value within the dynamic window. The grid point with the maximum value corresponds to the $(v_i, \omega_j)$ that optimizes the objective function. If the objective function correlates with the likelihood of achieving the goal, then the robot will move in an appropriate direction towards the goal.

# 5 Feature Identification

## 5.1 Laser Features

Given the quality of the laser data, it is possible to identify features in the data and differentiate obstacles in the environment. Visions systems can provide similar kinds information in the form of unique landmarks, but only a stereo camera setup also provides distance information. If you can identify features in the laser readings, then you automatically have their 2D location relative to the robot's position.

- If you can identify a unique obstacle and its location and orientation, that is sufficient information to localize your robot.
- If you can identify two unique obstacles, but not their orientation, that is also sufficient (SAS case).
- If your environment does not have unique obstacles, but has numerous easily identifiable obstacles like corners, you can use a method that integrates readings over time to localize the robot.

The key to all of the above is that you need a map that has the location of all of the obstacles, and you need the location of the obstacles detected by the laser in at least the robot's own coordinate system.

Converting each laser reading $z_i$ into robot coordinates requires knowledge of the sensor and its location on the robot. Let $\theta_i$ be the angle of the ith laser reading and $(d_x, d_y)$ be the offset of the laser from the center of the robot.

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}_R = \begin{bmatrix} z_i \cos(\theta_i) + d_x \\ z_i \sin(\theta_i) + d_y \\ 1 \end{bmatrix} = T(d_L, 0) R(\theta_i) \begin{bmatrix} z_i \\ 0 \\ 1 \end{bmatrix} \tag{35}$$

To find the global location of the laser readings, you need the state of the robot in global coordinates $(x_w, y_w, \theta_w)$.

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}_G = T(x_w, y_w) R(\theta_w) \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}_R \tag{36}$$

Given local robot coordinates, you can fit line segments or circular models to the readings. Note that it is possible to do the same thing with sonar and IR readings. However, given reflection and the wide angle nature of these sensors, it is much more challenging to identify specific features in the environment. Also, fine features like table legs, or even human legs will simply not show up on sonars.

A number of researchers have used the circular pattern of and predictable distance between human legs to identify obstacles that are likely to be people in the robot's environment.

### 5.1.1 Model Fitting

Whether we are trying to identify line segments, circles, or other shapes, there are two critical steps.

- Identify which points are part of the feature.
- Identify the best fit model for the selected points.

The set of points that correspond to a single feature or model are generally called *inliers*. Points that do not correspond to the feature of interest are *outliers*.

Both inlier identification and model estimation are necessary for accurate detection and characterization of features, but each step works best if the other part is known. Including points that are not part of the feature in the model estimation step causes errors in the model, and leaving out points that should be part of the model reduces its accuracy. However, accurately picking which points are inliers requires a model of the feature. The difficulty of the task is generally made more difficult when there are multiple conflicting features within the data, such as having several walls within range of the laser.

There are both deterministic and probabilistic methods of approaching the problem. All of the methods, however, interleave the two tasks and iteratively improve the estimate of both the model and the set of points that it represents.

### 5.1.2 Deterministic Line Estimation

The data provided by a standard 2-D laser range finder scans in a circular arc around the center of the device. Adjacent range readings are likely to correspond to the same surface. Therefore, we can assume that the readings from a single flat surface, such as a wall, will form a contiguous set of readings. This means that the task of identifying a line segment reduces to estimating four parameters: $(z_s, z_e, p_0, p_1)$. The pair $(z_s, z_e)$ are the indices of the start and end measurements of the contiguous segment, and the points $(p_0, p_1)$ represent the best estimate of the line, using a parametric representation of the line (37).

$$\vec{x} = \vec{p_0} + k(\vec{p_1} - \vec{p_0}) \tag{37}$$

Set up properly, $(p_0, p_1)$ will be the endpoints of the line segment, with the parameter $k \in [0, 1]$.

**Incremental Estimation**

Incremental estimation, or region growing, begins with a seed point and incrementally adds adjacent points to the set of inliers, re-estimating the model at each step and testing if the model still represents the set of inliers with a small error. The algorithm is as follows.

1. Begin with a seed measurement $z_s$ and an adjacent measurement $z_e = z_s + 1$.
2. Add one or more adjacent measurements, $z'_e = z_e + \Delta$.
3. Calculate a model $(p'_0, p'_1) = M(z_s, z'_e)$.
4. Estimate the error of the model $e = Q((p_0, p_1))$.
5. If the error is below a threshold $e < E_t$, set $z_e = z'_e$ and repeat from step 2.
6. Else, return the prior model $(p_0, p_1) = M(z_s, z_e)$.

The calling program can always filter results on the number of inliers, discarding line segments with too little support. Incremental estimation can also estimate many different line segments from a single scan. The error threshold $E_t$ determines how precisely the measurements must fit the model; a smaller threshold will generally create more and smaller line segments.

The drawback of incremental estimation is that the model estimate is likely to be slightly corrupted by the last few points included in the model, which are likely to be points belonging to the next segment.

**Splitting**

The opposite approach to growing is splitting. The overall concept is to start with some set of the measurement data, fit a model, and then subdivide the set of points into two sets if the model is not a good fit. Binary subdivision is typical.

1. Start with an empty list of line segments $L$.
2. Begin with all of the measurements in a single model $(z_s, z_e) = (z_0, z_{N-1})$
3. Call the splitting function $S(z_s, z_e, L)$.
   (a) Calculate a model $(p_0, p_1) = M(z_s, z_e)$.
   (b) Estimate the error of the model $e = Q((p_0, p_1))$.
   (c) If the error is below a threshold $e < E_t$, add the model to the list of line segments $L$ and return
   (d) Else, recursively call the splitting function $S(z_s, (z_s + z_e)/2, L)$ and $S((z_s + z_e)/2 + 1, z_e, L)$.
4. Return the set of line segments $L$.

The drawback of a pure splitting approach is that single features are likely to be split into several pieces because of the arbitrary nature of the splitting process. The error threshold controls how much splitting occurs.

**Split and Merge**

Split and merge combines the two methods. The general algorithm is to execute splitting and then go through adjacent segments and merge them if the line estimates are close enough. The merge decision can be based on a comparison of the models, such as the relative angle of the adjacent segments, or on an error metric for a single model estimated from both segments. Split and merge is generally more effective than either method on its own. Using a tight error threshold on the splitting step reduces the likelihood of outliers in any individual segment estimate, while the merge step is faster because it does not have to increment the inlier sets by individual points.

### 5.1.3 Hough Transform

The Hough transform is also a deterministic model estimation method, but it relies on voting to identify likely models within the data. Given an a priori model, such as a line, each measurement votes for all of the models of which it could be a part. The models that get the most votes are the most likely to exist within the data. The Hough transform requires quantizing the model space, creating a bucket for each possible model. The array of buckets collects the votes cast by the individual points. The dimensionality of the Hough transform space is the number of parameters required by the model.

The generic Hough algorithm is as follows.

1. Quantize the parameter space to generate an N-D histogram, or Hough accumulator.
2. Initialize the parameter histogram to zero.
3. For each point in the data set, let it vote for all parameter sets to which it could belong.
4. Identify maxima in the Hough accumulator as likely models.

**Example: Lines**

Line detection works best in a polar representation, with each line represented by its closest approach to the origin and the angle between the x-axis and a ray perpendicular to the line. In (38), $x$ and $y$ are any point on the line.

$$\rho = x \cos\theta y \sin\theta \tag{38}$$

Divide $\theta$ into angular buckets and $\rho$ into half the diagonal size of the sensor range, using the center of the robot as location $(0, 0)$.

Sensor reading votes for each line that could potentially pass through it. Using (38), step through all the values of $\theta$ required by the accumulator and calculate $\rho$ in order to identify the Hough accumulator locations. A simple improvement on this algorithm is to limit each reading's votes to only those lines that are appropriate given the local gradient direction. The local gradient of sensor measurement can be computed by looking at the relative locations of the two adjacent readings.

After filling the Hough accumulator, the maxima in the accumulator should correspond to high probability models. Note that, because of small roundoff errors, a single actual maximum will normally look like more than one high vote bins. Therefore, just selecting the top N bins in the Hough accumulator may generate the same line multiple times.

A reasonable process to follow is given below.

- Use a small box filter (e.g. 3x3) to identify the highest vote location in the accumulator.
- Store the line corresponding to that parameter bin.
- Suppress all of the accumulator values (set them to zero) in an area around the selected bin.
- Repeat N times if you want N lines, or until the best line fit has too few votes.

The same process also works for finding circles, boxes, and ellipses. Note that the Hough accumulator grows exponentially with dimension, so the method works efficiently only for small numbers of parameters (i.e. 2. or 3).

### 5.1.4 RANSAC

RANSAC, which stands for Random Sampling and Consensus, is a probabilistic method of identifying features in noisy data. It is commonly used in situations where there are non-Gaussian outliers in the data set, and it is able to produce reasonable results even when the percentage of outliers is potentially larger than 50%. RANSAC is a hypothesize-and-test model of feature identification. It uses the minimum required data to fit a model and then uses an error threshold to count the number of inliers for that model. Repeating the process many times enables the algorithm to identify the model with the most support: the consensus model.

For RANSAC, the key parameters are:

- An estimate of the percentage of inliers $w$
- The number of data points required to estimate a model $n$
- The desired probability that the algorithm will find a good solution $p$
- The number of iterations to run the process $k$

The probability of randomly picking an inlier point is given by $w$, so the probability of picking all inliers is $w^n$, and the probability of picking at least one outlier is $(1 - w^n)$. The probability of executing $k$ iterations and never picking a good model is $(1 - w^n)^k$. Therefore, if we want to pick a good model with probability $p$, we can identify the number of iterations $k$ as follows.

$$\begin{aligned} (1 - w^n)^k &= (1 - p) \\ k \log(1 - w^n) &= \log(1 - p) \\ k &= \frac{\log(1 - p)}{\log(1 - w^n)} \end{aligned} \tag{39}$$

The full algorithm is given below. The last step in RANSAC, once the algorithm has picked a consensus model, is to use all of the inliers to calculate a final model in a least-squares sense. The assumption is that errors within the inlier group are likely to be Gaussian, in which case a least-squares fit model for the inliers is the optimal model given the training data.

**RANSAC Algorithm**

1. Given: data set with N samples $\{s_i\}$ and a model $R(s)$ with $n$ degrees of freedom.
2. $maxInliers \leftarrow 0$
3. For $k$ iterations, or until $maxInliers > d$
   (a) Randomly pick $n$ points from the N samples
   (b) Fit a model $R_k$ to the $n$ points
   (c) Calculate the how many points $C$ are close enough to the model
   (d) if $C < maxInliers$ then update $maxInliers$ and store $R_k$ in $R_C$
4. Using all of the inliers for the best model, calculate the optimal model $R_o$ in a least squares sense.
5. Return $maxInliers$ and the corresponding model $R_o$

**RANSAC line fitting**

1. Lines require 2 points, so pick 2 points to form a line.

$$\vec{p} = \vec{x}_0 + k(\vec{x}_1 - \vec{x}_0) \tag{40}$$

2. Figure out how close all the other points $p_i$ are to the line
   - Calculate the normalized vector representing the direction of the line.

$$\hat{v} = \frac{\vec{x}_1 - \vec{x}_0}{|\vec{x}_1 - \vec{x}_0|} \tag{41}$$

   - Calculate the vector from point $p_i$ to $\vec{x}_0$

$$\vec{r_i} = \vec{x}_0 - p_i \tag{42}$$

   - Project $r_i$ onto $\hat{v}$ using a dot product to get the distance from $\vec{x}_0$ to the point of closest distance on the line to $p_i$.
   - Multiply the distance by $\hat{v}$ to get the actual closest point and then calculate the distance from there to $p_i$.

$$d_i = |(\vec{x}_0 + (\vec{r_i} \cdot \hat{v})\hat{v}) - p_i| \tag{43}$$

3. Count how many are within the error threshold
4. If the count is better than any model so far, save the model and which points it represents
5. Loop back to step one until the system has executed the desired number of iterations
6. Using the best model and all of its inliers, calculate the least squares line fit for the final model.

# 6 Decision-Making

Every robot must be able to make decisions about what to do next. Many times per second, the robot is scanning its current estimate of state and determining what motor controls to execute, how to interact with its environment, and what sensing mechanisms to activate or deactivate. Maxwell's rule for useful robots is:

> A robot should do about the right thing most of the time. A robot should avoid doing the wrong thing all of the time. (Bruce A. Maxwell, 1999)

There are many ways to handle a robot's decision-making process.

- Reactive - world is state, and the robot is simply reacting to its world state
- Finite state automata [FSA] - the robot has an action state, which changes based on inputs from the world or internal information.
- Case-based reasoning - divide the robot's state, defined as the current robot state and its sensor inputs, into a set of cases and specify the action for each case.
- Fuzzy logic - similar to case-based reasoning, but uses soft categories to define the cases, and the robot will generally combine actions from several cases.
- Probabilistic state machines - Markov Decision Processes [MDP] and other probabilistic methods define action sequences that have a high probability of attaining a goal given the current estimate of the robot's state.
- Reinforcement learning - through training, learn the best course of action to achieve a goal given the current estimate of the world state. Reinforcement learning is similar to case-based reasoning and Markov Decision Processes, but the robot has learned the most appropriate action in each situation.
- Planning - the robot has a knowledge base, a set of goals, and a planning system to select a series of actions to achieve the goal. Planning is usually combined with an FSA to implement the current plan.

State estimation, as in how is the robot situated in the world, is critical to decision-making.

## 6.1 State Machines

State machines, or finite state automata, are a common method of implementing high-level control of a robot. A state machine is defined as follows.

- A set of states. A state indicates the robot's current situation and defines which inputs will affect the robot's next state. A state may also define the actions, or outputs of the robot.
- A set of edges connecting the states. An edge is labeled with the input that causes that edge to be taken. Only edges that change the state of the robot need to be specified; all other inputs cause the state to stay the same. In a finite state automata, the edges leaving a state must be mutually exclusive; given the set of inputs, the choice of which edge to take cannot be ambiguous.
- A set of outputs, or actions. The outputs or actions may be associated with edges or with states. Either method can represent the same set of events in time. An FSM with the outputs or actions defined as part of the states is a Moore type state machine. An FSM with the outputs or actions defined as part of the transitions is a Mealy type state machine.

As an example, consider the Moore machine in figure 6 for a robot that wanders, halting and saying something if it sees a person. The unlabeled edges are default edges, meaning they always go to the next state in the next iteration of the control system. In the Moore machine, the output of the robot is defined by the state. In the "Begin Wander" state, the robot sends a command to the robot to wander. In the "Say Something" state, the robot tells the speech system to say something. In the other two states, the control module does not do anything. In the "Idle" state, if the control module gets a message saying the vision system as seen a face, then it moves to the "Say Something" state.

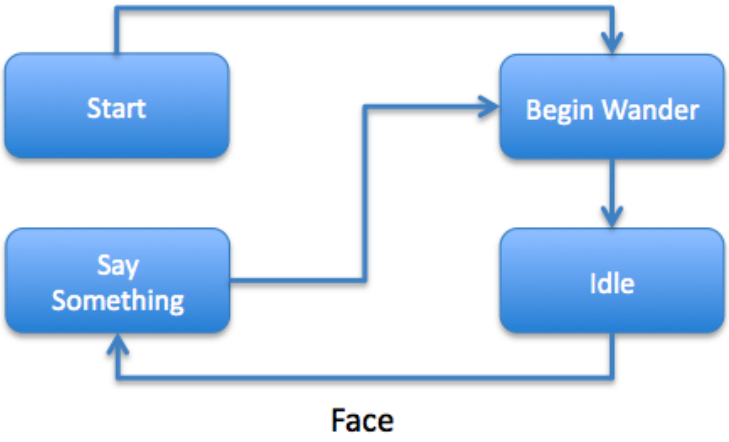



Figure 6: State machine for a simple interactive robot

When encoding a state machine in a program, Moore machines are generally simpler, more centralized, and less prone to errors. The only condition for executing an action is being in a particular state. For a Moore machine, the structure of the robot's control loop may look like the following.

```
// initialize the robot, including its initial state

while( robot_is_running ) {

  // listen for messages
  // messages may cause a state change

  switch( state ) {
  case State_A:
    // do action required in state A
    // evaluate the inputs and determine the next state
    break;
  case State_B:
    // do action required in state B
    // evaluate the inputs and determine the next state
    break;
  ...
  }
}
// shut down the robot
```

After initialization, the control program enters a loop. The actions of the control module, since they are a function only of the state of the robot, will all be defined in the switch statement. An incoming message may change the state of the robot, but no code outside the switch needs to implement any actions.

## 6.2 Case-based reasoning

Traditional case-based reasoning is a method of analyzing a situation and generating output based on prior experiences, or cases. It has its roots in early work in artificial intelligence.

Case-based reasoning, as it has been defined in robotics, is an approach to robot control that micromanages the behavior of the robot. Consider the robot's state to be defined by set of internal state variables and a set of external inputs. The combination of these two sets forms an N-dimensional vector. At any given moment, the robot's state is representable as a point in the N-dimensional space defined by the possible values of each element of the state vector.

A case, in case-based reasoning is a definable subset of the robot's state space. Each case must be exclusive of all other cases. The set of cases must span the robot's state space, although there can always be a catch-all case that handles any point in the state space not included in another case.

The action of the robot is completely defined by its current case. Since the case is a combination of both external inputs and internal variables, case-based reasoning as defined here is theoretically identical to a Mealy type state machine. In the most well-known case-based approach by Nourbakhsh, the programmer manually determined the robot's actions in each case.

Case-based reasoning, as originally implemented, does not allow cases to overlap. There is only one action defined for each case. Therefore, it is up to the programmer to ensure that transitions between cases result in smooth robot motion.

Pretty much the only reason to mention case-based reasoning is that someone implemented it and it worked ok for that robot. The method is brittle, requires the programmer to predict all possible situations, and is difficult to generalize. As such, it serves as a backdrop for introducing the following several methods of decision-making.

## 6.3 Fuzzy Logic

Fuzzy logic is a method for applying mathematical reasoning to semantic descriptions of a task. For example, we could express a set of rules for controlling the speed of a robot as the following set of statements.

- If the robot is far from an obstacle, go fast
- If the robot is medium distance from an obstacle, go a medium speed
- If the robot is close to an obstacle, go slow
- If the robot is very close to an obstacle, stop

There are two relevant variables in the above statements: distance and velocity. There are four adjectives that describe each of these variables. The set $\{far, medium, close, very\ close\}$ is defined on distance, and the set $\{fast, medium, slow, stop\}$ is defined on velocity.

The modifiers on the distance variable are like cases in case-based reasoning. However, in fuzzy logic, the boundaries of these cases are not sharp and will almost always overlap. A particular fuzzy set (e.g. $far$) is defined by a membership function $\mu(d)$ on the variable (e.g. distance). The membership function specifies a relationship between each value in the range of the variable and the fuzzy set. Formally, a fuzzy set $A$ is a

set of ordered pairs. Each pair consists of a value $x$ in the range of the variable of interest, $X$, and its degree of membership $\mu(x)$ in the fuzzy set.

$$A = \{(x, \mu_A(x)) | x \in X\} \tag{44}$$

For example, we might define the membership function for the fuzzy set $far$ as a one-sided trapezoid that starts at 0 at 2m, reaches its maximum value of 1.0 at 3m and stays 1.0 for all larger distances. A distance of 2.5m would be a member of the set $far$ with a value of 0.5. The graph in figure 7 shows four fuzzy sets defined on distance.

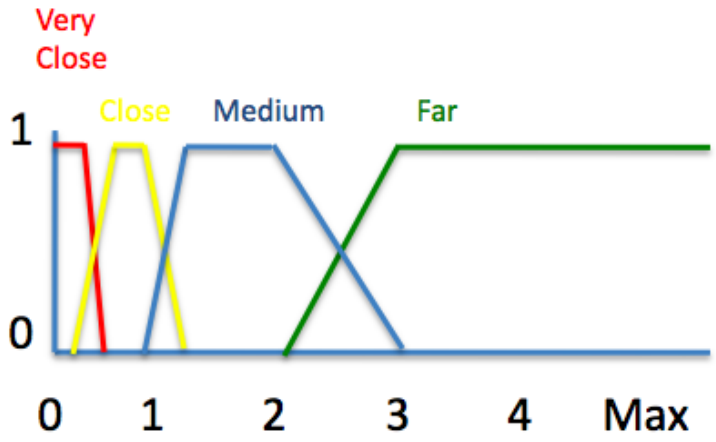



Figure 7: Fuzzy sets $\{far, medium, close, very\ close\}$ defined on distance (m)

The variable velocity has four values $\{fast, medium, slow, stop\}$. These fuzzy sets can be **fuzzy singletons**, single values, or they can be fuzzy sets themselves. A set of reasonable crisp values might be the following.

$$\begin{aligned} fast &= 1.0m/s \\ medium &= 0.5m/s \\ slow &= 0.25m/s \\ stop &= 0.0m/s \end{aligned} \tag{45}$$

If the consequents of the if-then control laws are crisp values, then we can use a simple method of **defuzzification** to implement them.

- Given a value $x$ in the range of the fuzzy variable.
- Apply the membership function for each antecedent fuzzy set in the if-then rule collection to generate a set of membership values for $x$.
- The final velocity should be a weighted sum of the velocities, using the membership functions as weights.

Given the sets defined in figure 7, the output velocities as the robot approaches an obstacle will step down from the maximum velocity at 3m to zero at around 0.2m. The transitions between steps will be linear, since

the sets are defined by trapezoids. A smoother definition of the sets–e.g. Gaussians or sigmoids–would result in smoother transitions.

If the output variables themselves are fuzzy sets, then we have to use more sophisticated methods of defuzzification. One method is to multiply each of the consequent fuzzy sets–defined on velocity, in our example–by the relevant antecedent membership value and then calculate the centroid of the resulting set of functions over the consequent variable (velocity).

ActivMedia, a commercial robotics company, has used fuzzy logic as the basis for its behavior-based control system. Programmers define fuzzy sets on selected variables and then define if-then rules to manage the robot's behavior. Fuzzy logic provides a rigorous framework for converting sets of fuzzy semantic rules into crisp actions, making it more effective, easier to build, and more general than case-based reasoning.

## 6.4 Markov Decision Processes

Uncertainty

- State machines and case-based reasoning build it in through the cleverness of the programmer
- Fuzzy logic handles it by make decision boundaries soft, but it still depends on good state estimates

There are two types of uncertainty. Both types occur in robotics.

- Uncertainty of the world state, which is an issue of perception
- Uncertainty of the results of actions, which is an issue of control

If you robot knows its state, the results of its actions are stochastic (uncertain), but it knows the reward structure of the environment, then you have a situation that can be modeled as a Markov Decision Process [MDP].

- A set of states $S$
- A set of actions $A$
- A reward function $R : (S \times A \rightarrow \Re)$
- A state transition function $T : (S \times A \rightarrow \Pi(S))$, where $\Pi(S)$ is a probability distribution over the states $S$ (state transitions are probabilistic). In other words, for each (state, action) pair, there is a probability (possibly zero) of transitioning to any other state.

The goal of the robot is to take the action at each time step that maximizes rewards. Some questions that arise in evaluating the utility of any particular action include: over what time period should the reward be maximized, and how important are future versus immediate rewards?

There are three models for making decisions

- Fixed-horizon decision-making: the horizon is $k$ steps away, and rewards beyond step $k$ are irrelevant. The horizon may be a particular $k$, in which case the agent resets after k steps, or the horizon can move in time along with the agent, staying $k$ steps ahead.
- Infinite horizon decision-making: the horizon is far away, and the agent must consider rewards far into the future. Infinite horizon decision-making is not particularly realistic, and very difficult to model, as large payoff possibilities can cause strange short-term behavior.

- Discounted infinite-horizon decision-making: while the horizon is infinitely far away, rewards in the future are not as important as rewards now. The discount factor $\gamma < 1$ is an exponential factor on rewards, giving a reward of $R\gamma^k$ for a reward $R$ $k$ steps in the future. Finite-time solutions exist for solving the discounted infinite horizon model, and it generally converges to a stable and optimal control policy.

Given an MDP: How do you figure out what to do now?

- Take the action that maximizes the total current and future reward (given the model)
- Need to look ahead, or pre-calculate what the best action is for each state

The value of a state, $V^*(s)$, is the expected value of moving to that state and experiencing all subsequent possible rewards that come after, discounted by the factor $\gamma$ over time.

$$V^*(s) = \max_{\pi} E\left(\sum_{t=0}^{\inf} \gamma_t r_t\right) \tag{46}$$

Another way of considering the problem is to specify the value of a state as the value of the action that maximizes the reward of a particular state action pair $R(s,a)$ and all the subsequent possible transitions from state $s$ to a subsequent state $s_i$.

$$V^*(s) = \max_{a}\left(R(s,a) + \gamma \sum_{s_i \in S} T(s,a,s_i)V^*(s_i)\right) \tag{47}$$

The latter equation can be written for all states in S, and forms a set of linear equations that determine the optimal value function $V^*$. For the discounted infinite horizon model, $V^*$ can be a stationary policy, which means it doesn't change over time.

An alternative algorithm for computing an approximation to $V^*$ is called value iteration

1. Initialize $V^*(s)$ according to known rewards (most states will have low or no reward)
2. loop until policy is good enough, or converges to a stable state
   (a) loop for all states $s$
      i. loop for all actions $a$
         - $Q(s,a) = R(s,a) + \gamma \sum_{s_i \in S} T(s,a,s_i)V(s_i)$
         - Save $Q(s,a)$ if it is the best so far
      ii. $V(s) = \max Q(s,a)$ over all actions $a$

The basic concept is to **relax** the values in the graph until they start to converge. Relaxation describes a process whereby information propagates through a graph by repeated iteration of a process. Good and bad rewards will propagate backwards through chains of actions that lead to those states.

Value iteration is generally fast for low-dimensional value functions (2 or 3), but becomes computationally difficult with more dimensions. A useful computational speedup is to approximate the central summation over $T(s,a,s_i)$ by taking the max of a probabilistic sampling of $T(s,a,s_i)$. For example, sampling two options and taking the max has performance asymptotic to sampling all options (Mitzenmacher, 2001).

## 6.5 Reinforcement Learning/Q-Learning

What if you don't know the reward structure beforehand, $R(s, a)$? Your robot has the ability to move from state to state, but it has no way to predict what are good choices and what are bad choices. It also may not know the structure of the state transitions or the probabilities of achieving a certain state given an action, $T(s, a, s')$. In the basic reinforcement learning structure, the robot does know its states.

Such a model-free situation is the most fundamental case for a robot, capable of action, trying to learn a task. The robot has to be motivated to explore, in which case it will receive positive or negative rewards, teaching it something about the structure of the state transition probabilities and reward structure. In other words , it is learning the value of each state and each action, $Q(s, a)$.

Q-learning is one algorithm for implementing the concept of reinforcement learning. Let $Q^*(s, a)$ be the expected discounted reinforcement of taking action $a$ in state $s$ and then choosing optimal actions in each subsequent state. One way to represent $Q^*$ is as a table, if the actions and states are discrete.

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s_i \in S} T(s, a, s_i) \max_{a_j} Q^*(s_i, a_j) \tag{48}$$

If we let the robot explore an environment, every time the robot moves from one state to the next it can build an experience tuple $(s_t, a_t, r_t, s_{t+1})$. The experience tuple provides a state $s_t$, an action chosen from that state $a_t$, a subsequent reward $r_t$, and a subsequent next state $s_{t+1}$. Each experience tuple provides a measurement for learning $Q^*$.

The update rule for learning $Q^*$ using an experience tuple is given in (49), where $\gamma$ is the discount parameter, and $\alpha$ is a learning constant that reduces over time as the robot learns the $Q$ function.

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_t + \gamma \max_{a_j} Q_k(s_{t+1}, a_j) - Q_k(s_t, a_t)) \tag{49}$$

Consider the behavior of the learning rule if the reward of the current state is 0 ($r_t = 0$). $Q_k(s_t, a_t)$ is the current expectation of the reward for taking action $a_t$ in state $s_t$. The expression $\gamma \max_{a_j} Q_k(s_{t+1}, a_j)$ is the current expectation of the reward for the optimal sequence of actions in the future. Recall that the experience tuple $(s_t, a_t, r_t, s_{t+1})$ says that the robot has evidence that taking action $a_t$ in state $s_t$ will get it to state $s_{t+1}$. Therefore, the value of $Q(s_t, a_t)$ should be equal to the discounted reward the robot would attain following the optimal future trajectory. If the current estimate of $Q(s_t, a_t)$ is less than the expected future reward, then we want to increase the value of $Q(s_t, a_t)$. If the current estimate of $Q(s_t, a_t)$ is greater than the expected future reward, then we want to decrease the value of $Q(s_t, a_t)$. The learning rule in (49) has exactly this behavior, moderated by the learning parameter $\alpha$.

Note that the learning parameter $\alpha$ is effectively a proportional control constant.

$$\begin{aligned} \Delta Q_{k+1}(s_t, a_t) &= \alpha(r_t + \gamma \max_{a_j} Q_k(s_{t+1}, a_j) - Q_k(s_t, a_t)) \\ &= \alpha(r_t + E) \end{aligned} \tag{50}$$

Q-learning can exhibit exactly the same behavior as a control system, depending upon the value of $\alpha$. If the learning constant is too small, then the system takes a long time to learn. If the learning constant is too large, then the learning system oscillates and takes longer to converge.

Once the robot has sufficiently learned the $Q$ model, it can act greedily and select the optimal action at each state. However, if $Q$ has not converged, then the robot still needs to explore states with lower reward values. Something like simulated annealing–selecting lower reward states with some probability–provide heuristic methods of exploration. However, exploration for Q-learning is still something not theoretically well defined.

Q-learning has problems generalizing over large state and/or action spaces, especially continuous ones. Sparse reward structures, in particular, make it difficult for the robot to learn the quality of states far removed from the rewards.

Q-learning tends to work best in situations with discrete states and actions. In such cases, the $Q^*$ may actually be providing the proper policy well before it reaches convergence, since the correct policy just has to have a higher value than the rest.

The strength of Q-learning is that we need to tell the robot only how well it's doing, not how to do it. It is generally much easier to judge the robot's performance than it is to design the performance itself.

### 6.5.1 Continuous action/state spaces

One of the issues with Q-learning is that the discrete form of the algorithm requires the spaces and actions to be both discrete and finite. Memory limitations also require that the number of spaces and actions not be too large.

Rather than explicitly representing the $Q(s, a)$ function, researchers have tried training a function approximator, such as a neural network, to learn the Q function. A neural network can represent an arbitrary function over the space of possible inputs. The key to successful using an ANN is how the system is trained and the robustness of the training set.

- A neural network is a trainable function approximator.
- The most common method of training a neural network is supervised training, which uses input/output examples to adjust the internal parameters of the network.
- During training, the network must see examples from all parts of the potential input space. If the training set is sparse or has incomplete coverage, the network will not properly learn the function.
- A neural network will learn a smooth function from a set of training examples.

Carreras et. al presented a method for Q-learning using a neural network and applied it to an underwater following task (Carreras, Ridao, and El-Fakdi, "Semi-Online Neural Q-learning for Real-time Robot Learning", IROS 2003). They directly addressed the most significant issues for training the neural network.

- The robot moved around its space, exploring different state/action combinations.
- As the robot moved, it collected training tuples and discovered rewards in the environment.
- The robot kept a set of sufficiently different samples in a training data base, replacing old examples as the robot discovered new similar examples.
- Each time a new experience tuple entered the system they trained the network on the whole set of samples in the database (more data, distributed around the space of possible inputs).
- Once trained, the robot used the current state $s$, its potential actions $\{a_i\}$, and the neural network to identify the $Q(s, a_i)$ pair with the largest reward.

### 6.5.2 Optimization of a Control System

Variations on reinforcement learning are also applicable to learning parameters for motion models. For example, Peter Stone and his students learned fast gaits for the AIBO (Kohl and Stone, "Policy gradient reinforcement learning for quadrupedal locomotion", ICRA 2004). They developed a 12 dimensional model for the motion of one leg of an AIBO robot. Optimizing a 12-dimensional space by hand is challenging, so they used a version of reinforcement learning to learn how to adjust the parameters to achieve an optimal solution.

**Optimization Algorithm**

```
P = InitialPolicy
while !done
  Calculate T random perturbations of P {R1, R2, ..., Rt}
  Evaluate each Ri

  for each dimension n
    Avg(+e, n) = average score of all Ri with a positive perturbation in n
    Avg(0, n) = average score of all Ri with zero perturbation in n
    Avg(-e, n) = average score of all Ri with negative perturbation in n

    if Avg(0, n) > Avg(+e, n) and Avg(0, n) > Avg(-e, n)
      An = 0
    else
      An = Avg(+e, n) - Avg(-e, n)

  A = A / |A| * alpha
  P = P + A
```

The main idea of the algorithm is to sample the space around the current policy P by making T random perturbations of P and using the perturbed policies to estimate how to improve performance (gradient descent). Each perturbation will modify some number of the N parameters. The middle loop looks at each parameter in turn, combining the results depending upon whether the perturbation increases, decreases, or leaves alone that parameter. It sets the change in each parameter according to the conditional test that leaves the parameter alone if the zero case had better average performance than the increase or decrease cases. Finally, the optimal policy moves towards the direction of improvement, mediated by a learning constant. The normalization step and the learning constant (alpha) ensures that the parameters change in a relatively controlled manner.

It is also possible to use other standard optimization algorithms, such as Genetic Algorithms, to explore the parameter spaces of control systems. A typical GA learning setup would create a set of control systems for the first generation, then evaluate each one on the actual robot to determine its fitness. Fitness may be defined as how fast the robot can go, how few obstacles it hits, or other measurable attributes. Based on the fitness, the GA would select the next generation of control systems, preferring those with higher fitness. GA learning in robotics takes a long time and can be physically demanding on the robot, so it is not uncommon for much of the learning to take place in simulation before evaluating the system on a physical robot.

## 6.6 Partially Observable Markov Decision Processes

What if we have to relax the requirement of perfect state knowledge? A robot rarely knows exactly where it is, but it may know with some distribution, or belief. For example, given a topological map, the robot may initially not know where it is on the map. Even if it does know its initial location its sensors may produce false readings about the robot's current state.

With a partially observable Markov decision process, the state of the robot can be sensed, but the sensing is imperfect. In other words, the robot may make an observation that indicates it is in a different state or position. A different way of thinking about it is that each state has a certain probability of producing each observation. Different states have different sets of probabilities for each action. By modeling the world as a POMDP, the robot tries to deduce its current state from a series of observations and the plan its future actions based on the value of the actions to the robot achieving its goal.

One way to view the difference between an MDP and POMDP, is that the state uncertainty is best handled by making the current state estimate a function of all prior observations, rather than a function of just the current observation. Note that POMDPs offer a unified method of combining uncertainty in state and action results, but they are not the only solution to the problem. One could use a Kalman filter or particle filter to generate the best current state estimate given all prior observations and then use an MDP derived policy to pick actions.

The ultimate goal of POMDPs, however, is not deducing the robot's current state, but deducing a deterministic policy the robot should follow to achieve its goal. Like MDPs, we want to know the value of each possible action with respect to achieving the goal, given the belief vector (probability of being in each state).

A POMDP contains the following elements:

- $S$: a finite set of states of the world
- $A$: a finite set of actions
- $T : S \times A \rightarrow \Pi(S)$: A state-transition probability matrix $T(s, a, s')$
- $R : S \times A \rightarrow R$: a reward function $R(s, a)$
- $\Omega$: a set of observations $(o_1, \ldots, o_N)$ the agent can experience
- $O : S \times A \rightarrow \Pi(\Omega)$: observation probability matrix $O(s', a, o)$. $O(s', a, o)$ is the probability of the agent taking action $a$, ending in state $s'$ and observing $o$.

The belief vector is a probability associated with each state. The action policy we want to learn is a function of the belief state, not a function of the actual state (which is not directly observable). Therefore, rather than stating the robot is in state $s_i$, when we use a POMDP we say the robot is in state $s_i$ with probability $\pi_i = p(s_t = s_i|(o_0 \ldots o_t), (a_0, a_t))$. The current belief state is the set of probabilities of being in each of the $M$ states $b = (\pi_0, \ldots, \pi_M)$.

The belief state changes in response to action/observation pairs.

$$\begin{aligned} b'(s') &= P(s'|o, a, b) \\ &= \frac{O(s', a, o) \sum_{s \in S} T(s, a, s') b(s)}{P(o|a, b)} \end{aligned} \tag{51}$$

The new belief state $b'(s')$ is the product of three terms. The first term is the probability of being in state $s'$ after taking action $a$ and seeing observation $o$, which is something we are given (or that the robot has learned previously). The second term is the sum of all the possible transitions from any state $s$ after taking action $a$ that end in state $s'$. The third term is the prior belief state. The denominator is a normalizing factor that guarantees the probabilities of the belief vector sum to 1.

The expected reward function for POMDPs is a function of the crisp reward function $R(s, a)$ and the belief vector.

$$\rho(b, a) = \sum_{s \in S} b(s) R(s, a) \tag{52}$$

The question now, is how to we learn the value of each belief vector? Just like with an MDP, what we want is a policy that lets us do a greedy evaluation of actions to obtain a plan to achieve the maximum reward. Researchers have developed several algorithms for discovering policies using POMDPs. We're not going to cover them here.

# 7 State Estimation

One of the most basic issues in robotics is that nothing is perfect. Sensors return incorrect values or miss objects altogether; imperfections in odometry and wheel slip cause the robot to lose track of where it is; and approximations within the robot control system cause a mismatch between velocity commands and actual robot speed. In other words, there is **uncertainty** between the predicted results of a robot's actions and the actual results in the world.

Another way to phrase the problem is that it is an issue with state estimation: the current state of the robot or the robot's sensors is uncertain. More accurate state estimation permits the robot to adjust its behavior to match its current state. State refers to anything from sonar or IR distance readings to an estimate of a landmark location, to a robot's current position and orientation. State estimation is a fundamental problem in control systems, and there is a significant amount of literature on the issue.

The most successful methods of state estimation work with probabilities. Some methods attempt to maintain a single belief, or estimate of the state, usually coupled with an estimate of the error in the current estimate of the state value. The most commonly used single state estimator is a Kalman filter or some variation thereof. Other methods explicitly maintain many beliefs, or estimates of the state, with a probability associated with each one. Sometimes these are continuous representations of probabilities, in other cases they are sets of discrete state estimates.

Hybrid methods may maintain multiple state estimates, with each individual estimate being the result of a single state estimator such as a Kalman filter. Some of the most successful localization and mapping methods use this approach.

## 7.1 Basic probability

Probability is based upon the idea of **random variables**. A random variable can take on any value within a range with some probability. If the probability of every value in the range is equally likely, then the variable is considered to be **uniformly distributed**, and every value of the random variable has an equal probability.

The probability of the random variable $x$ taking on a particular value $X$ is specified either as $P(X)$ or $P(x = X)$.

All probabilities must be positive, and the sum of the probabilities of all values within the range must be one. For a set of discrete values $x \in \mathbf{X} = \{X_1, \ldots, X_N\}$, (53) must hold.

$$\sum_{x \in \mathbf{X}} P(x) = 1 \tag{53}$$

Often we want to represent continuous random variables over a range. The distribution of values for a continuous random variable is represented as the **probability distribution function**, or PDF, which defines the relative likelihood of each value within the range. The integral of the PDF of a random variable must be one. An example of a commonly used PDF is the **Gaussian**, or normal distribution defined by (54).

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{54}$$

The Gaussian distribution is characterized by its mean value $\mu$, which defines the center of the distribution, and standard deviation $\sigma$, which defines the spread of the distribution.

Sometimes we want to use Gaussian distributions in higher dimensions, where the variables may be related to one another. For example, when values in one dimension are high, values in a second dimension may also be high. The relationship between normally distributed variables can be captured by a **covariance matrix** $\Sigma$. A multi-dimensional Gaussian distribution uses the covariance matrix to scale the differences in each dimension appropriately. The probability of a multi-dimensional random variable, represented as the vector $\vec{v}$, given a multi-dimensional mean vector $\vec{\mu}$ and the covariance matrix $\Sigma$ is given in (55), where $|\Sigma|$ is the determinant of the covariance matrix.

$$p(\vec{x}) = \frac{1}{|\Sigma|\sqrt{2\pi}} e^{\frac{1}{2}(\vec{x}-\vec{\mu})\Sigma(\vec{x}-\vec{\mu})} \tag{55}$$

Often we also want to know the probability of an event given that we know a piece of information. For example, what is the probability of there being an obstacle at a distance $d$ given that we have a sonar measurement $y$, represented as $p(d|y)$? These **conditional probabilities** let us integrate information with random variables in a systematic way.

Sometimes, however, it is difficult to measure a conditional probability such as the probability of an obstacle at distance d given a sonar measurement. To obtain such a probability we would have to take lots of sonar measurements, keep track of which ones returned a value of $y$, and then look at the distance of the obstacle, if there was one. It would be much easier to set an object at distance $d$ and then take lots of sonar measurements, but that would give us the reverse conditional probability distribution function, $p(y|d)$, or the probability of a particular sonar reading $y$ given an object at distance $d$.

Fortunately, we can use **Bayes rule** to relate inverse conditional probabilities. Bayes rule, given in (56), says that the probability of $x$ given $y$ is a function of the probability of $y$ given $x$, the probability of $x$, and the probability of $y$.

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} = \frac{p(y|x)p(x)}{\sum_X p(y|x_i)p(x_i)} \tag{56}$$

It is often the case, that the terms on the right side of the rule can be measured directly. For example, consider the case of a sonar reading $y$ and an obstacle at a distance $d$. We would like to know the probability of an obstacle, given a sonar reading, or $p(d|y)$. However, as noted above, that is difficult to measure directly. On the other hand, by placing objects at various known distances and taking many sonar readings we can easily estimate the PDF for the probability of a sonar reading given an object at a particular distance, or $p(y|d)$. If we assume that the probability of object distances is a uniform PDF, then $p(d) = C_d$ is some constant value $C_d$. Likewise, if we assume that the probability of sonar readings is also uniform, then $p(y) = C_y$ is also a constant value $C_y$. Alternatively, as shown in (56), we can use all of our measurements $y \in Y$ to estimate $p(y)$ directly, which limits our required assumptions to the probability of object distances.

## 7.2 Bayes Filter

It is often the case in robotics that we have an existing estimate of the state of relevant parts of the world. While such a state representation is not complete–a complete state would encompass anything remotely relevant to the robot's future actions–it is usually sufficiently complete to form a basis for predicting the relevant results of actions in the near future.

For example, we may have an estimate of the robot's pose and a map of some landmarks. If we also know the robot's current motor commands, then we can predict the new pose of the robot a short time into the future using a kinematic equation. If all the information we need to make that prediction is encompassed by our state information, the process is called a **Markov chain**. In a Markov chain, the behavior of a system is a function of only the current state. No prior information is required to determine the next state.

It should be clear, however, that small errors and unpredicted external forces make the prediction of the robot's future location uncertain, and that the level of uncertainty increases over time in the absence of feedback, or measurements. Therefore, we need a process the incorporates both prediction based on the current state and current actions and correction based on new information.

The most general probabilistic method of prediction-correction is called a Bayes filter. A Bayes filter does not work on static state values, but on probability distributions. Every value of each state variable has a belief associated with it, and the collection of beliefs–which sums to one–constitutes a probability distribution function. The algorithm for a Bayes filter requires the belief state of the robot at time $t-1$, $x_{t-1}$ and the motor commands and relevant measurements at time $t$, $u_t$ and $z_t$. The output is the new belief state at time $t$, $x_t$. Therefore, the Bayes filter is a mapping from a probability distribution, inputs, and measurements to a new probability distribution.

`Given:` $pdf(\vec{x_{t-1}}), u_t, z_t$
`for all state variables` $x_t \in \vec{x_t}$

1. Prediction: $bel(x_t(-)) = \int p(x_t|u_t, x_{t-1}) pdf(x_{t-1}) dx_{t-1}$
2. Correction: $pdf(x_t(+)) = \eta p(z_t|x_t(-)) bel(x_t(-))$

`return` $pdf(\vec{x_t}) <= pdf(\vec{x_t(+)})$

In the prediction step, the *a priori* estimate of the new state belief distribution function, $bel(x_t(-))$ is generated as a function of the prior state probability $pdf(x_{t-1})$ and the motor commands $u_t$. The belief value for any given $x_t(-)$ is the integral over the range of prior state values of the probability of the state value, $pdf(x_{t-1})$, and the probability of ending in $x_t$ given the state value and the motor commands.

Another way of thinking about the computation, is as a set of paths ending at location $x_t(-)$. The probability of ending up at $x_t(-)$ is the sum of the probabilities of all the paths ending there. The probability of any one path is the product of the probability of starting at $x_{t-1}$, given by $pdf(x_{t-1})$, and the probability of moving along a path from $x_{t-1}$ to $x_t(-)$ given the motor commands $u_t$, represented by $p(x_t|u_t, x_{t-1})$.

In the correction step, the belief in each new state $x_t(-)$ is adjusted by the probability of seeing the measurement $z_t$ in that new state. The normalization constant, $\eta$, converts the belief distribution back into a true probability distribution function by normalizing its integral to one.

The Bayes filter offers a general method of representing beliefs over time. However, it is not computationally tractable to compute in closed form, as the probability distribution functions quickly become complex for realistic models of motion and measurement noise. However, there are many approximations and simplifications to the method that are tractable without significant degradation of performance.

## 7.3 Kalman filters

One of the tractable versions of the Bayes filter is the Kalman filter.

The Kalman filter is an optimal Bayes filter algorithm for estimating state given the following conditions

- The system is linear (describable as a system of linear equations)
- The noise in the system has a Gaussian distribution
- The error criteria is expressed as a quadratic equation (e.g. sum-squared error)

Example of a linear system: Basic Newtonian physics

$$\begin{bmatrix} x_{i+1} \\ \dot{x}_{i+1} \\ \ddot{x}_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ \dot{x}_i \\ \ddot{x}_i \end{bmatrix} \tag{57}$$

Setting up the Kalman filter

1. A measurement $z_k$ is linearly related to a system state $x_k$ as in (58), where the noise term $v_k$ is a Gaussian (normal distribution) with zero mean and covariance $R_k$.

$$z_k = Hx_k + v_k \tag{58}$$

2. If variations in $x$ and $z$ are Gaussian (normally distributed), then the optimal estimator for the next state $\hat{x}_k(+)$ is also the optimal linear estimator, and the estimate of the next state is a linear combination of an a priori state estimate $\hat{x}_k(-)$ and the measurement $z_k$.

$$\hat{x}_k(+) = A\hat{x}_k(-) + Bz_k \tag{59}$$

3. The system to be measured is represented by a system dynamic model that says that the next state is a linear function of the current state plus some noise $w_{k-1}$. Note that, in the absense of any measurements, the error in the system state estimation will grow without bounds at a rate determined by $Q_k$, the process covariance matrix.

$$x_k = \Phi_{k-1}x_{k-1} + w_{k-1} \tag{60}$$

4. The system model, which we create for a specific system using our knowledge of its physics, tells us the a priori estimate of the next state (the estimate prior to any measurement information) based on the last a posteriori state estimate.

$$\hat{x}_k(-) = \Phi_{k-1}\hat{x}_{k-1}(+) \tag{61}$$

5. We also have an error model that tells us how much error currently exists in the system. The new covariances of the error are linear functions of the old error and the system update transformations in (61). The new covariance also increments by the process covariance matrix (the inherent error in the state update equation (60)).

$$P_k(-) = \Phi_{k-1} P_{k-1}(+) \Phi_{k-1}^T + Q_{k-1} \tag{62}$$

6. Now we have estimates of the new a priori state and a priori error based on the state and error update equations. These are open loop equations that do not use any new measurement information. Now we need to update the state based upon the measurements, which means we need to estimate the Kalman gain matrices for (59). The Kalman gain balances how much to trust the state update (using $P_k(-)$) and how much to trust the measurement (using $R_k$). The Kalman gain matrix equation is given in (63). The expression is the process noise projected into the measurement domain, so the Kalman gain matrix is the process covariance divided by the process covariance plus the measurement noise. The lower the measurement error relative to the process error, the higher the Kalman gain will be.

$$\bar{K}_k = P_k(-1) H_k^T \left[ H_k P_k(-) H_k^T + R_k \right]^{-1} \tag{63}$$

7. We can now complete the estimation of the error in the new state, $P_k(+)$, using the Kalman gain matrix, which tells us how much each piece will be trusted.

$$P_k(+) = \left[ I - \bar{K}_k H_k \right] P_k(-) \tag{64}$$

8. Finally, we can calculate the a posteriori state estimate using the Kalman gain, the a priori state estimate and the new measurement.

$$\hat{x}_k(+) = \hat{x}_k(-) + \bar{K}_k \left[ z_k - H_k \hat{x}_k(-) \right] \tag{65}$$

### 7.3.1 Kalman Filter Summary

Kalman filter variables:

- $\Phi_k$ = process matrix at step k defining how the system changes
- $Q_k$ = process noise covariance matrix at step k
- $H_k$ = measurement matrix relating the state variable and measurement
- $R_k$ = measurement noise covariance matrix at step k
- $\hat{x}_k(+)$ = system state estimate at step k
- $P_k$ = system covariance matrix at step k, estimate of uncertainty in $\hat{x}_k(+)$
- $K_k$ = Kalman gain estimate at step k, how much to trust the measurement

Kalman filter equations

$$\hat{x}_k(-) = \Phi_{k-1}\hat{x}_{k-1}(+) \tag{66}$$

$$P_k(-) = \Phi_{k-1}P_{k-1}(+)\Phi_{k-1}^T + Q_{k-1} \tag{67}$$

$$\bar{K}_k = P_k(-1)H_k^T\left[H_kP_k(-)H_k^T + R_k\right]^{-1} \tag{68}$$

$$P_k(+) = \left[I - \bar{K}_kH_k\right]P_k(-) \tag{69}$$

$$\hat{x}_k(+) = \hat{x}_k(-) + \bar{K}_k\left[z_k - H_k\hat{x}_k(-)\right] \tag{70}$$

**Example: temperature estimation**

Temperature is a slowly varying signal we can model as a constant value with a small system variance.

Setup

- State is a single value: $[x]$ (temperature in C)
- Estimate a process variance $Q$ (difficult to do, so probably just a small value)
- Estimate a measurement variance R by calculating the standard deviation for a large number of measurements of a constant temperature situation (or even just a constant voltage source). Want to separate the actual process noise and the measurement noise.
- Define a state estimation matrix

  Temperature is modeled as a constant value, so $\Phi = [1]$
- Relationship between measurements and the temperature H

  If we have C comes directly from the measurement process, then $H = [1]$

  If the measurement undergoes a transformation, put that in H
- Let the initial state be $[0]$
- Let the initial error be $[P]$

For each new measurement

- The a priori state is the same as the last a posteriori state ($\Phi$ is the identity)
- The a priori error is the same as the a posteriori error plus $Q$
- The Kalman gain is

$$K_k = \frac{P_k(-)}{P_k(-) + R} \tag{71}$$

- The new a posteriori state is

$$\hat{x}_k(+) = \hat{x}_k(-) + K_k \left( z_k - H x_k(-) \right) \tag{72}$$

- The new a posteriori error is

$$P_k(+) = \left( 1 - K_k H \right) P_k(-) \tag{73}$$

After each iteration, the temperature is updated based on the new measurement. Note that the value of $P$ will converge to a constant, as will $K$, so long as $R$ and $Q$ are constant. Both $P$ and $K$ can be precomputed using a differential equation, but it can be simpler to run the filter for a while and then preset $P$ and $K$ based on the stable values.

**Example: 1-D motion estimation**

Imagine a robot that can move in one direction estimating its distance from a wall as it moves. The robot is equipped with a noisy sensor, such as a sonar, and it also knows the velocity command last sent to the motors. We want to model not only the position of the robot, but also its current velocity.

Setup

- State is a pair of values, position and velocity: $\begin{bmatrix} x \\ \dot{x} \end{bmatrix}$
- Estimate a process covariance matrix $Q$ by measuring variation in the velocity and position given a command and extending that to position by multiplying by the time step $\Delta t$. Note that small differences in $\Delta t$ mean that the position error probably increases faster than the position error. The off-diagonal terms, in this case, will not be zero as errors in the velocity are strongly related to errors in the position.
- Estimate a measurement variance R by calculating the standard deviation for a large number of measurements of known distances.
- Define a state estimation matrix

  We are modeling the state as a constant velocity, so $\Phi = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$
- Relationship between the state and the distance measurement is defined by H.

  We can assume we are measuring distance directly, and velocity as $\dot{x} = (z_k - z_{k-1})$ in which case $H = \begin{bmatrix} 1 & 0 \\ 0 & 1/\Delta t \end{bmatrix}$ and $\begin{bmatrix} z_k \\ z_k - z_{k-1} \end{bmatrix} = H \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$.
- Let the initial state be $\begin{bmatrix} D \\ 0 \end{bmatrix}$ and let the initial error P be the identity $I$.

For each new measurement

- Update the a priori state using the last a posteriori state and the update matrix $\Phi$.
- The a priori error is the same as the a posteriori error plus $Q$, which is a known covariance matrix.
- The Kalman gain is

$$K_k = \frac{P_k(-)H^t}{HP_k(i)H^t + R} = P_k(-)H^t[HP_k(-)H^t + R]^{-1} \tag{74}$$

- The new a posteriori state is

$$\hat{x}_k(+) = \hat{x}_k(-) + K_k\left(z_k - Hx_k(-)\right) \tag{75}$$

- The new a posteriori error is

$$P_k(+) = \left(1 - K_k H\right) P_k(-) \tag{76}$$

After each iteration, the velocity and position are updated based on the new measurement. Note that the value of $P$ will converge to a constant, as will $K$, so long as $R$ and $Q$ are constant. Both $P$ and $K$ can be precomputed using a differential equation, but it can be simpler to run the filter for a while and then preset $P$ and $K$ based on the stable values.

## 7.4 Particle filters

In practice, Kalman filters can be used to estimate state for a wide variety of real systems. Even for non-linear systems, it is possible to calculate locally linear update matrices for the Kalman update equations. There are two limitations to Kalman filters, however, that restrict their application in robot localization.

- The Gaussian noise requirement is not realistic for many robot situations, which may be multi-modal.
- The need to invert a matrix during the calculation of the Kalman gain limits the number of possible state values, which can be large for robot localization tasks.

One successful alternative to Kalman filters is the **particle filter**, which represents probability distributions using samples. The Kalman filter also represents probability distributions using samples, but since the distributions are required to be Gaussian, only the mean and standard deviation are required for the representation. Arbitrary probability distributions require many more samples in order to ensure an adequate representation.

### 7.4.1 Definition

A particle filter is a sample-based Bayes filter. Each sample represents the probability of a particular state vector given all previous measurements. The distribution of state vectors within the samples is representative of the probability distribution function for the state vector given all prior measurements.

- $Z_k = (z_1, z_2, \ldots, z_k)$ is the set of all measurements.
- $X$ is a state vector
- $P(X|Z_t)$ is the probability of a state vector $X$ given all prior measurements $Z_t$.

Knowing $P(X|Z_t)$, we could pick high probability states or evaluate possible states as to their fitness. However, it is difficult to generate the distribution directly. Using the Bayes filter approach, we can express the probability of any particular state at time $t$, $x_t$ given the measurement sequence $Z_t$ as in (77).

$$P(x_t|Z_t) = \frac{p(z_t|x_t)p(x_t|Z_{t-1})}{p(z_t|Z_{t-1})} \tag{77}$$

Therefore, the probability of a state $x_t$ given all sensor readings is a function of three probabilities:

1. the probability of the state $x_t$ given all but the last sensor reading,
2. the probability of the last sensor reading $z_t$ given the state vector $x_t$, and
3. the probability of the last sensor reading $z_t$ given all prior sensor readings $Z_{t-1}$.

These probabilities represent an open loop update, $p(x_t|Z_{t-1})$, a closed loop correction, $p(z_t|x_t)$, and a normalization $p(z_t|Z_{t-1})$.

A particle filter represents the probability distribution function $pdf(x_{t-1}|Z_{t-1})$ as a set of samples, where each sample includes a state estimate $s^i_{t-1}$ and a state probability $\pi^i_{t-1}$.

$$pdf(x_{t-1}|Z_{t-1}) \approx \{(s^1_{t-1}, \pi^1_{t-1}), \ldots, (s^N_{t-1}, \pi^N_{t-1})\} \tag{78}$$

### 7.4.2 Particle Filter Update Algorithm

**Initialization**

The initial set of samples should be drawn from the initial distribution of state vectors. If the state vector is known at time zero, then all of the samples are assigned the known value with equal probability.

**Prediction phase:**

For each sample $(s_{t-1}^i, \pi_{t-1}^i)$, calculate a new state $s_t^{i-}$ using a motion model. A typical motion model uses the kinematic update equations plus additive Gaussian noise.

The end result of the prediction phase is the modification of the current particle set to represent the open loop update distribution.

**Update phase:**

The update phase weights the new samples according to the current measurement $z_t$.

$$\pi_t^{i+} = p(z_t|x_t) \tag{79}$$

**Re-sample phase:**

If the process stopped there, the samples would generally get less and less likely, because they would drift away from the high probability locations of the probability distributions. To correct for the drift, the next generation of samples get selected stochastically from the current set according to the relative weights; high likelihood samples are more likely to be selected, while low likelihood samples are likely to be dropped.

Resample process

1. Stochastically select $N$ samples from the set of existing samples according to the weights $\pi_t^{i+}$.
2. Renormalize the weights to sum to one: $\sum_N \pi_t^{i+} = 1$

### 7.4.3 Basic Process Summary

1. Given: a set of samples $\{(s_{t-1}^1, \pi_{t-1}^1), \ldots, (s_{t-1}^N, \pi_{t-1}^N)\}$
2. Prediction: For each sample, $(s_{t-1}^i, \pi_{t-1}^i)$, generate a new state $s_k^{i-}$.
3. Correction: For each sample, generate a new weight $\pi_k = p(s_k^{i-}|z_t)$.
4. Resample: Select a new set of samples based on the weights.
5. Normalize: Normalize the weights to one.
6. Repeat steps 2 through 5 to maintain the state probability distribution function over time.

### 7.4.4 Particle Filter Modifications

**Order of operation**

It turns out that you can sample at the end of the process based on the $\pi_t^{i+}$, or you can sample at the beginning of the process based on the $\pi_{t-1}^{i+}$. Either method seems to work, but drawing from the prior distribution lets you draw samples from different distributions.

- Sometimes you just want to insert random particles into the system drawn from $p(x_t)$. Inserting random particles lets the filter locate new high probability areas and recover from drift.
- Sometimes you know something about where particles ought to be.

**Sampling algorithm**

The simple way of selecting a next generation is $O(N^2)$. Let the sum of the weights be $\Pi$.

1. Pick a random number between 0 and $\Pi$.
2. Sum the weights of the particles until you reach the number.
3. Repeat from step one N times.

The simple algorithm requires $N$ summations of $N/2$ (on average) particles to select the next generation.

A more clever method of sampling is only $O(N)$.

1. Pick a random number between 0 and $\frac{1}{N}\Pi$.
2. Sum the weights of the particles until you reach the number
3. Select that particle
4. Add $\frac{1}{N}\Pi$ to the random number
5. Repeat from step 2

The algorithm samples from the weight space uniformly and regularly, but by aligning the grid at a random location it avoids aliasing. Due to the random alignment, any particle has a probability of landing on the grid that is proportional to its weight. The algorithm is order $N$ because it passes through the particles only once. Avoiding the $O(N^2)$ sampling time makes each iteration of the entire particle filter linear in the number of particles.

### 7.4.5 Example: Localization Using Particle Filters

Particle system localization has worked well in practice on numerous systems. They have proven robust to noise, robust to divergence in the state tracks (unlike a Kalman Filter), and they can be fast.

Source: Dellaert, Fox, Burgard, Thrun, "Monte Carlo Localization for Robots", ICRA'99, May 1999.

Video: http://www.cs.washington.edu/ai/Mobile_Robotics/projects/mcl/animations/global-vision.gif

Process:

1. Generate a visual map of the ceiling
   - From the robot, take pictures of the ceiling at regular intervals
   - Stitch the pictures together into a single mosaic
2. While the robot is moving
   - Look up with the camera and continuously watch the ceiling
   - Consider a single pixel in the middle of the image
   - Update a particle filter with the single pixel measurement

Results: The robot moved around the NMNH for several days

- After hours the robot moved fast (2m/s).
- System worked exceptionall well.
- Divergence would definitely have been an issue for a Kalman filter.

### 7.4.6 Example: Adaptive particle filters

One issue with particle filters is that as the number of dimensions grows, the number of particles required in the general case grows exponentially. For a 1-D system you may need only 40 particles, but for a 3-D system you may need $40^3 = 64000$ to ensure adequate representation of the probability distribution. That may stress the computational capabilities of the robot to the point where the sensor data is arriving faster than the particle system is being updated.

Fox (Fox, Adapting the sample size in particle filters through KLD-sampling, IJRR, 2003) proposed adaptive the number of particles used by the system based on the complexity of the probability distribution. When the distribution is unknown, it is necessary to use a large number of particles distributed around the space of possible locations. But when the distribution is strongly peaked, it is not necessary to represent the peak with a large number of points.

The key to making the system work properly is to have a useful estimate of the error in the representation of the probability distribution. When the error is large, the system needs to use more particles. When the error is small, the system can reduce the number of particles, speeding up the computation. In one localization example, the number of particles starts at 40,000 and then reduces to around 40 once the particles have condensed around the robot's true location.

Video:
http://www.cs.washington.edu/ai/Mobile_Robotics/projects/mcl/#_KLD-sampling:_Adaptive_particle_fil

# 8 Maps and Motion Planning

Motion planning covers a wide array of situations and techniques for handling them. In general, the more complexity in the environment, the more difficult motion planning becomes. Likewise motion planning complexity increases for a robot with more complex kinematics, or more joints. Nevertheless, it is possible to plan complex motions, such as parallel parking a double trailer.

What are the kinds of motion we might want a mobile robot to do?

- Random walks: collision avoidance and collision prediction
- Fixed goals: achieve a point or series of points
- Coverage: explore or cover an area efficiently
- Dynamic goals: go to or track a moving point
- Abstract goals: serve people food

Were going to look at a simple version of the fixed goals problem first.

Assumptions

- Robot is the only moving object
- Robot moves in such a way that we can ignore dynamic properties (slowly)
- Robot avoids contact with objects
- Robot is a single rigid object

Let $A$ be the robot. We assume the robot moves in a Euclidean space $W$ (**workspace**), represented as $R^N$ where $N \in \{2, 3\}$. The workspace is populated with $Q$ fixed rigid **obstacles** $B_1$ to $B_Q$ distributed in $W$. For this case, the geometry of $A$, the $B_i$ and the locations of the $B_i$ are accurately known, and there are no kinematic constraints on $A$ ($A$ is **free-flying**). A circular differential drive robot meets these conditions closely enough that the solution to this problem is a sufficient solution.

Given these assumptions, we can state the problem of motion planning as a purely geometric one.

> Given an initial position and orientation and a goal position and orientation of $A$ in $W$, generate a path for $A$ that avoids contact with the $B_i$'s, starting at the initial position and orientation and ending at the goal position and orientation. Report failure if no path exists.

It turns out that, despite these strong assumptions this problem is still hard.

## 8.1 Configuration Space

An important concept in motion planning is the **configuration space** of the robot, defined as the space $C$ of all possible robot configurations. Obstacles put constraints on the robot's feasible configurations.

Given the case of simple motion on the plane, what are all of the possible configurations of the robot?

- The robot is free-flying, so it can rotate and translate freely
- The robot has some finite size
- The robot has to avoid contact with the objects

The space occupied by the robot $A$ is a closed, finite area subset of worldspace $W$, with the pose of the robot defined as a geometric transformation between the coordinate systems of $W$ and $A$ as described previously. The obstacles $B_i$ are also closed subsets of $W$.

A **path** is a mapping $T[0,1] \rightarrow C$, where $T(0)$ is the initial state and $T(1)$ is the final state. The function $T(p)$ is a parametric representation of the curve, where $p \in [0,1]$. For a free-flying any definable path that does not cause the robot to pass through obstacles is possible between two points.

Defining objects/obstacles

- Every object maps to a region in configuration space.
- An obstacle's region is the union of all configurations of $A$ that intersect the object.
- $CB_i = \{q \in C | (A(q) \cup B_i) \neq 0\}$
- Obstacle space **C-obstacle** $= \bigcup B_i$
- **C-free** is the inverse of C-obstacle

Free paths

- A free path is a mapping onto C-free
- C-free can be divided by obstacles of infinite extent or by obstacles with holes
- A connected component of C-free is a set of configurations such that there is a free path between them

Mapping simple robots to configuration space

- C is equal to $R^N$, as long as the orientation of the robot can be ignored when planning a path
- Since a circular robot can rotate in place, orientation doesn't matter with respect to paths

Mapping obstacles to the configuration space of a simple robot

- Grow the obstacles by the radius of the robot plus a safety factor and then make the robot a point
- The path of the robot becomes 1D

Now that we have fairly clear definitions of obstacles and configurations, we can start to execute algorithms for identifying paths that achieve fixed goals. We can then build on that to achieve paths in dynamic environments where either the goal is changing or C-obstacle is changing over time.

## 8.2 World Representation

There are three major approaches to symbolic planning in mobile robotics, differentiated by the way the algorithm represents the world.

**Visibility graphs and retractions** are like roadmaps of the world that tell the robot where it can and cannot go. By searching the maps, the system can identify one or more paths, if they exist, between the start and goal locations.

**Cell decomposition** divides the world into a set of homogeneous areas; each area is either all free space or all obstacle. The decomposition includes links between the regions that indicate how the robot can travel from a region to its neighbors. By search the links between adjacent regions, the robot can find one or more paths, if they exist, between the start and goal locations. Cell decompositions can be **exact** or **approximate**. Exact cell decompositions divide the world according to obstacle boundaries, which means the decomposition tends to be irregular. Approximate decomposition divides the world into a regular grid, assigning a free/obstacle label to each grid square. While the approximate decomposition is easier to represent, it only approximates the shape of obstacles in the environment since the grid squares usually do not line up exactly with the obstacle boundaries.

**Potential fields** represent the world as a set of attractors and repulsers, where objects act as repulsive fields and the goal location as an attractor. The method uses the field gradients to execute both global and local navigation strategies.

All of the methods may be used in both static or dynamic environments, however, they have different modification costs, which approximate decomposition being the simplest to modify. All of the methods may also be used whether the obstacles are known a priori or discovered en route, again, with caveats about the ease of modifying the graphs. From limited observations visibility graphs seem to be closest to the method people use to navigate between locations.

## 8.3 Visibility Graphs

**Assumptions:**

- $A, B$ are polygonal
- $W = R^2$
- translational/holonomic robot, so $C = R^2$

The idea of the visibility graph is to construct a semi-free path as a polygonal line from the start location $Q_0$ to the goal location $Q_F$.

**Proposition 1**: Let $CB$ be a polygonal region of $C = R^2$. There exists a semi-free path between any two given configurations $Q_0$ and $Q_F$ if and only if there exists a simple polygonal line $T$ lying in a closed set of $C_{free}$, $cl(C_{free})$, whose endpoints are $Q_0$ and $Q_F$, and such that $T$s vertices are vertices of $CB$.

**Proof:** If there exists a path, then there exists a shortest path. Locally, the shortest path must also be the shortest path. Therefore, the curvature of the path must be zero except at the vertices of $CB$.

Therefore, it is sufficient to consider the space of lines between obstacle vertices when planning a path. The nodes of the graph are $Q_0$, $Q_F$, and the vertices of $CB$. The visibility graph itself consists of all lines

segments between the nodes that lie in $C_{free}$ except for possibly their endpoints, in the case of $Q_0$ and $Q_F$.

In English, if you have a set of obstacles, you go around them as closely as possible at the corners. Since the configuration space has grown the obstacles, the corners are "safe" for the robot to traverse. The drawback is that you come close to obstacles and have to rely on the safety factor built into the obstacle growing step.

**Generating visibility graphs**

Stupid method:

- For all line segments between all endpoints calculate their intersections with CB
- Keep the ones that don't intersect any other segments of CB
- Cost of the algorithm is $O(N^3)$

Line-sweep method:

- Sweep a line rotationally around each point, stopping at each other endpoint in CB
- Find the closest intersection with CB at each angle
- Cost is $O(N^2 \log N)$

Best method so far: (Hershberger and Suri, 1997):

- Subdivide the space into a conforming map
- Each vertex is within its own region of space
- Each edge of a region has a constant number of regions within a certain distance
- Propagate waves from all the vertices through the conforming map
- Calculate wavefront collisions, which tell you what the possible shortest paths are between vertices
- Cost of building the graph is $O(N \log N)$, with space requirements of $O(N \log N)$

The result of each method is a map of paths through the space from a source point P.

Searching the path is $O(\log N)$ once you have the graph (Dijkstras algorithm)

## 8.4 Retractions

Let $X$ be a topological space and $Y$ be a subset of $X$. A mapping $X \to Y$ is a retraction if and only if it is continuous and its restriction to $Y$ is the identity map. (The piece of $X$ in $Y$ maps identically to $Y$).

Let $P$ be a retraction. $P$ preserves the connectivity of $X$ if and only if for all $x \in X$, $x$ and $P(x)$ belong to the same path-connected component.

**Proposition 2**: Let $P$ be a connectivity-preserving retraction $C_{free} \to R$, where $R$, a subset of $C_{free}$, is a network of 1D-curves. There exists a free path between two free configurations $Q_0$ and $Q_F$ if and only if there exists a path in $R$ between $P(Q_0)$ and $P(Q_F)$.

**Proof:** If a path exists in $C_{free}$, then $P$ preserves connectivity and $Q_0$ and $Q_F \to R$. If a path exists in $R$, then a path in $C_{free}$ exists that has three components:

1. a path from $Q_0$ to $P(Q_0)$
2. a path from $P(Q_0)$ to $P(Q_F)$, and
3. a path from $P(Q_F)$ to $Q_F$.

Paths 1 and 3 must exists because $P$ is connectivity-preserving.

A **Voronoi diagram** is a retraction that meets these requirements. A Voronoi diagram is a set of paths that are equidistant from $CB$ at all points. Therefore, it maximizes the distance between the robot and the obstacles. In a polygonal world, it will consist of lines and parabolic line segments.

- Possible cases: [edge, edge], [edge, vertex], [vertex, vertex]
- $O(N)$ edges in the Voronoi diagram, where $N$ is the number of vertices in $C_{free}$
- Don't try to create a Voronoi diagram in an open-sided space!

Can also think of it as subdividing the space so that all of the points in each region have the same closest point or line segment in C-obstacle. There are $O(N \log N)$ algorithms for generating Voronoi diagrams. It is recommended that you use a library like CGAL to make them.

Once you have a retraction, you need to search it for the shortest path, which takes $O(\log N)$ time, similar to a visibility graph. In general Voronoi diagrams are simpler than visibility graphs. One benefit is that the robot stays as far away from obstacles as possible. Voronoi diagrams do not represent the shortest path between two points, in general, unless those points are on the same line segment of the diagram.

## 8.5 Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm for identifying the shortest path between one node of a graph and every other node in the graph. Dijkstra's algorithm builds a **spanning tree**, a graph with no loops, that gives the shortest path from each achievable node in the state space to the source node. Given a spanning tree, the shortest path to any node is possible to determine in time proportional to the length of the path.

**Algorithm:**

- G = (V, E): a graph with a set of vertices V and edges E
- S: the set of vertices whose shortest paths from the source have already been determined
- Q: the remaining vertices
- d: the array of best estimates of shortests paths to each vertex (one element per vertex)
- pi: an array of predecessors for each vertex (one element per vertex)

```
shortest_paths( Graph g, Node s )
    initialize_single_source( g, s )
    S := { 0 }          /* Make S empty */
    Q := Vertices( g ) /* Put the vertices in a priority queue */
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )

/* u is the newest node added to the spanning tree */
relax(Graph g, Node u, Node v, double w[][] )
    if g.d[v] > g.d[u] + w[u,v] then
       g.d[v] := g.d[u] + w[u,v]
       g.pi[v] := u

/* Initialize the source node to a distance of 0, and rest to inf */
initialize_single_source( Graph g, Node s )
   for each vertex v in Vertices( g )
       g.d[v] := infinity
       g.pi[v] := nil
   g.d[s] := 0;
```

## 8.6 General Search on Graphs

Once we have a symbolic representation of our space, we need a method of identifying good paths to get from the initial node to the goal node. Standard search algorithms offer an efficient solution to finding a path between two nodes given a graph.

The two basic modes of search are **breadth-first** and **depth-first search**. Breadth-first search [BFS] explores all of the nodes in order of distance from the start node. BFS explores all nodes directly connected to the start node, then explores nodes that are two steps away, and so on. BFS is expensive in terms of memory, but it is guaranteed to find a path, if it exists, and the first path it finds from the start to the goal node will be the shortest path.

Depth-first search [DFS] explores one branch of the graph as far as possible, backing up only when forward progress is impossible. While DFS is efficient in terms of memory–only the current path needs to be saved–it is not guaranteed to find a solution, nor is the solution it finds guaranteed to be the shortest solution.

A greedy search algorithm is a variant of DFS that expands nodes according to a function $h$ that provides an estimate of the distance to the goal from the current node. Greedy search algorithms can provide significant speedups in search time, but are not guaranteed to find the shortest path.

### 8.6.1 Iterative Depth-first Search

Depth first search expands nodes in a child-first method. Bounded depth first search will explore all of a tree down to a particular depth, identically to a breadth-first search to the same depth. The difference is that depth-first search uses much less memory: $O(\log N)$ instead of $O(N)$. The benefit of BFS is that it finds the shortest path. The benefit of DFS is that it uses very little memory. Iterative depth-first search provides both benefits without significantly more computation time (same $O$ complexity).

The IDS is give below.

1. Set $depth = 1$
2. Loop while the solution is not found and $depth < maxDepth$
   (a) Execute DFS to the specified depth
   (b) Return if the search finds a solution

IDS finds the shortest path because it explores each path of length $N$ before exploring any path of length $N + 1$. It uses the memory of DFS, and its complexity is no different than BFS because most of the nodes are in the last level searched.

### 8.6.2 A* Search

The A* search algorithm is a search method that takes into account both the estimated cost to the goal, $h$ and the cost of the path so far, $g$. By using their sum, $g + h$ to decide which node to expand next, A* gains some nice properties. In particular, if $h$ provides an exact or underestimate of the cost to the goal, then A* is guaranteed to find a path, if it exists, and the first path it finds will be the shortest path. Because A* takes into account the estimated cost of getting to the goal, it is more efficient than breadth-first, as it tends not to search nodes that do not bring it closer to the goal.

A search algorithm requires a **state space** in which the search takes place. A visibility graph, for example, represents a state space, where each node in the graph is one state. The **root node** is the starting node for the search, generally the robot's current location. The **children** of a node are all of the states accessible from the current node. In the case of the visibility graph, the child nodes are all of the nodes connected to the current node via an unblocked line.

In addition, A* requires the following components.

- OPEN list: sorted list of places to explore that are connected to places youve been
- CLOSED list: places you have already explored
- g() function: cost from root to the current node
- h() function: estimated cost from the current node to the goal

**Algorithm**

1. Put the root node on the OPEN list
2. While the OPEN list is not empty, extract the first item
   - Test if the node is the goal
     - If it is, terminate the search
     - return the path to the goal by linking through all the ancestors of the goal node
   - If it is not the goal, expand the node to find its children
   - For each child
     - If the child is on the CLOSED list
       - Carefully consider what needs to happen
       - If the node on the CLOSED list has a shorter g value, then delete the child
       - Else, make the current node the parent of the node on the CLOSED list and replace the g value of it with the g value of this child and propagate the g value to all of the descendents of this node, which may require resorting the OPEN list.
     - If the child is on the OPEN list
       - Carefully consider what needs to happen
       - If the node on the OPEN list has a shorter g value, then delete the child
       - Else, delete the node on the OPEN list and insert the child in the proper location
     - Else, if the child is not on either the OPEN or CLOSED lists
       - Put the child on the OPEN list with a link to its parent
   - Place the current node on the CLOSED list
3. If the OPEN list is empty, return failure in the search

The characteristics of the functions $g$ and $h$ determine the behavior of the A* algorithm.

- Breadth-first search: Estimate of cost to the goal (h) is zero
- Greedy/depth-first search: Estimate of cost from the root node (g) is zero
- A* is optimal if the h function is exactly the cost to the goal
- A* returns an optimal path if the h function under-estimates the distance to the goal
- A* is pretty good if the h function only sometimes over-estimates the distance to the goal

### 8.6.3 Any time search: IDA*

Sometimes in robotics we are computation-limited with real time deadlines. In that case, we may not be able to find a complete solution from start to finish. However, it is still possible to make an estimate of which direction the robot ought to move to achieve the goal even given the hard time deadline. The method is called IDA*, or **iterative deepening A*** search. The basic idea is to initialize a cost $f$ that bounds the extent of the search and then use the following algorithm.

1. Run A* search until the next node to be expanded has a cost $g + h > f$.
2. If the goal is not found, store the path with the least cost to the goal $h$, increase $f$, and repeat. Otherwise, return the path.

IDA* is an anytime search algorithm because the process can be interrupted and there will still be a best guess available from the prior search. Since the search algorithm always expands most of the nodes at the bottom level, the cost increase is not significant compared to the total cost of the search.

## 8.7 Approximate Decomposition

Approximate decomposition involves approximating the world with a set of polygons–usually regular polygons such as squares or rectangles. Hexagons also have nice properties, but are not as easy to represent and manipulate.

With approximate decompositions, each unit contains an internal state indicating the traversability of the unit. When the internal state is a binary variable, then each unit represents either free space or an obstacle. However, the internal state need not be a binary value. Multi-valued enumerated types are possible (e.g. open, blocked, or unknown), as are continuous value representations (e.g. the probability the unit is an obstacle). When each unit uses an enumerated type to represent its internal state, the map is known as an **occupancy grid**. When each unit uses a continuous value, the map is known as an **evidence grid**.

With respect to motion planning, the binary state representation of each unit provides a simple context for identifying a paths. If you can find a path through open regular polygons from $Q_0$ to $Q_F$, then there is a traversable path connecting the two. Remember, the robot is a point in configuration space so long as the obstacles are grown prior to motion planning.

Approximate decomposition is the simplest method of world representation to integrate with map generation because the basic units do not change their relationship with the addition of obstacles; only the state of the relevant units changes. While this can change the cost of paths that traverse those units, it does not change the connectivity of the units.

A grid maps onto a graph with connections between a grid cell and each of its neighbors. For smoother motion, the graph can also connect each cell with others inside a certain radius. However, care must be taken in expanding those connections as the paths they represented may be blocked by intermediate cells. The A* search algorithm is an appropriate algorithm for identifying paths on a grid between $Q_0$ and $Q_F$.

### 8.7.1 Evidence Grid

Evidence grids are a commonly used tool in robotics to integrate sensing, localization, mapping, and path planning. Evidence grids are usually a decomposition of the world into squares at a resolution appropriate for the obstacles in the environment, the noise level of the sensors and the type of paths the robot needs to traverse. The internal state of each grid cell is a single continuous value representing the probability of occupancy $p(o)$. For numerical reasons, the probabilities are normally represented as log probabilities.

Sensor models tell us what a particular reading from a sensor means in terms of a probability distribution. We have already explored how to go about creating sensor probability distributions by taking measurements and generating a probability distribution function for the probability of a sensor reading given an obstacle at a certain distance $p(z|O)$. Note that this distribution is two-dimensional because a sonar's response is based on a wedge of the map.

In order to maintain an evidence grid, we need an algorithm for discovering the probability of an obstacle given all of the measurements so far $p(O|Z_t)$. One commonly used method was proposed by Moravec.

Suppose we initialize the map with $p(O|Z_t) = 0.5$ at each grid cell. Initially, nothing is known about the world, so the probability of an obstacle is the same as no obstacle. The odds of an obstacle existing at a location is given by the ratio of an obstacle existing to an obstacle not existing at the grid. Using Bayes' rule, we can write the odds as a function of the noise models and a prior on the existence of obstacles.

$$\frac{p(O|Z_t)}{p(\bar{O}|Z_t)} = \frac{p(Z_t|O)p(\bar{O})}{p(Z_t|\bar{O})p(O)} \tag{80}$$

If we can assume the errors are independent, then $p(z_t, Z_{t-1}|O) = p(z_t|o_t)p(Z_{t-1}|O_{t-1})$. Therefore, we can rewrite (80) as the product of the existing probabilities and the new probabilities.

$$\frac{p(O|z_t, Z_{t-1})}{p(\bar{O}|z_t, Z_{t-1})} = \frac{p(z_t|o_t)p(Z_{t-1}|O_{t-1})p(\bar{O})}{p(z_t|\bar{o}_t)p(Z_{t-1}|\bar{O}_{t-1})p(O)} \tag{81}$$

Moravec's algorithm for updating the evidence grid is as follows:

1. $q_1 = p(O_{t-1}|Z_{t-1})/(1 - p(O_{t-1}|Z_{t-1}))$
2. $q_2 = p(z_t|o_t)/(1 - p(z_t|o_t))$
3. $q_t = q_1 * q_2$
4. $p(O_t|Z_t) = q_t/(1 + q_t)$

One benefit of using the above algorithm is that the numbers in the evidence grid can be straight probability values, so long as the update algorithm sets maximum and minimum values for the probability of occupancy that do not get too close to one or zero. Bounds such as 0.99 and 0.01 on $p(O_t|Z_t)$, for example, keep the update algorithm numerically stable.

It is also possible to run a Kalman filter or a simple Baysian filter at each grid location, with the internal state variable being the probability of occupancy. Moravec's implementation is a Bayesian filter method.

There have been many different implementations of occupancy/evidence grids since Moravec's first paper on the subject. There are two main issues: speed and accuracy. For a sensor like a sonar, which can have a wide beam, the pdf $p(z_t|o_t)$ is a somewhat complex 2D function mapped onto a grid. Often, a sonar is modeled as a Gaussian distribution centered at the measurement distance $d$ from the sonar's location and rotated through space for the angle of the sonar's coverage. A laser sensor is usually represented as a line, since its width even at maximum range is generally smaller than the size of the grid boxes.

Practical implementations using sonars will generally sample this function on a triangular polygon, which must then be rotated into the proper orientation and drawn into the grid. Computer graphics algorithms, such as the scanline polygon rasterization algorithm are fast implementations of this process. Likewise, Bresenham's line algorithm is a fast implementation of line drawing on a grid.

Almost all implementations of grid-based maps use continuous values when building the map, but may convert the map to a binary form for the purposes of planning. The simplest implementations draw the pdf (triangle or line) directly on the map, ignoring the existing values. More sophisticated algorithms use some variation of Moravec's method, with most using as single probability value representing $p(o_t|Z_T)$ and using Bayes' rule to update the value given a new measurement.

$$p(o_t|z_t, Z_{T-1}) = \frac{p(z_t|o_t)p(o_t|Z_{t-1})}{p(z_t|Z_{t-1})} \tag{82}$$

The terms on the right side can all be estimated or selected to achieve certain goals. The pdf $p(z_t|o_t)$ is the result of an experimental procedure based on placing the robot a certain distance from an obstacle and taking measurements. The a priori probability $p(o_t|Z_{T-1})$ is the prior estimate of an obstacle, possibly reduced by

some amount to let the map decay over time to reflect uncertainty introduced over time. The denominator is a normalizing constant and is generally treated as a constant value over time in most models. It must be less than one, or else the probabilities in the evidence grid will go to zero over time.

Executing an A* search on an evidence grid is straightforward. The level of occupancy of each grid point adds to the cost of traversing the path ($g$ function). Cells with high occupation probabilities should add a very high cost to the path, while cells with low occupation probabilities should not add much. Generally the costs are a nonlinear function (e.g. a step function) of the probabilities to discourage a search from selecting paths that go through likely obstacles.

## 8.8 Potential Fields

The potential fields method is a commonly used method of non-global path planning that can be easily integrated with either exact or approximate decomposition. It is simple to compute, provides a global solution for non-concave obstacle spaces, and provides a control scheme as well as a path.

One of the issues with something like A* search on an evidence grid, for example, is that the path exists as a set of connections between grid spaces and is rarely smooth. Following such a path, or generating a traversable smooth path is a non-trivial exercise. Because the potential field method always provides a continuous and locally optimal direction at every location on the plane, it provides a control input as well as the path.

The concept is that the goal attracts the robot while objects repel it. The goal attraction force is based on the distance between the robot and the goal, while the repellant force is based on all obstacles within a local window. The overall potential field at a location $q$ is the sum of the attractive and repulsive fields. The general model for the attractive and repulsive forces is generally an electrical field, which has a $\frac{1}{r^2}$ falloff.

$$U(q) = U_a(q) + U_r(q) \tag{83}$$

The robot's desired direction of motion is the downhill direction, or gradient of the potential field $U$ generated by the goal and the obstacles. We can also write this as the force on the robot being equal to the negative gradient of $U$. Conceptually, the goal should be the location of lowest potential, while obstacles should be high potential. To achieve the goal, the robot wants to move in the direction of the largest negative gradient.

$$F(q) = -\nabla U(q) \quad \nabla U = \left[ \begin{array}{cc} \frac{\partial U}{\partial x} & \frac{\partial U}{\partial y} \end{array} \right]^t \tag{84}$$

One method of generating the attractive force is to use a parabolic function of the Euclidean distance to the goal, $\rho = \sqrt{(q_x - q_g)^2 + (q_y - q_g)^2}$. The magnitude of the field is modulated by the factor $k_a$.

$$U_a(q) = \frac{1}{2} k_a \rho^2 \tag{85}$$

Given the goal position $q_g$, the derivative of the attractive field is straightforward to compute by taking the derivative of (85). The result is a simple proportional control statement based on distance to the goal.

$$\nabla U_a(q) = k_a(q - q_g) \tag{86}$$

The repulsive field is slightly more difficult to compute since we want the influence of obstacles to be local (not extend forever). One example of a typical repulsive field is given in (87), where the maximum extent of an obstacle's influence is given by $\rho_m$ and the magnitude of the repulsive field is modulated by $k_r$.

$$U_r(q) = \begin{cases} \frac{1}{2}k_r \left(\frac{1}{\rho(q)} - \frac{1}{\rho_m}\right)^2 & \rho(q) \leq \rho_m \\ 0 & \rho(q) > \rho_m \end{cases} \tag{87}$$

The derivative of the repulsive field is given by (88).

$$\nabla U_r(q) = \begin{cases} -k_r \left(\frac{1}{\rho(q)} - \frac{1}{\rho_m}\right) \frac{1}{\rho^2(q)} \frac{(q-q_b)}{\rho(q)} & \rho(q) \leq \rho_m \\ 0 & \rho(q) > \rho_m \end{cases} \tag{88}$$

The overall force on the robot at any location $q$ is given by (89), where the gradient fields are given by (86) and (88).

$$F(q) = -\nabla U_a(q) - \nabla U_r(q) \tag{89}$$

There are a number of problems with potential fields. In particular, concave obstacles or complex arrangements of convex obstacles can produce potential fields that lead to a local minimum or to oscillatory behavior. In many situations, however, they offer an approach that effectively combines both motion planning and obstacle avoidance. Local obstacle avoidance is a strength of the method.

Potential fields may also be combined with a global search method. The global search method provides a path that avoids local minima, while the potential field locally provides optimal obstacle avoidance behavior, such as keeping maximally away from obstacles on all sides.

**Extended potential field** methods extend the potential field concept by modulating the strength of the field depending upon the robot's orientation with respect to a gradient. For example, an extended potential field will reduce the effect of an obstacle that is parallel to the robot, such as a wall. Only obstacles toward the front of the robot exhibit their full repulsive effect.

# 9 Localization

Given a map, generated manually or automatically, an essential capability in robotics is to figure out where the robot is located on the map. This problem is **localization**. There are three variations on localization.

1. Path estimation: the robot starts in a known location and must maintain knowledge of its position.
2. Global localization: the robot starts in an unknown location and has to estimate and then maintain knowledge of its position.
3. Kidnapped robot problem: the robot has an estimate of its location, but it is moved to a new unknown location with no warning and must detect and correct the error.

The fundamental approach to localization is to use a Bayesian filter and integrate information from all of the robot's sensor information and motion commands over time. There are a number of variations on the concept, but all of them produce some kind of probability distribution indicating the robot's likely location. Examples include Kalman filters, Markov localization, and particle filter localization.

Kalman filter localization tends to be the fastest approach because of its closed form implementation and the fact it maintains only a single hypothesis. However, Kalman filter approaches behave poorly on the kidnapped robot problem because they don't have a built-in mechanism for recovering if the localization estimate fails (e.g. someone picks up the robot and puts it down in a different place).

The following sections describe Markov localization and particle filter localization, both of which can recover from failures in localization and deal with the kidnapped robot problem because they maintain multiple localization hypotheses.

## 9.1 Markov Localization

One of the simplest approaches to localization is **Markov localization**. Markov localization works with both topological maps and approximate decompositions (e.g. evidence grids). A topological map, for example may be a set of landmarks or rooms with their connectivity represented as a graph. An approximate decomposition, such as an evidence grid, is an actual metric map with obstacles indicated by probabilities.

In both cases there are a finite number of locations for the robot. Markov localization associates a probability with each possible location, with the sum of the probabilities equal to one. For an approximate decomposition and a differential drive robot on a plane, the space of possible locations is 3-dimensional $(x, y, \theta)$. Common grid sizes for each dimension are 10cm in $x$ and $y$ and 5 degrees in $\theta$.

Each grid location represents a single Bayesian filter. At each time step, the robot executes a motion and collects a set of measurements. The open loop step updates the probabilities of each grid location using the motion model. The a priori probability of the robot being in a particular map location $m_{xy\theta}$ is the sum of the probabilities of the robot being in all nearby cells and ending up in $m_{xy\theta}$.

The closed loop step updates the probability of each grid location using the current measurements. Given the map and the robot's location, the a posteriori probability is the probability of being in that location and the probability of seeing the robot's measurements.

The Markov localization algorithm is as follows.

1. Given: Map $M$, set of initial probabilities $P_0 = \{p_{0,xy\theta}\}$
2. for all time $t$
   (a) Given: set of measurements and motor commands at time $t$: $(z_t, u_t)$
   (b) for all map locations $m_{xy\theta}$
      i. $p_{t,xy\theta}(-) = \sum_{i \in \text{Neighborhood}} p_{t-1,i}(+)\text{MotionModel}(m_{xy\theta}, m_i, u_t)$
      ii. $p_{t,xy\theta}(+) = \eta\text{MeasurementModel}(z_t, m_{xy\theta}, \text{Map})p_{t,xy\theta}(-)$

The open loop step incorporates the motion model, and the closed loop step revises the probabilities based on the measurements and the map. The normalization constant $\eta$ ensures that the sum of the probabilities after update is one.

- If the robot knows where it is when it begins, then the initial probability distribution is a delta function at the starting location and zero elsewhere. Over time, dispersion due to the motion model will fill in non-zero probabilities along the robot's path.
- If the robot does not know where it is when it begins, then the initial probability distribution will be uniform across the grid (except where there are obstacles). Over time, the distribution should converge to the robot's likely location.

Thrun suggests that the number of measurements required to obtain reasonable results is small. Of the 600+ readings available from a laser, for example, they suggest good localization is possible using just 8 readings (possibly virtual sensor readings that combine several measurements).

The higher the grid resolution, the more quickly the system tends to converge to a strong estimate. Increasing the angular resolution tends to help more than increasing the spatial resolution. Small quantization errors in the angle can generate large differences between the actual and estimated sensor measurements.

One issue to consider in Markov localization is the relationship between grid size and the motion and sensor models. With a 10cm grid size, for example, the actual noise on a laser sensor is much smaller than the grid size. Within a single grid cell, the laser reading could vary by up to $\pm 7$ cm. Therefore, to avoid the true grid cell receiving a low probability, the sensor noise must be large enough that the entire 10cm x 10cm cell falls within a high probability range given the reading. Increasing the noise levels of the sensors, however, reduces the amount of information provided by each reading, reducing the speed of convergence.

The same relationship holds with the motion model. A large spatial grid size may require the robot to make several updates–with different sensor readings–within the same grid cell. The robot may obtain up to 10 sensor readings while traversing a 50cm grid size, for example. The probability distribution of the motion becomes much more accurate with a smaller grid size.

Integrating many different sensors into a Markov localization scheme is straightforward. Each sensor simply requires a sensor model to estimate the probability of a reading given the robot's location and the map. The framework enables the robot to integrate lasers, sonars, IRs, and visual landmarks into the same localization process.

## 9.2 Particle Filter Localization

The basic particle filter algorithm works well for localization, although for a three-dimensional robot state space up to 100,000 particles are necessary for accurate localization given an unknown starting location. Unlike Markov localization, the particle filter doesn't require discretization of the robot's state space. It can, therefore, provide more accurate localization with less computation. For comparison, a 10m x 10m map using a 10cm grid size and 3.6 degree angular resolution requires 1 million discrete probabilities.

The particle filter localization algorithm is very similar to the Markov Localization algorithm, with the addition of a sampling step after the update step.

**Particle Filter Localization Algorithm**

1. Given: map $M$, initial set of $N$ particles $X_0 = \{x_i, \pi_i\}$
2. for all time $t$
   (a) Given: set of measurements and motor commands at time $t$: $(z_t, u_t)$
   (b) $\bar{X}_t = \emptyset$
   (c) for all particles $i$
      i. $x_{i,t} = MotionUpdate(u_t, x_{i,t-1})$
      ii. $w_{i,t} = p(z_t|x_{i,t}, M)$
      iii. $\bar{X}_t = \bar{X}_t \cup (x_{i,t}, \pi_{i,t})$
   (d) $X_t = \text{WeightedSample}(\bar{X}_t, N)$

To discover where the robot is located, you can use the centroid of the particles, choose the highest probability particle, or pick the particle with the most local support (high probability nearby particles).

In practice, the above algorithm does not do well with the kidnapped robot problem because after some number of generations the samples will cluster around the robot's actual location. If we then moved the robot to a new location, there would not be any particles in the vicinity of its new location.

A variation on particles files we have already examined–adding random particles–is a simple solution to the problem. However, we can be more clever about how and how many particles we add.

- We want to add more random particles when the current estimate of the robot's location is poor, fewer if the current estimate is strong.
- Instead of adding completely random particles, we would like to add particles that are likely given the current sensor reading.

**Particle Filter Localization Algorithm with Random Samples**

1. Given: map $M$, initial set of $N$ particles $X_0 = \{x_i, \pi_i\}$
2. Given: $\alpha_{\text{fast}} > \alpha_{\text{slow}} > 0$
3. Given: $w_{\text{fast}} = 0, w_{\text{slow}} = 0$
4. for all time $t$
   (a) Given: set of measurements and motor commands at time $t$: $(z_t, u_t)$
   (b) $w_{\text{avg}} = 0$
   (c) $\bar{X}_t = \emptyset$
   (d) for all particles $i$
      i. $x_{i,t} = MotionUpdate(u_t, x_{i,t-1})$
      ii. $\pi_{i,t} = p(z_t|x_{i,t}, M)$
      iii. $\bar{X}_t = \bar{X}_t \cup (x_{i,t}, \pi_{i,t})$
      iv. $w_{\text{avg}} = w_{\text{avg}} + \frac{1}{N}\pi_{i,t}$
   (e) $w_{\text{fast}} = (1 - \alpha_{\text{fast}})w_{\text{fast}} + \alpha_{\text{fast}}w_{\text{avg}}$
   (f) $w_{\text{slow}} = (1 - \alpha_{\text{slow}})w_{\text{slow}} + \alpha_{\text{slow}}w_{\text{avg}}$
   (g) for $N$
      i. $\beta =$ Uniform(0.0, 1.0)
      ii. if $\beta < \max(0.0, 1.0 - w_{\text{fast}}/w_{\text{slow}})$
         A. draw a new random particle from an importance distribution
      iii. else
         A. draw a particle from $\bar{X}_t$ using weighted sampling
      iv. add the particle to $X_t$

The new part of the algorithm maintains the two values $w_{\text{fast}}$ and $w_{\text{slow}}$, which represent short-term and long-term weight averages, respectively. When the distribution is condensed, or condensing, then the short term average weight should be larger than the long-term average weight. In that situation, the system creates no random particles. If the short-term average is less than the long-term average, then the distribution is flattening, and there will be a non-zero probability of creating a new random particle.

New random particles do not need to be chosen uniformly and randomly from the set of possible robot states. Instead, we can use the sensor model and the map to identify potential likely locations for the new particle. Inserting the new particles in likely locations should help the system converge more rapidly both initially and in response to a localization failure.

The same reasoning used above to figure out how many particles to initialize randomly also applies to the overall number of samples required to localize the robot. If the particles are highly clumped around a likely robot state, then we do not need as many particles to represent the probability distribution. If, however, the particles are spread around the robot state space, then we need more particles to efficiently find the most likely robot state.

The mathematical reasoning behind the KLD, which stands for Kullback-Leibler divergence, particle filter variation is based on measuring the likelihood that a set of samples drawn from a distribution is representative of the true probability distribution. Consider, for example, a complex, multi-modal 1-D probability distribution. If we draw a few samples from the distribution, they will tend to be spread out, suggesting that the distribution is not compact. As we continue to draw more particles, the shape of the distribution will start to appear, and new samples will end up close to old samples. When most of the new samples provide little additional information, we can stop adding samples to the distribution representation.

If the 1-D distribution were a Gaussian with a small $\sigma$, then after just a few samples the shape of the distribution would become clear. Most of the samples would fall into a small range of the space, with just a few representing the tails. New samples will provide little additional information much more quickly with a compact distribution than with a complex distribution.

The KLD particle filter algorithm is as follows.

1. Given: map $M$, initial set of $N$ particles $X_0 = \{x_i, \pi_i\}$
2. for all time $t$
   (a) Given: set of measurements and motor commands at time $t$: $(z_t, u_t)$
   (b) Given: $C = 0$, $C_X = \infty$, $k = 0$
   (c) Given: particle set $X_t = \emptyset$
   (d) Given: histogram $\{b_j\} \in H$ with $H = \{0\}$
   (e) do
      i. draw sample $(x_{i,t-1}, \pi_{i,t-1})$ with probability $\propto \pi_{i,t-1}$
      ii. $x_{C,t} = \text{MotionUpdate}(u_t, x_{i,t-1})$
      iii. $\pi_{C,t} = p(z_t | x_{C,t}, M)$
      iv. $X_t = X_t + (x_{C,t}, \pi_{C,t})$
      v. if ( $x_{C,t}$ falls in an empty bin $b_j$ ) then
         A. $k = k + 1$
         B. $b_j = b_j + 1$
         C. if ( $k > 1$ ) then $C_X = \frac{k-1}{2\epsilon}\left(1 - \frac{2}{9(k-1)} + \sqrt{\frac{2}{9(k-1)}} z_{1-\delta}\right)$
   (f) $C = C + 1$
3. while( $C < C_X$ )

The counter variable $k$ measures the number of histogram bins filled with at least one particle. $C$ counts how many particles the process has drawn so far. $C_X$ specifies how many samples are needed to represent the true probability distribution with confidence represented by $z_{1-\delta}$ given the spread of the samples. For 99% confidence, for example, $z_{1-\delta} = 2.58$. The constant $\epsilon$ is the desired maximum error between the true probability function and the sample representation. Thrun suggests a value of 0.05 for $\epsilon$.

As the process draws samples, $C_X$ will increase whenever a sample falls into an empty bin. Each sample, however, increases $C$. As the number of particles becomes a good representation of the probability function, $C_X$ will increase more slowly than $C$, and eventually the iteration will complete when $C > C_X$.

The KLD variation on particle filters is applicable to not just localization, but any particle filter implementation. Researchers have demonstrated that the KLD filter outperforms a standard particle filter with much less computation. For example, using an average of 3000 particles, the KLD filter is able to perform as accurately as a standard particle filter using 50,000 particles. The key to that success is that in a standard particle filter, most of the particles represent a very small portion of the state space.

## 9.3 Dynamic Environments and Gating

The localization methods presented so far all assume a static environment. However, in most realistic situations there will be dynamic obstacles in the environment such as people or other robots. If we use the standard localization methods, those dynamic obstacles will quickly confuse the robot's current state estimate.

One approach to handling dynamic obstacles is to use a concept called **gating**. Given an estimate of the robot's current location and orientation, and a simple sensor model, we can predict the likelihood of seeing any particular reading. If the reading is surprising, we may want to drop the reading.

When dealing with dynamic obstacles, we can even make the further constraint that the issue will generally be readings that are too short, rather than readings that are too long (which may indicate the robot is not where it thinks it is).

In a particle system, each particle provides an estimate of the robot's state. Therefore, we can estimate the likelihood of each measurement for each particle. A simple approach to gating is to look at the collective likelihood of each measurement across all particles. Those measurements that have good support participate in the update process. Measurements without good support get discarded.

In the algorithm below, the process takes in a measurement $z_t$, a set of particles $X_t$, a map $M$, and a threshold $\chi$. The values $z_{\text{hit}}$, $z_{\text{short}}$, $z_{\text{max}}$, and $z_{\text{rand}}$ are pre-selected parameters that describe all the possible sources for a measurement $z_t$. The four parameters sum to one: $z_{\text{hit}}+z_{\text{short}}+z_{\text{max}}z_{\text{rand}} = 1$. Dividing the possible sources of $z_t$ into parts makes it easier to represent the total probability distribution for a particular sensor. If, over all the particles, the probability of the measurement being a hit is sufficiently large relative to the probability of all the possible interpretations for the reading, then the measurement should be accepted.

1. Given: $z_t, X_t, M$
2. let $p = q = 0$
3. for each particle $i$
   (a) $p = p + z_{\text{hit}}p_{\text{hit}}(z_t|x_{i,t}, M)$
   (b) $q = q+z_{\text{hit}}p_{\text{hit}}(z_t|x_{i,t}, M)+z_{\text{short}}p_{\text{short}}(z_t|x_{i,t}, M)+z_{\text{max}}p_{\text{max}}(z_t|x_{i,t}, M)+z_{\text{rand}}p_{\text{rand}}(z_t|x_{i,t}, M)$
4. return $p/q \leq \chi$

# 10 Simultaneous Localization and Mapping

Localization given a map is a challenging task, but Markov and MCL localization work reasonably well. Mapping given known positions works well using evidence grid techniques. The real challenge occurs when the robot doesn't have a map or an independent method of localizing itself.

Simultaneous localization and mapping [SLAM] is one of the holy grails for robotics. With SLAM capability, a robot can explore a novel situation, build maps, plan explorations, and navigate back to prior locations on the map. It is a challenging problem because it incorporates three kinds of uncertainty.

- Uncertainty in sensor measurements
- Uncertainty in the robot's motion
- Uncertainty in associating map features over time.

## 10.1 FastSLAM 1.0

FastSLAM is a particle filter based SLAM algorithm that takes advantage of the fact that the uncertainty associated with the location of landmarks, or features in the environment are independent except for the uncertainty in the robot's path. Given perfect knowledge of the robot's location, any uncertainty in the location of two landmarks would be completely independent.

Therefore, if we have an estimate of the robot's location, then we can represent the location and uncertainty of each landmark independently using a low-dimensional (e.g. 2) extended Kalman filter [EKF].

FastSLAM uses a particle filter to represent the probability distribution for the robot's location $(x, y, \theta)$, path, and map. Each particle contains the robot's current state $\mathbf{x} = (x, y, \theta)$, a complete path of prior robot locations, and a complete map, represented by a set of extended Kalman filters $M = \{(\mu_i, \Sigma_i)\}$, one for each feature in the map.

A feature in the map could be the piece of an obstacle seen by a single laser reading or an identifiable feature such as a corner or post that agglomerates several sensor readings. The latter is easier to match from time step to time step, a task called data association. The following algorithm assumes perfect data association.

The overall FastSLAM algorithm is as follows.

1. for each of M particles in $X_{t-1}$
   (a) Update the pose of the particle using the motion model to generate $x_t^i$
   (b) For each observed feature $z_t^j$, identify the correspondence $k$ for the measurement and incorporate the measurement $z_t^j$ into EKF$_k$ by updating the mean and covariance $(u_{k,t}^i, \Sigma_{k,t}^i)$.
   (c) Add new features to the map when a sensor reading has a low probability of corresponding to any existing feature.
   (d) Calculate a new importance weight $\pi_t^i$ based on all of the measured map features.
2. Execute a weighted resampling of the M particles to generate $X_t$

- **Problem representation:** The set of particles represents the joint probability distribution of the robot's location and the location of each feature in the map. Each particle represents a best estimate of the current location of the robot and the location of all of the features it has seen so far. Implied in the representation is a particular data association matching each sensor reading to a particular feature. Different particles my have different data associations.

  A single particle at time step $t$ has the following form.

$$\mathbf{x}_t = (\{\vec{x}_i\}_t; \vec{u}_{1,t}, \Sigma_{1,t}, \ldots, \vec{u}_{M,t}, \Sigma_{M,t}) \tag{90}$$

  The set of terms $\{\vec{x}_i\}_t$ give the robot's state from time 1 to $t$. Each $\vec{u}_{i,t}$ term is a 2-D vector representing the location in the map of the ith feature. Each $\Sigma_{i,t}$ is a 2x2 matrix representing the uncertainty in the location of the ith feature at time $t$.

- **Robot State Update**: The control signal $u_t$ defines a probability distribution for the next location of the robot given the current state estimate. Each particle selects a sample from from the probability distribution $p(\vec{x}_t|\vec{x}_{t-1}, u_t)$ and adds it to the set of path coordinates.

- **Map Feature Update**: The new estimate of the robot's location $\vec{x}_t$ and a data association mapping sensor readings to map features $c_t$ provide the information required to update the map features.

  If no sensor reading maps to a particular feature in the map, then the mean and covariance of the feature remain the same. Features only update when they are associated with a sensor value.

  For each observed feature, the algorithm executes a standard EKF update process to update the mean and covariance of the feature.

$$K_t = \Sigma_{c_t,t-1}(+)G_t \left[G_t^T \Sigma_{c_t,t-1}(+)G_t + R_t\right]^{-1} \tag{91}$$

$$u_{c_t,t})(-) = I u_{c_t,t-1}(+) \tag{92}$$

$$u_{c_t,t}(+) = u_{c_t,t}(-) + K_t(z_t - \hat{z}_t) \tag{93}$$

$$\Sigma_{c_t,t}(+) = \left[I - K_t G_t^T\right] \Sigma_{c_t,t-1}(+) \tag{94}$$

  $G$ is the linearized version of the matrix transforming the sensor measurement in the robot's coordinates to global map coordinates. Therefore, $G$ will be a rotation matrix combined with a translation relative to the robot's origin that converts the feature's $(x, y)$ location into the measured distances $(\Delta x, \Delta y)$ from the robot. $G$ also properly orients the errors in the sensor.

  The state vector for each feature includes only the location $(x, y)$, and the objects are assumed to be static. Therefore, the motion model update for features is the identity.

  The matrix $R$ is the covariance matrix of the noise in the sensor. Assuming a x-axis pointing forward, the noise in the $x$ direction, which is determined by the accuracy of the distance measurement, will generally be larger than the noise in the $y$ direction, which is determined by the angular resolution of the laser.

- **Reweighting**: The next step calculates the new particle weights based on how well the sensor readings correspond with the proposed map features (remember, all map features are also estimated). The key concept in this step is that the overall uncertainty for a particular particle is the combined uncertainty of how well its current readings match the current map estimate.

## 10.2 FastSLAM 2.0

FastSLAM 1.0 was one of the first effective, and fast SLAM implementations that could be run on a robot in real time. The key insight that made it possible was the factorization of the features of the map into individual state vectors and filters, rather than representing all of the features in a single state vector and filter.

One problem with FastSLAM 1.0 is that the set of particles generated in the motion model do not take into consideration the current measurement. While this is standard practice in most filters–open-loop update, followed by the closed-loop update–it means there are many particles generated by the process that may then be discarded after going through all the work of updating their features and calculating the weight of the particle. For a large map, this can be expensive.

FastSLAM 2.0 [Montemerlo et. al., IJCAI 2003], incorporates the current measurement into the motion model update step. If we consider the typical motion model to be the combination of two Gaussian distributions–one in angular velocity, the other in translational velocity–then FastSLAM 2.0 adds to that combination the probability of seeing a particular measurement given the prior map. By updating particles using the combined PDF, the filter sends more particles to high probability locations.

The main modification to the FastSLAM algorithm is the resampling step, which needs to modify the weighting function to take into account that the measurement has already been partially incorporated in the motion model step. The end result is effectively a Gaussian weighting scheme based on the measurement, but calculated slightly differently than in FastSLAM 1.0.

## 10.3 Data Association

Data association is a necessary part of all SLAM algorithms. There are two questions that must be answered for each measurement. First, does the measurement correspond to any existing feature in the map? Second, if the measurement does correspond to a feature in the map, which feature is it?

A common method of data association in particle-based SLAM algorithms is to use maximum likelihood data association. In other words, for each reasonable feature on the map, which feature has the highest likelihood of generating the measurement given the robot's estimated location? Features might be grid squares in an evidence grid, line segments, corners, circles, or other patterns. Generally, the likelihood model will be a 2D Gaussian distribution with a covariance matrix determined by the orientation of the sensor relative to the feature. Laser measurements, for example, will be precise in both distance and angular resolution, while sonar measurements will be much more precise in distance than in angular resolution.

If no feature in the map is found to have a likelihood higher than a selected threshold, then a new feature is added to the map and inserted into the particle as a new extended Kalman filter.

$$s_t \sim p(s_t|s_{t-1}, u_t) \tag{95}$$

$$\mu_{n,t} = g^{-1}(z_t, s_t) \tag{96}$$

$$\Sigma_{n,t} = (G_n R_t^{-1} G_n^T)^{-1} \tag{97}$$

$$w_t = p_0 \tag{98}$$

## 10.4 Feature Management

Features not seen by a sensor set should remain unchanged, as there is no new information to update their location. Features detected appropriately should be updated according to their EKF. However, in some cases there are features that should be detected, but are not.

As with all mapping systems, it is possible for spurious features, or dynamic features, to be placed into the map. To avoid this type of clutter, SLAM methods need to have a mechanism for removing features from the map. One method of managing features is to use the occupancy grid approach. The system can maintain a set of variables, one for each existing features, that increment each time a feature is detected and decrement each time a feature should be seen but is not. If the odds of the feature existing fall below a certain threshold, then the system removes the feature from the map.

By using a log representation of the probabilities, incrementing and decrementing the odds for a given feature is a fast update.

## 10.5 Least Squares SLAM

An alternative approach to SLAM is to treat the problem as one of graph optimization. The graph connects physical locations in space–the map–and the edges define the spatial relationships between the feature points.

Given an actual feature as $z_i$ and the robot's state $x$, there is a predicted location for the feature $f_i(x) = z_i'$ that is a function of $x$. We are interested in the error between the predicted location and the actual location. The following is a least squares solution for the localization problem, given a map and a data association method.

$$e_i(x) = z_i - f_i(x) \tag{99}$$

If we assume the error to be zero mean and normally distributed, then this becomes a linear least squares optimization problem. To properly represent the measurement error we use a noise covariance matrix $\Omega_i$, since the measurement $z_i$ is a vector (e.g. x, y, possibly z).

$$E(x) = \sum_i e_i^T(x)\Omega e_i(x) \tag{100}$$

The solution to this problem identifies the $x$ that minimizes the error. This requires a numerical approach, as there is no closed form solution.

$$x^* = \text{argmin}_x E(x) = \text{argmin}_x \sum_i e_i^T(x)\Omega e_i(x) \tag{101}$$

If a good initial guess is available, such as reasonable odometry, then we can solve the problem by iterative local linearization and use a gradient descent type methodology. Linearization involves taking the Taylor series of the error expression, which results in a constant term plus a term proportional to the Jacobian, which is the matrix of first derivatives of the error with respect to changes in the state.

$$e_i(x + \Delta x) \backsimeq e_i(x) + \frac{\partial e_i}{\partial x}\Delta x = e_i + J_i \Delta x \tag{102}$$

Substitution of the Taylor series expansion back into the original error expression results in a quadratic expression of $\Delta x$, which can be minimized. The process proceeds as follows.

Compute the local linear representation of the error, given the current estimated state.

$$e_i(x + \Delta x) \backsimeq e_i + J_i \Delta x \tag{103}$$

Compute the terms for the linear system.

$$b^T = \sum_i e_i^T \Omega_i J_i \tag{104}$$

$$H = \sum_i J_i^T \Omega_i J_i \tag{105}$$

Solve for the optimal update to the system $\Delta x^*$.

$$\Delta x^* = -H^{-1} b \tag{106}$$

Update the state estimate.

$$x \leftarrow x + \Delta x^* \tag{107}$$

To solve the SLAM problem in a least squares, graph-based method, we construct a graph of the robot's locations $x_i$ and build edges between them. The graph is a set of nodes in 2D or 3D each representing the pose of the robot at a time $t_i$. Constraints between the nodes can be measurement constraints–the robot observed the same part of the environment in the two states–or they can be odometry constraints. The key insight is that if we have really good odometry, the map is just a simple evidence grid. We get really good odometry by adjusting the robot's pose estimates so that we minimize the difference between measurements of stuff in the world seen at different locations.

The process is identical to the process for localization, but the equations are slightly more complicated.

1. Define an error function: minimize the overall error in the graph.
2. Linearize the error function (Taylor series).
3. Compute the derivative of the error function.
4. Set the derivative to zero and solve the linear system to identify the optimal step.
5. Iterate the procedure until convergence.

You do not have to explicitly represent features, or grow a list of features as the system explores, because you are optimizing the relationship between robot locations, not the location of features on the map. As new features become visible in a sensor reading, they will be used, as appropriate, to match nearby states in the graph. It is also possible do local updates on the graph as the robot traverses the space, because new measurements do not have a large effect on nodes that are far away.

# 11 Robot Motivation

## 11.1 Robot Emotion

Sloman and Croucher (IJCAI '81) argued that emotions are a necessary side-effect of intelligent systems with multiple motives and limited control over the world.

- The intellect has multiple overlapping (in time and energy) motivations and goals.
- The goals interact with external obstacles and entities in the environment.
- Motivations and goals constantly change and evolve over time.
- Emotions are one method of moving goals up and down the priority list.

More recently, a number of researchers have explored how to explicitly build emotions into robot control systems. Emotional models tend to follow a general structure.

- The robot is trying to achieve goals.
- The robot has a set of parameters, possibly with semantic labels, that define an emotional state.
- The robot has a set of attitudes, which are partitions of the parameter space.
- The robot's current attitude modulates the set of available actions, probabilistically or crisply.
- The robot's environment and internal state affect the parameters defining the emotional state.

Note that environmental and internal conditions will likely affect the emotional state differently than they impact the immediate decision-making process. Emotions will have a varying effect on decision-making. In some cases they enable or disable actions almost completely, while in other situations they have subtle effects on the robot's actions. Consider your own emotions and how they can affect your behavior in different situations.

Emotions, defined broadly, fall into three categories depending upon their drivers.

- Internal emotions: caused by internal state variables such as thirst, hunger, exhaustion, or a need for air. In a robot, these would correspond to power levels and the functionality of mechanical and electrical components.
- External emotions: caused directly by events in the environment. Examples include happiness, surprise, anger, sadness, satisfaction, and frustration. Are these are the most appropriate emotions for a robot? Satisfaction seems like it might be an appropriate emotion as a method of reinforcing successful action sequences. Fight or flight may also be appropriate in some situations.
- Social emotions: caused by comparison of the actor's behavior to cultural norms. Shame and pride or patriotism would fall in this category, since they require a social context. A person living alone in a forest would not necessarily experience these emotions, but could certainly experience the other two.

A complete emotional model will need to incorporate all three types of emotional state, possibly as separate state spaces. Internal emotions, for example, can provide generic long-term goals (e.g. eating) or completely override all other decision-making mechanisms. External emotions play a more short-term and modulated role. In a robot, we may not want external emotions to completely override decision-making, since it, arguably, often results in non-optimal decision-making by people. Social emotions may be necessary to assist the robot in human-robot interaction.

## 11.2 Novelty, Curiosity and Exploration

P-Y Oudeyer, F. Kaplan, and V. Hafner, "Intrinsic Motivation Systems for Autonomous Mental Development", *IEEE Trans. on Evolutionary Computation*, 11(2), 2007.

Emotion is not the same as motivation, or a tendency to explore and learn. The field of developmental robotics looks at the question of how we can build systems that not only learn about their environment, but do so autonomously. Developmental robotics has pulled from psychology, neuroscience, and experimental systems to derive theories about how we could create robots that not only have the capacity to learn, but are continually attempting to increase their knowledge.

Psychology

- Basic forms of motivation cannot account for all exploratory behaviors.
- Exploratory behavior must have its own reward, beyond satisfying basic needs.
- The reward of exploratory behavior must be an intrinsic motivation.
- Concepts such as novelty, surprise, incongruity, and complexity appear to be rewarding to humans.
- Situations with an intermediate level of novelty appear to have the most reward.
- Self-engagement in activities which require skills just above our current level are rewarding when we reach a new level of capability.
- The reward is greatest when the time spent mastering an activity is not too small and not too large.

Neuroscience

- Research has shown that dopamine neurons in the midbrain are responsive to the error in predicting expected reward delivery. These studies have focused on predicting extrinsic rewards.
- Recent research has explored the possibility that these systems are also sensitive to prediction error alone, not just prediction of rewards. A novel situation, therefore, may produce a positive response in the human brain, encouraging a search for novelty (but not too much novelty).

Developmental Robotics

- The goal is to create a system that is motivated to explore and learn.
- In order to create a system that likes a little bit of prediction error, we need to develop definitions of novelty, surprise, complexity, and challenge.
- What do these terms mean in the context of a robot, its sensors, and its hardware?
- How do we define what an intermediate level of novelty is? (We need a Goldilocks distance metric.)
- How do we store what is known and revise our definition of novelty once we have learned one task?

Active Learning

- One question in machine learning is how to present patterns to a system such that it learns as quickly as possible with the fewest number of examples.
- Active learning systems have been developed that work in discrete domains with nice noise properties.
- How do we extract what is relevant from the environment with respect to what we are learning?

Implementations of existing systems are built around a system that tries to predict what the robot will see in the next time step. Another way to state this is that the robot is trying to model the effect of its actions, and those effects are measured by the robot's sensors.

In order to learn, the robot must take actions that have unexpected effects.

- If the robot takes only actions with effects it can predict, it will not learn anything novel.
- If the robot takes actions that are completely outside the scope of what it has learned, it will be difficult for it to extrapolate any knowledge from the resulting effects.
- If the robot takes actions that have effects that are similar to, but somewhat different than its expectations, it will be able to usefully modify its predictor (learning mechanism).

A developmental robot, therefore, must have the following elements

- A machine learning system that can predict the effects of an action on what is measured by its sensors.
- A meta-learning system that can learn the expected differences between the predicted and actual results and thus identify situations that are actually novel, or outside the bounds of the types of errors normally expected.
- An action selection mechanism that uses the output of both systems to determine the next action of the robot.

Using only the meta-learning system output can result in the robot getting stuck in a situation where it is outside the bounds of what it has learned and the average error rate of the learning system is high and not improving. A good situation for the robot is when it is in a novel situation and the error of the machine learning system is decreasing as it learns to predict the effects of its actions. Once the error is sufficiently low and prediction errors are correspondingly small, it should be motivated to move to a new situation where it can learn something new.

Note, however, that in the real world, we may want a system to quickly change tasks–from grasping to visual tracking, for example–in which case there also needs to be some type of state information incorporated into the assessment of what is novel and useful to pursue. However, for developmental robotics, this task of identifying state, or categories in the world, also has to be developed autonomously. The robot has to be able to develop a concept of situation similarity and merge together experiences in similar situations.

Oudeyer et. al. propose a mechanism called Intelligent Adaptive Curiosity [IAC], which is an example of an implementation of a developmental robotic system with the ability to adaptively change its actions based on which actions will enable it to most effectively learn.

- The system relies on a memory which stores all of the experiences of the robot as vector exemplars.
- The system splits the space of exemplars into regions.
- Each region contains a learning machine, or expert, trained on the local exemplars.
- Each region stores a list of the prediction errors it has made on events that occur in its region.
- The lists permit evaluation of learning progress in each region.
- The robot selects actions based on which action will lead to maximal learning progress.
- As the learning machine in a region becomes better, other activities will take over the learning process.

### 11.2.1 Implementation: IAC

- An **experience** $SM(t)$ is a vector containing the sensor and motor data from a single time step
- An **exemplar** is a combination of the sensory-motor vector $SM(t)$ and the sensor response from the next time step $S(t+1)$: $(SM(t), S(t+1))$.
- A **region** is a set of exemplar vectors: $\Gamma_n = \{(SM(t), S(t+1))_i\}$
- Regions can be created simply or in a more complex fashion (e.g. online clustering)
  - Oudeyer et. al. use a simple counting threshold: split a region when the count is over $T = 250$.
  - The split is along one dimension of the exemplar vector, and the threshold is selected to minimize the within variance of the two groups. The split dimension has to be part of the $SM(t)$ component of the exemplar, since queries will not have the subsequent sensor vector.
  - Identifying the region of a new exemplar is a series of threshold tests.
- Experts can be KNN, simple neural networks or other ML methods. They use a simple 1-NN learning system, finding the best match between the query sensory-motor vector and the set of exemplars in the region.
- At each time step, the system computes the squared prediction error between the current sensor readings at time $t+1$ and the readings predicted at time $t$.
- The system then computes a smoothed derivative of the error, using a simple smoothing method that takes the difference between the mean error over windows that are some distance apart (e.g. a window 25 samples wide, with the window centers 15 samples apart).
- The learning progress function is the derivative of the learning error: $L(t+1) = -D(t+1)$
- The internal reward function is the learning progress function, as learning is rewarding: $r(t) = L(t)$
- At any given time $t$, the robot is in state $s$ and has to select an action $a$. In the context of the IAC system, a state is a sensor vector $S(t)$, an action is a set of motor commands $M(t)$, the reward for taking that action in that state is $r(t+1) = L(t+1)$, and the robot ends up in state $S(t+1)$. The goal of action selection is to identify the action that will produce the greatest reward over time. Given that we have perfect state recognition, but we do not know the reward space a priori, this problem is identical to reinforcement learning, or Q-learning. At each time step, the robot can take the prior state $s$, action $a$, and next state $s'$ and update the Q value $Q(s, a)$.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * \max_{a'}(Q(s',a')) \tag{108}$$

  The $a'$ are all of the possible actions to be taken from $s'$ and the $Q(s', a')$ are the possible next states. The discount parameter is $\gamma$, and $\alpha$ is the learning constant.
- Oudeyer et. al. simplify the discounted reward to be simply the reward at time $r(t+1)$, which can be estimated using the region exemplars. To estimate the value of a next state, the system uses the current sensor vector, combines it with various motor vectors, and then identifies to which region each sensor-motor vector belongs. Each region has a history of prediction errors and an estimate of the learning derivative, which is the estimated reward function.

- Given a set of expected rewards given a set of possible motor actions, the robot selects the maximal result with some probability. Otherwise, it selects one of the remaining actions uniformly and randomly. This balances exploration with learning. They select the maximal action 65% of the time and a randomly selected action 35% of the time.
- The first test situation was a simulated robot environment (Webots) with a two-wheeled robot and a ball. The robot can also emit a signal, giving a 3-value control vector. If the signal is within a certain range $f_1$, the ball moves randomly. If the signal is within a range $f_2$ the ball remains still. If the signal is within a range $f_3$, then the ball moves towards the robot. The sensor state of the robot is a single value that indicates how far away from the robot the ball is located.

  In this experiment, the robot created regions of the sensor-motor space that corresponded to the three regions of the $f$ space. It also divided up the two predictable areas of the $f$ space into more subregions to enable more accurate prediction.
- The second test situation was an Aibo robot with visual recognition tags. The robot is able to turn its head, bash, and bite, and there are five motor parameters controlling these actions. The robot has high level sensors that enable it to know when there is a visual tag in its vision, whether something is in its mouth, and whether an object nearby is moving. The robot initially knows nothing about what any of the parameters or sensors controls or measures.

  The results of this experiment are more difficult to measure, but the different runs tend to show the robot focusing on activities for a period of time, and as a robot comes back to an activity it appears to be learning a more complex behavior. This makes sense, because the robot is rewarded for faster learning and will only take on more complex tasks that are slower to learn when other options have been exhausted.

### 11.2.2 Discussion

Some of the larger issues discussed by Oudeyer et. al. include the development of hierarchical learning structures and goal-directed behavior. Complex behavior patterns in humans, and potentially other animals, tend to be hierarchically organized. Certainly, planned sequences of actions are hierarchical; we divide a problem up into a sequence of actions, and each action may consist of a series of sub-actions, and so on until the actions are defined at some atomic level. A major question in developmental robotics is how to build systems where these hierarchical structures are organized autonomously. So far, hierarchical structures have all be pre-built into network structures or the algorithms used to facilitate development.

Most AI systems are set up with built-in goal directed behavior, or schemas, where the goals are external and measurable results. The most successful AI systems are all built around schemas or designs where goals are explicitly represented in the system's architecture. Such goals are easily monitored, and progress towards the goals is possible to measure. A major question in AI is whether goal-directed behavior can arise from sub symbolic computational systems. Q-learning in situations where the reward is unknown is probably as close as robotics/AI has come to self-organizing goal-directed behavior.

Developmental robotics still has along way to go before we can turn on the robot and let it figure out the world for itself. But there is progress on developing the kind of structures for learning that are required for a robot to start the process.

# 12 Multiple Robot Systems

Multiple robot systems have the same set of requirements as a single robot, with the expansion of the motion parameters to multiple autonomous physical units. The system has to sense the environment, determine a plan of action, and execute the plan. The same requirements on responsiveness, adaptation to changes in the environment, complexity in sensing, and the challenge of planning robust actions apply to the system as they apply to the individual physical elements. However, the degree of communication between the individual physical entities, and any differences in sensing and capabilities, impact the types of algorithms required to make a working system.

Consider the following types of communication and decision-making structures.

- Homogeneous physical entities with no explicit communication ability. In this case, the only way the robots can communicate is by changing the world, thereby changing the sensor readings of other robots. For example, ants leave scent trails where they walk, and other ants can sense those trails, especially if many ants have walked along the same path. Likewise, if a robot team needs to move a number of boxes from one location to another, they do not need to communicate directly about which box to take; each robot takes a box that is available, and each robot appropriately places its box at the new location. Potential collisions can be treated like network or memory access collisions; when a robot senses a potential collision it waits for a randomly chosen amount of time and then re-evaluates its plan based on the new state of the world.
- Heterogeneous physical entities with no explicit communication ability. This case is similar to the prior case, except that robots in different capability classes execute different algorithms. In the box moving example, if some robots are intended to lift light boxes, and others are intended to lift heavy boxes, then each class has different selection and placement algorithms. The heavy lifters might need to wait for a heavy box to be uncovered at the source pile, and the light lifters might need to wait for a heavy box to be placed at the destination pile, but no explicit communication is necessary for the team to complete the task.
- Distributed decision-making with low bandwidth communication ability. Whether the teams are homogeneous or heterogeneous, the ability to send low bandwidth messages changes the types of algorithms the robots can use. Low bandwidth teams cannot be fully directed by a single decision-maker, as there is not enough bandwidth to send sensor and location data for every robot to a central source. A team using distributed decision making uses the communication bandwidth to enhance each robot's understanding of the state of the world. That state can include, for example, the intentions of each robot in the team, or at least those close by, allowing each individual robot to make a more informed decision.

  The key is to make use of the low bandwidth to send the most useful information, in the sense that it reduces the cost to the team of completing the task. Having each robot send its location and state at all times, for example, is not necessarily beneficial in the box moving example, as each robot can sense the other robots that are close by. On the other hand, having robots claim boxes in a first-claim, first-take algorithm as they approach the source pile reduces conflicts between robots approaching the pile at similar times.
- Centralized decision-making with low bandwidth communication ability. Centralized decision-making is potentially the most efficient use of robot resources as the decision maker is able to take into account the entire scope of the problem and all of the resources currently ring used. However, with low bandwidth communication, the central decision maker cannot be providing low level control information

to the robots, as it does not have access to its raw sensor information, or even necessarily each robot's location sufficiently often to enable an effective control loop. Instead, the central decision maker sends out high level goals or actions and the individual robots have to have sufficient autonomous capability to carry them out. In the box moving example, the central decision maker could decide the order in which the boxes are to be moved, and which robots should pick up which boxes, but the robots themselves would be responsible for picking up and putting down the boxes and acting in turn without colliding with one another.

- Robot teams with high bandwidth communication ability. The ability to send high bandwidth communication facilitates decentralized modular processing, central decision-making, and potentially low level control of individual robots. The latter depends primarily upon the communication lag time. What high bandwidth communications facilitates most is the use of differential processing capability. A stationary computational facility, or a robot with extra computational ability can act as a central resource for multiple simpler robots. For example, the individual robots can send sensor and odometry readings to a central location where localization and mapping is executed as a single application. Likewise, decision-making can be more fine-grained, even if the robots are ultimately responsible for their own tight control and response loops.

As noted above, decision making for multi-robot systems can be decentralized, centralized, or some hybrid combination. Many robot tasks, such as box-packing, exploration, or perimeter management can be stated as the equivalent of a traveling salesman problem, which means they are computationally challenging to solve optimally as the task becomes large. Because of this, few systems try to do optimal planning, but instead make use of heuristics to find good solutions, even if they are not optimal.

## 12.1 Robot Soccer

An early example of centralized control was the CMU robot soccer system described in (Bowling and Veloso, 1999). In that situation, the simple robots were sent motor commands directly, and the field was monitored by an overhead camera system provided each robot's location and orientation, along with the position of the ball and the opponents. The method of choosing plays (pass or shoot) and which robot should play which role was calculated dynamically by a central decision-maker using a local greedy algorithm. At each instant, the system made a determination as to the best course of action and sent the appropriate commands.

The next generation of control systems used a more hierarchical approach, incorporating the concept of a play and a playbook into decision-making. At regular intervals, the system would examine a library of plays, select the play to execute, and assign tasks to individual robots. It would then evaluate the result of the play and use that as part of future play selection. Plays have initial conditions that allow them to be selected, they have a set of potential results, which include aborting the play, and plays can be adapted to local conditions. This system uses a form of reinforcement learning to keep track of the expected reward from plays–which are initially unknown–and adjust play selection in the future.

Early non-global robot soccer systems did not have communication between robots, and so relied on each robot running the same strategy program and selecting its role individually based on its perception of the world. Even when communications were allowed, by having each robot run the same algorithm on the same data–shared across robots–each robot could come to the same conclusion about the best course of action and its own role based on the common data.

## 12.2 Exploration and Mapping

Exploration and mapping efficiently with multiple robots is an example of a computationally hard problem, equivalent to a traveling salesman (or N traveling salesmen) problem. A number of different methods of planning have been used to solve these situations.

- Simulated annealing (Mosteo and Montano, ) : the solution to the N-traveling salesman problem can be viewed as an ordered vector, laid out with the order of each robot's tasks. Identifying the optimal solution is NP-hard, but heuristic approaches like simulated annealing can usually find a good solution in a short amount of time. Simulated annealing is effectively gradient descent combined with small random perturbations of the solution vector. At each step in an iterative process a perturbation of the current solution is generated. If the perturbation is better than the current solution, it is taken. If the perturbation is worse, it is taken with some probability. In some implementations, better solutions are not always taken, but can be rejected with some probability.

  Simulated annealing requires a solution representation, the ability to generate reasonable perturbations, and temperature, and a cooling schedule. At the beginning of the optimization process, the temperature is high and worse solutions are more likely to be taken. As the temperature cools, worse solutions are less likely to be taken. At a zero temperature, only better solutions are accepted. The details of both representing the solution and generating perturbations are critical choices.

  Mosteo and Montano use hierarchical task trees to represent a solution. They have a number of perturbations that can be made to a task tree. These include: random insertion, random task swap, random plan switch, guided movement where an expensive task is switched out of a plan, and two other heuristics. The combination of random changes and guided heuristics allows both local modifications to the plan and large jumps through the plan search space. Their system can explore 2000-6000 plans per second on standard hardware.

  Simulated annealing is similar to Monte-Carlo Markov Chain search. The only difference is that MCMC does not have a cooling schedule. Theoretically simulated annealing will find the optimal solution if the cooling schedule takes long enough.

- Auctions (Zlot and Stentz, 2005): auctions are an alternative, distributed method of identifying efficient plans for multi-robot systems. The basic idea is that the overall task is described as a set of sub-tasks. Simple auction methods let each robot compete for simple tasks, giving each task to the robot with the lowest bid. A robot's bid is based on the cost to that robot of executing the task. Each robot gets paid when it completes a task, so it wants to set its bids to maximize the difference between its cost and how much it pays for the task.

  More complex auction systems allow individual robots to bid for complex hierarchical tasks and then auction off sub-pieces of the complex tasks. Any robot can run an auction, and any robot can make bids. This allows the system to adapt to a changing environment, because a robot can auction off tasks it may have taken on but which have become expensive because of changes in the environment. Using task trees allows individual robots to vote not only on tasks, but on plans.

  Bids in an auction are based on the marginal cost to the robot of inserting the task into its current task set. In some cases this might require a robot to undertake a complex action, in other cases the robot may be able to execute the task without making significant changes to its overall plan.